Submitted in part fulfilment for the degree of
MSc in Software Engineering.

Formal Modelling of Separation Kernels

ANDRIUS VELYKIS

18th September 2009

Supervised by Dr Leo Freitas

Number of words = 45327, as counted by detex <report.tex> | wc -w.
This report consists of 98 pages in total.
This includes the body of the report (without blank pages) and Appendix A,
but not Appendices B, C, D, E and F.

1Updated transactional operation proofs, 21st September 2009.
Abstract

A separation kernel is an architecture for secure applications, which benefits from inherent security of distributed systems. Due to its small size and usage in high-integrity environments, it makes a good target for formal modelling and verification. This project presents results from mechanisation and modelling of separation kernel components: a process table, a process queue and a scheduler. The results have been developed as a part of the pilot project within the international Grand Challenge in Verified Software. This thesis covers full development life-cycle from project initiation through design and evaluation to successful completion. Important findings about kernel properties, formal modelling and design decisions are discussed. The developed formal specification is fully verified and contributes to the pilot project aim of creating a formal kernel model and refining it down to implementation code. Other reusable artefacts, such as general lemmas and a new technique of ensuring transactional properties of operations are defined. The results will be curated within the Verified Software Repository.
Acknowledgements

I would like to thank Dr Leo Freitas for his supervision, encouragement and getting me hooked on formal methods. I am also grateful to Iain Craig for the original formal specification of separation kernels. Most importantly, I would like to thank everyone who made this happen.
## Contents

1 **Introduction** .......................................................... 1
   1.1 Background & motivation ........................................... 1
   1.2 Objectives ............................................................ 2
   1.3 Outline of the thesis ............................................... 2
   1.4 Statement of ethics ................................................. 2

2 **Literature Review** .................................................... 3
   2.1 Grand Challenge in Verified Software ............................ 3
   2.2 Separation kernel ................................................... 4
   2.3 Verification of OS kernels ......................................... 5
   2.4 Formal modelling of OS kernels .................................. 7
   2.5 Z formal specifications ............................................ 8
   2.6 Summary .............................................................. 9

3 **Project Aims & Problem Analysis** .................................. 10
   3.1 Separation kernel requirements .................................. 10
   3.2 Formal model requirements ....................................... 12
   3.3 Existing research ................................................. 12
   3.4 Research goals and scope ........................................ 13
   3.5 Summary ............................................................ 15

4 **Design & Methodology** ............................................... 16
   4.1 Intended approach ................................................. 16
   4.2 Structuring the specification .................................... 18
   4.3 Verification of the model ....................................... 19
   4.4 Naming and style conventions .................................... 22
   4.5 Transactional operations ......................................... 23
   4.6 Z/Eves theorem prover ........................................... 28
   4.7 Separation kernel architecture .................................. 29
   4.8 Summary ............................................................ 32

5 **Basic Types, Hardware & Supporting Types** ....................... 34
   5.1 General lemmas .................................................... 34
   5.2 Process identifiers .............................................. 35
   5.3 Process type & external identifiers .............................. 35
   5.4 Process state ..................................................... 36
   5.5 Memory address ................................................... 36
   5.6 Task state segments ............................................... 36
   5.7 Interrupts .......................................................... 37
   5.8 Hardware .......................................................... 37
   5.9 Context switch .................................................... 38
   5.10 Error handling .................................................... 38
   5.11 Operation preconditions ......................................... 40
   5.12 Messaging .......................................................... 40
   5.13 Segment descriptor ............................................... 41
   5.14 Summary .......................................................... 41

6 **Process Table** ....................................................... 42
   6.1 State ............................................................... 42
   6.2 State evolution .................................................... 43
   6.3 Interesting properties ........................................... 48
   6.4 Initial state ....................................................... 48
   6.5 Error cases ........................................................ 49
   6.6 Query operations ................................................ 49
   6.7 Simple process allocation operations .......................... 50
List of Tables

5.1 General lemmas in the project ......................................................... 34
5.2 ErrV and HW operation preconditions .................................................. 40
6.1 PTab operation preconditions .............................................................. 60
7.1 PQueue operation preconditions ........................................................... 67
8.1 Sched operation preconditions .............................................................. 79
9.1 Summary of paragraphs ........................................................................ 84
9.2 Summary of proofs ............................................................................... 85
9.3 Summary of proof steps ...................................................................... 85

List of Figures

4.1 Top-level Established Strategy for separation kernel ................................... 17
4.2 Separation kernel memory overview ......................................................... 30
4.3 Separation kernel architecture, dissertation scope ....................................... 32
A.1 Separation kernel architecture ................................................................. 96
Chapter 1

Introduction

In this chapter the project is introduced and established within its context. Furthermore, general objectives are explored. Finally, the explanation of thesis structure is given and project ethical implications are justified.

1.1 Background & motivation

Computer systems span a wide range of devices, from the common desktop and laptop systems to embedded devices, which are used in everyday or high-integrity environments. With the computer-based systems handling the brakes of our cars, controlling engines of the aeroplanes we’re flying and setting the pace of our hearts, there is not much else to do than trust them.

For such safety-critical and high-integrity applications, security and reliability are mandatory. If taken even further, the applications should work flawlessly - thus emerges the notion of correct software. Correct programs are constructed with full assurance and knowledge of what they do, how they do it and why they work.

Formal methods allow achieving such level of assurance due to their roots in fundamental mathematics. It is a collection of mathematically-based languages, techniques and tools used throughout the life-cycle of software and hardware computer systems. The mathematical nature of formal methods enables users to produce precise and unambiguous documentation for domain modelling, requirements engineering, specification, design, development, testing and maintenance. The models can be verified to achieve necessary confidence or reasoning about model properties at appropriate abstraction level [8, 76, 78].

Long being the tool to document the domain, requirements and behaviour, formal methods now encompass all areas of software development. Maturing tools and increasing technological capabilities allow creating verified and correct models and refining them down to code level or even performing automatic code generation, while keeping the properties of the original model.

With the appropriate scene established, Tony Hoare united formal methods practitioners under the Grand Challenge in Verified Software with the belief that a global long-term initiative can achieve the ideal of correct software [28, 30, 62]. Within the Grand Challenge, researchers strive to create a strong engineering toolkit, the Verifying Compiler, which could automatically guarantee correctness of a program. Furthermore, the Verified Software Repository serves as a collection of appropriate range of examples, case studies and resources.

Following the success of initial pilot projects in verifying bank smart-card and flash storage with a file system [24, 35], a larger challenge of Verified Operating System Kernels was embarked. This pilot project strives to create correctly constructed kernels using top-down development through refinement [26].

The appreciation to formal modelling for operating system (OS) kernels comes from their importance within any computer system. Being a central part of most computer systems, a verified and reliable kernel increases its reliability and security. A number of projects verifying existing microkernels within the industry illustrate this fact [9, 39].

The separation kernel [55] is valued for its simplicity and security, therefore its usage is increasing within the applications in high-robustness environments. The partitioning properties of the kernel allow running processes on the same processor with inherent security properties of a distributed system. The clearly defined separation kernel requirements and its small size make it a good candidate for formal modelling and verification.

Craig [11] argues that formal methods can build new foundations for operating systems design. The abstract reasoning about the properties of the system can help define new models for operating systems. At least, a formally specified abstract operating system allows exploration of possible implementations, because all the properties applicable to the abstract model hold for the properly refined application.

A formal model of separation kernel presents a significant contribution to the verification of security kernels, embedded kernels and operating systems in general. This thesis is part of the pilot project of OS kernels verification and is concerned with the development of a formal model for separation kernels.
1.2 Objectives

The objectives set out for this thesis are in line with the associated pilot project. The main course is to define a correct and verified kernel model. Mathematical reasoning and proofs can be performed to verify kernel properties. Then, a correctly constructed kernel can be created by applying refinement techniques and tools.

As the basis of such endeavours, a book on formal specification and refinement of operating system kernels by Craig [12] is used. Craig is a domain expert in OS development and has devised, among all, an abstract model of separation kernel as a formal Z specification. Furthermore, the model is accompanied by a refinement towards an implementation code, which is also available.

Based on Craig’s specification and other contributions already existing within this pilot project, a mechanised model of a separation kernel needs to be defined. Mechanisation is done using a theorem prover, which is essential to finding various problems as well as proving mandatory and creatively identified properties of the model. A mechanised model can be reused by various tools and in other projects, benefiting from its proved properties. Furthermore, a critical analysis of the model needs to be applied to identify more serious semantic errors and mistaken design decisions. Such effort improves the model and enables further reasoning about it.

Finally, the project is not standalone but is a part of a larger initiative. The achieved results need to contribute to the pilot project and Grand Challenge in general. The model and the whole research must serve as a case study for future work and must identify results and resources that can be beneficial for related research.

1.3 Outline of the thesis

The thesis consists of 10 chapters of project report, starting from the project context and literature review, identifying the problem and finally solving it.

Chapter 1 (Introduction) places the project within the wider context of formal methods and operating system kernels verification. Furthermore, the initial objectives are discussed.

Chapter 2 (Literature review) highlights the key concepts that will guide project’s development. The full formal methods context is established and related work on operating system kernels is analysed.

Chapter 3 (Problem analysis) builds on the reviewed literature and identifies requirements for both the separation kernel and its formal model as well as the available resources to facilitate project development. With this information, concrete project goals are stated and project scope is defined.

Chapter 4 (Design) selects and adapts a development process, which will guide the creation of formal specification. The approach is examined in detail, investigating structure and verification requirements for the formal model. Furthermore, original research contribution in terms of Transaction design pattern and the full design of a separation kernel are presented. The kernel model is evaluated against the separation kernel requirements.

Chapters 5-8 (Formal specification) describe the formal model of separation kernel components: the basic types, process table, process queue and the scheduler. The most important parts of the formal specification, all justification and design decisions as well as evaluation in the form of theorems and proofs are presented.

Chapter 9 (Evaluation) summarises the project achievements and evaluates them against the project goals, established in problem analysis, to show that all aims have been successfully reached. Furthermore, a deeper analysis is performed with proof examination and project statistics comparison to related projects.

Chapter 10 (Conclusion) discusses the project achievements and provides guidelines for further research. Moreover, the impact of project contributions is examined.

1.4 Statement of ethics

No human participants and no personal or confidential data were required for this project. The outcome of this project - the formal specification - can to some extent be applied to a variety of kernels or other software products. While separation kernels are used within military applications, this is not restricted as the kernel is general-purpose. There are no restrictions against creating harmful or unethical systems making use of the kernel components, however that is out of my control and must be accepted as a consequence of any such general tool. Apart from this, the thesis and formal specification in their direct applications do not have any other identifiable ethical implications.
Chapter 2

Literature Review

Hoare and Misra [30] argue that the long-term ideal of correct software can be reachable with a global effort in the research and industry. Formal methods form one of the cornerstones of software correctness, allowing mistakes to be found even before a program is tested.

This chapter intends to overview the current state of the formal methods application and examine the Grand Challenge in Verified Software, a project to achieve automatic software verification. The current achievements within the project are presented and discussed. Then, the security of operating system kernels is investigated. First, a secure kernel architecture - a separation kernel is presented. Then, different approaches to kernel verification using formal methods are examined. Finally, a discussion about Z notation presents this formal specification language and approaches to its application.

2.1 Grand Challenge in Verified Software

A recent survey on formal methods [78] reports the increase of formal methods usage in many areas of the industry. The examined projects report (in average) reduced development time and cost as well as significant improvement in quality. The practitioners agree that formal methods usage has been both appropriate and successful. Furthermore, the cost effectiveness of using formal methods is more evident in repeated application within a domain.

As an example, Microsoft employs static analysis and invariant detection tools for device drivers verification. The Static Driver Verifier is used to eliminate specific classes of errors in device drivers - a prominent cause for the famous blue screen crashes in Windows operating system [2].

Reliability is an assumed fact in classic engineering. For instance, a constructed bridge is expected to stand properly and perform its function. In software engineering, however, faults and errors are forgivable things, as users associate unreliability as one of the properties of computer software. Unreliable software, malicious attacks and viruses cost enormous amounts of money to the industry and public [54].

To address the fundamental problems of unreliable software, Tony Hoare proposed and initiated the Grand Challenge in Verified Software [28, 30, 62]. The challenge envisions creation of tools and techniques - a verifying compiler - to produce reliable, verified and correct software. The UK effort within the project is coordinated as the Grand Challenge in Dependable Systems Evolution (GC6) [29].

Hoare presents the Grand Challenge as a long-term goal, involving a global research action. He argues that with the recent improvement in experience, technology, tools and techniques within areas of formal methods and software verification, such initiative is feasible [28].

Proposed as one of the first steps in the Grand Challenge, the Verified Software Repository [6] serves as a scientific repository in the area of software engineering and the mechanical certification of computer programs. The repository is intended to curate the various products of experimentation within the Grand Challenge. These products are mainly verified components, specification for key parts, libraries of lemmas, and the experiments themselves.

Hoare and Misra [30] envisaged the initial years of the Grand Challenge as a series of pilot projects. A pilot project is a smaller scale research exercise with intent to gather evidence and experience for intermediate and long-term goals within the challenge. Pilot projects should attempt to explore development methods and establish practices and approaches. Furthermore, they should produce foundations for the long-term goals and may contribute with reusable artefacts. The results of pilot projects together with the produced theory, formal models and reusable artefacts must be curated within Verified Software Repository [30].

2.1.1 Mechanising Mondex

Mondex smart card development in the 1990s used Z to construct formal abstract security policy model and lower-level architectural design. The security of money transactions within the card had to be ensured. Mondex smart card was...
the first product ever to achieve high-assurance ITSEC Level E6 [34] certificate [66, 78].

The formal models of Mondex smart card were verified and refined. The involved mathematical proofs were carried out manually, under the impression that due to inadequate tool support, mechanisation and automated proof would be more expensive [78].

The Mondex smart card project was revisited as the initial pilot project for the Grand Challenge [35]. The project challenged researchers in formal methods to mechanise Mondex specification and proofs. The original work has been done manually, with over 200 pages of mathematics completed [77]. Since Z notation was used originally, application of different tools require to translate the model into appropriate languages and formats. Eight groups took up the challenge and employed different tools and methods (Alloy, ASM, Event-B, OCL, PerfectDeveloper, π-calculus, Raise, Z). Some of their results have been collectively published in [35].

The Mondex pilot project investigated the level of automation achievable using verification tools. The automation revealed several unknown bugs in the original work, showing the benefits of such exercise. Freitas and Woodcock [23] observed that the cost of Z proofs automation was only 10% of the original effort. Almost all attempted techniques achieved similar results in both automation and effort [78].

Research on Mondex contributed to the Verified Software Repository by evaluating existing tools and techniques. The experiences have been recorded and new ways of applying correctness tools to industrial-scale projects have been found [35, 74]. Finally, the Mondex project proved that a collaborative worldwide effort can be accomplished in the area of software verification. The project can serve as a benchmark for evaluation of tools, techniques and effort in the future.

2.1.2 POSIX file store

Next pilot project mechanised a POSIX-compliant [36] file store [24]. The project was initiated by NASA Jet Propulsion Laboratory to create a verified file-store interface for flash memory hardware, used by forthcoming NASA missions [37]. A subset of POSIX standard has been successfully mechanised using a theorem prover [56], conforming to the initial proposal. Furthermore, the abstract model was refined to a Java `HashMap` implementation [24].

In addition to the successful execution of the pilot project, it is interesting how the project benefited from and contributed to the Verified Software Repository and in general to the Grand Challenge in Verified Software. Freitas et al. [24] report successful reuse of general findings and experience with proving theorems from Mondex pilot project. Furthermore, the reusable verified Java `HashMap` formal model was constructed and general properties for injective entities were proved, supplementing the extended toolkit of reusable lemmas and theorems for Z [20].

2.1.3 Verified OS kernels pilot projects

Operating system (and its kernel) is a central part of most computer systems. Microkernels in particular are a suitable target for formal verification due to their small size [69]. The Grand Challenge hosts several pilot projects attempting to verify OS kernels.

FreeRTOS [17] and Microsoft Hyper-V Hypervisor [9] are existing microkernel implementations, written in pointer-rich C (FreeRTOS) and C/assembly (Hypervisor). Verification of each kernel is a challenge to detect its specification from the implementation code and guarantee it is being enforced. FreeRTOS may benefit from rational reconstruction of the kernel, starting from abstract top-level specification and refining it down to code level. The top-level verification and all refinement proofs need to be discharged to produce a verified real-time OS kernel [78]. The Hypervisor is being verified using C code annotations and a verifying compiler [9]. The verification is intended to guarantee security properties of the kernel, which forms a part of upcoming Windows Server releases.

A different approach is taken in Verified OS Kernels pilot project. Inspired by abstract top-level models and refinement using Z notation by Craig [12], the project’s intent is to create abstract verified formal models of microkernels. From these models, refinement can be applied to down to implementation code, in what is known as correctness-by-construction development [26].

The discussion on verified OS kernels is continued by stepping back from the context of Grand Challenge to examine security and verification of operating system kernels in general.

2.2 Separation kernel

A separation kernel is an architecture for secure applications introduced in [55]. The separation kernel applies distributed system security approach to process interaction within an operating system kernel. The concept is utilised by a number of systems and projects, including military and virtual machine applications [42].
Rushby [55] argues that separation kernel cannot be a general-purpose multilevel secure system. Environments that require high security usually are special-purpose, single function systems, with specific security requirements. Therefore, a separation kernel can be minimally small and simple, providing just the necessary functions for the system, thus making its verification easier.

The main concept within separation kernel is the partitioning of processes. Each process in the kernel is given a complete execution environment that is isolated from other partitions. The information flow between partitions is available using established communications channels only. The verification of such partitioning within a separation kernel guarantees that the system benefits from the inherent security properties of distributed systems [55].

To address increasing separation kernel usage for military applications, US National Security Agency has produced a Separation Kernel Protection Profile (SKPP) under Common Criteria framework [32] to define requirements for separation kernels used in environments that require high robustness [63]. SKPP introduces more specifics and requirements for separation kernels and their interaction with the platform and active processes, as well as additional concepts to achieve high robustness.

2.3 Verification of OS kernels

Verification of abstract (top-level) and medium-level models of computer systems has been successfully attempted using formal methods. In such models, the higher-level properties of the system are examined and verified, however the proven mathematical relationship between the abstract model and implementation code are scarce.

The improvement of techniques and tools to verify low-level implementation code spurred the attempts to formally verify operating system (OS) kernels. OS is at the core of most computer systems and its verification is a natural choice towards secure and reliable systems. While the size of monolithic general-purpose kernels (Linux, Windows) is discouraging to attempt their verification, microkernels [69] are small programs (around 10 kLoC) that can be verified. The verified properties range from basic, for instance memory safety, to full formal specification of kernel behaviour [38].

This section presents significant attempts to verify high-level functional correctness or security properties of OS kernels.

2.3.1 Monolithic kernels & early attempts

Verification of monolithic general-purpose kernels is limited by the size of such kernels (e.g. over 11 MLoC of Linux kernel 2.6.30 [41]). Nevertheless, some of the large kernels have undergone verification process: Trusted Solaris, Windows NT, and SELinux (Red Hat Enterprise Linux 4.1) have been certified to Common Criteria EAL 4 [32]. However, this level does not require any formal modelling [38]. Klein [38] notes that high-level formal analysis has been applied to the security policies of SELinux, however there are no proofs showing that SELinux kernel implements the analysed policies correctly.

First (to some degree) successful OS kernel verification projects have been undertaken in the 1970s and 1980s. They focused of verifying small operating system kernels (size and functionality similar to that of a microkernel) [38].

UCLA Secure Unix [75] attempted formal modelling and verification of a Unix kernel written in simplified Pascal. The formal refinement process was undergone from top-level abstraction down to Pascal code. The project completed 90% of specification and about 20% of proofs in a 5 man-year effort.

Walker et al. [75] report that specification alone was enough to uncover significant security error, however the effort required and difficulty of proofs due to the lack of automation was overwhelming. Finally, Walker et al. note the poor performance of secure kernel, a magnitude slower than standard Unix kernel in some cases.

UCLA Secure Unix formal model assumed that the underlying hardware is working correctly and abstracted hardware using axiomatic pre/post conditions [75]. The assumption of compiler, hardware and verification/proof tool correctness is common in other projects as well [38].

The provably secure operating system (PSOS) verification [47] in late 1970s included both the OS kernel and the underlying hardware into the design. The design decisions contributed substantially to computer science research, however the actual implementation is unclear [38].

Finally, KIT (Kernel for Isolated Tasks) [5] is a fully formally verified kernel, albeit with minimal functionality. The kernel has undergone full formal refinement down to the implementation in assembler instructions. Bevier [5] proves the security properties for the abstract level and show that they hold because of proved refinement relations.
The early attempts at OS kernel verification have shown that adequate automation and tool support is necessary for successful completion. Furthermore, operating system design with verification in mind can substantially facilitate its verification [38].

With the emergence of microkernels, subsequent attempts at OS kernel verification (e.g. VFiasco) focused on them [38]. The size of microkernels allows to verify them, while the increasing performance (in particular L4 [43]) allows them to be used for general-purpose applications [69].

The most promising recent verification attempts, namely Verisoft and L4.verified, are presented next.

### 2.3.2 Verisoft/Verisoft XT

The Verisoft and its successor Verisoft XT projects are large-scale efforts at formal verification of integrated computer system design. The projects are funded by German government and feature large collaboration between academic and industry organisations [73].

The projects dismiss the assumptions of compiler or instruction set model correctness and strive to verify full formal chain from hardware to applications [38].

The project verified and published a number of artefacts, such as string and big-integer libraries, e-mail client, C0 (a subset of C) compiler. The results are publicly available at [72].

The recent work on Verisoft XT has shown progress in the verification of commercial OS kernels, Microsoft Hyper-V Hypervisor and PikeOS.

*Microsoft Hyper-V Hypervisor* is a micro-kernel that runs directly on a multi-processor x64 hardware, turning it into a number of virtual multiprocessor x64 machines. The hypervisor consists of a 105 kLoC C/Assembler code, written without verification intent [9].

Verification of the hypervisor drives the development of the associated verification tool, the Verifying C Compiler (VCC). Verification is performed by annotating C code according to interface specification and then using VCC to verify its correctness. Also, invariants about data types have been specified and proved amissable. Microsoft currently has completed verification of about 20% of hypervisor codebase [9].

*PikeOS* system consists of a microkernel with partitioning features, which allows virtualisation of a number of applications or operating systems on a single kernel [4, 52].

PikeOS uses Microsoft VCC to verify correctness of the microkernel and have succeeded to verify low-level functions and initial system calls [4].

### 2.3.3 L4.verified/seL4

seL4 is a high-performance L4 [43] microkernel which correctness has been verified according to its abstract specification, L4.verified. The full refinement from the abstract specification to C/assembler implementation code has been mechanised and proved. The proofs have just been completed and the results are being published [39].

seL4 microkernel is claimed to have the first formal proof of functional correctness of a complete, general-purpose OS kernel. The functional correctness is ensured by complete refinement - all properties that hold for the abstract specification, are true for the implementation as well. The abstract specification has been verified for security and crash prevention and allows to predict the behaviour of the kernel in all situations by reasoning at the top abstraction level [39].

The kernel was developed as a prototype in Haskell programming language. This allowed OS programmers to create the kernel implementation. Furthermore, Haskell was translated into theorem prover Isabelle/HOL [49] as the executable specification and then abstracted into the abstract specification.

Finally, to achieve high-performance of L4 kernels, the Haskell prototype was manually reimplemented as a C application. The C implementation was translated into theorem prover to prove refinement between it and the executable specification [39]. Note that the actual implementation code is not being proved (e.g. by using C annotations as in Verisoft project), but a translated copy of the implementation in the theorem prover.

Klein et al. [39] report useful statistics about the verification process. Their C implementation including Haskell prototype is a cost-saving development process in comparison with other kernel implementations “from scratch”. The cost of full proof is split into 9 person-years (py) invested into proof tools, automation and frameworks and 11 py of the seL4-specific proof.
Klein et al. [39] predict that with existing experience and tools, re-doing similar verification would result in 8 py total effort (kernel + proof). The authors illustrate the cost of security by noting that a certification of a seL4 size kernel to Common Criteria EL6 specification would cost $87M but would not guarantee correctness as the formal verification.

The recent verification of OS kernels reports success at proving correctness of a crucial component of computer systems. The presented recent initiatives are proprietary and show that industry involvement is active in the formal methods area.

The following section examines a different approach to OS kernels verification - formal modelling.

2.4 Formal modelling of OS kernels

All discussed OS kernel verification attempts are using the bottom-up approach: the project starts with a working kernel implementation\(^1\), then its abstract model is specified and verified. Research has shown that verification would be much easier if the OS was designed with verification in mind [38]. Such findings mandate investigation of a top-down approach, of which correctness-by-construction using refinement calculus has been chosen for the pilot project associated with this thesis.

Craig [11] argues that the formal methods approach to OS design allows to explore kernel functionality without the burden of an implementation code. An abstract OS kernel model can be verified for necessary properties and adjusted if mistakes emerge. Furthermore, formal reasoning can help define new models of operating systems, as well as prove important security and assurance properties.

When the model is defined and verified, it can be refined down to implementation level. Again, the choice of actual implementation language, platform or algorithms can be chosen by the implementor, as long as the abstract model invariants are satisfied. Such approach guarantees a correct implementation of the abstract model.

OS kernels is an interesting area for formal methods practitioners. A number of attempts exist to specify a formal model for parts of OS kernels, e.g. interrupt handling [64] or data separation in a separation kernel [27]. The proprietary Mechanically Analyzed Separation Kernel (MASK) has a verified formal specification and an implemented kernel [45]. However, the proofs of implementation and abstract model conformity have not been performed [38].

2.4.1 Full formal models of OS kernels

The formal modelling of operating system kernels has been explored by Craig [11, 12], where he attempts to provide full specifications of a broad range of kernels, including a simple classic, swapping and separation kernels. In the initial work Craig formalises standard components, present in operating system kernels, such as process queues, process table and hardware abstraction. Then he continues on producing top-level models of several kernels using Z and Object-Z notation.

The research is continued and elaborated in the second book. Craig attempts to develop a detailed formal top-level model for a simple kernel, which is then refined to a near-code level. The standard components from previous work receive more polish and detail modelling, as well as integration into the larger model of the kernel.

Finally, building upon the model of a simple kernel, the formal model and refinement of a separation kernel is developed. The model employs and adapts some of the standard components already modelled for the simple kernel, while exploring the required process separation, special treatment for device processes and a different scheduler model.

2.4.2 Verified OS kernels pilot project

Here we revisit the earlier mentioned Verified OS Kernels pilot project. A substantial amount of research has been performed as part of this project and the results are available within the Verified Software Repository [6, 74]. Formal specifications have been developed by taking the simple kernel in [12] as a base formal model and mechanising it. Furthermore, complete and verified formal models of individual reusable kernel components and data types have been produced [19, 21, 44].

Chain datatype

The process tables in both simple and separation kernels [12] are modelled as data structures for process information. The top-level process table definition using abstract sets is then refined into a chain-like data structure suitable

---

\(^1\)L4.verified started with a prototype implementation in Haskell.
for code-level implementation. The initial analysis, mechanisation and verification of the process table by Freitas [19] produced a pure formal model of a reusable Chain datatype. The abstract mathematical model is suitable for application in various structures within both simple and separation kernels.

**Mechanisation of simple kernel components**

A broader mechanisation and verification of the main simple kernel components (process table, process and priority queues, and the priority scheduler) has been achieved by Liu [44]. Using Z/Eves theorem prover to prepare the models allowed the author to address and fix syntax, type and domain application errors as well as semantic inconsistencies in [12] model. Further analysis involved precondition calculation and proof, resulting in corrections within totalled (interface) operations.

In addition to verified and mechanised formal specification of main simple kernel components, Liu [44] produced a number of general reusable lemmas, which can be employed in further modelling of OS kernels.

**Formal models of simple kernel components**

The full formal models of main simple kernel components are presented in [21]. The formal specifications of process table, process queue and priority queue are developed as general purpose data types, based on the simple kernel specification in [12].

Freitas [21] goes further than just the mechanisation of original specification and analyses the design decisions made throughout the model. This allows him to address problems in the original model: add missing error types and complete partial interface operations, adjust erroneous under- and over-specifications throughout all components as well as locate serious mid-point insertion problems in the priority queue.

The completed and verified formal models of simple kernel components contribute to the available kernel design resources in Verified Software repository [74]. The design decisions of these formal models can be reused in similar applications within other OS kernels.

Finally, the overall presentation and structure of the formal specification in [21] is enhanced to comply with the best practices within the industry and academia [3, 67].

The work in Verified OS Kernels pilot project shows good progress towards implementation of formal abstract specification of a simple kernel. The project further aims to explore refinement tools and techniques by refining the model down to implementation code. Also, a similar process of abstract specification and refinement is necessary for the separation kernel.

In the final section of this chapter, a standard in the formal methods area - Z notation - is presented.

### 2.5 Z formal specifications

The Verified OS Kernels pilot project employs Z notation and associated tools to formally specify and verify models of operating system kernels. As the thesis is associated with this pilot project, Z notation should be used to benefit from existing resources and previous work.

The Z notation is a widely used formal specification language, based on set theory and mathematical logic [65, 76]. It is supplemented with a notion of schema and a number of extended operators. Furthermore, proof techniques of mathematical logic can be used to reason about these specifications and the models they describe. Also, refinement techniques can be used to derive more concrete specifications, which are closer to executable code.

The notation was developed by the Programming Research Group at University of Oxford in late 1970s. In 2002, it became an international ISO standard [33].

Z application in industrial projects spurred emergence of literature on how to use Z in real projects. Z design patterns [67] and best practices [3] guide the user to make the most of the notation. Z does not define a method (procedure) of software development using formal methods. Barden et al. [3] suggest that a specific development process should be selected and evaluated for each project involving Z.

The Established Strategy (also known as “State and Operations” style or Delta/Xi pattern) is a widely used approach, which covers development from Z specification to code [3, 67]. The development is based on defining user’s functional requirements in the formal specification of the system. The developed system is modelled with enough detail at an abstract level. A Z specification contains definition of given sets and global constants, abstract system state and its operations, together with associated comments and calculations of preconditions and properties.
The strategy defines specification structure and approach to the abstract model. Furthermore, the specification is reviewed and refined to the specific level of abstraction or code. The development process can be iterative to incorporate changes or adjustments in the requirements, design or actual model [3].

Structured methods cover the analysis and design of information systems. Various parts of the process can benefit from using Z specifications, such as analysis, design, validation or quality assurance. Combination of Z and SSADM version 4 [16] structured method has been investigated in [53] and defined how to incorporate Z specifications into the process.

Object-Z introduces the object-oriented concepts to Z specifications [7]. It is an extension to Z that allows composition of Z variables, state and operations into reusable classes. Moreover, class inheritance and other object-oriented principles are introduced.

Other ways of creating Z specification, such as Morph or Event traces [67], represent specific approaches for systems that do not follow the common “state and operations” style.

2.6 Summary

This chapter established formal methods as a successful way of achieving software assurance. The Grand Challenge in Verified Software attempts to unite researchers with the goal of correct software. Initial work on the pilot projects are showing promise in both scientific collaboration and the accumulation of experience, tools and reusable artefacts. Each pilot project improves the state of verified software and facilitates following research.

Furthermore, the different approaches to kernel verification have been explained. The common solution is to attempt verification of an existing kernel. An abstract specification of the existing kernel is created and verified, however proving conformance between the abstract specification and the implementation code proven difficult for the majority of attempts.

With the increase of computing power and maturity of formal methods and tools, proprietary OS microkernel verification projects have achieved breakthroughs and the first formally verified general-purpose kernel has been announced [39]. The verification of some kernels is simply a proof of correctness. In other cases, abstract models allow reasoning about kernel security properties.

A different approach is to start with a formal model of an OS kernel, verify it and then refine down to implementation code. Such approach may benefit research in OS design as well as allow different implementations of the kernel. This chapter presented current achievements to create formal kernel models, including the Verified OS Kernels pilot project.

In this chapter, a need for abstract formal OS kernel models, which are not limited by underlying implementation, has been identified. The existing separation kernel specification by Craig [12] forms a basis for a full abstract model of a separation kernel. Such kernel is used within military and other high-robustness environments, where formal verification can prove beneficial.

This concluded the review of the research areas fundamental to this project. With the gained knowledge and identified general need for a formal model of a separation kernel, a deeper analysis can be undertaken to formulate the problem and investigate the requirements.
Chapter 3

Project Aims & Problem Analysis

The investigation of the subject areas surrounding the project (Chapter 2) has revealed the huge interest and effort put into the application of formal methods to verify operating system (OS) kernels.

The difficulties in kernel verification suggest that a different (top-down) approach of kernel modelling needs to be explored. Results and reusable artefacts emerged from such approach can benefit further research and be used to explore new OS design or implementation models.

The successful pilot projects within the Grand Challenge in Verified Software have accumulated the resources, knowledge and practices that will facilitate this project. Furthermore, the reported scale of kernel verification projects suggest that the project needs to be limited to a certain scope to enable successful completion.

The project aims to develop a partial formal model of the separation kernel as part of the Verified OS Kernels pilot project. This chapter examines the requirements for a separation kernel and its formal model. Then, existing research and resources that can facilitate the project is examined in detail. Finally, the specific project goals and scope are identified.

3.1 Separation kernel requirements

The separation kernel is an architecture for secure application which partitions processes into isolated environments with controlled information channels. For the development of separation kernel formal model, both the original definition [55] and Separation Kernel Protection Profile (SKPP) extensions [63] are used as sources for requirements. Furthermore, it is assumed that the requirements for operating system kernels in general are well understood and established in the literature, e.g. [68], therefore the following analysis focuses on separation kernel requirements only.

In this section, the main requirements for a separation kernel are given, loosely grouped into general categories. The particular requirements are identified using [Req-*] tag to reference them directly. Note that these requirement tags represent high-level requirements and are used within this thesis. The detailed separation kernel requirements can be found in SKPP [63].

3.1.1 Functionality and security

The original arguments by Rushby [55] that the separation kernel should be small and simple, corresponding to the security application, are enforced in [63]. SKPP requires kernel’s functionality, architecture and design to be of minimal size and complexity to support high robustness [Req-F1]. Furthermore, the separation kernel must protect all of its resources from unauthorised access [Req-F2].

The separation kernel is only concerned with the security of its own functionality and adherence to its architectural principles. This means that an implementation of a verified separation kernel can be reused in various systems, because enforcement of system security policy is not the concern of the kernel [55].

SKPP suggests that a separation kernel does not need to provide a multi-user environment together with the identification, authorisation and access control functionality. Furthermore, the kernel is not concerned about data encryption (both for storage and transfer) and provides no user interface during initialisation and execution, as it is configured before launch [63]. However, if any of this functionality is needed by the system using the separation kernel, it must be provided by either separate components or by extending requirements for the kernel [63].

3.1.2 Separation and information flow

The main requirement for a separation kernel is the separation of components (processes). Each component must be provided an individual execution environment with communication channels to separate the component from others.
in a similar way as by physical isolation in distributed systems [Req-S1] [55].

SKPP elaborates on the separation by introducing more general concepts of isolated partitions (sets of kernel exported resources) and subjects (actual components/processes), executing within the certain partitions [63].

The partition isolation requires that actions of subjects within the partition cannot be detected or communicated to by subjects in other partitions, unless the information flow between the partitions has been explicitly allowed by an established communication channel [Req-S2] [63].

The separation also applies to other kernel components, as SKPP requires that the desired isolation properties are preserved by the internal resources within the separation kernel as well as the exported resources [Req-S3] [63].

3.1.3 Configuration

The definition of partitions (available resources, memory size, quotas, etc.) and established communication flows are explicitly controlled by kernel configuration data [Req-C1] [63]. Furthermore, the configuration covers separation kernel self-test parameters and audit function behaviour [Req-C2] [63].

The separation kernel can support either static or dynamic configuration [63]. The static configuration results in a simpler kernel, when the configuration data can only be changed offline [Req-C3]. The separation kernel is initialised with a certain configuration vector, which cannot be changed during execution [63].

Nguyen et al. [48] notice that a static configuration separation kernel is simple and small, therefore can be evaluated for high robustness. With the verification of kernel’s fundamental security services, it can then be reused as a trusted building block for more complex systems.

Dynamic configuration separation kernel must support configuration change either during kernel restart or an online reconfiguration operation [63]. SKPP does not define the actual configuration change as long as the secure state of the kernel is maintained all the time [Req-C4].

3.1.4 Principle of least privilege

The principle of least privilege [14, 59] requires that each user of the system operates under the minimum privileges that are necessary to complete its job. SKPP requires this principle to be applied both for separation kernel subjects and its resources [Req-L1] [63]. The kernel must only provide functionality and information flows that are needed by the component to complete its job. These restrictions are established using kernel configuration data [Req-L2] [63].

3.1.5 Memory management

In general, because of the single function nature of the secure systems, separation kernel can permanently allocate fixed memory space and resources for each partition, in contrast to supporting paging and virtual memory management in general-purpose kernels [55].

SKPP relaxes the original definition and allows memory resources to be accessible to subjects either by direct (but controlled) access or using virtual memory techniques. However, it requires that each partition is allocated a distinct, non-overlapping portion of memory [Req-M1] [63]. Certain address spaces, necessary to establish information flows between partitions, can be shared between these partitions. This is controlled using separation kernel configuration data [Req-M2] [63].

3.1.6 Execution and scheduling

There are no scheduling requirements as long as it supports the separation of each component. Rushby [55] suggests executing components on a round-robin basis without preemption, allowing each component to either to suspend voluntarily or terminate to give up control.

Certain kernel resources can only be accessed by a single component at a time. The partitioning requirement means that every resource set must appear isolated. To achieve complete separation for such kernel resources, SKPP requires temporal separation in addition to the spatial separation [63]. This requires that only a single component is active at each given time [Req-E1] - a requirement satisfied by the original proposition of a non-preemptive scheduler [55].

3.1.7 Platform considerations

The separation kernel platform (both hardware and firmware) is considered an integral part of the kernel. For this reason, the hardware and software components comprising the platform are assumed to have been evaluated and
verified for correctness and security under Common Criteria [32] requirements, making them trusted entities within the kernel [Req-H1] [63].

Nguyen et al. [48] argue that in general hardware vendors are reluctant to share design documents for verification and only basic evaluation of kernel dependencies and hardware security mechanisms is performed. The problem of hardware assurance is not resolved in SKPP.

If the underlying platform exports privileged (kernel mode) interfaces, they are internal to the separation kernel but may be exported as the kernel interface, if necessary [Req-H2]. The unprivileged (user mode) interfaces are accessible to both kernel and the components [Req-H3] [63].

Furthermore, a separation kernel can benefit from the memory management and segmentation features of the underlying hardware to achieve physical separation of the partitions [55].

Finally, the underlying hardware is considered to be part of the kernel, therefore a separation kernel has the responsibility to handle interrupts and direct them to appropriate components [Req-H4] [55].

3.1.8 Verification & deployment

A verified separation kernel must show that components can communicate only by explicitly provided channels [Req-V1] [55]. It must be verified that the partitioning and information flow controls cannot be altered or bypassed [Req-V2]. Furthermore, the verification of a separation kernel is performed offline and an executional security management is not required [63].

The separation kernel must also be able to generate audit data about its execution [Req-V3] [63].

Finally, SKPP establishes the separation kernel deployment requirements, such as trusted delivery, configuration, load and initialisation, to guarantee secure kernel deployment, configuration, launch and recovery [Req-D1] [63].

Having defined requirements for separation kernel, the requirements for the formal specification and the project in general are identified.

3.2 Formal model requirements

The main objective of the pilot project is to achieve a correctly constructed separation kernel using top-down development through refinement. For that, we start from Craig [12], since as a domain expert in OS development, he devised a whole Z specification of the abstract separation kernel, as well as the refinement towards code, which is also available (in C).

Mechanisation of the separation kernel requires all hand-written Z models in Craig [12] to be recreated in a Z tool, correct errors [MReq-A1], verify for inconsistencies [MReq-A2], calculate the operation preconditions [MReq-A3] and ensure the totality of interface functions [MReq-A4].

During the formal modelling and refinement, Craig [12] declares a number of proof and discharges them manually. The project requirement is to formalise all proofs [MReq-A5] - declare and prove them using a Z theorem prover.

The refinement of abstract model to implementation level code is done using manual proofs and declarations [12]. The project aims to automate the refinement using available refinement tools. The initial proposal is to investigate feasibility of a tool chain [MReq-A6]:

\[
\text{Z/Eves} \rightarrow \text{ZRC-Refine/Gabriel} \rightarrow \text{Spec#-Boogie/PL}.
\]

Finally, the results must be curated in the Verified Software Repository. This means that the created formal specification must be reusable for future research [MReq-A7]. Appearing artefacts, such as general lemmas, theorems and reusable data types must be prepared for reuse [MReq-A8]. The process, best practices, new patterns must be documented [MReq-A9] to facilitate future research.

Having specified the requirements for separation kernel and its formal model, the available resources to facilitate the project are investigated.

3.3 Existing research

The existing research in the formal modelling of OS kernels area has been reviewed in Section 2.4. Now it is necessary to examine the related research in detail. In particular, the reusability of the results, including formal specifications, component models and designs, formal lemmas and proofs, mechanisation and modelling experiences, are of significant
interest. As the research in this dissertation will closely relate to existing works, it is important to investigate what is available and what contribution is still needed.

Craig [12] provides a substantial work on a formal separation kernel model based on requirements defined in the original architecture by Rushby [55] and the US National Security Agency (NSA) guidance for separation kernels [50]. The model is not of a full security kernel but delivers the majority of separation kernel requirements and functionality:

- Process table for basic process management;
- Process spatial separation in terms of non-overlapping address space allocation;
- Communication channels by the means of an asynchronous kernel-based messaging system;
- Process temporal separation using a non-preemptive scheduler and the messaging system;
- Kernel protection by employing external process identifiers and clear interfaces for processes. Also, a secure kernel exit is defined in case of invalid interface access;
- Exported kernel resources as device processes;
- Separation and principle of least privilege for device processes by providing special interfaces within the kernel and otherwise treating them as other processes;
- Synchronous user process - device interaction by supporting priority execution for devices;
- Basic verification of several kernel properties using mathematical proofs.

The model is based on Intel IA32 [31] architecture, as it relies on the underlying hardware to support segmentation and cross-boundary access restriction, as well as the context switching when next process is executed.

The model of a separation kernel in [12] is an excellent attempt at developing a formal specification of separation kernel in Z notation, benefiting from author’s expertise in operating system kernels.

Craig’s separation kernel model can be further supplemented using resources produced as part of the Verified OS Kernels pilot project.

The Chain datatype [19] is an abstract mathematical model suitable for application in various structures within separation kernel refinement.

Liu [44] and Freitas [21] have defined a number of general reusable lemmas, which can be employed in analogous applications (as in simple kernel) within a separation kernel. Moreover, the completed and verified formal models of simple kernel components [21] contribute to the available kernel design resources in Verified Software repository [74]. The design decisions of these formal models can be reused in similar applications within separation kernel.

The rigorous process queue model [21] can be directly adapted for the use within separation kernel scheduler, as [12] uses the same queue model for both kernels. The adaptation is simplified even further, as the Z/Eves proofs for the model are available from the associated technical report [22].

The existing work on formal specification of simple and separation kernels forms a strong base for the development of a verified formal model for separation kernel. In the next section the actual problems that are the goals of this dissertation are identified and defined.

3.4 Research goals and scope

In this section, the necessary work towards the formal specification of a separation kernel is identified. The existing problems are examined and possible research goals are set by analysing the requirements, existing research and previous experiences in similar exercises.

Then, a certain subset of the goals is selected to limit the scope for the thesis, as the full solution is impossible due to the large size of a separation kernel and limited time available for writing the dissertation.

3.4.1 Problems

The formal specification provided in [12] covers the majority of main requirements for separation kernel (Section 3.3). Furthermore, its use of Z notation and similarity to simple kernel specification makes it a perfect candidate as the base of verified formal model for separation kernel.
The work in [12] is appreciated as an important attempt to model a separation kernel in Z, nevertheless it is hindered by lack of mechanisation, proof and poor Z application. The results of simple kernel mechanisation in [44] prove that tool usage and rigorous approach to formal model verification is mandatory to make the formal model useful for future applications. Without that, the model cannot be used as a trusted component within larger systems or further research.

The initial analysis of the original specification in [12] has shown that it suffers from similar problems as found in the mechanisation of a simple kernel [44]. It can be expected that the mechanisation and verification of the separation kernel yields similar results.

Furthermore, the analysis of design decisions in separation kernel by [21] has shown that mechanisation is not enough to produce a verified formal model. The separation kernel needs to be examined with scrutiny to verify that the design decisions in [12] satisfy the requirements and avoid mistakes as well as under- or over-specification.

The problems found in Craig’s work are offset by the reuse of invariants and design decisions in the original specification. This saves a lot of time and effort and gives a full model - a candidate for refinement to code.

Craig [12] provides the refinement of the abstract separation kernel model, however possible mistakes in the top-level render the refinement incorrect, because it is based on false pretence of correct abstract model. With the successful development of the top-level formal model, the refinement calculus can be applied to refine the model, either following Craig’s guidance or in a different way.

Finally, the original specification can benefit from layout, naming and consistency updates and corrections to bring it up to the standard for curation within Verified Software Repository [74].

During the development of formal model, various reusable artefacts such as general lemmas, formalised kernel components, implemented design decisions or created best practices and new design patterns may appear. They highly contribute to the overall knowledge of modelling OS kernels or formal modelling with Z in general. Such artefacts are valuable and effort must be put to make them reusable for other applications.

3.4.2 Goals

Surfacing from the reasoning above and the formal model requirements (Section 3.2), a number of goals to develop a formal separation kernel model have been identified. Where the goal addresses a model requirement, it is indicated using \[\text{MReq-*}\] tag:

\[\text{G-1} \] Mechanise top-level separation kernel specification in [12, Ch. 5] using Z/Eves:

\[\text{G-1a} \] Correct the syntax and type errors. Prove domain checks \[\text{MReq-A1}\];

\[\text{G-1b} \] Verify the formal specification for inconsistencies \[\text{MReq-A2}\];

\[\text{G-1c} \] Calculate and verify the preconditions of operations \[\text{MReq-A3}\];

\[\text{G-1d} \] Create total operations for interface functions \[\text{MReq-A4}\];

\[\text{G-2} \] Analyse and update specification semantics and design decisions:

\[\text{G-2a} \] Examine the established predicates and justify their appropriateness for corresponding requirements;

\[\text{G-2b} \] Check for mistakes due to under- or over-specification, resulting in erroneous non-determinism, missing constraints or, conversely, too restrictive invariants. Correct the emerged problems;

\[\text{G-3} \] Establish equivalence or refinement relationships between the result and original specifications. If not possible, provide full justification of changes;

\[\text{G-4} \] Investigate the interesting and useful properties of the model. Formulate and prove theorems about the findings \[\text{MReq-A5}\];

\[\text{G-5} \] Identify, formulate and prepare for reuse emerging artefacts such as general lemmas, formalised kernel components, implemented design decisions or created best practices and new design patterns \[\text{MReq-A8, A9}\];

\[\text{G-6} \] Verify the created formal specification. Prove all established theorems and domain checks \[\text{MReq-A9}\];

\[\text{G-7} \] Upgrade the presentation of Z specification to conform to best practices (e.g. [3]). Improve overall layout, naming and specification structure \[\text{MReq-A7}\];

\[\text{G-8} \] Update the model to fully satisfy requirements for separation kernel (Section 3.1);

\[\text{G-9} \] Perform refinement for the top-level specification down to code implementation \[\text{MReq-A6}\];
[G-10] Produce the mechanised formal specification for curation within Verified Software Repository [MReq-A7].

The goals defined here are a superset of model requirements [MReq-*]. They expand on the original requirements with the need to evaluate the model and design decisions critically, find higher level problems, provide corrections and/or alternatives. Furthermore, goal [G-8] defines the activity to ensure that the model satisfies all separation kernel requirements (3.1).

Note, that in this dissertation, the phrase “original specification” in most cases refers to formal specification and refinement of a separation kernel in [12, Ch. 5].

3.4.3 Scope

The identified goals comprise material for a project of a very large size. The original specifications of simple and separation kernels took the author about three months’ working time to produce [12]. Furthermore, Craig benefited from his expertise in operating system kernels and experience in formal refinement.

The total of 900 hours (90 credits × 10 hours per credit [71]) allocated for the whole dissertation (May - September 2009, about 9 hours per work day on average) is nowhere near enough to complete all goals, therefore it is necessary to limit the scope to a subset of goals. Furthermore, the lack of experience in large formal Z modelling and no prior knowledge of formal proofs needs to be taken into account, as proofs are expected to make up a large part of the work and they are hard to perform [18].

To evaluate the scope of this dissertation, comparable thesis by Liu [44] has been analysed. During the same available time, the mechanisation of simple kernel basic types, process table, process and priority queues, and the scheduler have been delivered. It is necessary to note that the work was based on existing process table exploration in [19] and several proofs were not completed.

The initial scope for this dissertation therefore has been defined as mechanisation of basic types, process table, process queue and the scheduler of a separation kernel, which is comparable to the work accomplished in [44]. Furthermore, in case of success, the challenge is extended to an attempt for a verified formal model of the indicated components.

A very successful development would result in the mechanisation of the selected components (goals [G-1] and [G-6]) as well as production of a full verified top-level formal specification of the components, alongwith related reusable artefacts and model improvements (goals [G-2]-[G-7] and [G-10]).

The development of formal specification for the remaining components is left for future work. Furthermore, after the completion of a formal model based on [12], it can be extended to satisfy goal [G-8]: and/or refined to code as indicated in goal [G-9].

This concludes the identification of specific goals expected in this thesis.

3.5 Summary

This chapter explored the requirements for a formal model of separation kernel. The defined separation kernel requirements are necessary to evaluate the design of separation kernel in [12] and identify the compatibility and limitations.

The formal model requirements describe the formal specification that needs to be developed. Based on these requirements and existing reusable resources, a set of specific goals have been stated. The goals define work that is needed to achieve within the pilot project.

Since the effort necessary for full project completion is very large, a subset of the goals has been selected as the scope of this dissertation. Both minimum (a comparable size to previous thesis) and maximum (in case of success, full top-level model of selected component) scope boundaries have been set.

With the goals set, an approach how to achieve them can be defined.
Chapter 4

Design & Methodology

The problem analysis (Chapter 3) has revealed that the project needs a twofold approach: the kernel model must satisfy separation kernel requirements and it must be a proper formal model.

This means that the design stages need to examine how to prepare a correct formal specification that is fully mechanised, defined and verified. Then, an appropriate Z formal model must be defined to satisfy separation kernel requirements, that follows the specification design.

This chapter evaluates and selects a development model, which is then applied to the examined problem. The development model and style governs the formal specification structure, required definition and verification steps. This constitutes the formal model design and methodology. A Z theorem prover is evaluated and selected to support the mechanisation effort.

New findings in Z application, namely the Transaction pattern, are presented to supplement the established Z methodology.

The chapter concludes with the architectural overview of separation kernel design. The main components within the kernel, their interactions and interfaces are presented.

4.1 Intended approach

Formal methods, although named so, cannot be classified as actual development methods [3]. Z, in particular, is a notation used to formally specify system models, define system properties and formulate statements and reasoning about the system [33]. However, it does not define a method (procedure) of software development using formal methods [3].

Software engineers have developed methods to use Z specifications within development process. The most prominent approaches have been presented in Section 2.5.

4.1.1 Development approach evaluation

The given Z development approaches have been evaluated for suitability to the project. The Established Strategy (“State and Operations”) development model is the most suitable Z specification approach for the separation kernel. It governs development and verification of design in the form of Z specification, theorems and proofs [3].

Furthermore, the Established Strategy has a refinement process with associated refinement calculus, which is used to create more definite and complete specifications to the actual implementation code, while preserving the invariants and properties of the abstract specification [76].

Refinement of separation kernel is out of scope of this dissertation, therefore we focus on the development model for the top-level Z specification of separation kernel components. Moreover, the verification and validation part of the Established Strategy is followed to evaluate the model and its properties [3].

Finally, the available work regarding the separation kernel [12] and a related simple kernel [19, 21, 44] is defined following the Delta/Xi pattern. The selection of this process allows the project to benefit from the available research.

The other evaluated approaches, Z with structured methods and Object-Z were rejected. SSADM (structured method) is too big and the size of analysis necessary for it is not applicable to the scope of this dissertation. Object-Z requires learning an extended notation and, in the case of separation kernel, the process table, scheduler and other components are individual objects in the whole system and do not benefit from reusability of corresponding classes.
4.1.2 Established Strategy overview

The application of Established Strategy to develop formal model of separation kernel is examined with focus on the iterations within the scope of this dissertation. An overview of the process is given in Figure 4.1.

The following brief overview discusses the specifics of selected development approach.

Requirements

The requirements for separation kernel and operating system kernels in general are known and established in the literature and industry (Section 3.1). Therefore requirements elicitation consists of examining them in associated research and industry technical manuals.

Analysis, design and specification

The analysis and design involve both the modelling decisions and choosing the right notation to translate requirements into the design of the model. The created Z specification must be specified at an appropriate abstraction level (under- and over-specification should be considered). Also, specification should be readable and avoid clutter or inappropriate usage of Z constructs to describe intent.

The model is specified using Z in the following manner [3]:

1. Basic data types, given sets and constants are identified and specified, describing important parts of the problem;
2. Abstract component state is defined as a Z schema, specifying state variables and invariants;
3. Initial state of the component is established by constraining important variables;
4. Operations define functional properties of the component by relating initial and final states together with input and output variables. These operations are composed to build the complete robust component interface;
5. Specification is reviewed to ensure quality, abstraction and coherence.

Although the analysis and specification are separated as different activities, one involving decisions about the model and another producing the formal specification, they are closely related and analysis/design is applied throughout the construction of specification.

Verification and validation

In the context of formal models and in particular Z specifications, verification is the process of formulating and exercising formally justifiable proofs about the specified constructs [3]. The proofs are constructed to show specification consistency and feasibility, calculate operation preconditions and reason about properties of the model.

Validation involves demonstrating and justifying that the formal specification satisfies requirements and produces a correct model of the system [3].

The verification in particular is an important part of the dissertation, as the mechanisation involves proof about specification correctness and investigation of component properties.
Iterative process

The development process involves iterations, as arising verification and validation problems or later specification issues may require modification of early models. It has been established that requirements for separation kernel do not change, therefore the iterations involve analysis, specification and verification stages only. Iterative nature of the process allows tackling emerging design flaws, specification shortcomings and other problems appearing throughout the design/implementation.

Component dependency

The major components identified in the high-level design of separation kernel (Section 4.7) are modelled one-by-one. Furthermore, the established dependencies between components dictate the order of their formal specification, as Z requires all constructs to be defined before use.

The Established Strategy process is applied to each component to develop its formal specification. This indicates another axis of the development process - formal model construction for the whole system, based on formal component specifications. The abstract data types, constants and basic schemas are defined first, then process table is built. The process queue definitions are reused and adapted from previous work [21]. Finally, the scheduler model is produced.

This part of the development process is also iterative, as definition of subsequent components may require modifications in earlier specified parts of the system. Furthermore, change in early parts of the specification is very likely to ripple through the dependent components, as the schemas, invariants and proofs may need updating to accommodate the change.

The Established Strategy development process has been adapted to suit the project. The following sections provide certain specifics about the process application in regards to specification structure, correctness and verification. The associations between formal specification and certain proofs are established, as well as fine points and pitfalls of certain modelling approaches are discussed.

4.2 Structuring the specification

4.2.1 Delta/Xi pattern

The Delta/Xi pattern, also called the “State and Operations style” is a widely used way of describing a schema using Z specification [67]. The main actors are abstract state elements and operations defined on those states, defined as Z schemas.

Along with the general structure of specification, Delta/Xi defines special semantics to support the state/operations approach: schemas and variables decorated with dash (e.g. State’) represent after-state, ? and ! are used to identify input and output variables, respectively (e.g. in?, out!) [67]. Furthermore, Δ and Ξ schema names are used to specify relationship between before- and after-states and are defined as following [33, 67]:

\[\Delta State \triangleq State \land State’\]

\[\Xi State \triangleq [\Delta State \mid \theta State = \theta State’]\]

These names are central to the style (even lending the name to the pattern) and are used in operations to define relationships between before- and after-states. ΔState puts no constraints on state change, while ΞState requires the entire state to be unchanged [67]. Note that \(\theta\) operator denotes the characteristic bindings of schema and \(\theta State = \theta State’\) means that each variable in State is equal to corresponding decorated variable in State’.

Note that schemas in Z can be defined using the usual schema box or the horizontal notation [76]. Both styles are widely used within Z specifications, with the shorter horizontal notation usually employed for small schema definitions.

4.2.2 Disjoining error cases

The successful operation behaviour is usually defined to require a certain state. This state is operation precondition, and if the precondition puts additional constraints on abstract state, the operation is partial [3].

The precondition specifies state, under which operation invocation is successful. In order to achieve robust and complete interfaces with operations that can handle any input and are executable with any system state, operation error cases are defined and disjoined to produce a total operation as identified in Delta/Xi: disjoin errors design pattern [67].
Error cases are also operations with their own preconditions that describe the state other than that of successful operation [3]. Error cases usually provide report on the error and leave the state unchanged or perform some recovery.

A total operation joins the success and error cases to produce a robust solution using of schema disjunction [3].

\[ \text{FullOp} \equiv \text{SuccessOp} \lor \text{ErrFailOp} \lor \text{ErrAnotherFailOp} \]

The precondition operator \( \text{pre} \) distributes through disjunction [76], therefore the precondition of a total operation is a disjunction of each operation precondition. Constituting operation preconditions must cover all possible states together, producing \( \text{true} \) precondition of the total operation. This means that a total operation handles all inputs and can be executed for any system state - in each case, a specific execution path is taken based on the preconditions.

Disjoining the error cases can introduce non-determinism in the specification, when the error preconditions overlap. However, such non-determinism can be helpful at this level of abstraction, allowing the concrete error handling to be defined by implementations.

### 4.2.3 Operation partitioning

A common way to increase modularity and reuse in a Z specification is to partition schemas into smaller ones and then reuse them where applicable (Assemble from chunks design pattern [67]). Larger schemas, usually operations, are then constructed by joining the smaller schemas using conjunction (\( \land \));

This approach has great benefits, however the operations should be constructed with care, especially when used within total operations (Section 4.2.2) or in a schema composition.

The operation partitioning style is widely (often too much) used by Craig [12]. Craig defines small operations to change individual variables of large states. Then he combines them into larger operations, intermixed with schema compositions and error cases, which produces errors in some cases.

The safe rule would be to constrain all state variables before using operation in disjunction (\( \lor \)) or schema composition (\( o \)). Although in certain cases, under-specification can be useful at an appropriate level of abstraction. It would indicate that the constraint of a certain variable is not of particular interest for this model.

A more detailed example explaining operation partitioning and possible pitfalls is given in Appendix B, together with full formal specification and proofs.

### 4.2.4 Promotion

Promotion is a structural design pattern to permit local state / global state structure within Z specification. It defines a good way of increasing modularity and separation of concerns. Promotion is defined and analysed in a number of works [3, 67, 76] in its plain form and elaborations.

Promotion is used when multiple instances of a local state with operations are referenced in a global state. The local states are referenced using some identifier, allowing them to be accessed uniquely within the global state [3].

For such structure, a general promotion operation is defined, which ‘promotes’ operations on a local state within the global state. Using it, operations on global state can be defined that execute a corresponding operation on the identified local state.

Furthermore, a special initialisation promotion operation is defined to initialise new local state instances within global state. The general promotion operation updates existing local state values based on the executed operations, while the initialisation promotion instantiates a new local state and adds it to the global state [67].

The main components in the separation kernel, process table, process queues and the scheduler are individual data structures (2 instances for process queues), therefore promotion is not applicable to them. However, the messaging subsystem in the separation kernel contains message queues for each process, allowing to benefit from structure using promotion.

### 4.3 Verification of the model

#### 4.3.1 Syntax, type and domain checking

A Z specification is merely a document containing schemas, predicates and other Z constructs, typeset and laid out according to Z notation. The Z ISO standard [33] defines \LaTeX{} mark-up for Z notation, allowing to express special
symbols and constructs in Z using plain-text \text{l}\text{a}\text{t}e\text{x} commands. Furthermore, taking advantage of \text{l}\text{a}\text{t}e\text{x} document preparation system \cite{40} allows creating correctly laid out specifications without much difficulty.

\textbf{Syntax and type checking}

Without proper syntax and type checking, Z specification is just a document and may contain mistakes. The benefit from using a Z tool to prepare Z specification is invaluable. Z/Eves tool can read and analyse Z specifications prepared using \text{l}\text{a}\text{t}e\text{x} mark-up. Furthermore, it provides automatic syntax and type checking \cite{57}. Syntax checking helps to find mistakes such as missing brackets, incorrect set comprehension formulations and so on \cite{3}. Type checks are performed to ensure that operator and operand (function and parameter) types match, otherwise the specification is incorrect \cite{57}.

The syntax and type checking is performed when the specification is read, marking failed parts of the specification as incorrect. Using Z/Eves to prepare and analyse the specification allows to construct a syntactically correct formal Z specification.

The approach of using \text{l}\text{a}\text{t}e\text{x} mark-up to prepare the specification is employed for separation kernel model. Such specification has a standardised format \cite{33}, thus is suitable for inclusion and curation in the Verified Software Repository \cite{74}.

\textbf{Domain checking}

Domain checking is performed to verify that all functions in the formal model are applied in their domains. Such errors as division by zero, function application for elements not belonging to their domain, produce undefinedness in the specification. While the undefinedness is Z treated differently in the literature, Z/Eves takes the approach of showing that the expressions are meaningful. For cases when the Z/Eves type checking system and basic reasoning about total functions/operations is insufficient, Z/Eves generates a domain check predicate about function application \cite{57}.

The domain checks are formulated as theorems, therefore when such function application is defined, the domain check is added to unproved theorem list. The generated predicate must be proved using Z/Eves facilities to show that the application is within domain. To satisfy the requirement of verified separation kernel model (Section 3.1), all domain checks must be proven.

\subsection{4.3.2 Global inconsistency checking}

An axiom box, generic box or a predicate that is too strong can render whole specification inconsistent. In such case there are no possible systems that satisfy the specification \cite{57}. To avoid inconsistent specifications, consistency checks are necessary before each axiomatic definition or similar construct that can produce inconsistency.

Consistency check verifies that there exist instances of constrained items for the defined invariants.

\[
\begin{array}{c}
D \\
\hline \\
P
\end{array}
\]

For an axiomatic box defined above, the system is consistent if a certain $D$ can be found that satisfies predicate $P$: $\exists D \cdot P$ \cite{57}.

Consistency check needs to be defined before the actual constraint, as otherwise the axiomatic definition is used to prove the theorem - it will always be true.

To demonstrate consistency calculation, consider the following axiomatic definition of minimum and maximum process identifier (PID) values $\text{minpid}$, $\text{maxpid}$:

\[
\begin{array}{c}
\text{minpid}, \text{maxpid} : \mathbb{N} \\
\hline \\
\text{minpid} \leq \text{maxpid}
\end{array}
\]

The constraint $\text{minpid} \leq \text{maxpid}$ requires us to show that values $\text{minpid}$ and $\text{maxpid}$ exist, which can satisfy the constraint. Thus the consistency check theorem is defined preceeding the above declaration, which shows that these values exist.

\textbf{theorem} tPIDConsistency

$\exists \text{minpid}, \text{maxpid} : \mathbb{N} \cdot \text{minpid} \leq \text{maxpid}$
The theorem is very simple to prove, by instantiating \( \text{minpid} = 1 \) and \( \text{maxpid} = 9999 \). Its result shows that a system can be constructed where the constraint \( \text{minpid} \leq \text{maxpid} \) is valid.

### 4.3.3 Initial state verification

The operation to create initial state is part of the Established Strategy process of constructing Z specifications. The initialisation operation produces only the after-state, which is considered to be the initial state of the modelled component or system. Furthermore, predicates can be indicated to constrain initial variables to particular values [3].

\[
\text{StateInit} \equiv [\text{State'} \mid \text{predicate for initial state}]
\]

In a manner similar to the global inconsistency checking (Section 4.3.2), it is necessary to verify that the initialisation predicates are not too strong and a possible system state, satisfying state invariants, can be established - check for local inconsistency. This is done by proving the the initialisation theorem [76]:

\[
\text{theorem } t\text{StateInit} \\
\exists \text{State'} \cdot \text{StateInit}
\]

Furthermore, proving this theorem also verifies that no local inconsistency has been introduced in abstract state definition [57].

### 4.3.4 Precondition calculation

Following the Delta/Xi pattern, operations are specified as schemas, relating before-state and input variables to after-state and output variables [3]. An operation precondition is defined by existentially quantifying all output and after-state variables [76]. Z defines a core word \text{pre} to define precondition of operations [33, 76]:

\[
\text{pre Operation} = \exists \text{State'} \cdot \text{Operation} \setminus \text{outputs}
\]

Precondition calculation determines the state, under which the operation execution is successful, that is operation predicates always define an after-state that satisfies state invariants. The precondition state is specified in a so called “signature schema”, which contains the before-state and input variables, constrained by precondition predicates. Then the precondition is verified using precondition calculation theorem [18]:

\[
\text{theorem } t\text{OperationPre} \\
\forall \text{OperationSig} \cdot \text{pre Operation}
\]

Such proof verifies that the operation can be executed successfully (a valid after-state can be found) for the state defined in \text{OperationSig}. The precondition calculation also checks for local inconsistency in operations, as a too strong operation predicate results in \text{false} precondition [3].

### 4.3.5 Equivalence and refinement proofs

An existing Z specification can be updated without changing its model. The change of definition style, usage of more appropriate operators/functions, removal or addition of derived predicates and other refactorings are performed to increase the quality, readability and cohesion of a Z specification. These activities do not change the actual mathematical definitions.

To ensure that the change was only stylistic, it is necessary to prove the equivalence theorem between the original and changed schemas, which shows that both schemas are mathematically equivalent.

\[
\text{theorem } t\text{StateEquiv} \\
\text{State } \Leftrightarrow \text{StateOriginal}
\]

Sometimes new predicates or variables are introduced or other change performed that makes the schemas no longer equivalent. While justifying all the change is mandatory, it is wise to show that the change is a refinement of the original schema.

\[
\text{theorem } t\text{StateRefinement} \\
\text{State } \Rightarrow \text{StateOriginal}
\]

Proof that new schema implies the original indicates that the changed schema satisfies the invariants of the original, while introducing additional information.
4.3.6 Interesting properties

The specification structure and verification techniques, defined in the previous sections, arise from the Established Strategy development model. They should be applied when constructing the formal model to ensure basic correctness of the specification.

Further analysis involves creative examination of the formal model. Various properties of the model can be formulated and proved using theorems. In certain cases, the requirements state that certain properties of the system must hold [3]. Otherwise, the modeller may be able to spot desirable properties to show the results of operations, reason about the data structure contents and so on.

There is no general way to specify such properties, as they must be formulated for each case using Z notation. For example, proof about certain properties of the operation outcome can be formulated as a universally quantified predicate. The following theorem states that predicate \( P \) on operation result (after-state) \( State' \) holds for all successful executions of \( Operation \).

\[
\text{theorem } t\text{OpProperty} \\
\forall Operation \bullet P \text{ about } State'
\]

Proving such properties gives insight to the formal model and increases confidence in it.

4.4 Naming and style conventions

Good naming conventions, when used consistently within the specification, make the specification easy to read and follow. Z specifications easily grow in size and real specification can consist of several hundred pages [67]. Thus, appropriate naming style may suggest the meaning, type or purpose of the Z construct and may save the effort of finding its actual declaration. Stepney et al. define this as Name consistently presentation pattern [67].

4.4.1 Declarations and layout

This project adopts the Z naming conventions from [3]. Barden et al. define a house style for Z specifications and govern the naming and layout of sets, schemas and variables.

The specification is structured according to the general Delta/Xi pattern [67]. The given set, generic parameter, free type names are given in upper case, while schema names are written in CamelCase, with first letter capitalised. Variable names are also in CamelCase, starting with lower case letter.

In Z schema declarations, the included schemas are written before simple declarations, and precondition predicates preceding postcondition ones.

The initial states for state schemas carry a suffix Init. For example, the initial state of state schema \( PTab \) is called \( PTabInit \).

4.4.2 Theorems

Moreover, a consistent style for Z theorems and lemmas, as well as the general document structure is borrowed from [18]. All theorems carry a prefix, describing their type/usage. Please refer to [18] for full explanation of Z/Eves proof environment.

- \( g \) - assumption rule (grule);
- \( f \) - forward rule (frule);
- \( l \) - lemma or rewriting rule (rule). Freitas [18] suggest to have \( r \) for rewriting rules and \( I \) for lemmas, however rewriting rules are often used as lemmas thus a common \( l \) prefix is used for both;
- \( t \) - theorem (not a rule).

Theorem names are designed to describe in short its contents. To preserve consistency, some of the most popular theorem types are indicated with common suffixes, to indicate their meaning:

- **Consistency** - variable consistency calculation (see Section 4.3.2);
- **Equiv** - equivalence theorem (see Section 4.3.5);
- **Init** - data type (schema) initialisation calculation (see Section 4.3.3). Note that the same suffix is used for the initial states - the difference is that initialisation theorems always carry a prefix \(\text{t}\). For example, the initial state \(\text{PTabInit}\) and its initialisation calculation theorem \(\text{tPTabInit}\);

- **Original** - the schema as is in [12];

- **Pre** - operation (schema) precondition calculation (see Section 4.3.4);

- **Sig** - operation (schema) signature (see Section 4.3.4);

- **Type** - maximal type rule;

### 4.4.3 Disjoining error cases

Total operations (Section 4.2.2) consist of success operations and appropriate error cases composed using schema disjunction. The main (success) operations composing a total operation are indicated with suffix \(0\) ("zero") and all failure cases with \(\text{Err}\) prefix for easier readability [21].

The following example shows a total operation definition following these naming conventions.

\[
\text{FullOp} == \text{SuccessOp}_0 \lor \text{ErrFailOp} \lor \text{ErrAnotherFailOp}
\]

### 4.4.4 Naming inconsistencies in original specification

The original specification in [12, Ch. 5] has various style and naming inconsistencies, which make it hard for the reader to follow and comprehend.

One of the noticeable attempts in the original specification is to keep uppercase names for data types, e.g. \(\text{PID}\), \(\text{PTAB}\), \(\text{SKSCHED}\). However, as the data types get longer names, such as \(\text{DEVPROQUEUE}\), it is hard to separate words because of full capitalisation. For example, using CamelCase, such name is written as \(\text{DevProcQueue}\) and is much easier to read. Moreover, concatenation of multiple data types, such as \(\text{SetHWTSS}\) (for \(\text{HW}\) and \(\text{TSS}\)) puts even more burden to the reader. Finally, some operations are given full capitalisation, undermining all “all capitals for data types” idea, e.g. \(\text{CtxtSw}\).

Craig also features name of the data type (in capitals) in the names of some operations for that data type, such as \(\text{DequeuePROCESSQUEUE}\) for \(\text{PROCESSQUEUE}\) data type. However, this is used without any consistency, because there are a lot of operations such as \(\text{AddIdleProcess}\), \(\text{SetPreviousProcess}\), etc. Even more, some operations with abbreviated CamelCase names, e.g. \(\text{AddPD}\) (Add Process Definition) or \(\text{SKSchedNext}\) (separation kernel Schedule Next) can leave the reader searching for non-existent data types \(\text{PD}\) or \(\text{SK}\). Finally, capitalisation in some of the names just do not follow any guessable pattern, e.g. \(\text{AddPDESC}\) (possibly Add Process Description) or \(\text{IDLEPROCESSIdent}\) (Idle Process Identifier).

To avoid arising naming problems in the original specification, the CamelCase naming style is adopted for schemas (both data types and operations) as indicated in Section 4.4.1.

### 4.5 Transactional operations

Error handling via disjunction is a common way to specify total operations [3, 76]. Larger operations that involve sequential execution of several total operations can be defined in Z using a schema composition (\(\circ\)) operator. Now, if this larger operation is required to be a transaction [25], schema composition is not enough to ensure atomicity of the operation.

In this section we examine shortcomings of schema composition for transactions and propose a rough Transaction design pattern for Z notation to achieve atomicity for sequential composition of total operations. Furthermore, the pattern is derived and applied in a simple example, which illustrates the problem and results.

The Transaction pattern and its intermediate states can be used for composite operation modelling within the scheduler of separation kernel.

#### 4.5.1 Transaction construction

First we investigate how a sequential execution of operations is constructed and the required transaction properties. Then we examine an error handling style by [3], which solves early operation failures and error propagation. Finally, we define the rough design pattern to achieve atomicity of the full operation.

23
Operation and its properties

In general, a total operation can be executed for any state of the data, i.e. has a precondition \textit{true}. Furthermore, the error cases usually do not change the state - this means that even if an operation fails, the data is not affected.

While construction of total operations is a desirable goal for all data types, using them at a higher level causes some problems. In particular, we investigate the case when two total operations (which contain error cases) are executed sequentially.

In Z notation, schema composition ($\circ$) is used to define this intent. For example, a composite operation that involves sequential execution of total operations - first \textit{UpdateFirst} and then \textit{UpdateSecond} - is defined as

\[ \text{UpdateFull} \equiv \text{UpdateFirst} \circ \text{UpdateSecond} \]

Problems arise when it is required that the composite operation has transaction processing properties [25]. While the \textit{consistency}, \textit{isolation} and \textit{durability} properties are implied by the use of formal methods (e.g. the correctness of after-state is verified by consistency/precondition checks), the \textit{atomicity} cannot be guaranteed when using schema composition of total operations. A transaction is \textit{atomic} if its operations either all succeed or no change is done.

The basic use of schema composition puts no restrictions on operation execution based on success/failure of other operations. While the usual error case handling for failed operation is “no change”, the other operation may be executed and produce change to the state.

Early failure handling

A solution to the case if the first operation fails can be found in [3]. An \textit{Errors with memory} error handling method [3, Sec. 18.3] defines operations that are executed only if the system is in a valid state. A state variable is employed to hold the system status and a failed operation sets it to an error.

Furthermore, if the system is in erroneous state (i.e. previous operation failed), the error is propagated by the next operation without any change to the state. We enhance the full operation to check whether the first operation failed and propagate the error without state change:

\[ \text{UpdateFullCheckFirst} \equiv \text{UpdateFirst} \circ (\text{IsOk} \land \text{UpdateSecond}) \lor \text{ErrNotOk} \]

Such definition ensures atomicity in the case of first operation failure. Error handling mechanism within \textit{UpdateFirst} would set system to erroneous state and leave the data unchanged. Then \textit{IsOk} checks whether the system is in valid state and otherwise propagates the error with \textit{ErrNotOk}. For a full example, please refer to Section 4.5.2.

However, this solution does not achieve atomicity when the second operation fails - the first operation may be executed successfully. Nevertheless, this model is used further when we define the full transaction design pattern.

Transaction

The \textit{atomicity} property requires that if the second operation fails and therefore does not change the state, the first operation must not change the state as well. This cannot be defined in a straightforward fashion, as in the schema composition, operation before the \( \circ \) operator does not have access to second operation’s after-state.

Furthermore, we cannot use second operation success precondition to allow/disallow execution of the first operation, e.g. \( \text{pre UpdateSecond} \Rightarrow \text{UpdateFirst} \). The failure of the second operation could be caused because of the state produced by \textit{UpdateFirst} and such check is not able to consider that state.

We propose a \textit{Transaction} pattern as a simple solution to achieve \textit{atomicity} of operations, which is employed in transactions within other systems. In the event of a failure of any operation within transaction, the data is rolled back to its previous state before transaction [25].

The design pattern then is quite simple. We record the initial before-state before the execution of operation. Then the operations are executed and results produced. Successful operations mark system state as OK, while the failures indicate specific errors. At the end of operation execution we check the system state. In case of error, we roll back the changes by using the recorded initial state as the after-state of the transaction. If everything succeeds, the resulting state is kept.

The definition of revert (roll back) and transaction validation in Z is done using several constructs. Consider a system state schema \textit{Numbers} and error state \textit{ErrState}. For a full example, please refer to Section 4.5.2.
We use a decorated state $Numbers''$ to store the initial state. Then we define backup and revert operations to store the initial before-state and apply it to the after-state, respectively.

$$NumbersBackup \equiv [\Delta Numbers; Numbers'' | \theta Numbers = \theta Numbers'']$$

$$NumbersRevert \equiv [\Delta Numbers; Numbers'' | \theta Numbers' = \theta Numbers'']$$

Refer to Section 4.2.1 and Z literature [76] for more information on used notation.

During transaction handling, if the system is in valid state ($err = ok$), the results and error state are kept (using $\Xi$ notation). Otherwise, $NumbersRevert$ is employed to roll back the state while keeping the error to indicate failure.

$$CompleteTransaction0 \equiv [\Xi Numbers; \Xi ErrState | err = ok]$$

$$ErrRevertTransaction \equiv [\Delta Numbers; \Xi ErrState; NumbersRevert | err \neq ok]$$

$$CompleteTransaction \equiv CompleteTransaction0 \lor ErrRevertTransaction$$

The composite operation must be defined with error propagation (see early failure handling), otherwise successful execution of second operation overwrites the error value. Furthermore, the $Numbers''$ state is hidden using existential quantification.

$$UpdateFullTransaction \equiv \exists Numbers'' \bullet ((UpdateFullCheckFirst \land NumbersBackup) \Rightarrow CompleteTransaction)$$

This pattern can be easily extended for the composition of multiple total operations and other error handling styles, however we chose not to expand more on this in this dissertation, as the we focus on formal modelling of a separation kernel.

We have defined Transaction pattern which models atomicity within composition of total operation in Z. Next we analyse its derivation, usage and results via a simple example.

### 4.5.2 Transaction example

In this example we investigate the schema composition of total operations. For each composite operation variant in Section 4.5.1, we analyse the results for different system states and find their shortcomings and successes. In the end we construct a transactional operation according to the Transaction design pattern (Section 4.5.1).

In this section we guide through the example by indicating the important parts. For full formal specification of this example and associated theorems and proofs, please refer to Appendix C.

#### State and component operations

Consider a simple state schema with numeric variables $x$, $y$ and no invariants.

$$\text{Numbers}$$

$x, y : \mathbb{Z}$

Furthermore, we construct a state $ErrState$ to store system status, i.e. whether a system is in a valid ($ok$) or an invalid (any $errorX$) state. Each error is used for different operation, as we want to illustrate correct error propagation.

$$Status ::= ok \mid error1 \mid error2$$

$$\text{ErrState}$$

$err : Status$

Now we can define simple operations for the state. For each operation we introduce arbitrary preconditions to construct total operations. The first operation simply increments $x$ and $y$ when $x < 10$. 

25
We assume that the opposite condition, when $x \geq 10$, is an error case. We construct an error operation that keeps the state unchanged and updates the system state to $\text{error}\,1$.

Then the total operation is defined by disjoining the cases in the usual fashion.

$\text{UpdateFirst} \equiv \text{UpdateFirst}_{0} \lor \text{ErrFirst}$

The second total operation is defined in a similar way. It has a different precondition ($y < 10$) and the failure sets system status to $\text{error}\,2$.

Next we investigate ways of defining sequential execution of $\text{UpdateFirst}$ and then $\text{UpdateSecond}$.

**Plain composition**

First we investigate the plain composition of both operations. We use schema composition ($\circ$), because both operations have constraints on the same variables and thus a conjunction is not possible.

$\text{UpdateFullNoCheck} \equiv \text{UpdateFirst} \circ \text{UpdateSecond}$

Such composition, if both cases succeed, increments the variables twice - once for each operation. We formulate the execution of successful state as a theorem and prove it. For full definitions and proofs, please refer to Appendix C.

**Theorem** $\text{tUpdateFullNoCheckSuccess}$

\[ \forall \text{Numbers} \mid x < 10 \land y < 9 \bullet \]

\[ \text{UpdateFullNoCheck} \Rightarrow (x' = x + 2 \land y' = y + 2 \land \text{err}' = \text{ok}) \]
We also prove that if both operations fail, nothing is changed and one of the errors is reported.

However, for the state when only one success precondition is satisfied, the composed operation is no longer atomic - one operation succeeds and changes Numbers state, while the other handles an error state and does not change the state. We show one such case in tUpdateFullNoCheckFailFirst.

\[
\text{theorem} \ t\text{UpdateFullNoCheckFailFirst} \ \forall \text{Numbers} \mid x \geq 10 \land y < 10 \bullet \ \\
\quad \text{UpdateFullNoCheck} \Rightarrow (x' = x + 1 \land y' = y + 1 \land \text{err}' = \text{ok})
\]

We get an intermediate state as a result - only one operation has succeeded and variables have been updated only partially. Even more - the second operation overrides error state and loses the error. This is clearly a non-desirable result, which the proof of this theorem shows that there is something wrong with the way one could choose to structure one’s models.

Composition with check for first operation failure

To solve the case when the first operation fails, we use early failure handling technique. We define operations to check for valid system state and propagate error in the case of failure.

\[
\text{IsOk} = [\Delta \text{ErrState} \mid \text{err} = \text{ok}]
\]

\[
\text{ErrNotOk} = [\Xi \text{Numbers}; \Xi \text{ErrState} \mid \neg \text{err} = \text{ok}]
\]

We use the operations to check for failure on the execution of second operation.

\[
\text{UpdateFullCheckFirst} = \text{UpdateFirst} \circ (\text{IsOk} \land \text{UpdateSecond}) \lor \text{ErrNotOk}
\]

We show that the first operation failure is handled correctly: state is not changed and a corresponding error is reported.

\[
\text{theorem} \ t\text{UpdateFullCheckFirstFailFirst} \ \forall \text{Numbers} \mid x \geq 10 \land y < 10 \bullet \ \\
\quad \text{UpdateFullCheckFirst} \Rightarrow (x' = x \land y' = y \land \text{err}' = \text{error}1)
\]

Furthermore, we formulate theorems for some regression testing and verify that the earlier defined cases of both operations succeeding or both failing are also handled correctly.

Such operation structure is simple and is useful for handling first operation failures in schema composition. It should be used when the second operation always succeeds. However, if the second operation has failure cases, such composition fails.

\[
\text{theorem} \ t\text{UpdateFullCheckFirstFailSecond} \ \forall \text{Numbers} \mid x < 10 \land y \geq 9 \bullet \ \\
\quad \text{UpdateFullCheckFirst} \Rightarrow (x' = x + 1 \land y' = y + 1 \land \text{err}' = \text{error}2)
\]

The theorem shows that on the failure of the second operation, the error is reported correctly but the state is changed by the first operation.

Composition within transaction

Following the Transaction pattern definition (Section 4.5.1) we define the full transaction operation.

\[
\text{NumbersBackup} = [\Delta \text{Numbers}; \text{Numbers}'' \mid \theta \text{Numbers} = \theta \text{Numbers}'']
\]

\[
\text{NumbersRevert} = [\Delta \text{Numbers}; \text{Numbers}'' \mid \theta \text{Numbers}' = \theta \text{Numbers}'']
\]

Transaction handling in success case propagates the results using Ξ operators. As this operation will be joined in transaction using schema composition, the before-state of CompleteTransaction0 will actually be the result after-state of the operation.

\[
\text{CompleteTransaction}0 = [\Xi \text{Numbers}; \Xi \text{ErrState} \mid \text{err} = \text{ok}]
\]
For failure case, we discard the results and revert to initial state using using \textit{NumbersRevert}. Furthermore, we preserve the error state so that we keep the log of what happened.

\[
\text{ErrRevertTransaction} \triangleq [\Delta \text{Numbers}; \exists \text{ErrState}; \text{NumbersRevert} \mid \text{err} \neq \text{ok}]
\]

\[
\text{CompleteTransaction} \triangleq \text{CompleteTransaction}_0 \lor \text{ErrRevertTransaction}
\]

Now we can define the full transactional operation. We reuse \textit{UpdateFullCheckFirst} definition, because we need error propagation if the first operation fails. The initial state is backed up by \textit{NumbersBackup} and results are handled using \textit{CompleteTransaction}. We hide \textit{Numbers''} using existential quantification, because it is a local variable state and should not be visible outside the transaction.

\[
\text{UpdateFullTransaction} \triangleq \exists \text{Numbers''} \bullet ((\text{UpdateFullCheckFirst} \land \text{NumbersBackup}) \sqsubseteq \text{CompleteTransaction})
\]

We verify that the transactional definition handles all defined cases correctly. Also, we formulate and prove a general theorem about transaction results. For any given state of \textit{Numbers}, the transaction either succeeds with \textit{ok} signal, or fails and does not change the state.

\[
\text{theorem tUpdateFullTransaction} \forall \text{Numbers} \bullet \text{UpdateFullTransaction} \Rightarrow
\begin{align*}
(x' = x + 2 \land y' = y + 2 \land \text{err'} = \text{ok}) \lor (x' = x \land y' = y \land \text{err'} \neq \text{ok})
\end{align*}
\]

Finally we make sure that the transaction is a total operation - it’s precondition is \textit{true}.

This example illustrates the application of \textit{Transaction} design pattern for Z specifications. We have successfully applied the principle of rolling back the transaction from databases and other systems and developed the \textit{Transaction} design pattern. It contributes as an original research to the existing knowledge and practices of using formal methods and in particular Z notation.

### 4.6 Z/Eves theorem prover

Z/Eves theorem prover is a simple and user-friendly tool for preparation and reasoning about Z specifications [56]. It has an easier learning curve in comparison to related theorem provers; and is equipped with powerful tactics that enable high levels of automation during proofs [18].

Z/Eves has proven successful for carrying out Z projects during the pilot projects within Grand Challenge in Verified Software (Section 2.1). A substantial material has been accumulated during its use in pilot and other projects in addition to the official documentation [46, 57, 58]. Experiences and guidelines how to drive the theorem prover to achieve desired results [18, 23] as well as useful lemmas and rules to supplement the official toolkit [20, 21] are available in the literature and within the Verified Software Repository.

Finally, the directly reusable proofs from simple kernel mechanisation using Z/Eves facilitate this project, therefore Z/Eves (version 2.3) is a natural choice as the main tool.

#### 4.6.1 Using Z/Eves

Z/Eves identifies as an interactive system for composing, checking and analysing Z specifications [57]. A Python-based frontend to the prover is a user-friendly interface for specification preparation and an excellent way of introducing users to Z. However, it has several significant shortcomings and suffers from frequent crashes [18].

Z/Eves can be interacted with using its textual (command-line) interface [46, 57]. Z notation within the command-line is entered using LATEX notation [33]. This works naturally with specification preparation, as the formal specification is prepared using Z LATEX notation. Furthermore, this interface provides a more verbose information between proof steps, which is useful to see what automatic reasoning has been applied.

However, the textual interface has poor integration with available LATEX editors and interaction with command-line in Windows operating system is limited. While an interface to emacs addresses some of the limitations, an integration with modern development environments is desirable. As Woodcock et al. [78] note, integration of formal methods tools with existing design, programming and static analysis environments should be one of the directions in tool development.
4.6.2 Automation

The Z/Eves prover is powerful and can perform difficult automatic reasoning and proof of the theorems. However, sometimes certain assumptions or rules need to be added to prover’s toolkit in order for it to carry the rest of the proof automatically [18].

The rewriting rules (rule) are applied to transform the proof goal when their pattern matches. Assumption (grule) and forward (frule) rules do not transform the goal, but are used by the prover to discharge type checks and side conditions, especially when complex data types are involved. For a detailed discussion on the automation rules, please refer to [18].

4.7 Separation kernel architecture

This section presents the general design decisions about used separation kernel model. The model is based on separation kernel in [12]. This high-level design justification explains how the separation kernel requirements are addressed by the model (Section 3.1). When a particular requirement is addressed, it is indicated by its [Req-] tag.

Design of full separation kernel functionality is analysed, which goes beyond the scope of the dissertation. This is because the components within the scope, such as process table, are central to the kernel and interact with the majority of other components. For this reason, they cannot be examined in isolation.

Note that this section covers only the architecture and major design decisions. It is intended to give a high-level overview of the separation kernel design. The subsequent top-level Z specification also serves as a complete formal design document. Thus the details of the model, concrete design decisions, alternatives, justification and evaluation is examined later in Chapter 5.

4.7.1 Architecture overview

The separation kernel has minimal functionality and therefore a simple model [Req-F1]. Kernel provides basic process management features, simple scheduling and inter-process communication facilities. Furthermore, basic hardware abstraction is attempted. The kernel contains a memory allocation manager, raw interrupt handling facilities and provides a synchronous I/O model for interactions with hardware.

An overview of kernel architecture is presented in Figure A.1. Due to its large size, the diagram is displayed in Appendix A.

Note that the diagram is not an exact and detailed representation of the architecture, but instead a rough overview for easier comprehension. Figure A.1 indicates main components in the separation kernel (blue rounded rectangles), such as the Process Table, Scheduler or Storage Pool. Furthermore, relationships between the components (solid lines) and the promoted interfaces (dotted lines) are established.

The notation used to render the architecture is borrowed from Service Component Architecture (SCA) assembly model [61]. SCA assembly model defines the relationships and composition of service components within distributed domains. Each service component has references to other services (green arrow) or provides own services (purple arrow). Provided and required services are linked together to compose the whole system. Furthermore, the assembly model defines hierarchical composition of components, when certain services/references of internal components are exposed as the interface of a larger component [61].

While the separation kernel and its components are not SCA constructs, the notation and ideas seemed appropriate to describe the formal kernel architecture. Each component within the kernel is modelled as an individual construct with clear interface to other components. Where appropriate, components using the defined interfaces. Moreover, some of the kernel operations are exposed (promoted) to be accessible for user processes outside the kernel.

The SCA notation [61] is not followed blindly as the goal is to present an architectural overview, not to produce a perfect diagram. Furthermore, labels for relationships (services) are provided only where clarity is needed. The notation allows to clearly indicate interfaces available for user and device processes as well as present the rough internal structure of a separation kernel model.

In the remainder of this section, the prominent components, relationships and design decisions are briefly examined. It is advised to use Figure A.1 as a reference for the “big picture”.
4.7.2 User and device processes

Separation kernel processes are classified as user or device processes. User processes (subjects) are completely untrusted code and main security properties of the kernel are applied to them. Device processes are the software for handling devices (for simplicity device process and device are used interchangeably within the thesis).

The device processes (kernel exported resources) are modelled as “trusted” code, which resides in kernel address space. “Trusted” code is assumed to have been formally verified [Req-H1]. The device process abstraction is modelled similarly to untrusted user processes and to some extent addresses requirement [Req-S3].

To protect internal kernel resources from access by user processes, verification by external identifiers is employed [Req-F2]. Both user and device processes are assigned external identifiers (device numbers for devices), which are used to reference processes outside the kernel. All references within the kernel are handled using internal process identifiers. When a user process initiates action within the kernel, the identifiers are verified and translated to internal representations.

The external user identifiers allocation is modelled within the Process Table. Craig [12] defines a concrete allocation algorithm. Each new process is assigned an increasing natural number. While this is not a wrong approach, an alternative design for a higher abstraction level can be employed, where external identifier is allocated non-deterministically.

4.7.3 Process table

Process Table acts as a central information resource for processes. All process attributes and associated constructs are stored within this data structure. The process table keeps the list of active processes, provides allocation and termination facilities for both user and device processes. Note that device processes are allocated at the start and considered to never terminate.

When a process is allocated, the process table generates its internal and external identifiers. At the top abstraction level, internal identifier allocation is modelled non-deterministically.

The process management operations in process table form part of kernel interface actions to create and terminate processes [Req-F1].

4.7.4 Memory management

An abstraction for real memory is modelled using the Raw Storage component. Then Storage Pool component provides memory allocation and management within this storage. Each user process is allocated a separate partition of non-overlapping address space within the memory [Req-M1, S1]. An overview of memory space allocation within separation kernel is provided in Figure 4.2.

Craig [12] simplifies the model by assuming that the kernel is running on Intel IA32 processor [31]. This allows to rely on hardware support for segmentation. An illegal attempt to access memory segments of another process is detected by the hardware, and can be handled within the kernel [Req-V2].

Also, a dedicated portion of memory is allocated as a message buffer for inter-process messaging subsystem.

Figure 4.2: Separation kernel memory overview
Finally, the trusted device processes are considered part of the kernel and therefore reside in kernel address space (Figure 4.2). The interrupt service routines (ISRs) are small operations for handling interrupts and also share kernel address space.

Memory management facilities form part of process creation/termination interface but are not accessible to user processes directly [Req-H2].

4.7.5 Messaging

Each process in separation kernel model is treated as an isolated instance. For inter-process communication, an asynchronous messaging system is modelled [Req-S1]. To keep the separation of address spaces, the sent message is copied to a dedicated message buffer memory space, thus address space cannot be shared between partitions [Req-S1, M2]. Refer to [12] for full discussion, as messaging system out of scope of this dissertation.

The interaction between user processes and devices follows a synchronous I/O model. A different messaging system based on “request-reply” interaction is modelled. When a user process sends a request to a device, it suspends and waits for device reply. To ensure synchronous interaction, device process must have priority to execute within the scheduler. When the device posts a reply, it is removed from execution.

The device messaging system models user process interaction with kernel resources (devices) that allow access through unprivileged interfaces [Req-H3].

4.7.6 Scheduling

Separation kernel employs a non-preemptive round-robin Scheduler. A running process can only suspend voluntarily or terminate. Scheduler provides operations that form part of kernel interface for process execution - suspend, terminate or ready process.

It is assumed that the kernel is running on a single-processor machine, therefore non-preemption means that only a single process is executing at each given time [Req-E1].

Craig [12] takes the round-robin scheduling approach from the original suggestion by Rushby [55]. The algorithm is chosen for its simplicity. Round-robin is modelled using FIFO Process Queue components, which store scheduled processes before execution.

Device processes are executed with priority over user processes, to support synchronous I/O access. Priority execution is modelled with two distinct Process Queue components - one is a priority (device) queue, the other is a normal (user) queue.

The assumption of Intel IA32 processor allows to simplify the context switching activity. It is assumed that context switch is performed by executing JMP hardware instruction. This instruction changes the process registers and executes target process. Because of this simplification, a detailed formal model for task switching operations is not provided.

4.7.7 Kernel interface

The formal model defines robust external interfaces for both user and device processes (shown in Figure A.1 as purple arrows on kernel rectangle edge). The exposed operations are limited and represent only the necessary functionality [Req-L1]. Note that the interface operations are indicated in the specification as such by associated comments, since Z does not support the notion of encapsulation.

4.7.8 Hardware considerations

In addition to device process model for kernel resources, the separation kernel has an abstraction model for hardware (processor) and interrupt handling facilities.

While the interrupt facilities in the kernel are widely used, the full formal model of interrupts is not provided. At the current level of abstraction, only interrupt raising operations and resulting interrupt handlers are formally modelled. The actual link between raising an interrupt and handling it is not specified, therefore [Req-H4] is only partially addressed.

The internal kernel resources, such as processor and memory management, are contained within the kernel. No interface is provided for user processes to manipulate these components [Req-H2].
4.7.9 Limitations of the model

The requirements in [63] for kernel configuration are important to ensure full kernel security. The established communication channels, allowed interfaces, memory allocations and other parameters must be explicitly configured. The original separation kernel model in [12] does not include configuration support, therefore requirements [Req-S2, C1, C3, C4, L2, V1] are either not satisfied or only partially satisfied within the model. Nevertheless, the design of the model can be extended to include necessary configuration requirements. The limited scope of the dissertation requires these extensions to be left for future research.

The missing audit and deployment functionality [Req-C2, V3, D1] is considered an extension to the core model of separation kernel and therefore left for future research.

Furthermore, the verification of process separation is addressed only partially. The provided interfaces allow process communication using asynchronous messaging only. Hardware support for memory segmentation guarantees that address space of other process cannot be accessed [Req-V1, V2]. However, the actual proofs about the security properties of a separation kernel are not performed in the original specification [12]. Messaging and virtual store components are out of scope of this dissertation, therefore verification is left for future research.

4.7.10 Dissertation scope

The scope of this dissertation is limited to basic types (e.g. process identifier definitions), process table, process queue and the scheduler (Section 3.4.3). Nevertheless, the consideration of full kernel design - the “big picture” - is necessary, as the modelled components are central to separation kernel. The usage of components and relationships with other parts of the kernel must be examined for better understanding of kernel design and to avoid mistakes that render the model not usable in the future.

The overview of kernel architecture within the dissertation scope is given in Figure 4.3.

![Separation Kernel Architecture](image)

**Figure 4.3:** Separation kernel architecture, dissertation scope

The final kernel interface operations are out of scope (even though depicted in the diagram), since they depend on other components, such as the Storage Pool. Nevertheless, the clear interfaces of the process table and scheduler must be specified to be composed into kernel interfaces in the future.

This concludes the high-level overview of formal separation kernel model, based on formal specification by Craig [12]. The design and development of its formal specification is continued in Chapter 5. There, the detailed design decisions, particular data structures and model verification are given.

4.8 Summary

This chapter explained how the formal model for separation kernel will be constructed. The iterative *Established Strategy* is a widely used approach to formal Z specification that satisfies the formal model requirements. This
approach has been adapted for the project and detailed requirements and guidelines on specification structure and verification have been identified.

Then, a Transaction design pattern to achieve atomicity in schema composition has been proposed and developed. The design pattern is a result of iterative development and has been developed to solve problems that have arisen during the formal modelling, which is presented in the following sections. Transaction pattern contributes to the existing Z knowledge body and can be used for subsequent research.

The Z/Eves theorem prover has been selected as the main tool for the project, due to its proven track record in the area and plenty of resources to benefit from.

Finally, a high-level overview of separation kernel architecture examined the “big picture” of kernel design. The prominent components and their interactions have been identified and the separation kernel requirements have been addressed. The analysis of architecture limitations have shown that some of the requirements are not resolved, however the design can be extended to satisfy them. Due to the limitations of dissertation scope, this has been deferred for future research.

Following the Established Strategy, a detailed formal model of separation kernel can be constructed according to the identified architectural design.
Chapter 5

Basic Types, Hardware & Supporting Types

Having established both the high-level design of the separation kernel and the formal specification structure, a detailed construction of formal model can begin.

The formal specification spans several chapters, each chapter designated for a prominent part of the separation kernel. The Established Strategy is applied to specify each component definition and operations and then verify it.

This means that the formal specification contains both the design (detailed decisions about formal model) and evaluation (theorems and proofs). We try to avoid separation of these parts, as the development process allows small iterations, when the verification is performed on designed schemas or operations. Furthermore, such presentation style makes it easier to comprehend the model, when the proposed design decisions are immediately evaluated with formal proofs.

The initial part of formal specification sets up the scene for the main kernel components. This chapter presents the general lemmas, used to reason about Z constructs, relationships and their properties. Then, basic kernel data types are defined.

The modelling continues with hardware abstraction and a brief definition of necessary constructs from messaging and memory management components, which in general are out of scope of this dissertation. Constructed formal model is based on separation kernel definitions in [12, Sec. 5.1-5.2].

The full formal specification is given in Appendix E. Consult the full specification for details, as the less important or repetitive definitions are not included here. Furthermore, proof scripts for all theorems in the specification are provided in Appendix F.

5.1 General lemmas

Each pilot project in the Grand Challenge involves mechanisation and verification of various data types. While proving various aspects of different mathematical constructs, new properties of core Z operators, functions or more complex entities emerge. These properties are formulated as reusable lemmas and are collected in the extended Z/Eves toolkit [20, 21].

Mondex pilot project contributed with rules about functional overriding and finiteness [23], subsequent projects expanded the toolkit with lemmas about functions, sequences, mid-point insertion and cardinality [21].

The extended toolkit lemmas that were reused from previous projects are listed in Appendix D.

The defined general lemmas usually reason about simple and obvious statements. However, the Z/Eves prover has trouble reasoning about these properties and their derivation is usually quite challenging. However, once the proof is completed, lemma can be easily reused manually or within Z/Eves automation capabilities.

Extraction of such lemma and proving it separately can make an impossible proof solvable [21]. This approach is continued within this project and a number of useful reusable lemmas has been defined. Some of them are presented in Table 5.1.

<table>
<thead>
<tr>
<th>Table 5.1: General lemmas in the project.</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 ⊢ [X, Y] ∀ A : X \rightarrow B \land B : Y \rightarrow A \land P = Q \land P \in A \rightarrow B \land Q \in B \rightarrow A</td>
</tr>
<tr>
<td>L2 ⊢ [X, Y] ∀ A : X \rightarrow B \land B : Y \rightarrow A \land P \cup Q = P \cup Q</td>
</tr>
<tr>
<td>L3 ⊢ [X, Y] ∀ Y ∈ \text{iseq} Y \mid ¬ s = \langle⟩ \land head s \in ran (tail s)</td>
</tr>
<tr>
<td>L4 ⊢ [X, Y] ∀ A : X \rightarrow B \land B : Y \rightarrow A \land ran[X, Y] P = {} \Rightarrow P = {}</td>
</tr>
</tbody>
</table>
Lemma L1 states that inverse relationship between partial functions is a bijection, or separately - both functions are injective. This is proved by investigating partial injection definition. Lemma L2 verifies property already indicated in [76], that for functions with disjoint domains, relational override behaves as union operator.

Lemma L3 contributes to the toolkit about sequences. It states that for an injective sequence (no duplicate elements), the first element (head) is not among the remaining elements (tail). While this is implied by “no duplicates”, proving it has been a great challenge. The proof involves a number of creative steps and usage of targeted lemmas from both Z/Eves toolkit ([58]) and the extended toolkit (Appendix D).

Furthermore, lemmas L4 and L5 are similar and state that if function relationship domain (range) is empty, the whole relationship has no elements as well. Again, an obvious property, which proof is challenging due to the fact that relationship domain is defined in terms of relationship, however nothing is said about the relationship in terms of domain.

All general lemmas in Appendix D have been proved. With the necessary proof lemmas and theory established, basic data types of separation kernel can be examined in the following sections.

5.2 Process identifiers

All processes within kernel are internally referenced using unique process identifiers (PID). The PID data type is defined as a bounded range of natural numbers. Furthermore, this range is extended with a nullpid identifier to represent the null reference to a process. These definitions are analogous to the simple kernel [21].

As the range is defined axiomatically, it is necessary to prove a global consistency (see Section 4.3.2). Then the range limits can be defined axiomatically and the process identifier abbreviation PID can be stated.

\[
\begin{align*}
\text{minpid}, \text{maxpid} & : \mathbb{N} \\
\langle \text{rule lRangePID} \rangle & \text{minpid} \leq \text{maxpid} \\
\text{PID} & == \text{minpid} .. \text{maxpid}
\end{align*}
\]

The minpid and maxpid are defined in a way that minpid defines the lower bound and the definition allows at least one process identifier (when minpid = maxpid).

The nullpid is defined as a value, which is not a valid PID. This allows an arbitrary identifier selection. The suggestion in [12] is followed that defines nullpid to be larger than values in PID.

\[
\begin{align*}
\text{nullpid} & : \mathbb{N} \\
& \langle \text{rule lRangeGPID} \rangle \forall p : \text{PID} \bullet p < \text{nullpid} \\
\text{GPID} & == \{ \text{nullpid} \} \cup \text{PID}
\end{align*}
\]

The GPID set is used when referencing process identifiers that can have null values. Such definition of nullpid is prescriptive and a simple nullpid \notin PID would have sufficed. In that case, a more specific restriction could be introduced during refinement process, depending on the implementation choice. While the original nullpid definition is kept, we prove a theorem to show that nullpid cannot be confused with a valid value:

\[
\text{theorem tNullpidNotPID} \\
\text{nullpid} \notin \text{PID}
\]

Although this theorem seems obvious, the actual proof requires facts about ranges and guiding the prover through the proof.

5.3 Process type & external identifiers

User and device processes within the kernel have significant differences in how they are handled. To distinguish between them, a free type PTYPE is defined to indicate process type.

\[
\text{PTYPE ::= uproc | dproc}
\]
For each process type, a different external identifier data type is defined. The external identifier is intended to be unrelated to process identifier inside the secure kernel. Craig [12] suggests to denote each user process by a natural number identifier:

\[
UPID \equiv \mathbb{N}
\]

The \textit{UPID} is used to reference a user process outside the kernel, e.g. by other user processes. Natural numbers set suggests that the number of possible user processes is unbounded.

The external device number \textit{DevNo} is defined as a bounded range of available numbers. This follows from the assumption that the number of devices in a system is limited and they all are initialised at the start.

\[
\text{DevNo} \equiv \text{mindev} \ldots \text{maxdev}
\]

\textit{Remark 5.1.} Craig [12] originally defined the constraint on device numbers as \(\text{mindev} < \text{maxdev}\). Such specification requires the implementation to have at least two numbers allocated for devices (e.g. \(p_1 = \text{mindev}\) and \(p_2 = \text{maxdev}\)). We consider such specification to be too restrictive, e.g. there is no reason why a single-device system could not be defined. With a lack of reasoning about this restriction in [12], the constraint was relaxed to \(\text{mindev} \leq \text{maxdev}\).

### 5.4 Process state

Each process in the separation kernel always has an associated state. The state data type is defined as a free type:

\[
PSTATE \ ::= \text{psterm} \mid \text{psrunning} \mid \text{psready} \mid \text{psdevwait} \mid \text{pswtgdev}
\]

Within the scope of this dissertation, process scheduling states, namely \textit{psterm}, \textit{psrunning} and \textit{psready} respectively define states of terminated, currently executing, or ready to be executed processes. The \textit{psdevwait} and \textit{pswtgdev} are “waiting” states for user and device processes in synchronous I/O messaging.

### 5.5 Memory address

The memory addresses, which are used by processes, are defined as a range between \textit{nulladdr} = 0 and \textit{maxaddr}:

\[
\text{Addr} \equiv \text{nulladdr} \ldots \text{maxaddr}
\]

### 5.6 Task state segments

Hardware abstraction is modelled with the assumption that separation kernel is running on Intel IA32/64 architecture processor. To define process registers, which are used for context switching, a Task State Segment (TSS) structure from IA32 architecture is used. TSS contains all registers of a process as well as the stack pointer [31].

The TSS is an abstraction of a data structure and the model is not concerned with its actual representation. For this reason, \textit{tss} is best suited as a given type, \(\text{TSS}\).

However, such definition (used in [12]) is not enough and it needs to be extended. Subsequent proofs in process table require to show that, for example, when a new process identifier \(p? : \text{PID}\) is allocated, the system is in a valid state. To show that, it is necessary to satisfy a condition that there exists a mapping \(p? \mapsto t\), where \(t : \text{TSS}\) is “some TSS value”. Therefore, \(\text{TSS}\) must have at least one value:

\[
\text{theorem ITSSNotEmpty}
\] \(\exists t : \text{TSS} \cdot \text{true}\)

When a given set is defined in Z, nothing is said about its elements. A given set can be an empty one, which would very likely cause global inconsistency in the system. Therefore \(\text{ITSSNotEmpty}\) cannot be proved with the original definition.
One solution is to specify that the set is not empty. In our specification, we adopt a method suggested by Leo Freitas. Another given set \( TSS_0 \) is first defined and then \( TSS \) is defined as a non-empty subset of \( TSS_0 \).

\[
\begin{align*}
\{ TSS_0 \} \\
| TSS : \mathbb{P}_1 \ TSS_0
\end{align*}
\]

With such definition, \( ITSSNotEmpty \) can be proved and subsequent proofs can show that a valid system can be instantiated.

5.7 Interrupts

Craig [12] models a simplified interrupt mechanism. Interrupts are raised by the kernel in the hardware, then appropriate interrupt service routine (ISR) handles the interrupt, e.g. performs a context switch or halts the kernel.

The operations to raise interrupts and contents of important ISRs are defined within the model. However, the actual link between raising the interrupt and handling it is not defined. It is assumed that the execution is interrupted and ISR handling is executed [12].

Full hardware abstraction is out of scope of this dissertation, therefore the model in [12] is mechanised, but a full interrupt handling is not attempted.

Interrupts are defined as a bounded range of natural numbers.

\[
\begin{align*}
\text{minint, maxint} : \mathbb{N} \\
\langle\langle \text{disabled rule lRangeIntNo} \rangle\rangle \text{ minint} < \text{maxint}
\end{align*}
\]

\[
\text{IntNo} == \text{minint} . . \text{maxint}
\]

The strict “less than” relationship \( \text{minint} < \text{maxint} \) must be preserved, because the model must accommodate at least 2 distinct interrupt numbers: kill (kernel halt) interrupt and context switch interrupt.

Kill (\( \text{killintno} \)) and context switch (\( \text{ctxtswintno} \)) interrupts are defined axiomatically as valid distinct interrupt numbers. A consistency check is necessary to verify that it is possible to have 2 distinct interrupts. This is guaranteed by the above definition of IntNo range.

\[
\begin{align*}
\text{killintno} : \text{IntNo} \\
\text{ctxtswintno} : \text{IntNo}
\end{align*}
\]

\[
\langle\langle \text{rule lDistinctIntNo} \rangle\rangle \sim \text{killintno} = \text{ctxtswintno}
\]

Interrupt usage is further analysed in the following sections.

5.8 Hardware

The hardware (IA32 architecture processor) is modelled in \( HW \) schema with corresponding operations to raise interrupts within processor. Within the scope of dissertation, the hardware state schema \( HW \) is simplified to contain only the current interrupt.

\[
\begin{align*}
\text{HW} \\
\text{intno} : \text{IntNo}
\end{align*}
\]

Remark 5.2. The decision to store interrupt value in a hardware state can be justified as a modelling simplification. In x86 architecture, an interrupt is raised in the processor and it is immediately handled, while the running process is interrupted. Then an interrupt handler is located in the interrupt descriptor table (IDT) and executed [31].

Such algorithm does not require storage of the last called interrupt, as the raised interrupt must be immediately handled. As it is demonstrated in interrupt operation \( \text{RaiseInterrupt} \) later, the previous state is disregarded and the new interrupt value is set.

This affects the initial state of hardware schema \( HWInit \). In the same manner when previous interrupt value is disregarded upon raising a new interrupt, the initial interrupt value is also not important. This means that no invariants are necessary on the initial state and the initial interrupt value can be allocated non-deterministically.
 HWInit

HW'

The initialisation theorem (Section 4.3.3) for \( HWInit \) is proved by showing that we can instantiate \( intno \) value with some interrupt value, e.g. \( minint \), thus creating a valid \( HW \) state.

The operation to cause an interrupt is modelled as simply setting the interrupt value to be the one indicated in input variable \( ino \).

\[ \text{RaiseInterrupt} \equiv [\Delta HW; \ ino? : IntNo \mid intno' = ino?] \]

In order to model a proper interrupt handling operation, it should select an appropriate ISR for the input interrupt and executes the associated operation. However, it is out of scope of this dissertation.

5.9 Context switch

The context switch is performed to switch registers between TSSs when an executing process is changed. The context switch is initiated from the kernel by raising a \( ctxtswintno \) interrupt. This operation is defined by reusing \( \text{RaiseInterrupt} \) operation.

\[ \text{CtxtSw} \equiv [\text{RaiseInterrupt} \mid ino? = ctxtswintno] \setminus (ino?) \]

Such definition style is suggested by Barden et al. [3] for reusing operations with inputs. The input variable is assigned a concrete value and then hidden (existentially quantified) to be inaccessible from outside when this operation is reused. This style is easier to read and better represents the intent. Nevertheless, it is just a change of style and the equivalence to the original style in [12] is easily proved.

The interrupt handler (ISR) for \( ctxtswintno \) interrupt number performs the actual context switch. It is assumed that the context switch is done using IA32 hardware instruction \( \text{JMP} \) (jump). The \( \text{JMP} \) instruction performs unconditional one-way program control transfer to the indicated destination instruction (specified by instruction address) [31].

A sketch of this operation is provided as \( \text{ContextSwitch} \), however it is not syntactically correct and therefore is not included in the specification.

\[ \text{ContextSwitch} \]

\[ \Delta HW; \ \Xi \text{PTab} \]

\[ \text{outproc}? : \text{PID} \]

\[ \text{JMP} \ tss(\text{outproc}?) \]

5.10 Error handling

When the conditions for kernel operations to succeed are not satisfied, an error must be generated. This section presents a model to report specific errors. Then a secure error handling is defined, which stops the kernel in case of error.

5.10.1 Error types

Error types are defined as an enumerated free type \( \text{SYSERR} \), in a way as suggested in various sources [67, 76].

\[ \text{SYSERR} ::= \text{sysok} \mid \text{unusedpd} \mid \text{ptabfull} \mid \text{emptyqueue} \mid \text{alreadyqueued} \mid \text{notuserpid} \mid \text{notdevicepid} \]

\[ \mid \text{badpidcurr} \mid \text{badpididle} \mid \ldots \]

The number of possible errors is quite large, therefore only the ones within the scope of the dissertation are provided here. Please refer to the specification in Appendix E for the full error list.

With the continuing model mechanisation and verification, new errors are discovered to construct total operations. In addition to error cases in [12], \( \text{alreadyqueued} \), \( \text{notuserpid} \), \( \text{notdevicepid} \), \( \text{badpidcurr} \) and \( \text{badpididle} \) errors were encountered in process queue and scheduler specifications (Chapters 7 and 8, respectively).

The actual meanings of the errors will be examined when they are used later in the specification.
5.10.2 Error state

The error handling is modelled as in Errors with memory style [3]. A state variable is defined to store the error. A persistent error can be used to report about current error state, e.g. produce a kernel status report.

Furthermore, kernel status is necessary for operations inside the kernel. An operation may need to check if the kernel is still in valid (e.g. if previous operation did not fail). This is necessary to achieve atomicity of composite operations using early failure handling (Section 4.5.1).

The global variable for storing errors is defined as a error state schema ErrV (Error Variable), where serr stores the actual error value.

\[
\begin{align*}
\text{ErrV} \\
\text{serr : SYSERR}
\end{align*}
\]

The system is initialised in a valid state, namely sysok.

\[
\text{ErrVInit} \equiv \{ \text{ErrV} | \text{serr} = \text{sysok} \}
\]

The initialisation theorem for ErrVInit is discharged with a simple prove by reduce proof command, because all variables are constrained by the initialisation operation.

Basic operations to set and query the error value are provided, following the Delta/Xi pattern (Section 4.2.1).

\[
\begin{align*}
\text{SetSysErr} & \equiv \{ \Delta \text{ErrV}; \ e? : \text{SYSERR} | \text{serr} = e? \} \\
\text{SysErr} & \equiv \{ \Xi \text{ErrV}; \ e! : \text{SYSERR} | e! = \text{serr} \}
\end{align*}
\]

5.10.3 Valid system state

When a system is in a valid (“ok”) state, the error variable should carry the sysok value. Therefore, when an operation succeeds (in contrast to failure cases), it sets the error state to sysok, where appropriate.

As successful operations are frequent, an operation to set sysok is defined explicitly.

\[
\text{SysOk} \equiv \{ \text{SetSysErr} | \ e? = \text{sysok} \} \setminus (e?)
\]

Furthermore, for certain operations it is necessary to check that the system is in valid state in order to continue. A query to check for valid state is defined - the operation succeeds when the before-state system error is sysok.

\[
\text{IsSysOk} \equiv \{ \Xi \text{ErrV} | \text{serr} = \text{sysok} \}
\]

5.10.4 Security exit error handling

The separation kernel is modelled to be a secure kernel. The security is emphasised by having a paranoid error handling system, when the kernel is killed (halted) whenever any error occurs. This solution is also chosen because of its simplistic modelling [12].

The kernel is killed by raising a killintno interrupt. In general, this interrupt is raised to signal a fatal error. Just like with CtxtSw operation, a separate operation is defined to raise the kill interrupt.

\[
\text{RaiseKillInterrupt} \equiv \{ \text{RaiseInterrupt} | \ \text{ino}? = \text{killintno} \} \setminus (\text{ino}?)
\]

Throughout the original specification, Craig [12] employs the kill kernel error handling system by adding RaiseKillInterrupt operation to all operations, which signal error. Operation reuse is improved if an additional RaiseError operation is defined at the beginning. It joins two operations - setting an arbitrary error and raising killintno interrupt. This allows definition of the logic in a single place its reuse everywhere.

\[
\text{RaiseError} \equiv \text{SetSysErr} \land \text{RaiseKillInterrupt}
\]

To ensure homogenous interfaces for total operations (Section 4.2.2), an analogous operation needs to be defined to signal success.

\[
\text{RaiseOk} \equiv \{ \text{SysOk}; \ \Xi \text{HW} \}
\]

RaiseOk operation signals sysok state and keeps the interrupt unchanged. Otherwise, the interrupt would be undefined and when used with schema composition, interrupt propagation would fail.
5.11 Operation preconditions

For each operation the preconditions are calculated. A precondition proofs is completed to show that the precondition signature is valid for the operation (see Section 4.3.4). The preconditions are summarised in Table 5.2.

<table>
<thead>
<tr>
<th>Operation name</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetSysErr, SysOk</td>
<td>true</td>
</tr>
<tr>
<td>SysErr</td>
<td>true</td>
</tr>
<tr>
<td>IsSysOk</td>
<td>serr = sysok</td>
</tr>
<tr>
<td>RaiseInterrupt, RaiseKillInterrupt, CtxtSw</td>
<td>true</td>
</tr>
<tr>
<td>RaiseError, RaiseOk</td>
<td>true</td>
</tr>
</tbody>
</table>

An operation with true precondition can be executed for any given state. All state updating operations have true preconditions, therefore no additional checks are necessary to use them in other components.

In the specification, the precondition signatures are defined as schemas with Sig suffix, and the precondition proofs as theorems with Pre suffix. For example, operation SetSysErr precondition signature is SetSysErrSig and the precondition proof theorem is tSetSysErrPre.

5.12 Messaging

The messaging subsystem was declared out of scope of this dissertation. Nevertheless, it is necessary to define basic data types of the messaging system, because they are referenced in process table. Each process in separation kernel is associated with its individual message queue. The actual message is referenced in memory using a message pointer to the dedicated message buffer. The following messaging specification is based [12, Sec. 5.9].

The message pointer type, MPtr is defined axiomatically as a strict subset of memory address type, Addr.

\[ MPtr : \mathbb{P} \text{Addr} \]

\[ \lessdot \text{rule \ IMPtrDef} \lessdot MPtr \subset \text{Addr} \]

The message queue MsgQ stores messages in a sequence. The queue is limited to the maximum number of messages maxMs.

\[ \begin{align*}
\text{mq} : \text{seq} & \ MPtr \\
\text{maxMs} : \mathbb{N}_1 \\
\# \text{mq} & \leq \text{maxMs}
\end{align*} \]

When a process is allocated in the process table (see Section 6.9), its related information is also initialised. A message queue initialisation operation is used to allocate message queue for a new process. It is compatible with the Promotion Z design pattern (see Section 4.2.4) used in modelling process message queues.

A message queue can be initialised with a certain capacity, specified at creation time.

\[ \begin{align*}
\text{MsgQInit} \\
\text{MsgQ'} \\
\text{maxMs'} : \mathbb{N}_1 \\
\text{maxMs}' & = \text{maxMs} \\
\text{mq}' & = \langle \rangle
\end{align*} \]

An initialised message queue has no messages and is capable of carrying the indicated number of messages. Furthermore, the initialisation theorem proves that such message queue state can exist, which is easily discharged by indicating a possible message size, e.g. 1.

\[ \text{theorem \ tMsgQInit} \]

\[ \exists \text{MsgQ'}; \ \text{maxMs'} : \mathbb{N}_1 \bullet \text{MsgQInit} \]
5.13 Segment descriptor

The memory, used by a process, is referenced by code and stack segments. This information is also stored within process table and needs to be defined, even though memory management is out of scope of this dissertation.

Segments are fully defined by their start memory address (a value of type $Addr$) and a segment size (a natural number). A compound data type, which carries both values, is defined as a Cartesian product of $Addr$ and natural number set $\mathbb{N}$. It is used to reference process segments.

$$SDesc == Addr \times \mathbb{N}$$

Again, when a new user process is allocated, its code and data/stack segments need to be allocated. A certain constructor function, as defined in [12, Sec. 5.10], produces the $SDesc$ tuple out of indicated memory address and segment size.

$$mkSDesc : Addr \times \mathbb{N} \rightarrow SDesc$$

This function relates already allocated memory address and segment size. The actual allocation is handled by memory store in [12, Sec. 5.7, 5.10] and is out of this dissertation scope.

The labeled rule $lSDescFunDef$ enforces the construction algorithm as a simple combination of the indicated memory address and segment size.

5.13.1 Automation

The segment allocation plays an important part in the whole process allocation within process table. This results in constructor function being featured in several proofs. To benefit from automation of Z/Eves prover, several automation theorems need to be defined about $SDesc$.

One of the more challenging proofs is used in domain checks for when the constructor function is used. Even though the $mkSDesc$ is defined as a total function ($\rightarrow$), we need to prove that every address/size pair can be transformed to a $SDesc$ - in other words, is in the domain of the constructor function.

$$\forall a : Addr; \ s : \mathbb{N} \bullet mkSDesc(a, s) = (a, s)$$

The proof involves dissecting the $mkSDesc$ declaration as a total function and guiding the prover by reasoning about function domain and showing that the variables satisfy the total function definition.

5.14 Summary

This chapter has established formal models of supporting data types for process representation and hardware abstraction. Furthermore, kernel error handling system has been specified. With the associated information defined, we can model the main data structure for process information - the process table.
Chapter 6

Process Table

The processes in separation kernel are collected in a process table data structure. This structure contains full representation of processes it contains - the process list, process attributes and related data.

A process table, \( PTab \), is the central schema in the formal specification. Other components of the separation kernel (e.g. scheduler, messaging subsystem, etc.) reference and manage information in the process table.

Within the scope of this thesis, we model the process table itself and its main operations of process allocation and deletion. We analyse process table aspects related to process queues and scheduling, which we will examine with more detail in later chapters. However, the messaging subsystem, hardware-related information (memory access, segments, etc.) and full device modelling are out of scope of the dissertation and therefore these aspects of process table are analysed to a lesser extent.

In this chapter we examine the process table state schema, its evolution and interesting properties. Then we construct and analyse main operations regarding process allocation and deletion as well as information setting and querying. Finally we will examine preconditions for the operation success and their proofs.

6.1 State

First we present the improved version of process table state schema \( PTab \). The presented schema corrects all mistakes and redundancies in original [12, Sec. 5.4]. Moreover, additional predicates have been added to ensure several important process table properties. Full evolution of state schema from Craig’s original is demonstrated in Section 6.2.

The separation kernel uses 2 types of processes: user and device. The process table carries information about all processes in the system. Both process types share properties such as state, type and task state segments. However, additional information for each process type is required to be represented separately.

The “known” processes in separation kernel are recorded in \( used \) set within \( PTab \). Usually, when a process is referenced, it is required to exist within \( used \) processes set.

Moreover, the process table carries information about external identifiers for both user and device processes. The external identifiers are used to reference user processes and devices outside of the kernel, e.g. in user process execution. For storing user process external identifiers, \( UPID \), an inverse pair of functions \( extpid \) and \( pidext \) is used. The external device numbers \( DevNo \) are stored in \( devmap \) variable.

The \( PTab \) state schema contains 14 variables to record information about processes:

- \( used \) contains all “known” (used) process identifiers to the kernel. The \( free \) variable is used to denote remaining unused process identifiers.
- \( tss \) defines task state segments (TSS) for each process, which are used during context switch, to reference process registers.
- \( state \) indicates the current state of each process, e.g. running, ready, terminated, etc.
- \( ptype \) defines process type, either user (\( uproc \)) or device (\( dproc \)) processes.
- User process variables:
  - \( nextupid, extpid \) and \( pidext \) are related to generating and storing external user process identifiers. The variable \( nextupid \) denotes next available user identifier, while \( extpid \) and \( pidext \) are inverse functions between external and internal identifiers, \( UPID \) and \( PID \), respectively.
  - \( msgq \) links to the message queue of each user process.
  - \( cdseg \) and \( dsseg \) are used to reference locations in memory for each user process. \( cdseg \) has information about code segment location, while \( dsseg \) is used to indicate segment for process data and stack.
Device process variables:

- `devmap` is similar to `extpid` and defines the external identifier (device numbers) for each device process. The injective relationship (\(\mapsto\)) shows that each identifier is unique among devices.
- `devmsg` and `devrpy` are concerned with device messages. `devmsg` is used to store a request (message) from a user process to a device process, while `devrpy` stores the reply from device to user process.

\[
\begin{align*}
PTab & \\
nextupid & : \text{UPID} \\
\text{extpid} & : \text{UPID} \rightarrow \text{PID} \\
pidext & : \text{PID} \rightarrow \text{UPID} \\
\text{used, free} & : \mathbb{F} \rightarrow \text{PID} \\
tss & : \text{PID} \rightarrow \text{TSS} \\
devmap & : \text{DevNo} \mapsto \text{PID} \\
\text{state} & : \text{PID} \rightarrow \text{PSTATE} \\
\text{ptype} & : \text{PID} \rightarrow \text{PTYPE} \\
\text{msgq} & : \text{PID} \rightarrow \text{MsgQ} \\
devmsg & : \text{PID} \rightarrow (\text{GPID} \times \text{MSG}) \\
devrpy & : \text{PID} \rightarrow \text{MSG} \\
cdseg & : \text{PID} \rightarrow \text{SDesc} \\
dssseg & : \text{PID} \rightarrow \text{SDesc}
\end{align*}
\]

\[
\begin{align*}
\text{free} & = \text{PID} \setminus \text{used} \\
\text{used} & = \text{dom} \ \text{state} = \text{dom} \ \text{ptype} = \text{dom} \ \text{tss} \\
\forall \text{dprocs, upprocs} : \mathbb{F} \rightarrow \text{PID} \ |
\begin{align*}
\text{dprocs} & = \text{ptype}^{-1}(\{ \text{dproc} \}) \\
\text{upprocs} & = \text{ptype}^{-1}(\{ \text{uproc} \}) \\
\text{dprocs} & = \text{ran} \ \text{devmap} = \text{dom} \ \text{devmsg} = \text{dom} \ \text{devrpy} \\
\text{upprocs} & = \text{dom} \ \text{cdseg} = \text{dom} \ \text{dsseg} = \text{dom} \ \text{msgq} = \text{ran} \ \text{extpid}
\end{align*}
\end{align*}
\]

Schema invariants define `used` and `free` to partition available `PID`s set. Moreover, the common properties `state`, `ptype` and `tss` carry information of the used processes only.

By using the process type information in `ptype` we can define local sets of all device or all user processes (`dprocs` and `upprocs`, respectively). Then the process type specific variables can be restricted to processes of their respective type.

The `extpid` and `pidext` both share the mapping between external (UPID) and local (PID) user process identifiers. The variables allow inverse querying for process identifier value, either `PID` for `UPID` in `extpid` or `UPID` for `PID` in `pidext`.

Finally, when modelling the allocation of user processes within separation kernel, Craig enforces an algorithm of increasing natural number value for external identifier. To ensure that the algorithm is valid, we must make sure that certain values (\(u \geq \text{nextupid}\)) are available for allocation. For more discussion of this aspect of `PTab`, see Section 6.8.

6.2 State evolution

The original `PTab` schema contains syntax and structuring errors, redundant and missing statements. The schema syntax can be easily simplified for the benefit of both the reader and the person modelling the specification.

We remove redundant predicates or simplify them into better suiting mathematical constructs. This results in a specification with clear intents of predicates. Moreover, we add additional predicates to strengthen the state and avoid errors because of loose specification.

6.2.1 Evolution chain

We correct the original process table schema as `PTabOriginal`. Then we update and refine it in small steps to prove equivalence or refinement between changed schemas. A proved equivalence between schemas shows that the change
did not cause any logical changes. When equivalence is not possible, we prove that the new schema refines the previous one. That is the old schema can be implied from the new one.

For all changes, we provide justification of why it is done. We will show the evolution in 6 steps (with original corrections), culminating in the presented \(PTab\) state. The full \(PTab\) evolution can be shown as:

\[PTab \Rightarrow PTabv4 \Leftrightarrow PTabv3 \Leftrightarrow PTabv2 \Rightarrow PTabv1 \Leftrightarrow PTabOriginal\]

This chain shows that most transformations produced an equivalent version of the schema. Next we will explain each step highlighting the important parts of the schema. For full schema definitions, please refer to the specification in Appendix E.

### 6.2.2 Initial syntax and structure corrections

First of all, we correct syntax mistakes, such as variable name inconsistency (\(denvs\) and \(dprocs\)). Moreover, the predicates ran \(extpid = uprocs\) and dom \(pidext = uprocs\) were specified outside \(\exists dprocs, uprocs\ldots\) predicate originally. This is a syntax error since name \(uprocs\) is not defined outside the quantifier.

Also, \(free\) was defined outside the schema as a remark, yet it needs to be specified in \(PTab\) as it references \(used\).

Predicates \(used = \text{dom} \ state\), \(used = \text{dom} \ ptype\) and \(used = \text{dom} \ tss\) are concerned with the whole \(used\) set. Originally they were defined as a part of \(\exists dprocs, uprocs\ldots\) predicate. This is not appropriate, as they do not depend on either \(dprocs\) or \(uprocs\). Furthermore, \(used = \text{dom} \ ptype\) is required outside the predicate to discharge the \(dprocs\) and \(uprocs\) definition domain check.

Finally, after thorough investigation of full separation kernel specification [12], we concluded that \(ddevrs\) variable in original specification is redundant. The variable is not used anywhere in remainder of the specification. The variable to possibly store device requests (trying to decode the name), it is very likely has been obsoleted by \(devmsg\) variable, which is also used when device request is made. Craig has made a number of similar organisation mistakes in the original specification and this is very likely to be one of them. Because of this, we chose to remove the \(ddevrs\) variable from \(PTab\) schema altogether.

The original process table schema with these corrections is shown as \(PTabOriginal\). Note that the layout is compacted to save space for nicer presentation.

\[
\begin{align*}
PTabOriginal \quad \text{nextupid : UPID} \\
\text{extpid : UPID} & \Rightarrow \text{PID}; \ \text{pidext : PID} \Rightarrow \text{UPID} \\
\text{used, free : F} & \ \text{PID} \\
\text{tss : PID} & \Rightarrow \text{TSS}; \ \text{state : PID} \Rightarrow \text{PSTATE}; \ \text{ptype : PID} \Rightarrow \text{PTYYPE}; \ \text{msgq : PID} \Rightarrow \text{MsgQ} \\
\text{devmap : \text{DevNo}} & \Rightarrow \text{PID}; \ \text{devmsg} \ := \ \text{PID} \Rightarrow \text{(GPIID \times MSG)}; \ \text{devrpy} \ := \ \text{PID} \Rightarrow \text{MSG} \\
\text{cmsgq} & \ := \ \text{PID} \Rightarrow \text{SDesc}; \ \text{dsseg} \ := \ \text{PID} \Rightarrow \text{SDesc}
\end{align*}
\]

\[
\begin{align*}
\text{free} = \text{PID} \ \setminus \ \text{used} \\
\text{used} = \ \text{dom} \ \text{state} = \ \text{dom} \ \text{ptype} = \ \text{dom} \ \text{tss} \\
\exists \ dprocs, \ uprocs : \ \text{F} \ \text{PID} \mid \\
\text{dprocs} = \ \{ \ p : \ \text{PID} \mid \ p \in \ \text{used} \ \land \ \text{ptype}(p) = \ \text{dproc} \} \ \land \\
\text{uprocs} = \ \{ \ p : \ \text{PID} \mid \ p \in \ \text{used} \ \land \ \text{ptype}(p) \neq \ \text{dproc} \} \ \bullet \\
\text{dprocs} = \ \text{ran} \ \text{devmap} = \ \text{dom} \ \text{devmsg} = \ \text{dom} \ \text{devrpy} \ \land \\
\text{uprocs} = \ \text{dom} \ \text{cmsgq} = \ \text{dom} \ \text{dsseg} = \ \text{dom} \ \text{msgq} = \ \text{dom} \ \text{pidext} = \ \text{ran} \ \text{extpid} \\
\text{pidext} = \text{extpid}^\sim \\
\forall \ d : \ \text{DevNo} \ \bullet \ d \in \ \text{dom} \ \text{devmap} \ \Rightarrow \ (\exists_1 \ p : \ \text{PID} \ \bullet \ p = \ \text{devmap}(d))
\end{align*}
\]

### 6.2.3 Redundant predicate about device numbers

Next we analyse the last predicate in \(PTabOriginal\), which was defined in original specification:

\[
\forall \ d : \ \text{DevNo} \ \bullet \ d \in \ \text{dom} \ \text{devmap} \ \Rightarrow \ (\exists_1 \ p : \ \text{PID} \ \bullet \ p = \ \text{devmap}(d))
\]

The predicate state that each element in the domain of \(devmap\) function has only one value. However, this is just the redefinition of a \textit{function} - there are no diverging relationships from an element in function domain.
To be completely sure about this, we formulate this claim as a theorem \( lDevmapPFunImpliesUnique \) and prove it by reasoning about function domain, range and function membership.

\[
\text{theorem} \quad \text{disabled rule} \ lDevmapPFunImpliesUnique \\
\text{devmap} \in \text{DevNo} \rightarrow \text{PID} \Rightarrow \\
(\forall d:\text{DevNo} | d \in \text{dom devmap} \bullet (\exists_1 p:\text{PID} \bullet p = \text{devmap}(d)))
\]

The theorem states that the definition of partial function \( (\text{DevNo} \rightarrow \text{PID}) \) implies the mentioned predicate.

We remove the predicate in next evolution process table \( PTabv1 \). Then, using the proved rule we show that this change produced an equivalent schema - the predicate was redundant.

\[
\text{theorem} \quad tPTabOriginalv1Equiv \\
PTabOriginal \iff PTabv1
\]

### 6.2.4 Additional device number restrictions

We think that Craig made a mistake in the original predicate about device numbers, which was removed as redundant. We assume that he actually wanted to indicate that each device process has a unique device number.

This intention can be deduced from the device number allocation in operations \( \text{InitDeviceNum} \) and \( \text{NewDeviceProcess} \) in the original specification. We can see there that the \( \text{devmap} \) variable is updated only in the cases when the device number \( d:\text{DevNo} | d \notin \text{dom devmap} \). Therefore we introduce a new predicate

\[
(\forall p:\text{PID} \bullet p \in \text{ran devmap} \Rightarrow (\exists_1 d:\text{DevNo} | d \in \text{dom devmap} \bullet p = \text{devmap}(d)))
\]

This predicate is very similar to the original one. Its intention is to enforce that for each process identifier in \( \text{devmap} \) range, only a single element in its domain exists. Basically, we indicate that each device process has a unique device number among all devices.

We consider such predicate to be the original Craig’s intention. Moreover, it is useful for reasoning about device numbers and enhances the specification, therefore we add it in \( PTabv2 \) schema.

In the following sketch of \( PTabv2 \) we show the predicate’s place, where it replaces the predicate, removed as redundant earlier.

\[
PTabv2 \\
\vdots \\
\vdots \\
(\forall p:\text{PID} \bullet p \in \text{ran devmap} \Rightarrow (\exists_1 d:\text{DevNo} | d \in \text{dom devmap} \bullet p = \text{devmap}(d)))
\]

The introduction of new predicate means that the new \( PTabv2 \) schema is not equivalent to the previous evolution \( PTabv1 \). However, we can prove implication \( PTabv2 \Rightarrow PTabv1 \) to show that the new schema is a refinement of the previous one.

\[
\text{theorem} \quad tPTabv1v2Refinement \\
PTabv2 \Rightarrow PTabv1
\]

### 6.2.5 Partial injection for device numbers

The predicate was introduced as a universal quantifier over device processes to show the similarity to similar (yet redundant) predicate in original Craig’s specification [12]. The predicate’s intent can be written in a simpler form as a partial injection \( (\rightarrow) \) function. The injective property ensures that for there are no converging relationships in the function, i.e. for each element in function range there is a single domain element.

We remove the predicate and change definition to \( \text{devmap} : \text{DevNo} \rightarrow* \text{PID} \) in \( PTabv3 \). Then we prove the equivalence between schemas.

Here we provide a sketch of \( PTabv3 \) to illustrate change to partial injection.
As we prove equivalence between schemas by splitting them into separate cases of inverse implications \(((PTabv2 ⇒ PTabv3) \land (PTabv3 ⇒ PTabv2))\), we prove the equivalence between our predicate and partial injection function in the same way.

First we formulate and prove that the predicate implies partial injection in \(lDevmapUniqueImpliesPInj\). Then we prove the implication the other way in \(lDevmapPInjImpliesUnique\).

\[\text{theorem} \ lDevmapUniqueImpliesPInj\]
\[
\begin{align*}
\text{devmap} &\in \text{DevNo} \mapsto \text{PID} \\
(\forall \ p : \text{PID} \bullet p \in \text{ran} \ \text{devmap} \Rightarrow (\exists_1 \ d : \text{DevNo} \ | \ d \in \text{dom} \ \text{devmap} \bullet p = \text{devmap}(d))) &\Rightarrow \\
\text{devmap} &\in \text{DevNo} \mapsto \text{↣} \text{PID}
\end{align*}
\]

\[\text{theorem} \ lDevmapPInjImpliesUnique\]
\[
\begin{align*}
\text{devmap} &\in \text{DevNo} \mapsto \text{↣} \text{PID} \\
(\forall \ p : \text{PID} \bullet p \in \text{ran} \ \text{devmap} \Rightarrow (\exists_1 \ d : \text{DevNo} \ | \ d \in \text{dom} \ \text{devmap} \bullet p = \text{devmap}(d))) &\Rightarrow
\end{align*}
\]

Using these implications, we prove the equivalence between schemas.

\[\text{theorem} \ tPTabv2v3Equiv\]
\[PTabv2 \Leftrightarrow PTabv3\]

### 6.2.6 Relational image instead of set comprehension

Next we address the user and device process sets. In the original specification, Craig defines user and device process sets using set comprehension, e.g.

\[
dprocs = \{ p : \text{PID} \ | \ p \in \text{used} \land \text{ptype}(p) = \text{dproc} \}
\]

Such definition is very cumbersome for proofs, because of the lack of rules in Z/Eves toolkit [58] to manipulate and reason about the set as a whole. The expressions within set definition are not exposed to outside of the set and existing rewriting rules are applied to individual members of such sets, not the whole set [46].

The set comprehension of \(dprocs\) and \(uprocs\) defines the sets as known \(\text{PID}\) values, which type is \(\text{dproc}\) and \(\neg \text{dproc}\), respectively. The same intent can be defined using relational image of inverse \(\text{ptype}\) function, for example

\[
dprocs = \text{ptype}^{-\bowtie}(|\{\text{type}\}||)\]

A relational image \(R[\ A \ ]\) of a relation \(R : X \leftrightarrow Y\), where and \(A\) is a subset of \(X\), is the set of all elements in \(Y\) to which some element of \(A\) is related [76].

In our case, we apply the relational image on inverse \(\text{ptype} : \text{PID} \leftrightarrow \text{PTYPE}\) function. So the inverse is \(\text{ptype}^{-\bowtie} : \text{PTYPE} \leftrightarrow \text{PID}\). This means we are looking for \(\text{PID}s\), which are related to a particular set of types. By using singleton \(\{\text{dproc}\}\) or \(\{\text{aproc}\}\) sets, we get all \(\text{PID}s\) of type \(\text{dproc}\) and \(\text{aproc}\), respectively.

Predicate \(\text{used} = \text{dom} \ \text{ptype}\) satisfies the \(p \in \text{used}\) predicate in set comprehension definition. Therefore, the relational image definition is equivalent to the set comprehension one. We formulate this as theorem \(lPTypeSetComprehension\) and prove it by reasoning at set member level.

\[\text{theorem} \ lPTypeSetComprehension\]
\[
\forall \\text{ptype} : \text{PID} \leftrightarrow \text{PTYPE}; \ \text{type} : \text{PTYPE} \bullet \\
\text{ptype}^{-\bowtie}(|\{\text{type}\}||) = \{ p : \text{PID} \ | \ p \in \text{dom} \ \text{ptype} \land \text{ptype}(p) = \text{type}\}
\]

We replace set comprehension with appropriate relational image expressions in \(PTabv4\).

Moreover, in the same \(PTab\) evolution we remove predicate \(\text{dom} \ \text{pidext} = \text{uprocs}\). This predicate is redundant, because it is guaranteed by the inversion \(\text{pidext} = \text{extpid}^{-\bowtie}\) and predicate ran \(\text{extpid} = \text{uprocs}\). This follows from the definition of inverse function, that domain of one function is the range of another.
Here we provide only a sketch of $PTabv_4$, showing only the important changed parts in this evolution. For full schema definitions, please refer to Appendix E.

\[\begin{align*}
\text{used, free} & : F \ \text{PID} \\
\text{ptype} & : \text{PID} \rightarrow \text{PTYPE} \\
\end{align*}\]

Before we prove the equivalence of changed schema to the previous one, we need to address another issue. When defining user process set, Craig uses expression $\text{ptype}(p) \neq \text{dproc}$. As $\text{PTYPE}$ has only 2 values, $\text{dproc}$ and $\text{uproc}$, it is obvious that the type of $p$ is $\text{uproc}$. However, the prover cannot automatically assume that.

As $\text{PTYPE}$ could possibly have more than 2 values, $\text{ptype}(p) \neq \text{dproc}$ would mean that $\text{ptype}(p)$ can get any of the remaining values. We need to prove explicitly that the other remaining type is $\text{uproc}$. We formulate that as a theorem $\text{lPtypeNotDproc}$ and prove it using declarations of $\text{PTYPE}$ member elements.

\[\text{theorem } \text{lPtypeNotDproc}
\forall \text{ptype} : \text{PID} \rightarrow \text{PTYPE}; \; p : \text{PID} \mid p \in \text{dom} \; \text{ptype} \bullet
\text{ptype}(p) \neq \text{dproc} \Leftrightarrow \text{ptype}(p) = \text{uproc}\]

Now we can easily prove a rewrite rule that a set of non-device processes is actually a set of user processes. Again, we have to reason at set member level.

\[\text{theorem disabled rule } \text{lPtypeNotDprocSetEqual}
\forall \text{ptype} : \text{PID} \rightarrow \text{PTYPE} \bullet
\{p : \text{PID} \mid p \in \text{dom} \; \text{ptype} \land \text{ptype}(p) \neq \text{dproc}\} =
\{p : \text{PID} \mid p \in \text{dom} \; \text{ptype} \land \text{ptype}(p) = \text{uproc}\}\]

Using the defined lemmas and rules, we prove the equivalence between this schema and the previous one.

\[\text{theorem } \text{tPTabv3v4Equiv}
\begin{align*}
\text{PTabv3} & \Leftrightarrow \text{PTabv4} \\
\end{align*}\]

### 6.2.7 Additional restrictions on UPID generation

The process table is used to store and generate external identifiers for user processes. In the original specification Craig models the operations in a way that defines a concrete way of generating new UPIDs [12]. For each new identifier, the value in $\text{nextupid}$ is used, which is then increased for next allocation.

We add a predicate to require that $\text{nextupid}$ and subsequent identifiers are available for allocation.

\[\forall u : \text{UPID} \mid u \geq \text{nextupid} \bullet u \notin \text{dom} \; \text{extpid}\]

Such constraint allows us to avoid unnecessary and unwarranted precondition checks in $\text{PTab}$ operations. The predicate necessity is shown in Section 6.8. However, this predicate is only necessary because of the abstraction decision done by Craig.

We add the predicate to the next evolution version of process table. Since this is the final evolution, we name the state schema $\text{PTab}$, which name will be used afterwards in the operations.
Here we provide a sketch of $PTab$ schema to indicate the added predicate. This final $PTab$ version is the one presented in Section 6.1.

$$
\begin{align*}
&\text{PTab} \\
&\vdots \\
&\vdots \\
&\forall u : UPID \mid u \geq nextupid \land u \notin \text{dom extpid}
\end{align*}
$$

As this change adds new predicates, we cannot prove it to be equivalent to previous schema. Instead, we show that this schema is a refinement of the previous one.

\begin{itemize}
\item \textbf{Theorem} tPTabv4v5Refinement
\item $PTab \Rightarrow PTabv4$
\end{itemize}

### 6.3 Interesting properties

With the process table state specified, we can explore its interesting properties. Specifying such properties and proving them gives us good insight into the process table and its components.

#### 6.3.1 Injective $extpid$ and $pidext$ functions

Because of inverse relationship between $extpid$ and $pidext$ functions, we can prove that they injective. Thus we are guaranteed that each user process has a single external identifier and vice-versa. We show it by proving that $extpid$ and $pidext$ are partial injections ($\mapsto\Rightarrow$).

\begin{itemize}
\item \textbf{Theorem} tPTabExtpidPinj
\item $\forall PTab \Rightarrow extpid \in UPID \mapsto\Rightarrow PID \land pidext \in PID \mapsto\Rightarrow UPID$
\end{itemize}

#### 6.3.2 Disjoint user and device processes

We can prove that user and device process sets are disjoint within $PTab$ - no process can be both user and device process. To formulate that we use $dprocs$ and $uprocs$ as relational images (from $PTab$ definition).

\begin{itemize}
\item \textbf{Theorem} tPTabDprocsUprocsDisjoint
\item $\forall PTab \Rightarrow \text{disjoint} \langle ptype \sim \{\{dproc\} \cup \{uprocs\}\} \rangle$
\end{itemize}

#### 6.3.3 Next $UPID$ is available

As a part of external user process identifier $UPID$ generation, we can show that the variable $nextupid$ always carries a value valid for allocation and thus can be used for new processes.

The theorem is formulated as a forward rule, to aid in the automation of proofs.

\begin{itemize}
\item \textbf{Theorem} disabled frule fPTabNextupidUnused
\item $PTab \Rightarrow \neg nextupid \in \text{dom extpid}$
\end{itemize}

### 6.4 Initial state

Building on the defined $PTab$ state, we specify the initial process table state as well as a number of operations to manipulate variables stored in $PTab$.

The process table is initialised with no process information - all processes and their information are allocated afterwards using $PTab$ operations. The initial state is established using $PTabInit$:

$$
\text{PTabInit} \triangleq [ PTab' \mid \text{used}' = \emptyset \land nextupid' = 1 ]
$$

The $used$ set is the domain of all functions carrying process information in $PTab$. Therefore, specifying it to be empty in turn ensures that all other functions are also empty (to satisfy schema state predicates). Moreover, we initialise the available external user identifier $nextupid$ to be 1, which will be afterwards increased as processes are allocated.

To show that $PTabInit$ initialisation is valid and produces a consistent system, an initialisation theorem is proved.
6.5 Error cases

Before we define the operations to manipulate the process table, we examine errors encountered during the execution of the PTab operations. The error cases are reported using error identifier SYSERR and raising a kill interrupt. Here we reuse the RaiseError schema to provide a concise error definition, as discussed in Section 5.10.4. Moreover, we add Err prefix for error cases to follow the naming style indicated in Section 4.2.2.

RaiseErrUnusedPD error occurs when an invalid process identifier (PID) is referenced, e.g. when an indicated process identifier is not “known” (used) to the kernel.

\[ \text{RaiseErrUnusedPD} \equiv [\text{RaiseError} \mid e? = \text{unusedpd}] \setminus (e?) \]

RaiseErrPTabFull signals that all available process identifiers in the system are in use (full table).

\[ \text{RaiseErrPTabFull} \equiv [\text{RaiseError} \mid e? = \text{ptabfull}] \setminus (e?) \]

As we use the new style of specifying error signals, we prove equivalence between these schemas and the ones indicated in the original specification. Please refer to Appendix E for full specification and proofs.

6.5.1 Redundant pdinuse error

The original specification in [12] contains another error case - ErrPDInUse (with error value pdinuse). We think that this error case is carried from simple kernel specification [12, Sec. 3.3] without proper verification.

The error pdinuse denotes the state in which a process identifier is already in use. In simple kernel it is used during process identifier allocation.

The ErrPDInUse error case is not used anywhere in separation kernel. Moreover, we can show that it is redundant in simple kernel as well, as the process allocation preconditions are very similar in both kernels. This is analysed in more detail in Remark 6.3.

Because of these reasons, we chose to exclude ErrPDInUse in our specification.

6.6 Query operations

First we define various small operations to access information within process table. A query operation does not change the process table, which we indicate using ΞPTab (see more about Delta/Xi pattern in Section 4.2.1).

Since these operations are simple, we define them using horizontal schema notation, in order to save space.

\[ \text{UsedPID} \equiv [\text{ΞPTab}; p? : \text{PID} \mid p? \in \text{used}] \]

UsedPID determines whether the input process identifier p? is known to the kernel. Most operations require process identifier to be known, in order to reference it.

Operation GotFreePIDs is used to check whether there are available PIDs in the system. It is necessary for allocating new processes, as no new processes can be allocated once the process table is full.

\[ \text{GotFreePIDs} \equiv [\text{ΞPTab} \mid \text{used} \subset \text{PID}] \]

The proper subset (⊂) relationship in predicate used ⊂ PID means that not all PIDs are in used. To clarify this even more, we show that this predicate is equivalent to free ≠ ∅. This actually means what the operation name implies - there are free PIDs.

\[ \text{theorem tPTabGotFreePIDsEquiv} \]
\[ \forall \text{PTab} \bullet \text{used} \subset \text{PID} \iff \text{free} \neq \emptyset \]

We discharge this proof starting with definition free = PID \ used and reasoning about set difference and set membership.

User process external and internal identifiers are related using extpid and pidext functions. An operation for the UPID → PID translation is available:

\[ \text{PIDforUPID} \]
\[ \text{ΞPTab}; u? : \text{UPID}; p! : \text{PID} \]
\[ u? \in \text{dom extpid} \]
\[ p! = \text{extpid}(u?) \]
Z/Eves generates domain check proofs in schemas to ensure that functions are applied in their domain. In order to prove the domain check for PIDforUPID schema, we had to add a precondition \( u? \in \text{dom} \ extpid \). The original specification lacked it and domain check proof was impossible.

Such necessity of additional predicates for domain checks is one of the drawbacks of a verbose modelling style employed by Craig. These operations usually form a part of a bigger operation and are not used anywhere else (except the ones intended to be exposed as an interface). Thus additional predicates have to be defined to prove domain checks, instead of reasoning about variables in bigger operations (which may satisfy these constraints).

### 6.6.1 Process attributes

To query specific attributes of a process, we define operations, which output the attribute value for valid process identifier. Again, Craig missed the necessary precondition \( p? \in \text{used} \), which is required for domain checks.

Furthermore, we reuse operation \( \text{UsedPID} \), which defines the input variable \( p? \), does not change process table (\( \Xi \ PTab \)) and has the necessary check \( p? \in \text{used} \). The remaining part is to define output variable for the attribute value and how this value is accessed.

\[
\text{ProcType} \triangleq [\text{UsedPID}; \ pt! : \text{PTYPE} | pt! = \text{ptype}(p?)] \\
\text{ProcState} \triangleq [\text{UsedPID}; \ st! : \text{PSTATE} | st! = \text{state}(p?)]
\]

\( \text{ProcType} \) is used to query process type, while \( \text{ProcState} \) is used to access current process state. The value is presented in output variables \( pt! \) and \( st! \), respectively.

### 6.7 Simple process allocation operations

The processes in separation kernel are allocated using operations provided by the process table. Craig models the allocation by constructing simple operations, which change single variables, and then composing them in larger operations.

Such verbose modelling style is unnecessary long winded, especially when some of the smallest operations are not reused in multiple operations - it would have been simpler to just define logic directly in the larger operations.

Furthermore, most of the operations update just a part of process table, while leaving the remaining state variables in undefined state. In the final operation composition, all variables must be defined. This means that these operations cannot be exposed as \( PTab \) interface. Please refer to Section 4.2.3 for further discussion about such operations.

Nevertheless, we try to follow Craig’s style of modelling with full precondition checking of small operations. However, we will also provide “compact” versions of larger operations, which have all necessary predicates contained, instead of being composed of small operations.

#### 6.7.1 PID allocation

First of all we employ a non-deterministic process identifier \( PID \) allocation operation as it was defined in [12]. \( \text{AllocPID} \) allocates a new process identifier and outputs it using \( p! \).

\[
\begin{align*}
\text{AllocPID} & \triangleq \Pi \ PTab; \ p! : PID \\
\Pi & : \text{used} \\vdash \ p! \notin \text{used} \ \land \ \text{used}' = \text{used} \cup \{p!\}
\end{align*}
\]

The available process identifier pool is defined by \( PID \) data type. This means that there is a limit of total available identifiers. Moreover, as \( PIDs \) are allocated and then unallocated, they become available again - identifier cycling is employed (similar to index cycling in array-based implementations).

The \( \text{AllocPID} \) operation does not define which identifier is selected from free identifiers pool, therefore it is non-deterministic. It is only required that allocated \( PID \) was not used in the before-state, and is used in the after-state. This means that any suitable algorithm to determine next allocated \( PID \) can be chosen during the refinement and implementation of this operation and the algorithm is not important at current abstraction level.
6.7.2 UPID allocation

Craig specifies a concrete algorithm to allocate new external user process identifiers UPID. The external identifier is modelled as a natural number, thus a way to get unused UPID value is to increase it with every allocation. AllocUPID operation outputs a new UPID value for each operation call.

\[
\text{AllocUPID} \\
\Delta \text{PTab}; \ u! : \text{UPID} \\
\quad u! = \text{nextupid} \\
\quad \text{nextupid}' = \text{nextupid} + 1
\]

If this operation is analysed without its usage context in the original specification [12], we encounter several modelling problems. For example, we cannot be guaranteed that nextupid is actually unused. Thus in our specification we introduced additional predicates to ensure that nextupid and higher UPID values are not used by PTab state schema. Please refer to Section 6.8 for further justification.

6.7.3 UPID – PID mapping

Finally an operation to map the allocated UPID and PID values for a process is defined. AddProcUPID sets the external-internal identifier mapping in the process table. Identifier values are given as inputs for the operation.

\[
\text{AddProcUPID} \\
\Delta \text{PTab}; \ p? : \text{PID}; \ u? : \text{UPID} \\
\quad u? \notin \text{dom extpid} \\
\quad p? \notin \text{ran extpid} \\
\quad \text{extpid}' = \text{extpid} \cup \{u? \mapsto p?\}
\]

We require this operation to be used on unused UPID and PID values only, thus adding a new mapping to process table.

Originally Craig modelled this operation using relational override (⊕) function [12]. We present the original way in AddProcUPIDOriginal:

\[
\text{AddProcUPIDOriginal} \\
\Delta \text{PTab}; \ p? : \text{PID}; \ u? : \text{UPID} \\
\quad \text{extpid}' = \text{extpid} \oplus \{u? \mapsto p?\}
\]

Relational override is used to update an existing function value or add a new one, if the previous does not exist [76]. We consider its usage in this operation inappropriate. To show that, we analyse the various pairs of (u?, p?) values:

1. \(u? \notin \text{dom extpid}, p? \notin \text{ran extpid}\): when neither identifier is mapped in the process table (e.g. when new process identifier values are allocated), override adds the mapping to extpid as if the union (\(\cup\)) operation was used.

2. \((u?, p?) \in \text{extpid}\): such mapping already exists and the operation does change anything. In this case the operation is useless and thus having it defined is confusing.

3. \(p? \in \text{ran extpid}, (u?, p?) \notin \text{extpid}\): if process with \(p?\) already has an assigned UPID value, which is different from \(u?\), relational override produces a relationship when two different UPID values pointing to the same PID value. This means that extpid is no longer injective and thus PTab invariant is no longer valid.

4. \(u? \in \text{dom extpid}, p? \notin \text{ran extpid}\): if external user identifier \(u?\) is already used, but maps to a different PID than \(p?\). Relational override changes the UPID value to point to the new PID. However, to satisfy the PTab state invariant, the old PID value needs to be removed from used, and thus from all other associated functions. For example, to update the ptype function (remove previous pointed PID value and set newly added \(p?\) type to uproc), we need a cumbersome predicate:

\[
\text{ptype}' = (\text{extpid}(u?) \bowtie \text{ptype}) \cup \{(p? \mapsto \text{uproc})\}
\]

Basically from items 2 and 3 we deduce a predicate \(p? \notin \text{ran extpid}\). We add this predicate to AddProcUPID to avoid useless case of the operation and illegal partial injection. Also this predicate means that this operation is only used for new user processes, when \(p?\) is not known to the kernel.

51
From the analysis of item 4, we can conclude that updating an existing UPID is unnecessarily cumbersome. In general this means that we are reusing external process identifiers. Since we model UPID as natural numbers, there should be no need to reuse them, as natural number set is infinite.

If we did decide to reuse UPID values, a much clearer and better structured approach is to first deallocate (free) a process and then allocate a new one.

Because of the mentioned reasons, we also add predicate \( u_? \not\in \text{dom} \text{extpid} \) to AddProcUPID. Because of the predicates we can abandon relational override \((\oplus)\) operator in favor of a simple union \((\cup)\). Union operator is clearer and speaks better about operation intent - to add new mappings, instead of the “update” intent of relational override.

We prove that for new elements, relational override acts as a union using the following theorem:

**Theorem** disabled rule \( \text{lExtpidOplusIsCup} \)

\[
\forall PTab; \; p : \text{PID}; \; u : \text{UPID} \; | \; u \not\in \text{dom} \text{extpid} \bullet \\
\text{extpid} \oplus \{(u, p)\} = \text{extpid} \cup \{(u, p)\}
\]

Finally using \( \text{lExtpidOplusIsCup} \) we prove the equivalence between updated and original schemas, when both carry the mentioned predicates about \( p_? \) and \( u_? \):

**Theorem** \( t\text{AddProcUPIDEquiv} \)

\[
\text{AddProcUPID} \Leftrightarrow \text{AddProcUPIDOriginal}
\]

### 6.7.4 Using AddProcUPIDOriginal definition

The new definition of AddProcUPID is simpler, clearer and more suitable for its usage. We use it instead of the original one to provide a clear reusable component in PTab. Furthermore, such definition makes the precondition and other proofs simpler.

However, if for some reason we actually needed to use the original relational override definition, it is quite simple to adjust the proofs. With the current context of this operation, the relational override for both extpid and pidext can be easily transformed into union using \( \text{lExtpidOplusIsCup} \) and \( \text{lExtpidInvCup} \) rewrite rules.

To show that, we have proved the precondition calculation of AddProcUPIDOriginal in \( t\text{AddProcUPIDOriginalPre} \). Its proof involves transforming the relational override into union and then following the same path as the precondition proof for the new AddProcUPID operation. Please refer to Appendices E and F for the formal specification and proof scripts.

### 6.8 User process identifier allocation

We define the full user process identifier allocation operation as a conjunction of the defined individual allocation operations. This results in an operation, which allocates new internal and external identifiers and sets their mapping. In order to reduce the unnecessary verbosity, we introduce a simpler operation:

\[
\begin{align*}
\text{NewUPIDForProcess} & \quad \Delta \text{PTab}; \; p !_! : \text{PID}; \; u !_! : \text{UPID} \\
& | \; p !_! \not\in \text{used} \\
& | \; \text{used} \; = \; \text{used} \cup \{p !_! \} \\
& | \; u !_! \; = \; \text{nextupid} \\
& | \; \text{nextupid} \; = \; \text{nextupid} + 1 \\
& | \; \text{extpid} \; = \; \text{extpid} \cup \{u !_! \mapsto \text{p}_!\}
\end{align*}
\]

The definition of simple operations allowed us to explore the allocation algorithm and its preconditions. Furthermore, we did not want to clutter the definition with excess predicates, as some of the predicates are implied by others. To indicate that this was just a style change, we retain the original definition using Craig’s verbose style from [12] and show the equivalence of both schemas:

**Theorem** \( t\text{NewUPIDForProcessEquiv} \)

\[
\text{NewUPIDForProcess} \Leftrightarrow \text{NewUPIDForProcessOriginal}
\]

52
6.8.1 Process type implications

Such process identifier allocation can only be used for user processes. While *AllocPID* could be reused for both user and device processes, all information regarding *extpid* variable is related to the user processes only. Furthermore, we prove this claim using *tNewUPIDIsUser* theorem.

\[ \text{theorem } t\text{NewUPIDIsUser} \]
\[ \text{NewUPIDForProcess } \Rightarrow \ p\text{type'} } p!' = \text{ uproc} \]

This proof allows us to look ahead to the process allocation modelling. Originally, Craig defined process type input parameter for *AddIPD* operation, suggesting that a process type can be chosen upon allocation [12]. Following the proof that this operation can only be used for user processes, we remove the process type choice facilities in *AddIPD* and require that the allocated process has *uproc* type.

6.8.2 Predicates about *nextupid*

When defining *AddProcUPID*, we reasoned about the prohibition of *UPID* reuse (Sec. 6.7). Also, in the way that we are modelling process allocation, predicate \( u'? \notin \text{ dom } \text{extpid} \) (of *AddProcUPID*) must be satisfied in *NewUPIDForProcess*. In this case, it translates to \( \text{nextupid} \notin \text{ dom } \text{extpid} \).

Craig [12] took no actions to achieve that \( \text{nextupid} \) is not used for new allocations. In our specification, we model this as a predicate in \( \text{PTab} \):

\[ \forall u : \text{UPID} | u \geq \text{nextupid} \bullet u \notin \text{ dom extpid} \]

This ensures that all external identifiers starting with *nextupid* are available. Such large scope of *UPIDs* is required to ensure that the next identifier (\( \text{nextupid} + 1 \)) is also unused.

*Remark 6.1.* An alternative, non-deterministic approach to *UPID* allocation can be taken to produce a model of higher abstraction level. If *AllocUPID* allocated identifier non-deterministically, i.e. \( u! \notin \text{ dom extpid} \), the *nextupid* algorithm would not be necessary. The process table would be simplified and would not need additional predicates about *nextupid*.

Such approach is desirable for abstract models. It allows a different external identified allocation algorithm, if necessary, while not compromising the proofs and reasoning about kernel properties and security.

6.9 User process information allocation

We model user process allocation operation *AddPD* as a combination of 3 main tasks:

1. Allocate process identifiers (defined in *NewUPIDForProcess*, Section 6.8);
2. Initialise process information;
3. Handle allocation error cases.

In order to create a nice, reusable model with appropriate separation of concerns, we define smaller operations and then combine them into larger ones to create full operation. While doing that, we follow our guidelines as indicated in Section 4.2.3.

6.9.1 Shared attributes

First we update a reusable operation *AddPDesc*, indicated in [12], which sets the initial values of process attributes, shared between both process types. In this way, the same operation can afterwards be reused for device process allocation.

\[
\begin{align*}
\text{AddPDesc} \\
\Delta \text{PTab}; \ p'? : \text{PID} \\
tss'? : \text{TSS} \\
\text{state'} = \text{state} \cup \{p'? \mapsto \text{psready}\} \\
\text{tss'} = \text{tss} \cup \{p'? \mapsto \text{tss'}\}
\end{align*}
\]

Originally, Craig modelled this allowing a process state can be indicated upon allocation. Then this operation was used inconsequently, i.e. for idle process allocation only, with *psready* state (*AddIdleProcess* operation). For other processes their state was not defined initially at all [12].
This suggests that Craig assumed separation kernel scheduling operations (Sections 8.7 and 8.8) to be responsible for setting process states. However, the PTab state invariant requires a process state to be defined at all times. We have corrected the operation to always set the initial process state as psready, which seems natural.

Furthermore, we have added facilities in AddPDesc to set process task state segment (TSS) information. This information is common for both user and device processes and is used for context switching.

During our modelling, we try to separate data structure (as in PTab) and hardware-related (HW) concerns in our operation, where appropriate. Therefore, we leave the actual TSS allocation to be performed elsewhere and then be used as a parameter for process allocation.

### 6.9.2 Message queues

The full modelling of messaging subsystem of the separation kernel is out of scope of this dissertation. However, we need to initialise message queue for new user processes. This means that we need to model message queue initialisation within PTab. We use messaging operations from [12, Sec. 5.9] with the assumption that they are correctly modelled.

Craig [12, Sec. 5.9] suggests modelling message queue operations using promotion. Each operation is defined within scope of MsgQ and promoted to incorporate specific PID within PTab. Unfortunately, Craig fails to provide message queue initialisation and allocation in his original specification.

We add message queue initialisation and allocation using promotion as defined in [67]. To do that, we need to define a special promotion (framing) operation, used for the initialisation of local state (in our case, MsgQ).

\[
\begin{align*}
\text{PromotePTabMNew} & \quad \Delta \text{PTab} \\
& \quad \text{MsgQ}' \\
p? : \text{PID} \\
\text{msgq}' = \text{msgq} \cup \{p? \mapsto \theta \text{MsgQ}'\}
\end{align*}
\]

The promotion operation PromotePTabMNew adds a new message queue mapping for the indicated PID. Now we can use MsgQ initialisation operation, MsgQInit, with this promotion operation in order to define global message queue allocation operation.

\[
\text{NewUProcMsgQ} \equiv \exists \text{MsgQ}' \cdot \text{PromotePTabMNew} \land \text{MsgQInit}
\]

As only the MsgQ’ is existentially quantified (hidden), the input variables from MsgQInit are added to the interface of global operation NewUProcMsgQ. To illustrate that, we also provide an expanded version of the global operation.

\[
\begin{align*}
\text{NewUProcMsgQExpand} & \quad \Delta \text{PTab}; \ p? : \text{PID} \\
& \quad \text{maxMs}? : \mathbb{N}_1 \\
\text{msgq}' = \text{msgq} \cup \{p? \mapsto \theta \text{MsgQ}[\text{maxMs} := \text{maxMs}?, mq := \langle \rangle]\}
\end{align*}
\]

The operation instantiates a message queue with initial values and adds it to the indicated PID.

### 6.9.3 User process attributes

The remaining user-specific attributes are initialised with a separate operation, which includes shared attribute and message queue initialisation operations. The initial values for process type and code/data segments are set. The segment address/size information is passed as input and is used to configure specific process attributes.

\[
\begin{align*}
\text{NewUProcInfo} & \quad \text{AddPDesc} \\
\text{NewUProcMsgQ} & \quad \text{cdAddr}? : \text{Addr}; \ \text{cdSize}? : \mathbb{N} \\
& \quad \text{dsAddr}? : \text{Addr}; \ \text{dsSize}? : \mathbb{N} \\
\text{ptype}' = \text{ptype} \cup \{p? \mapsto \text{uproc}\} \\
\text{cdseg}' = \text{cdseg} \cup \{p? \mapsto \text{mkSDesc} (\text{cdAddr}?, \text{cdSize}?)\} \\
\text{dsseg}' = \text{dsseg} \cup \{p? \mapsto \text{mkSDesc} (\text{dsAddr}?, \text{dsSize}?)\} \\
\text{devmap}' = \text{devmap} \land \text{devmsg}' = \text{devmsg} \land \text{devrpy}' = \text{devrpy}
\end{align*}
\]

54
This operation is concerned with user process attributes, thus uproc process type is indicated.

Furthermore, process code and data/stack segments are allocated based on the corresponding input variables. Segment allocation using mkSDesc constructor function is taken as it is in SetCodeSegInfo and SetStackDataSegInfo in [12, Sec. 5.10]. Again, as it was with TSS allocation, we separate the hardware-related concern from PTab. The actual memory allocation is done elsewhere and only the allocated memory parameters, address and size, are used as process information within PTab.

We have chosen not to define operations SetCodeSegInfo and SetStackDataSegInfo separately, as they are only necessary for user process allocation and are not reused afterwards. Thus defining the actual allocation directly within a larger operation allows us to avoid unnecessary clutter in the specification.

Finally, we indicate that during user process allocation, device-related information does not change. We do not see a good case when device information should be changed in conjunction with allocating new user process. For example, even if two processes, user and device, would be allocated simultaneously, the operation to allocate common variables must be extended to consider two processes, which is unnecessary.

The resulting NewUProcInfo constrains all process attributes within PTab, making it a complete operation. In general, we could skip constraining some of the attributes and thus causing under-specification, if we considered it unnecessary for our abstraction level. However, the segment allocation is used in erroneous way further in the original specification [12]. We included it here to provide a correct way of modelling process allocation within the separation kernel. Please refer to Section 4.2.3 for more discussions regarding operation completeness and under-specification.

6.10 User process allocation

Full user process allocation combines the allocation of process identifier and initialisation of its attributes. Furthermore, it introduces error cases to handle illegal operation calls, when some preconditions for success are not satisfied.

6.10.1 Correct allocation

We combine the separate operations of process attribute allocation are into a single operation to define the “success” case of operation - when a process is allocated with all attributes initialised.

\[
\begin{align*}
& \text{AddPD0} \\
& \text{NewUPIDForProcess} \\
& \text{NewUProcInfo}[\!p!/\!p?] \\
& \text{used} \subseteq \text{PID}
\end{align*}
\]

We materialise the precondition that new processes can only be allocated when there are available PIDs. In the original specification, Craig employs GotFreePIDs operation to check this precondition in an expense of a schema composition (s) [12].

\[
\text{AddPD0Original} \equiv \text{GotFreePIDs} \sqsubseteq (\text{NewUPIDForProcess} \land \text{NewUProcInfo}[\!p!/\!p?])
\]

The corrected and adapted AddPD0Original is different from the actual one in original specification. However, we kept it to show a different modelling style - to reuse query operations with schema composition. To show that it’s only a difference in style, we have proved the equivalence between these operations.

Sequential composition introduces additional existential quantifier and variable mapping between composed schemas. This results in cluttered expressions within further proofs. Thus in this specification we strive to achieve simplicity and elegance in trying to avoid compositions where appropriate.

6.10.2 Full process table error case

An attempt to allocate a process when no process identifiers are available results in an error case. This error case is modelled as an operation, which does not change the process table but instead raises error ptabfull.

\[
\begin{align*}
& \text{ErrPTabFull} \\
& \Xi\text{PTab} \\
& \text{RaiseErrPTabFull} \\
& \neg \text{used} \subseteq \text{PID}
\end{align*}
\]

55
We indicate the preconditions when such error case occurs using predicate \( \neg \text{used} \subset \text{PID} \). Usually, when error cases are modelled, they represent the negation of “success” preconditions. Then, when joined using disjunction, “success” and error operations cover all possible conditions.

Craig [12] overlooks this requirement and does not specify concrete preconditions for error cases. That is a serious mistake, because the specification allows errors to be raised for “success” conditions (see Section 4.2.2).

6.10.3 Total operation and homogenous interfaces

Finally we can define the total operation of user process allocation, which encompasses all defined allocation and error cases. Furthermore, we add raising of \( \text{sysok} \) signal for the “success” case to indicate that the system is in good state. We join the execution cases using disjunction (see Section 4.2.2 for more information on this style).

\[
\text{AddPD} \equiv (\text{AddPD0} \land \text{RaiseOk}) \lor \text{ErrPTabFull}
\]

One of the issues with such definition of total operations is a possibility of non-homogenous interfaces. Interfaces are considered homogenous, when the same set of input and output variables is defined for all branches within operation (a branch is usually a disjoint predicate set or schema).

When we have non-homogenous interfaces, we cannot guarantee, for example, that a certain output variable exists. In this case, \( \text{ErrPTabFull} \) outputs neither of process identifiers \( p! \) and \( u! \). Non-homogeneity presents problems in schema expansions. For example, in \( \text{AddPDExpand} \) (we do not give its representation due to large size, please refer to Appendix E for the specification) we can only define the input and output variables for both branches, thus ensuring homogeneity for all cases. However, the equivalence proof cannot be done unless the non-homogenous \( \text{AddPD} \) has a common interface as well. We overcome this obstacle by defining an interface-only schema and joining it with \( \text{AddPD} \):

\[
\text{theorem tAddPDEquiv} \\
(\text{AddPD} \land \text{AddPDInterface}) \Leftrightarrow \text{AddPDExpand}
\]

Ensuring that all operations are homogenous can result in an over-specified model. For example, current \( \text{ErrPTabFull} \) definition can be easily reduced for device processes. If the operation had the interface of user processes, it would not apply for devices.

Furthermore, attempt to ensure homogeneity would produce operations that do not use input variables and provide arbitrary output values. When directly refined to code, it may be expressed as unnecessary parameters in method signatures for simple operations, which leads to difficult maintenance.

Finally, the desired interfaces can be achieved using refinement of interfaces [10]. Therefore, at the current abstraction level we chose to have non-homogenous interfaces for clearer modelling.

6.10.4 Process allocation properties

We have mechanised and proved the process allocation property indicated in [12, Th. 58].

\[
\text{theorem tAddPD0NotFree} \\
\text{AddPD0} \Rightarrow p! \notin \text{free'}
\]

This property shows that an allocated process identifier is no longer available (is not free). However, we had to adapt this property for “success” case only (\( \text{AddPD0} \)). If the process table is full, no new process is allocated and therefore \( p! \) is not defined (because of non-homogenous interface in \( \text{ErrPTabFull} \)).

Actually, the error case can be proved to be \text{true} as well, however it is only possible because when \( \text{used} = \text{PID} \), we get \( \text{free} = \text{PID} \setminus \text{used} = \text{PID} \setminus \text{PID} = \emptyset \). Now, whatever undefined value \( p! \) takes, predicate \( p! \notin \emptyset \) is always \text{true}. Still, we do not include this into our modelling, because it proves a different property than the one intended.

6.11 Idle process allocation

An idle process is used by the scheduler to occupy the processor when no other processes are running. This means that it has the lowest possible priority and usually does not perform any actions. An idle process can be either contained inside the kernel or be treated as a common user process, thus being visible outside the kernel, e.g. the System Idle Process in Windows NT family of operating systems. Furthermore, such process can be used do implement processor power saving.
Our formal model of separation kernel does not define the contents of idle process, thus we simply assume that it “does nothing”. We define the idle process allocation operations as an equivalent to user process allocation. This means that we treat the idle process as a user process.

\[ AddIdleProcess \equiv AddPD[ip]/p! \]

We redefine the user process allocation operation \( AddPD \) and change the output variable name, as suggested in [12]. \( AddIdleProcess \) operation is therefore exactly the same as \( AddPD \).

We will distinguish idle process later by using it in scheduler initialisation. That process will then be marked as having the lowest priority and used when no other process is running.

### 6.11.1 Original idle process model

We would also like to explore the attempt in [12] to model idle process as a “lightweight” process of arbitrary type. Craig argues that idle process does not have external process identifier and can be of any type.

Such claim is incorrect, because the state invariant requires all process attributes to be set. Therefore, all user processes must have external identifier \( UPID \) and all devices - device number \( DevNo \). Also, deducing from idle process creation in [12, Sec. 5.10], the idle process must be a user process, because it has code and data/stack segments (albeit empty) allocated outside the kernel address space.

In order to model a “lightweight” idle process without external identifier, message queues and other properties, we would need to define additional process type and its invariants in process table \( PTab \). This approach would unnecessarily burden the rest of the model. Therefore we ignore the original definition of \( AddIdleProcess \) and model idle process equivalent to user processes.

### 6.12 Process deletion

Terminated processes are deleted from the kernel. Deletion must deallocate process identifier and remove all information regarding the process from the process table \( PTab \).

In a complete model, other deallocation actions are likely to be performed as well, such as reclaiming allocated memory or updating device/user message queues to remove references to the terminated process. However, memory management and messaging subsystem are out of scope of this dissertation, therefore currently we focus on the deletion of process identifiers and directly related information.

#### 6.12.1 Attribute deletion

We define a universal operation to free process identifier and remove all related information. The same operation can be used for both device and user processes.

| FreePID |
| \( \Delta PTab \) |
| \( p? : PID \) |
| \( nextupid' = nextupid \) |
| \( used' = used \setminus \{p?\} \) |
| \( tss' = \{p?\} \leftarrow tss \) |
| \( state' = \{p?\} \leftarrow state \) |
| \( ptype' = \{p?\} \leftarrow ptype \) |
| \( extpid' = extpid \triangleright \{p?\} \) |
| \( msgq' = \{p?\} \leftarrow msgq \) |
| \( cdseg' = \{p?\} \leftarrow cdseg \) |
| \( dsseg' = \{p?\} \leftarrow dsseg \) |
| \( devmap' = devmap \triangleright \{p?\} \) |
| \( devmsg' = \{p?\} \leftarrow devmsg \) |
| \( devrpy' = \{p?\} \leftarrow devrpy \) |

The operation removes indicated process identifier \( p? \) from the process table, also deleting all related information from \( PTab \) variables. We use domain and range antirestriction (\( \leftarrow \) and \( \triangleright \), respectively) operations to ensure that the mapping with indicated \( p? \) is removed, while all remaining mappings stay the same. This means that we do not allow
attributes of other processes in $PTab$ change, when one process is removed. We also make sure that $nextupid$ value does not change during process deletion.

$FreePID$ is defined as a single operation for both process types. Domain/range antirestriction does not change the mappings if $p?$ does not belong to the variable domain/range, respectively. This means that when a user process is deleted, device-related variables are unaffected and vice versa.

A careful specification of constraints for all $PTab$ variables follows the guidelines in Section 4.2.3. This operation was erroneously under-specified originally in [12]. The original operation $DelUserPD$ puts constraints on $used$ and $extpid$ variables only. While state invariant guarantees that $p?$ is removed from all variables, Craig puts no restrictions on attribute values of other processes.

### 6.12.2 Total operation

The precondition of $FreePID$ is $true$, which means that the operation can be executed for all valid states of $PTab$ (see Section 6.14 for preconditions). This is caused by the nature of set difference and domain/range antirestriction operations - if process in question $p?$ is not know to the variable, it is unchanged. Thus, when a process is unknown to the kernel, its removal operation does not perform any changes.

While such behaviour is correct, it is not convenient from the modelling perspective. If an operation is executed in a way that it does not perform any changes, doubt is cast over the necessity and correctness of such execution. Therefore we want to separate the “success” case when a certain process is actually removed from the error case, when unknown process is being removed.

To achieve this error handling, we artificially introduce a predicate that allows process removal for known (used) processes only. Craig [12] also argues that such precondition is adequate. We define the known process deletion as a new operation, which reuses $FreePID$.

$DelPD0 \equiv [FreePID | p? \in used]$

Now we define an operation to handle the error case of unknown process. It does not change the process table $PTab$ and raises $unusedpd$ error.

\[
\begin{array}{ll}
\text{ErrUnusedPD} & \Xi_{PTab} \\
RaiseErrUnusedPD & p? : PID \\
p? \notin used
\end{array}
\]

This operation is also reused in further sections of the specification, when used $PIDs$ are referenced.

Finally, we define total operation $DelPD$ for process deletion, which has error case handling, in the usual manner.

\[
DelPD \equiv (DelPD0 \land RaiseOk) \lor ErrUnusedPD
\]

Note that even though Craig attempted to define total operation for process deletion in simple kernel specification [12, Sec. 3.3], he failed to do the same for the separation kernel.

### 6.12.3 Deletion of all processes

When a kernel is halted (killed) because of a serious error, all processes are terminated at once. Therefore we specify an operation to delete all processes thus clearing the process table, as defined in [12].

\[
DeleteAllProcesses \equiv |\Delta PTab | used' = \emptyset|
\]

The state invariant in $PTab$ ensures that when $used$ set is cleared, all other process variables are also cleared and we do not need to specify that explicitly. To show this, we investigate one case and show that this operation clears $extpid$ variable.

**Theorem** $tDeleteAllExtpid$

$DeleteAllProcesses \Rightarrow extpid' = \emptyset$
This implication is correct because of the domain/range relationships between used, uprocs and extpid in PTab. However, it is quite challenging to prove these relationships for empty set. We defined a general lemmas to show that empty range and domain implies empty relationship. Here we provide lemma for range relationships.

\[ \text{theorem gEmptyRan } [X, Y] \]
\[ \forall A : P \ X ; \ B : P \ Y \bullet \forall P : A \leftrightarrow B \cdot \text{ran}[X, Y] P = \{\} \Rightarrow P = \{\} \]

The proof of tDeleteAllExtpid involves examining members of the concerned sets, using lDomExtpidSubsetUsed lemma about the relationship between used and uproc and lEmptyRan to get from function range to statements about the whole function.

### 6.13 Process state change

A number of process attributes are initialised at process creation and are not changed afterwards. These include process type, external identifiers, task state, code and stack/data segments. Other variables contain information, which changes during process lifecycle.

Process state defines the execution state for a process and is managed by the separation kernel scheduler. When a process is waiting to be scheduled, it’s state is psready, when it is being executed - psrunning and so on.

#### 6.13.1 Generic state change operation

We provide operations to change process state as part of PTab interface. The process state is changed by the scheduler, which does not perform any further changes to PTab variables. Therefore we define the operation to change only process state and leave the remaining variables unchanged.

**Remark 6.2.** When an operation is defined, which we consider as being exposed to PTab users, we explore how it is going to be used, in a similar manner when software developers create public API.

If an operation is used as a part of larger operation, which involves simultaneous changes of PTab variables, we define it without restrictions on remaining variables to allow joining multiple operations using conjunction. Otherwise, we localise the change done by the operation and constrain the remaining variables to be unchanged (we keep process table “stable”).

To indicate “everything else stays the same” intent for process state change, we use Delta/Xi : change part of the state Z design pattern [67]. Using schema hiding, we can employ \( \Xi \text{PTab} \) to constrain all variables except the hidden ones to remain the same. Schema hiding existentially quantifies the variables we intend to change thus allowing us to specify change constraints.

As Z/Eves does not allow schema expressions to be used as predicates, we indicate \( \Xi \text{PTab} \) with state variable hiding as a separate schema.

\[ \text{PTabChangeState } \equiv \Xi \text{PTab} \setminus (\text{state}) \]

Then we use it to construct the process state change operation.

\[ \begin{align*}
\text{SetProcState} & \equiv \Delta \text{PTab} \\
\text{PTabChangeState} & \\
p? : \text{PID}; \ st? : \text{PSTATE} & \\
\text{state}' = \text{state} \oplus \{p? \mapsto st?\}
\end{align*} \]

We define the rest of SetProcState operation as indicated in [12]. The relational override (\( \oplus \)) operation updates \( p? \) mapping to point to indicated process state \( st? \).

Note that the “everything else stays the same” condition implies that this operation can only be used for known \( \text{PID}s, \) ergo SetProcState has a precondition \( p? \in \text{used} \).

Originally, Craig argued that this precondition is “implied by the invariant” [12], however it was not true for his original specification, where other PTab variables were left unconstrained.
6.13.2 Specific state changes

For easier reusability, we define shortcut operations for certain process changes. We use the variable hiding style to indicate concrete process states in the same fashion as with error definitions.

\[
\text{SetStateToReady} \equiv [\text{SetProcState} \mid \text{st}? = \text{psready}] \setminus (\text{st}?)
\]

\[
\text{SetStateToRunning} \equiv [\text{SetProcState} \mid \text{st}? = \text{psrunning}] \setminus (\text{st}?)
\]

\[
\text{SetStateToTerminated} \equiv [\text{SetProcState} \mid \text{st}? = \text{psterm}] \setminus (\text{st}?)
\]

For all these operations we prove equivalence theorems with original definition style. Furthermore, we provide an expanded version of \(\text{SetStateToReady}\) for better operation overview. Please refer to Appendix E for full formal specification.

6.14 Operation preconditions

We calculate preconditions for all process table \(\text{PTab}\) operations and prove their precondition theorems (see Section 4.3.4). Several process allocation operations, which can be reused for both device and user processes have different preconditions depending on whether it is used in allocating device or user processes. We calculate and prove preconditions for both cases, in order to show that the operation can actually be reused.

First, we summarise the process table operation preconditions in Table 6.1 and then we will provide further analysis on the preconditions and their proofs.

<table>
<thead>
<tr>
<th>Operation name</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{RaiseErrUnusedPD}, \text{RaiseErrPTabFull})</td>
<td>true</td>
</tr>
<tr>
<td>(\text{UsedPID})</td>
<td>(p\ ? \in \text{used})</td>
</tr>
<tr>
<td>(\text{GotFreePIDs})</td>
<td>(\text{used} \subset \text{PID})</td>
</tr>
<tr>
<td>(\text{PIDforUPID})</td>
<td>(u\ ? \in \text{dom extpid})</td>
</tr>
<tr>
<td>(\text{ProcType, ProcState})</td>
<td>(p\ ? \in \text{used})</td>
</tr>
<tr>
<td>(\text{AllocPID (device)})</td>
<td>(\exists p : \text{PID} \cdot p \notin \text{used} \land \exists d : \text{DevNo} \cdot d \in \text{dom devmap})</td>
</tr>
<tr>
<td>(\text{AllocPID (user)})</td>
<td>(\exists p : \text{PID} \cdot p \notin \text{used})</td>
</tr>
<tr>
<td>(\text{AllocUPID})</td>
<td>true</td>
</tr>
<tr>
<td>(\text{AddProcUPID})</td>
<td>(p\ ? \notin \text{used}, u\ ? \geq \text{nextupid})</td>
</tr>
<tr>
<td>(\text{NewUPIDForProcess})</td>
<td>(\exists p : \text{PID} \cdot p \notin \text{used})</td>
</tr>
<tr>
<td>(\text{SetProcType (device)})</td>
<td>(p\ ? \notin \text{used} \land \exists d : \text{DevNo} \cdot d \in \text{dom devmap} \land pt? = \text{dproc})</td>
</tr>
<tr>
<td>(\text{SetProcType (user)})</td>
<td>(p\ ? \notin \text{used} \land pt? = \text{uproc})</td>
</tr>
<tr>
<td>(\text{AddPDesc (device)})</td>
<td>(p\ ? \notin \text{used} \land \exists d : \text{DevNo} \cdot d \in \text{dom devmap})</td>
</tr>
<tr>
<td>(\text{AddPDesc (user)})</td>
<td>(p\ ? \notin \text{used})</td>
</tr>
<tr>
<td>(\text{PromotePTabMNew}, \text{NewUProcMsgQ}, \text{NewUProcInfo})</td>
<td>(p\ ? \notin \text{used})</td>
</tr>
<tr>
<td>(\text{AddPD} 0)</td>
<td>(\text{used} \subset \text{PID})</td>
</tr>
<tr>
<td>(\text{ErrPTabFull})</td>
<td>(\neg \text{used} \subset \text{PID})</td>
</tr>
<tr>
<td>(\text{AddPD}, \text{AddIdleProcess})</td>
<td>true</td>
</tr>
<tr>
<td>(\text{FreePID})</td>
<td>true</td>
</tr>
<tr>
<td>(\text{DelPD} 0)</td>
<td>(p\ ? \in \text{used})</td>
</tr>
<tr>
<td>(\text{DelPD}, \text{DeleteAllProcesses})</td>
<td>true</td>
</tr>
<tr>
<td>(\text{SetProcState}, \text{SetStateToTerminated}, \text{SetStateToReady, SetStateToRunning})</td>
<td>(p\ ? \in \text{used})</td>
</tr>
</tbody>
</table>
Most of the preconditions are quite simple from engineering perspective, for example query and setter operations require referenced process identifier \( PID \) to be known to the kernel \( (p? \in used) \). This means that they will fail if used with an unknown \( PID \). However, some of the operations have quite tricky preconditions, which we will try to analyse.

6.14.1 User process allocation preconditions

A new \( PID \) for user processes is allocated non-deterministically. However, since we have a limited number of available identifiers, the allocation can be executed only if there exists at least one unused \( PID \). If this condition is satisfied, we can find a new identifier and allocate it. This precondition is expressed as \( \exists p : PID \cdot p \notin used \). Since \( PID \) is allocated for both user and device processes, this precondition is repeated for most of the remaining process allocation operations.

If the \( PID \) of the process that is being allocated is used as an input variable to operation \( p? \), we adapt the precondition to require that exactly that \( PID \) is not used \( (p? \notin used) \). Thus these preconditions indicate process identifier availability is necessary to successfully allocate a process.

6.14.2 Device allocation preconditions

A device is allocated with both internal \((PID)\) and external \((DevNo)\) identifiers. The available device numbers \( DevNo \) are limited, contrary to unlimited \( UPID \) numbers. This means that when a device process is allocated, we must make sure that both identifiers are available. The calculated precondition for shared operations, when used for device processes, e.g. \( AllocPID \) (device), states just that. Together with a precondition about available \( PIDs \), it is necessary that there exists at least one available device number: \( \exists d : DevNo \cdot \sim d \in dom \ devmap \).

6.14.3 Merged preconditions

Allocation operation \( AddPD0 \) is composed of \( NewUPIDForProcess \) operation and \( used \subseteq PID \) predicate (which comes from \( GotFreePIDs \) operation), among others (Section 6.10). When operations are joined, their preconditions are also joined as a common precondition of the larger operation. In this case, the precondition \( \exists p : PID \cdot p \notin used \) of \( NewUPIDForProcess \) is merged together with \( used \subseteq PID \) and only one is kept. We have proved that these preconditions are equivalent and therefore having them both is unwarranted.

\[
\text{theorem} \ lExistFreePIDs \\
\forall used : \mathbb{P} \ PID \cdot (\exists p : PID \mid p \notin used \cdot true) \iff used \subseteq PID
\]

Remark 6.3. The equivalence between \( \exists p : PID \cdot p \notin used \) and \( used \subseteq PID \) preconditions was not spotted originally in [12]. These specific parts of process allocation are common to both kernels, simple kernel and separation kernel.

The usual style of defining error cases for total operations is negating the preconditions of “success” case. Following that, Craig likely identified error cases for both of the mentioned preconditions, defining \( \text{pdinuse} \) and \( \text{ptabfull} \) errors, respectively.

As we have shown that these preconditions are equivalent, we can remove one of the error cases as redundant, because it is used to cover exactly the same conditions as the other one. We have chosen to remove \( \text{pdinuse} \), because \( \text{ptabfull} \) better represents the error case specifics.

6.14.4 Interface operations

The calculated preconditions indicate system state when an operation can be executed successfully. Most of these operations are used within other operations and are not accessible outside the kernel. However, in order to achieve a robust system, we must ensure that no partial operations are exposed outside the kernel.

The interface (top-level) operations of the separation kernel process table are constructed with error cases and therefore put no restrictions on the system. This is shown by the precondition \( \text{true} \) of \( AddPD \), \( AddIdleProcess \), \( DelPD \) and \( DeleteAllProcesses \). We consider these operations (marked in bold in Table 6.1) the interface of \( PTab \).

If (possibly) malicious user of \( PTab \) is limited to the access of these operations, the precondition \( \text{true} \) indicates that the operations can be executed successfully (as in a controlled way) for any \( PTab \) state. If the state is erroneous, an error will be raised but the state will not be compromised in general sense.

This means that the remaining operations must be kept internally and used only in a verified way, because operation execution in invalid state results in operation failure and non-determinism within the system.
6.14.5 Proving preconditions

When we calculate the operation precondition, we prove a theorem to show that it is correct. We show that for all
the indicated preconditions, there is a valid after-state, defined by the operation (see Section 4.3.4).

If an operation does not constrain all state variables, we instantiate the remaining variables to some valid values, thus
showing a possible after-state. The process allocation operations in PTab are joined into larger ones until all variables
are constrained. Thus each smaller operation actually constrains a small part of a larger constraint.

Realising this, we can define one possible after-state of user process allocation, which satisfies all these operations
and simplify the proof by reusing it. We formulate the reusable part of the proof as a separate lemma to reuse it
afterwards.

\[ \text{theorem } \text{lAllocPIDUproc} \]
\[ \forall P\text{Tab}; p : PID; t : TSS; st : PSTATE; u : UPID; \text{maxMs : } \mathbb{N}_1; \]
\[ \text{cdAddr : Addr; cdSize : } \mathbb{N}; \text{dsAddr : Addr; dsSize : } \mathbb{N}; \]
\[ p \notin \text{used} \land u \geq \text{nextupid} \rightarrow \]
\[ \text{PTab}_\text{used} := \text{used} \cup \{p\}, \text{free} := \text{free} \setminus \{p\}, \]
\[ \text{state} := \text{state} \cup \{(p, st)\}, \text{ptype} := \text{ptype} \cup \{(p, u\text{proc})\}, \]
\[ \text{nextupid} := 1 + u, \text{tss := tss} \cup \{(p, t)\}, \]
\[ \text{msgq := msgq} \cup \{(p, \theta \text{MsgQ}[mq := ()]\}\}, \]
\[ \text{extpid := extpid} \cup \{(a, p)\}, \text{pidext := pidext} \cup \{(p, u)\}, \]
\[ \text{cdseg := cdseg} \cup \{(p, (\text{cdAddr, cdSize}))\}, \text{dsseg := dsseg} \cup \{(p, (\text{dsAddr, dsSize}))\}\]

The defined after-state is the same as defined by the final AddPD0 operation. Using this proof, the majority of
precondition proofs is simplified to instantiating corresponding variables and reusing lAllocPIDUproc lemma to finish
the precondition proof.

The proof of lAllocPIDUproc itself is more challenging. We have to show that the defined set union (∪) satisfies PTab
state invariant. We achieve that by using Z/Eves toolkit to reason about union within partial injection functions.
Furthermore, we define additional rewrite rules ITypeCupImageDifferent and ITypeCupImageSame, which are used to
prove program behaviour within relational images.

The preconditions for device process allocation and process deletion are proved in a similar fashion. We define separate
lemmas, lAllocPIDDproc and lFreePID, accordingly, and reuse them where appropriate. The proof of device process
allocation lemma lAllocPIDDproc is similar to that of user processes.

The proof of process deletion lemma lFreePID requires to show that process removal using domain/range antirestric-
tions satisfies the PTab invariant. We define and use general lemma lElemNotInDomNrres, which states that element
does not belong to the set reduced with range antirestriction if it does not belong to the actual set in the first
place.

\[ \text{theorem } \text{disabled rule lElemNotInDomNrres } [X, Y] \]
\[ \forall P : X \rightarrow Y; R : P P \rightarrow \gamma x \in \text{dom } P \Rightarrow \gamma x \in \text{dom } (P \bowtie R) \]

Finally, proofs of total operations, which have disjoint execution branches, are simplified by reusing precondition proofs
of each execution branch. For example, when proving precondition of AddPD, we reuse precondition proofs of both
AddPD0 and ErrPTabFull.

6.15 Automation

The Z/Eves prover is powerful and can perform difficult automatic reasoning and proof of the theorems. However,
sometimes certain assumptions or rules need to be added to prover’s toolkit in order for him to carry the rest of the
proof automatically. For this reason, we define a number of theorems as assumption, forward or rewrite rules for the
process table PTab to help the prover reason about PTab and its components. Please refer to Section 4.6 for more
information about Z/Eves prover automation.

PTab schema contains a large number of variables and predicates, relating them to each other. When this amount of
information is expanded in a proof, it makes completing the proof significantly more difficult. To simplify the proofs,
we try to avoid expansion of PTab, therefore we define rules which expose the variables outside the schema such as

\[ \text{theorem } \text{frule lPTabDomPidext} \]
\[ \forall P\text{Tab} \rightarrow \text{dom pidext = ptype} \setminus \{(u\text{proc})\]
Most of these proofs are trivial and require only to expand $PTab$ definition and rewrite the expanded predicate. Furthermore, we split the proof into smaller pieces to help with the proof. Then we define the certain pieces as rules, which Z/Eves prover can use automatically. For example, we show that the process identifiers with external user $UPIDs$ are used within the kernel.

\[
\text{theorem disabled rule } \text{IDomExtpidSubsetUsed}
\forall PTab \bullet \text{ran extpid} \subseteq \text{used}
\]

To prove this rule, we need to prove the relationship between these sets by reasoning about powersets, relational images and domain members. However, when this proof is complete, we can easily reuse its results in other proofs.

6.16 Summary

Departing from the formal methods view of process table, we can evaluate the created process table model as a data structure. We will try to give brief process table analysis from an engineering perspective.

An implementation of the defined formal process table model benefits from the various invariants about its state. We can be sure that this data structure contains only used (known) processes and each of them is always fully defined by its properties. The two supported process types are distinct and associated variables carry only the information of processes of the respective type. The external identifiers of both user and device processes are unique among used processes.

These properties are verified and give confidence in the data structure when it is used. Furthermore, we define robust interface operations, which implementation benefits from the formal specification. We guarantee that process allocation and deletion does not affect other existing data within process table. The allocated process identifier is no longer available until freed, while external user process identifiers can be generated without limitations following a specific incremental algorithm.

The definition of the interface (top-level) operations guarantees that they can be executed in a controlled manner for any process table state. When the state does not allow successful execution, an error is raised, however the data within process table is preserved unchanged. This shows the robustness of the data structure and its operations.

Such process table data model is defined with reusability in mind. The defined top-level operations constrain all process table information, therefore they can be reused with nice separation of concerns in other components of separation kernel, such as the scheduler, or be part of the kernel’s interface exposed to user processes.

Formal modelling and verification of this process table data structure increases the trust in its implementations and allows to use them with confidence in both high-integrity (where trust and security is mandatory) and common systems, with the goal of creating dependable systems.

Next we continue separation kernel modelling with definition of process queue data structure, which stores process identifiers in a successive fashion.
A process queue data structure is used to store process identifiers in a sequential order. Being a quite basic data structure, it has a number of applications within operating system kernels. In separation kernel, a process queue is used to represent scheduling queues in the round-robin scheduler. Furthermore, it can be used by kernel semaphores to control interprocess communication, as they are employed in the simple kernel [12].

The process queue is modelled as a FIFO queue with 3 main operations:

- Head (query for the first \((head)\) element);
- Enqueue (append the element to the end of the queue);
- Dequeue (remove the element at the start of the queue).

Process queue is associated with an underlying process table, which is the source of queue’s elements (process identifiers). However, we model it with the aim that when queue operations are performed, the process information in process table is not affected.

The process queue in separation kernel [12, Sec. 5.5] is equivalent to its counterpart in the simple kernel [12, Sec. 3.4]. Their only difference is the underlying process table \(PTab\). This similarity allows us to benefit from research already done on mechanising and verifying process queue within simple kernel by Liu [44] and Freitas [21].

Freitas [22] updates the specification of a process queue within simple kernel. The specification is mechanised and necessary initialisation and precondition proofs are performed. The report provides full definitions and associated proofs for further use as a part of Verified Software Repository.

The available resources creates a great incentive to reuse the definitions and proofs for the process queue within separation kernel. In this chapter, we adapt the process queue \(PQueue\) together with its operation definitions and proofs from [22] to separation kernel. Also, we update the naming and layout style according to Section 4.4 to keep uniformity within the thesis.

As with the definition of all data types and structures, first we provide the process queue state schema and its initialisation operation. Then we model query, enqueue and dequeue operations for managing queue’s contents. Finally we examine the operation preconditions and their proofs.

### 7.1 State

The process queue \(PQueue\) is modelled as an injective sequence of process identifiers \(PID\). The injective sequence ensures that each element in the sequence is unique. This means that the same identifier cannot be enqueued twice. Furthermore, the process queue relies on the underlying process table \(PTab\) for information about available identifiers.

\[
PQueue \quad PTab \quad \text{proc} : \text{iseq} \quad \text{PID} \quad \text{ran proc} \subseteq \text{used}
\]

The process queue requires that only processes known (used) in the kernel are queued. This invariant improves the usage of process queue data type in other constructs, when the referenced processes must be known within the kernel.

Furthermore, Freitas [21] argues that the subset constraint \((\subseteq)\) allows modelling a more generic, reusable process queue data structure, as opposed to the original usage of proper subset \((\subset)\) in simple kernel [12, Sec. 3.4]. The requirement
that not all processes are enqueued can then be enforced for specific usages of process queue. We prove this property for the separation kernel scheduler in Section 8.2.

In the original specification, Craig [12] suggests to not bother with stronger \textit{PQueue} invariants and model it as a simple sequence without any restrictions. He argues that the properties imparted by injective sequence and relationship with \textit{PTab} are guaranteed by \textit{PQueue} usage and in general by the design of separation kernel. However, formal methods are used with the purpose of proving such properties mathematically instead of relying on verbal arguments. Furthermore, access to full definition, mechanisation and proof of the \textit{PQueue} for simple kernel in [21] allowed us to easily reuse the strong \textit{PQueue} model.

### 7.2 Initial state

The process queue is initialised without any elements [12]. There are no constraints put on the underlying \textit{PTab} initialisation as the process queue can be initialised for any valid state of \textit{PTab}.

\[
PQueueInit \equiv [\text{PQueue'} \mid \text{procs'} = \langle \rangle]
\]

Together, initialisation of \textit{PTab} and \textit{procs'} constrain all initial values of process queue and underlying process table. We also prove the initialisation theorem to show that \textit{PQueueInit} produces a valid system.

**Remark 7.1.** In order to produce a strongly defined model of \textit{PQueue}, Freitas [21] suggests including the process table initialisation \textit{PTabInit} within the process queue initialisation. While this is reasonable in the context of just \textit{PQueue}, it puts restrictions on how the queue can be reused afterwards.

During the initialisation of scheduler component (Section 8.3), it is required that the idle process already exists in the system. As the process queues are initialised together with the scheduler, inclusion of \textit{PTabInit} creates an inconsistency as a valid \textit{Sched} state cannot be produced. It is impossible to have an empty process table with an allocated idle process.

For this reason we define \textit{PQueueInit} operation that leaves the underlying \textit{PTab} unconstrained.

### 7.3 Error cases

We define two errors regarding the queue in the usual fashion. An empty queue prevents element dequeue and first element query operations from succeeding. This case is identified with \textit{emptyqueue} error. Furthermore, an element that is already queued cannot be again enqueued using enqueue operation. The \textit{alreadyqueued} error is defined in [21], though missing in the original specification.

\[
\text{RaiseErrEmptyQueue} \equiv [\text{RaiseError} \mid e? = \text{emptyqueue}] \setminus (e?)
\]

\[
\text{RaiseErrAlreadyQueued} \equiv [\text{RaiseError} \mid e? = \text{alreadyqueued}] \setminus (e?)
\]

The state invariant that all elements in the process queue must be used within \textit{PTab} requires all referenced \textit{PIDs} to be known. The error of unused identifiers has already been defined in \textit{PTab} specification (\textit{unusedpd}) and is reused in \textit{PQueue}.

### 7.4 Query operations

The process queue provides access to its contents using simple FIFO operations: query the head of the queue, enqueue at the end and dequeue from the start. Freitas [21] refines the original operations in [12, Sec. 3.4] to not modify the underlying process table. The original operation specifications put no restrictions on \textit{PTab}.

The additional constraint decreases coupling between \textit{PQueue} and \textit{PTab}, because operations on the queue do not affect the process information, and improves refinement to ANSI-C code [21]. Furthermore, Freitas [21] employs an elegant specification style and defines a reusable schema for queue operations (i.e. \textit{enqueue} and \textit{dequeue}).

\[
\begin{align*}
PQueueOp & \triangleq \text{PQueue}; \ p : \text{PID} \\
\end{align*}
\]

\[
\Xi PTab
\]

\textit{PQueueOp} defines a reusable process identifier variable and requires that the process table does not change (\textit{\Xi PTab}).
A simple query operation for the process queue is to check whether the queue is empty. This operation is used for querying, therefore it does not change the queue.

\[ \text{EmptyPQueue} \equiv [\exists \text{PQueue} \mid \text{procs} = []] \]

If the process queue is not empty, we can get the first element in the queue, called the head of the queue. Z defines operations head and tail to access the first element in the sequence and the set of remaining elements, respectively. Thus the following operation outputs the first element in the queue as \( p! \).

\[
\begin{align*}
\text{HeadPQueue0} & \quad \exists \text{PQueue}; \ p! : \text{PID} \\
\text{procs} & \neq [] \\
p! & = \text{head procs}
\end{align*}
\]

HeadPQueue0 operation has a precondition \( \text{procs} \neq [] \). To construct a robust operation, which can be executed for all states of \( \text{PQueue} \), we handle the opposite case as an error. The empty queue operation has already been defined as \( \text{EmptyPQueue} \) thus we reuse it in error construction and define the total query operation \( \text{HeadPQueue} \) in the usual fashion.

\[
\begin{align*}
\text{ErrEmptyQueue} & \equiv \text{EmptyPQueue} \land \text{RaiseErrEmptyQueue} \\
\text{HeadPQueue} & \equiv (\text{HeadPQueue0} \land \text{RaiseOk}) \lor \text{ErrEmptyQueue}
\end{align*}
\]

### 7.5 Enqueue operations

A process is enqueued by adding it to the end of the FIFO process queue. Freitas [21] reuses the \( \text{PQueueOp} \) operation to state the variables and keep the \( \text{PTab} \) unchanged.

\[
\begin{align*}
\text{EnqueuePQueue0} & \quad \text{PQueueOp}[p?/p] \\
\text{procs}' & = \text{procs} \uplus \langle p? \rangle
\end{align*}
\]

A successful operation concatenates input variable to the end of the existing sequence \( \text{procs} \). The state invariants of \( \text{PQueue} \) require that \( p? \) is known to \( \text{PTab} \) and not yet enqueued, which is calculated as EnqueuePQueue0 precondition (Section 7.7).

To satisfy the injective sequence invariant of \( \text{PQueue} \), the concatenated sequence must be injective - there must be no duplicate elements. With the precondition that \( p? \) is not yet enqueued, this is satisfied. This property of injective sequences is defined as a lemma and used in the consequent proofs [21].

**Theorem** rule IPQCatPIDIsInseqPID \([X]\)

\[
\forall A : \mathbb{P} \ X \cdot \forall s : \text{iseq} \ A; \ x : A \mid \neg x \in \text{ran} \ s \cdot s \uplus \langle x \rangle \in \text{iseq} \ A
\]

The lemma states that for the injective sequence concatenation to remain injective, the added element must not belong to the sequence already. While such definition produces a sequence without duplicates, this cannot be proved using Z/Eves due to limitations in the toolkit about sequence concatenation [21].

Next we define error operations to handle cases when preconditions of EnqueuePQueue0 are not satisfied. We have already defined the error operation for unused process identifiers \( \text{ErrUnusedPD} \) in \( \text{PTab} \), therefore we reuse it here with an additional requirement that \( \text{PQueue} \) does not change.

\[
\begin{align*}
\text{ErrUnusedPDPQueue} & \quad \text{ErrUnusedPD} \\
\text{ErrUnusedPD} & \quad \exists \text{PQueue}
\end{align*}
\]

If the input variable \( p? \) is already queued, we raise the alreadyqueued error.

\[
\begin{align*}
\text{ErrAlreadyQueued} & \quad \exists \text{PQueue} \\
\text{PQueueOp}[p?/p] & \quad \text{RaiseErrAlreadyQueued} \\
p? & \in \text{ran} \ \text{procs}
\end{align*}
\]
The total enqueue operation \( EnqueuePQueue \) is constructed by disjoining the “success” operation with both error cases.

\[
EnqueuePQueue \equiv (EnqueuePQueue_0 \land RaiseOk) \lor ErrUnusedPDPQueue \lor ErrAlreadyQueued
\]

### 7.6 Dequeue operations

Following the FIFO definition, dequeue operation removes the first (\( head \)) element from the queue and returns it via output variable \( p! \). As with the previous operations, the process table remains unchanged due to invariants in \( PQueueOp \) [21].

\[
\begin{align*}
\text{DequeuePQueue}_0 & \quad \text{PQueueOp}[p! / p] \\
\text{procs} & \neq \langle \rangle \\
p! & = \text{head } \text{procs} \\
\text{procs}' & = \text{tail } \text{procs}
\end{align*}
\]

The first element is removed and the queue is shortened by using \( \text{tail} \) operator. A \( \text{tail} \) of a sequence is a sequence of all elements starting from the second one. In order to apply \( \text{head} \) and \( \text{tail} \) operators, the sequence must be not empty - we cannot dequeue an element from an empty queue.

To construct a total dequeue operation \( \text{DequeuePQueue} \) the already defined error case \( \text{ErrEmptyQueue} \) for empty queue is reused.

\[
\text{DequeuePQueue} \equiv (\text{DequeuePQueue}_0 \land \text{RaiseOk}) \lor \text{ErrEmptyQueue}
\]

The definition of these operations completes the model of the process queue \( PQueue \).

### 7.7 Operation preconditions

In order to verify the conditions under which process queue operations succeed, the operation preconditions are calculated and precondition proofs are performed (see Section 4.3.4). We summarise the process queue operation preconditions in Table 7.1.

<table>
<thead>
<tr>
<th>Operation name</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{RaiseErrEmptyQueue}, \text{RaiseErrAlreadyQueued}, \text{PQueueOp} )</td>
<td>true</td>
</tr>
<tr>
<td>( \text{EmptyPQueue}, \text{ErrEmptyQueue} )</td>
<td>( \text{procs} = \langle \rangle )</td>
</tr>
<tr>
<td>( \text{HeadPQueue}_0, \text{DequeuePQueue}_0 )</td>
<td>( \text{procs} \neq \langle \rangle )</td>
</tr>
<tr>
<td>( \text{EnqueuePQueue}_0 )</td>
<td>( p? \in \text{used} \land p? \notin \text{ran } \text{procs} )</td>
</tr>
<tr>
<td>( \text{ErrUnusedPDPQueue} )</td>
<td>( p? \notin \text{used} )</td>
</tr>
<tr>
<td>( \text{ErrAlreadyQueued} )</td>
<td>( p? \in \text{ran } \text{procs} )</td>
</tr>
<tr>
<td>( \text{HeadPQueue}, \text{EnqueuePQueue}, \text{DequeuePQueue} )</td>
<td>true</td>
</tr>
</tbody>
</table>

The precondition \( \text{true} \) in for interface (top-level) operations \( \text{HeadPQueue} \), \( \text{EnqueuePQueue} \) and \( \text{DequeuePQueue} \) indicate that the \( PQueue \) interface is robust. The top-level process queue operations can be executed for any queue state.

Well structured \( PQueue \) specification and appropriate automation (Section 7.8) rules make the precondition proofs for \( PQueue \) operations very simple and well automated. For most cases, a guided \texttt{reduce/prove by reduce} command is enough to discharge the proof. Precondition proofs for total operations are performed by reusing precondition proofs of their component operations.
7.8 Automation

To achieve great automation and simplicity in PQueue operation precondition proofs, several rewrite rules are defined. They supplement the prover toolkit with properties about injective sequence head and tail operators, as well as sequence concatenation. These rules are automatically applied by Z/Eves prover when reasoning about sequences - the underlying structure of PQueue.

The property about injective sequence head element being of the same type as the whole sequence is quite obvious to state and prove, yet improves the automation of proofs substantially.

\[ \forall \text{procs} : \text{iseq PID} | \text{procs} \neq \langle \rangle \bullet \text{head procs} \in \text{PID} \]

The rule that the tail of injective sequence of PIDs is injective, is actually an instantiation of the same general lemma (Section 5.1) for PID data type. Sometimes the prover has troubles applying a generic rule and a tailored rule is necessary for successful automation.

\[ \forall \text{procs} : \text{iseq PID} | \text{procs} \neq \langle \rangle \bullet \text{tail procs} \in \text{iseq PID} \]

All PQueue automation rules and their proofs in this specification are directly reused from [21], because they are not related to PTab and thus are the same for both kernels.

This concludes the process queue definition. For the separation kernel needs, it is necessary to define a clone of the process queue in order to allow 2 process queues coexist within the scheduler.

7.9 Device queue

Separation kernel scheduler (Chapter 8) is a round-robin scheduler with a priority for device processes. This is modelled as two separate process queues - one for user processes and one for devices. Therefore the scheduler contains two instances of PQueue data structure to represent each queue.

In Z notation, variable name is used as its reference, thus two variables with the same name are considered to be the same variable. To differentiate between the queues, we need to rename one of them - in particular, the device queue [12, Sec. 5.6].

It is only necessary to rename the procs (sequence) variable, which is the only one defined by PQueue in addition to PTab. When both queues are used together in the scheduler, they will have different sequences but the same underlying process table. This results from the fact that PTab schema in both queues has the same variable names - thus is considered the same structure.

We define the device queue and its main operations by renaming procs variable in corresponding PQueue schemas to devs and giving these schemas new names [12, Sec. 5.6]. Where necessary, we need to rename the variable in both pre- and post- states.

\[ \text{DeviceQueue} \equiv \text{PQueue}[\text{devs}/\text{procs}] \]

\[ \text{DeviceQueueInit} \equiv \text{PQueueInit}[\text{devs'}/\text{procs}'] \]

\[ \text{EnqueueDeviceQueue} \equiv \text{EnqueuePQueue}[\text{devs}/\text{procs}, \text{devs'}/\text{procs}'] \]

\[ \text{DequeueDeviceQueue} \equiv \text{DequeuePQueue}[\text{devs}/\text{procs}, \text{devs'}/\text{procs}'] \]

For all these operations we prove the initialisation and precondition theorems. The renaming does not actually change anything within the schema and all reasoning and proofs hold true from PQueue. However we chose to add these theorems to indicate that this is a separate data structure. If the device queue is changed or redefined at some point in the future, the initialisation and precondition theorems need to be updated to verify the new data structure.

All DeviceQueue proofs are simple to complete because they reuse the corresponding proofs in PQueue with renamed variables.
7.10 Summary

Process queue is a simple data structure with strict invariants about its contents. From an engineering point of view, it is a data structure to store elements in a sequential fashion. The model enforces no-duplicates policy on the queue and provides basic FIFO queue management operations.

A FIFO queue is a common data structure in the operating system kernels as well as other different software systems. The relationship with the process table gives us confidence on the queue’s contents - we can be sure that the process identifiers it contains are valid.

The top-level query and update operations of the process queue are robust - if an operation is used when it cannot succeed, an error is raised but the queue (and thus the process table) information is not changed. Furthermore, the requirement that queue operations do not change process information in the process table allows us to construct data structures with lesser coupling.

A formal methods treatment for this data structure allows us to create a verified and reusable component for various different uses. The fact that the process queue model is reused from its definition for a simple kernel in [21] is a good reusability evidence. The effort required to adapt the process queue to a different process table is diminished to the level of cosmetic (specification style) and minor adjustments.

Having constructed process queues, we can move on the the formal modelling of round-robin scheduler for the separation kernel.
Chapter 8

Scheduler

In this chapter we provide a formal model of a separation kernel scheduler. It contains two process queues for user and device processes. A basic priority support is in place, with device processes having priority over user processes. When no device or user processes exist, an idle process is run.

The scheduler is non-preemptive and runs processes within each queue on a round-robin basis. Furthermore, operations to enqueue (ready) and suspend processes are defined.

Following the style of previous data structures, we provide the scheduler state schema and its properties first. Then we investigate initialisation and operations as well as calculate and prove the precondition theorems for each operation.

8.1 State

The scheduler contains two FIFO process queues, one for devices and one for user processes. The FIFO queues are employed to support round-robin scheduling - a new process goes to the end of the queue and moves towards the start when other processes are scheduled. The distinction between queues for devices and user processes allows a simple model that gives priority to devices.

Formal specification of the scheduler reuses the defined \( PQueue \) data structure (Section 7) for FIFO queues [12]. However, since both queues must co-exist in the scheduler schema, one of them must be renamed to avoid name clash. The definition of \( DeviceQueue \) (Section 7.9) is just a renaming of \( PQueue \) to define a new name for the internal sequence variable [12].

The \( PQueue \) data type (underlying sequence variable \( procs \)) and its operations are used to store user processes within the scheduler. Then the \( DeviceQueue \) (variable \( devs \)) holds device processes. Furthermore, scheduler contains references to currently running (\( curr \)) and idle (\( ipid \)) processes.

\[
\begin{align*}
\text{Sched} & \quad PQueue; \ DeviceQueue \\
& \quad curr, ipid : \text{PID} \\
& \quad queued : \mathbb{P} \ PID \\
& \quad \{ curr, ipid \} \subseteq used \\
& \quad \text{ran procs} \subseteq \text{ptype}\sim(\{ uproc \} ) \\
& \quad \text{ran devs} \subseteq \text{ptype}\sim(\{ dproc \} ) \\
& \quad queued = \text{ran procs} \cup \text{ran devs} \\
& \quad \text{curr} \notin queued \land \text{ipid} \notin queued
\end{align*}
\]

We add a number of predicates to constrain variables within the scheduler in regards to the properties of referenced processes.

8.1.1 Predicates about the scheduler

Original state schema of separation kernel defines no predicates to accompany defined variables [12, Sec. 5.6]. Many of scheduler’s properties are implied by Craig in the specification comments however they are not specified formally.

The lack of predicates about scheduler variables is a direct result of Craig’s reluctance towards a proper implementation of process queue \( PQueue \) [12, Sec. 5.5]. In the original specification, \( PQueue \) has no reference to a process table \( PTab \) and no predicates. As a result, scheduler does not have reference to \( PTab \) and no predicates about process properties can be formulated. The original specification is therefore under-specified and allows loose implementation as well as restricts reasoning about scheduler properties.
The updated process queue (Section 7) allows us to define predicates about processes within the scheduler. The PQueue state invariants require that all queued processes to be used (known) in kernel (Section 7.1). We expand on this concept and require that the current and idle processes in the kernel are known to the system as well.

Furthermore, we add an invariant that current and idle processes cannot be queued, i.e. a process that is currently running cannot be also queued for execution. When a process is run, it must be removed from a corresponding queue. For this invariant and subsequent scheduler properties we define a derived variable queued, which is a set representing all queued processes.

Finally we formally state that proc (PQueue) holds only user processes and devs (DeviceQueue) is for device processes. Without these invariants, a malicious operation may enqueue a user process into the device queue gaining a higher priority execution, which is not a desirable property.

The added predicates highly increase the quality of the specification. We do not need to draw attention to supporting comments about the model of the scheduler when defining new operations or reusing the formal model. Defined predicates are mathematical statements about certain properties of the system that form a state invariant, which must be met during all times of system execution.

The implementation of such constrained model boasts high confidence, as we can reason about it and prove its properties as well as restrain non-determinism and possibly malicious system states. We formulate and prove several interesting properties about the scheduler in Section 8.2.

The added invariants, while improving the Sched state, require various additions and changes to the original specification [12] when modelling scheduler operations. We need to introduce new error cases (Section 8.4), change operation invariants and so on.

8.1.2 Previously run process

In addition to the added invariants, we have done several design decisions when modelling the scheduler, which deviate from the original specification [12].

Remark 8.1. In the original specification Craig models a reference to previously run process. When a new process is scheduled, the previously running process reference is stored in variable prev using prev’ = curr predicate. A careful examination of the separation kernel specification in [12] shed no light into how variable is then used. In all instances, the variable is only set but never “read” (used in other predicates apart from the mentioned one).

The only use of a corresponding prev variable in the scheduler for simple kernel is to formulate a statement about process state within scheduler [12, Corollary 2] as

\[ \text{prev} \in \text{used} \lor \text{state(prev)} = \text{pterm}. \]

A similar statement is done about current process curr in [12, Corollary 1]. However, these statements are invalid in the context of PTab. He states that a process is either used or terminated, however when process is terminated and removed from used, its attribute information such as state is deleted as well, thus state(prev) value is undefined. Even more, the reasoning is wrong as prev would point to a valid used process if the process is suspended instead of terminated.

Finally, we did not find any real use for the previously run process reference in a general design of a round-robin scheduler. The scheduling is fully controlled by process queues and the currently running process.

As we strive to create good quality, reusable data structures for kernel design, we chose to remove redundant reference to previous process in the original specification [12].

8.1.3 Promotion pattern for scheduler

Remark 8.2. Craig attempts to model process queues within the scheduler using Promotion pattern (Section 4.2.4) [12]. His specification has a large number of syntax and type-checking mistakes throughout scheduler model because of such approach, as he applied the promotion erroneously.

Promotion is normally used when there is some identifier-based reference to a schema-type value, e.g. a mail system that contains a number of mailboxes, which are referenced using and identifier [76]. In Sched state, there are no identifiers and the queues are accessed directly in the schema.

For a correct usage of promotion in the original specification we need to define two specific promotion schemas, each for one process queue, because we cannot reference the queues based on some identifier. Furthermore, usage of promotion introduces unnecessary number of existential quantifiers in the proofs, thus putting a considerable burden on the already large proofs within Sched.
Even if we go such lengths as to correct the mistakes in using promotion as in the original specification, the application of technique is out of place and will raise questions among the users of this formal model.

Instead, we choose to take a simpler and more elegant approach by including process queues and their operations directly within the scheduler using schema conjunction (Sections 8.6 and 8.8).

The final scheduler state schema $Sched$ carries a large number of changes from the one indicated in the original specification [12]. Starting with the addition of $PTab$ reference to process queues, we add new $Sched$ state invariants, remove previous process reference and abandon promotion in favour of simple conjunction. All these changes mean that we can no longer prove neither equivalence (because of new additions), nor refinement (because of removed variable).

However, we closely follow the arguments, justification and formal specification of scheduler for separation kernel in [12] when constructing the new formal model. We believe that we translate all requirements and intentions into the new model while enhancing it along the way. We provide complete justification for all additions and changes.

8.2 Interesting properties

The added invariants to $Sched$ allow us to reason about scheduler properties. We formulate and prove them as theorems to show that they always hold.

While the use of injective sequence in $PQueue$ (Section 7.1) prevents duplicates within the queue, we show there are no duplicates in the scheduler as well. We prove that both queues are always separate and no process can be be enqueued in both queues.

\[ \text{theorem} \ disabled \ rule \ LSchedSeparateQueues \]
\[ \forall Sched . ran \ proc \cap ran \ dev = \emptyset \]

Next we investigate a possibility for deadlock in the kernel, when all processes become queued and no process is running. We prove that it is never the case that all known processes are queued (indicated by proper subset ($\subset$) relation).

\[ \text{theorem} \ tSchedQueuedSubsetUsed \]
\[ \forall Sched . queued \subset used \]

These properties give confidence about the execution of the scheduler. Having proper invariants and using formal methods, we can explore and ascertain certain properties about the model implementations.

8.3 Initial state

8.3.1 Scheduler initialisation

A scheduler is initialised with a known idle process and empty user/device queues. For the initialisation of process queues, we reuse respective initialisation operations $PQueueInit$ and $DeviceQueueInit$. Furthermore, the scheduler is initialised with a reference to the idle process, which is passed as an input variable $p$.

\[
\begin{array}{|c|}
\hline
SchedInit \\
Sched' \\
PQueueInit \\
DeviceQueueInit \\
p? : PID \\
\hline
ipid' = p? \\
curr' = ipid' \\
\hline
\end{array}
\]

The scheduler is initialised with the idle process as currently running one. This means that the scheduler can be initialised for a separation kernel that has at least 1 process (idle process).

We prove the initialisation theorem for this operation to show that it is possible to construct a valid $Sched$ state following these invariants.
8.3.2 Initially running process

Remark 8.3. The original specification defined Sched to initialise with minpid as the initially running process identifier [12]. This came from the assumption that the idle process is allocated first and therefore is assigned minpid identifier. Such definition is too restrictive. The processes at this level of separation kernel specification are allocated non-deterministically. It is impossible to guarantee that the idle process gets assigned minpid value - the actual algorithm to choose PIDs is left for refinement or implementation.

Without the ability to state that idle process p? = minpid, in general case, these identifiers can reference different processes. Then, to satisfy state invariants of Sched, it can only be initialised when PTab has at least two known processes, p? and minpid, unless p? = minpid. This is too restrictive as a single existing process should be enough to initialise scheduler - the idle process.

This is one of the mistakes arising from under-specification in the original model [12]. While various assumptions are done during modelling and only mentioned in comments, the lack of actual formal invariants in the model allow an implementation that does not follow the original intentions.

We correct the initialisation by stating that initially the idle process is running in the kernel, as it is expected when no other processes are queued for execution.

8.3.3 Initialisation with process table

To satisfy the state invariant of Sched, scheduler initialisation requires the idle process to be already allocated. This means that scheduler cannot be initialised in parallel to the process table, and must follow the the PTab initialisation (see Remark 7.1).

We explore the two-step definition of both process table and the scheduler. First the empty process table is initialised and then the idle process is allocated and used to initialise the scheduler.

We define the full initialisation as a new operation. Here we reuse the idle process allocation operation AddIdleProcess. Furthermore, dependency on existing error (ErrV) and hardware (HW) schemas is required.

\[
\begin{align*}
& PTabFullInit \equiv PTabInit \land ErrV' \land HW' \\
& SchedPTabInit \equiv PTabFullInit \land (AddIdleProcess \land SchedInit[ip!/?]) \setminus (ip!, u!)
\end{align*}
\]

Note that ErrV' and HW' are not restricted, because this initialisation requires allocated hardware-related information (memory segments) and therefore these schemas are likely to be initialised before or in parallel.

We provide an expanded version of SchedPTabInit to show that it is actually a complex initialisation operation (defining only post- variables) for process table, queues and the scheduler. The complexity arises from the need to satisfy Sched state invariant. Furthermore, additional hardware and other configuration information is required and is passed using input variables.

\[
\begin{align*}
& SchedPTabInitExpand \\
& Sched'; ErrV'; HW' \\
& tss? : TSS; maxMs? : N_1 \\
& cdAddr? : Addr; cdSize? : N \\
& dsAddr? : Addr; dsSize? : N \\
& \text{used}' = \{ipid'\} \\
& \text{nextupid}' = 2 \\
& \text{ezlpid}' = \{1 \mapsto \text{ipid}'\} \\
& \text{state}' = \{\text{ipid}' \mapsto psready\} \\
& tss' = \{\text{ipid}' \mapsto tss'\} \\
& ptype' = \{\text{ipid}' \mapsto uproc\} \\
& msgq' = \{\text{ipid}' \mapsto \theta MsgQ[maxMs := \text{maxMs}'?, mq := \emptyset]\} \\
& cdseg' = \{\text{ipid}' \mapsto (cdAddr?, cdSize?)\} \\
& dsseg' = \{\text{ipid}' \mapsto (dsAddr?, dsSize?)\} \\
& devmap' = \emptyset \land \text{devmsg}' = \emptyset \land \text{devrpy}' = \emptyset \\
& serr' = sysok \\
& curr' = \text{ipid}' \\
& procs' = \emptyset \land \text{devs}' = \emptyset
\end{align*}
\]

We verify the equivalence between expanded and short definitions of the full initialisation. Moreover, we prove the initialisation theorem for this operation to indicate that it produces a valid state of the separation kernel.
8.4 Error cases

The addition of new invariants allows the creation of secure and controlled operations. We can verify properties of scheduler state and input variables to ensure that operation is successful and robust.

The original specification does not define any errors because of the lack of invariants in \textit{Sched} (although some are mentioned in the associated comments) \cite{12}. We define new error types in Section 5.10 and construct operations to raise these errors in the usual fashion.

Errors \textit{notuserpid} and \textit{notdevicepid} are used when process of incorrect type is being added to a certain queue, e.g. an operation to enqueue a user process is executed with device process parameter.

\[
\text{RaiseErrNotUserPID} \triangleq [\text{RaiseError} | e? = \text{notuserpid}] \ (e?)
\]

\[
\text{RaiseErrNotDevicePID} \triangleq [\text{RaiseError} | e? = \text{notdevicepid}] \ (e?)
\]

Furthermore, errors \textit{badpidcurr} and \textit{badpididle} signal that the currently running or the idle processes, respectively, are being enqueued to a process queue.

\[
\text{RaiseErrBadPIDCurr} \triangleq [\text{RaiseError} | e? = \text{badpidcurr}] \ (e?)
\]

\[
\text{RaiseErrBadPIDIdle} \triangleq [\text{RaiseError} | e? = \text{badpididle}] \ (e?)
\]

These errors are used to define secure operations for processes to enter the scheduler (enqueue) in the following sections.

8.5 Query and update operations

Scheduler model provides query operations to reference idle and currently running processes \cite{12}. These operations do not change the scheduler information.

\[
\text{IdleProcess} \triangleq [\exists \text{Sched}; \ p! : \text{PID} | p! = \text{ipid}]
\]

\[
\text{RunningProcess} \triangleq [\exists \text{Sched}; \ p! : \text{PID} | p! = \text{curr}]
\]

Furthermore, an operation to update the currently running process to the indicated process is provided \cite{12}. This operation must be used with care and in special circumstances. For common use, scheduling operations (Section 8.8) should be used to ensure correct implementation of round-robin scheduling with basic priority support.

\[
\text{UpdateCurrentProcess} \triangleq [\Delta \text{Sched}; \ p? : \text{PID} | \text{curr}' = p?]
\]

The idle process reference is established during the initialisation of the scheduler and must not change afterwards. For this reason no operations are provided to change it and other operations must keep \textit{ipid} variable constant as well.

8.6 Enqueue operations

A process that needs to be executed is introduced into the scheduler by enqueuing it on a respective process queue and readying it. To model enqueue operations, we reuse corresponding operations of respective process queues. As the \textit{PQueue} operations do not change the underlying process table, we extend this intent to the scheduler. When a process is enqueued in the scheduler, its respective queue is updated while other information must remain unchanged, e.g. when a user process is enqueued, the device queue and running/idle processes does not change.

We define two sets of operations - for device and user processes. The operations are very similar, therefore we define them side by side. For simplicity, we provide justification for user processes, as analogous arguments are used for devices.

The reused \textit{EnqueuePQueue} operation is total and defines a number of error cases (Section 7.5). As we reuse it by inclusion into corresponding scheduler operation, these error cases still hold. Moreover, the remaining variables are kept the same, thus if an error case in \textit{EnqueuePQueue} is encountered, \textit{Sched} state does not change.

\[
\begin{array}{|c|}
\hline
\text{EnqueueUserSched} \\
\Delta \text{Sched} \\
\text{EnqueuePQueue} \\
\text{curr}' = \text{curr} \land \text{ipid}' = \text{ipid} \land \text{devs}' = \text{devs} \\
\hline
\end{array}
\]
For situations when the process does not satisfy Sched state invariants, we define additional error operations. The state invariants of Sched require that the enqueued process is of correct type and is neither idle process nor the currently running one. We define operations to handle preconditions for these errors. In case of an error, scheduler state is not changed.

$$\text{NotUserPID} 0 \equiv [\exists \text{Sched}; \ p? : \text{PID} | p? \in \text{used} \land \neg \text{ptype} p? = \text{aproc}]$$

$$\text{NotDevicePID} 0 \equiv [\exists \text{Sched}; \ p? : \text{PID} | p? \in \text{used} \land \neg \text{ptype} p? = \text{dproc}]$$

Next the operations are joined with corresponding errors to signal error cases. For current and idle process checks we reuse previously defined query operations.

$$\text{ErrNotUserPID} \equiv (\text{NotUserPID} 0 \land \text{RaiseErrNotUserPID})$$

$$\text{ErrNotDevicePID} \equiv (\text{NotDevicePID} 0 \land \text{RaiseErrNotDevicePID})$$

$$\text{ErrBadPIDCurr} \equiv (\text{RunningProcess}[p?/p!] \land \text{RaiseErrBadPIDCurr})$$

$$\text{ErrBadPIDIdle} \equiv (\text{IdleProcess}[p?/p!] \land \text{RaiseErrBadPIDIdle})$$

The total operations to enqueue processes in their respective queues are defined by disjoining the error cases. Note that OK signal (RaiseOk) is not added to the success case, because it is already indicated in the included EnqueuePQueue operation. Additional definition of RaiseOk would conflict with error cases in EnqueuePQueue.

$$\text{EnqueueUserSched} \equiv \text{EnqueueUserSched} 0 \lor \text{ErrNotUserPID} \lor \text{ErrBadPIDCurr} \lor \text{ErrBadPIDIdle}$$

$$\text{EnqueueDeviceSched} \equiv \text{EnqueueDeviceSched} 0 \lor \text{ErrNotDevicePID} \lor \text{ErrBadPIDCurr} \lor \text{ErrBadPIDIdle}$$

Next we extend these operations to indicate that enqueued process is ready for execution.

### 8.7 Ready operations

The full operation of readying user process in the scheduler involves adding it to the queue and setting its state to ready. Since queue management does not change process table and setting process state requires to update PTab, we use schema composition to join the operations. Both enqueue and state operations are complete (constrain all variables within PTab), the composition is fully defined and does not involve non-determinism.

The enqueue operation is total and has error cases. Using plain composition causes problems, because if enqueue fails and state change does not, the operation results in an intermediate state, which is not desirable. To achieve atomicity, we employ early failure handling (Section 4.5.1) to check if the first operation succeeded.

The error propagation in case the first operation fails is handled by ErrSysFailPTab. In this case, the state is preserved and error indication is held.

$$\text{ErrSysFailPTab} \equiv [\exists \text{PTab}; \ \exists \text{ErrV} | \neg \text{serr} = \text{sysok}]$$

Now the ready operation is defined as enqueuing the process and, if operation was successful, setting its state to psready. Otherwise, the error describing the failure is kept.

$$\text{MakeReady} \equiv \text{EnqueueUserSched} \lor ((\text{IsSysOk} \land \text{SetStateToReady}) \lor \text{ErrSysFailPTab})$$

Operation SetStateToReady has a precondition $p? \in \text{used}$ (Section 6.14). However, we do not specify an error case for it because the same error is handled by EnqueueUserSched. If $p? \in \text{used}$, the first error is reported and SetStateToReady is not executed. This proves that the chosen order of operations is suitable for this simplified definition.

MakeReady is a complex operation, which updates the user process queue and process state in PTab. If expanded, it contains 6 disjoint branches (5 error cases and 1 successful execution). It no longer makes sense to examine full expansion of the operation. Instead, we can slice it and investigate success conditions or particular errors. We show
that the successful execution updates target process queue and process state in \( PTab \), while leaving other variables unchanged. In case of an error, no change is performed. Please refer to the full specification in Appendix E for the expanded versions.

An analogous operation to ready devices, \( ReadyDeviceProcess \), is defined in the same way. All justification done for the user processes apply here as well.

\[
\begin{align*}
\text{ReadyDeviceProcess} & \triangleq \text{EnqueueDeviceSched} \ (\ (\text{IsSysOk} \land \text{SetStateToReady}) \lor \text{ErrSysFailPTab}) \\
\end{align*}
\]

The complex process readying operations provide one of the main capabilities of the scheduler. Using them, a process is introduced to the scheduler and is prepared for execution.

### 8.8 Schedule operations

The main activity of the scheduler is to govern the executing processes and allocate the processor to them in a particular order. Our separation kernel is modeled with a round-robin scheduler with device process priority. The scheduling algorithm to select the next running process out of the readied processes in this case is simple [12]:

1. If queued device processes are available, select the first in the device process queue (which has been waiting for the longest);
2. If no queued device processes are available but there are queued user processes, select the first in the user process queue;
3. If no processes are queued at all, select the idle process.

This algorithm gives devices priority over user processes and respects round-robin practice by using FIFO process queues. When a process is selected to run, it is removed from the respective queue and marked as currently running. Finally, an interrupt is raised in the processor to perform context switch to the new running process.

In order to have good modularisation and separation of execution branches, we define each scheduling operation separately before joining them in one operation for full scheduling action. Originally, Craig modelled the \( SchedNext \) operation by defining full scheduling algorithm in one place [12]. Because of the complexity of these operations, we chose to compose the operation of its smaller counterparts.

#### 8.8.1 Schedule idle process

The idle process is a constant reference within the scheduler, therefore scheduling and running it does not require to change the process queues. For that reason, \( SetStateToRunning \) operation, which indicates the process to be running, can be included in the operation schema.

\[
\begin{align*}
\text{RunIdleProcess} & \\
\Delta \text{Sched} & \\
\text{SetStateToRunning}[ipid/p] & \\
\text{curr}' = ipid \land ipid' = ipid \land \text{procs}' = \text{procs} \land \text{devs}' = \text{devs}
\end{align*}
\]

The action of running idle process consists of marking the idle process as currently running, updating its state to \( psrunning \) and keeping the rest of the system unchanged. The state change operation constrains all variables in underlying process table, therefore \( \text{RunIdleProcess} \) needs only to include invariants for keeping other scheduler variables unchanged.

Note that \( SetStateToRunning \) precondition \( p? \in used \) (Section 6.14) is guaranteed by state invariants of \( Sched \), which require idle process to be known (used). The same can be stated about user and device processes in the later definitions.

Next we supplement the idle process operation with scheduling conditions, which indicate that the idle process is run only if both process queues are empty.

\[
\begin{align*}
\text{RunIdleNext0} & \\
\text{RunIdleProcess} & \\
\text{RaiseOk} & \\
\text{devs} = () \land \text{procs} = ()
\end{align*}
\]
### 8.8.2 Schedule user process

When no device processes are waiting to be scheduled, the first element in the user process queue is scheduled, if available. Removal of the first user process in the queue is specified by reusing `DequeuePQueue` operation.

\[
\begin{align*}
\text{SchedUserNext0} & \quad \Delta \text{Sched} \\
\text{DequeuePQueue} & \\
\text{devs} & = \langle \rangle \land \text{procs} \neq \langle \rangle \\
\text{curr}' & = p! \land \text{ipid}' = \text{ipid} \land \text{devs}' = \text{devs}
\end{align*}
\]

`DequeuePQueue` removes the first process from the user queue and outputs it in `p!` variable. This variable is indicated as currently running and the remaining variables are kept unchanged.

The definition of `DequeuePQueue` (Section 7.6) specifies an error case for empty queue. However, the defined scheduling rules (empty device queue, non-empty user queue) ensures that this error is never reached. We formulate and prove that with the scheduling rules in place, user process schedule operation always succeeds.

**Theorem** \( \text{tSchedUserNext0AlwaysOk} \)

\[ \forall \text{SchedUserNext0} \quad \bullet \quad \text{serr}' = \text{sysok} \]

Queue management in the scheduler restricts underlying \( PTAB \) from changing, therefore the operation to update process status to \( psrunning \) must be added using schema composition. As we have shown that scheduling operation always succeeds, the early failure handling (Section 4.5.1) is not needed for this composition.

\[ \text{RunUserNext0} \equiv (\text{SchedUserNext0}[p] [\text{SetStateToRunning}[n/p?]]) \setminus (n) \]

The precondition of `SetStateToRunning` that the referenced process is used, is guaranteed by `PQueue` invariants, where all queued processes must be used in `PTAB`.

### 8.8.3 Schedule device process

Device processes have priority over user and idle process. The first device process in the device queue is scheduled whenever there are device processes waiting in the scheduler. Apart from this precondition, the definition of device scheduling operations are analogous to these of user processes.

\[
\begin{align*}
\text{SchedDeviceNext0} & \quad \Delta \text{Sched} \\
\text{DequeueDeviceQueue} & \\
\text{devs} & \neq \langle \rangle \\
\text{curr}' & = p! \land \text{ipid}' = \text{ipid} \land \text{procs}' = \text{procs}
\end{align*}
\]

We show that device process scheduling is always successful with the indicated scheduling rules and define the full device running operation.

**Theorem** \( \text{tSchedDeviceNext0AlwaysOk} \)

\[ \forall \text{SchedDeviceNext0} \quad \bullet \quad \text{serr}' = \text{sysok} \]

\[ \text{RunDeviceNext0} \equiv (\text{SchedDeviceNext0}[n/p] [\text{SetStateToRunning}[n/p?]]) \setminus (n) \]

Having defined individual operations and scheduling rules for each type of schedulable process, we join them all for the combined scheduling operation.

### 8.8.4 Schedule next process

Each scheduling operation handles a distinct subset of possible scheduler states. By disjoining them in a single operation, a scheduling operation for any scheduler state is defined.

\[ \text{SchedNext} \equiv (\text{RunIdleNext0} \lor \text{RunUserNext0} \lor \text{RunDeviceNext0}) \setminus \text{CtxtSw} \]

The actual running process change within the processor is initiated by raising context switch interrupt [12]. Context switch is modelled using \( CtxtSw \) operation (Section 5.9). Originally \( CtxtSw \) was added to each individual scheduling
operation [12], however with composition distribution over disjunction, the \textit{CtxSw} operation can be moved outside.

Further precondition calculations show that \textit{SchedNext} operation has no preconditions (Section 8.10). This means that scheduling operation can be executed for any scheduler state. Moreover, it is failproof, as the error cases possible in dequeing are dismissed by the scheduling algorithm. We formulate and prove that the execution of scheduling operation is always successful.

\textbf{theorem} \ tSchedNextAlwaysOk
\[ \forall \text{SchedNext} \bullet \text{ser}^t = \text{sysok} \]

\textit{SchedNext} is the central operation for the scheduler, as it selects and executes the next process. Proving that it never fails and can be executed for all scheduler states, gives us confidence about the implementation of this formal model of a separation kernel scheduler.

The defined operations to ready a process in the scheduler and schedule next process provide all required functionality for a separation kernel scheduler. Next we define an operation to suspend a user process, which combines the defined scheduler operations.

\section{Suspend operations}

A suspend operation is executed by user processes when they want to relinquish execution to other processes. One of the cases when this may happen is during inter-process messaging - a process sends a message to other process and suspends itself allowing the other process to eventually execute and handle or reply to the message.

According to separation kernel definition, processes are never pre-empted and can only suspend themselves voluntarily [55]. The principle is held in this formal model, therefore a suspend operation is defined.

\subsection{General suspend operation}

Specifically, the suspended user process must stop executing and the next available process should be run. Furthermore, the suspended process must be placed back on the user queue to be executed when its turn comes. This definition is easily satisfied by reusing \textit{SchedNext} and \textit{MakeReady} operations.

Both operations are total and complete (constrain all variables in \textit{Sched}), therefore they are joined using schema composition (\(\circ\)). Note that we do not need to use early failure handling (Section 4.5.1) for these operation, because we have proved that \textit{SchedNext} always succeeds (Section 8.7).

\[ \text{RequeueUserProcess} \equiv (\text{SchedNext} \circ \text{MakeReady}) \]

The general definition of the operation takes the suspended process as an input variable [12]. Thus basically it defines an operation to put a user process onto a queue while scheduling next process at the same time.

When it is actually used to define suspend operation afterwards, currently running process is used as the input - thus satisfying the suspend operation definition. This means that the scheduling operation must precede readying operation, because currently running process cannot be enqueued without first stopping it.

\subsection{Transactional suspend operation}

The general operation \textit{RequeueUserProcess} is not atomic - in case of \textit{MakeReady} failure, \textit{SchedNext} still executes successfully. The problem of achieving atomicity for the failure of second operation in schema composition is quite challenging. We analyse this problem in depth and propose a solution in the form of a rough \textit{Transaction} design pattern for Z in Chapter 4.5.

\textbf{Remark 8.4.} \ (Necessity of operation atomicity) While the atomicity is not explicitly specified as a requirement in the original specification [12], its achievement is an improvement to the specification. It guarantees that the operation never leaves the system in an intermediate state, even though valid, and operation intent is realised correctly.

Furthermore, one may argue that the simplified security exit error handling (Section 5.10.4) kills the kernel in case of an error, therefore a complex operation upgrade to guarantee atomicity is not necessary. However, we want to create a reusable component that contributes to formal modelling of operating system kernels research. This means that is is very likely that in the future the kernel is upgraded to have a more sophisticated error handling system, e.g. one that allows recoveries on certain error situations. In that case the users of this model will highly benefit from considerations about operation atomicity.
To achieve atomicity, we employ the defined Transaction design pattern (Section 4.5.1), which allows to roll back the transaction if any of the operations fail. We define operations to backup and revert Sched state using intermediate decorated state Sched′′.

\[
\begin{align*}
\text{SchedBackup} & \equiv [\Delta \text{Sched}; \theta \text{Sched} = \theta \text{Sched}''] \\
\text{SchedRevert} & \equiv [\Delta \text{Sched}'; \theta \text{Sched}'' = \theta \text{Sched}']
\end{align*}
\]

The transactional handling either accepts operation results when system is in sysok state, or reverts then to initial state using SchedRevert. Note that both the error and hardware state are kept in case of the error, because it is important to retain the interrupt raised by the operations.

\[
\begin{align*}
\text{SchedTrans} & \equiv SchedTrans0 \lor \text{ErrSchedRevertTrans} \\
\text{ErrSchedRevertTrans} & \equiv [\Delta \text{Sched}'; \theta \text{Sched}'' = \theta \text{Sched}']
\end{align*}
\]

Finally, the full transactional operation is defined following Transaction pattern (Section 4.5.1).

\[
\begin{align*}
\text{RequeueUserProcessTrans} & \equiv \exists \text{Sched}' \bullet ((\text{RequeueUserProcess} \land \text{SchedBackup}) \circ SchedTrans)
\end{align*}
\]

Remark 8.5. (Complexity of scheduler operations) The defined operations are complex, because they are constructed of a number of smaller operations. As witnessed before, MakeReady has 5 error cases plus 1 success case (Section 8.7). When combined with 3 execution paths in SchedNext, we get a total of \(3 \times 6 = 18\) execution paths in RequeueUserProcess.

The added transactional support in RequeueUserProcessTrans has 2 additional execution paths (complete or revert), making the number of execution paths in RequeueUserProcessTrans total at \(18 \times 2 = 36\). For the sheer amount of execution paths, the expanded versions of these operations would be enormous. Even the success case needs to analyse all 3 paths of SchedNext. For this reason, we do not provide expanded versions of the operations.

### 8.9.3 Device process suspend operations

Again, an analogous operation is defined to suspend device processes. The transactional support for the operation reuses the same operations as defined for user processes.

\[
\begin{align*}
\text{RequeueDeviceProcess} & \equiv (\text{SchedNext} \circ ReadyDeviceProcess) \\
\text{RequeueDeviceProcessTrans} & \equiv \exists \text{Sched}'' \bullet ((\text{RequeueDeviceProcess} \land \text{SchedBackup}) \circ SchedTrans)
\end{align*}
\]

### 8.10 Operation preconditions

We calculate preconditions for all scheduler Sched operations and prove that the operation can be executed under the indicated preconditions (see Section 4.3.4). The preconditions are summarised in Table 8.1.

<table>
<thead>
<tr>
<th>Operation name</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RaiseErrNotUserPID, RaiseErrNotDevicePID, RaiseErrBadPIDCurr, RaiseErrBadPIDIdle</td>
<td>true</td>
</tr>
<tr>
<td>IdleProcess, RunningProcess</td>
<td>true</td>
</tr>
<tr>
<td>UpdateCurrentProcess</td>
<td>(p? \in \text{used} \land p? \notin \text{queued})</td>
</tr>
<tr>
<td>EnqueueUserSched0</td>
<td>(p? \in \text{used} \Rightarrow (\text{ptype} p? = \text{uproc} \land p? \notin {\text{curr}, \text{ipid}}))</td>
</tr>
<tr>
<td>EnqueueDeviceSched0</td>
<td>(p? \in \text{used} \Rightarrow (\text{ptype} p? = \text{dproc} \land p? \notin {\text{curr}, \text{ipid}}))</td>
</tr>
<tr>
<td>ErrNotUserPID</td>
<td>(p? \in \text{used} \land \neg \text{ptype} p? = \text{uproc})</td>
</tr>
<tr>
<td>ErrNotDevicePID</td>
<td>(p? \in \text{used} \land \neg \text{ptype} p? = \text{dproc})</td>
</tr>
<tr>
<td>ErrBadPIDCurr</td>
<td>(p? = \text{curr})</td>
</tr>
</tbody>
</table>
The precondition calculation shows that all top-level (interface) operations, which should be exposed for usage by other components, have a precondition true. This means that their execution is defined for all scheduler states, which makes it a robust interface to interact with the scheduler. The interface operations are marked in bold in Table 8.1.

The complexity of operations within the scheduler (e.g. see Remark 8.5) poses a challenge to prove the calculated precondition theorems and other proofs. Scheduler operations reuse constructs in process table PTab and process queue PQueue data structures. This means that no fundamental proofs of certain properties are necessary (hence the lack of new automation theorems and general lemmas for Sched). Nevertheless, the sheer size of expanded scheduler operations does not allow us to benefit from automatic proving in Z/Eves. When put against a huge amount of invariants, Z/Eves theorem prover fails to calculate the result and proving each step may take minutes and more.

We tackle the proofs by carefully guiding the prover towards the expected goal. Good scheduler model and structure, allows us to build up precondition proofs by reusing corresponding precondition proofs of composing operations. We slice the proof into several parts and prove them individually. When guiding the prover through the proof steps, we expand only these schemas, which benefit towards the expected goal.

### 8.11 Summary

This chapter defined the separation kernel scheduler. The scheduler supports two main operations - introduce a process (make ready) to the scheduler, preparing it for execution; and schedule next process. Other operation, such as suspend, are derived by composing next scheduling with process reintroduction to scheduler.

The scheduler can only be initialised if there exists the idle process. That is, the idle process must always be defined for the scheduler, and it is executed when no other processes are queued. This means that to initialise the whole system, the process table needs to initialised first, then an idle process must be allocated to initialise the scheduler. This event has been explored within this chapter.

Next process scheduling operation is robust in a sense that it can be executed for any scheduler state and it will always succeed. A process can only be introduced to its respective queue if it is not already running. Furthermore, the composite process ready and suspend operations benefit from transactional properties - if part of the operation fails, no change is done to the scheduler and an error is reported.

The formal specification of the scheduler completes the separation kernel model within dissertation scope. With all specifications complete, the project can be evaluated to verify its successful completion.
Chapter 9

Evaluation and Results

The formal model of separation kernel (Chapters 5-8) contains both the design (definitions) and evaluation (theorems and proofs) of each component and its parts. In this chapter, the project is evaluated as whole. First of all, project goals and requirements established in the problem analysis are reviewed, as the project success is evaluated against them. After that the project results are examined in detail to show requirements have been successfully achieved. Finally, the proof effort is discussed and projects benchmarks are used to compare it with related projects.

9.1 Project evaluation & achievements

Problem analysis (Chapter 3) examined the need for this project and identified requirements for separation kernel and its formal model. It is difficult to evaluate the project against these requirements, as they encompass a much broader effort within the pilot project than it is possible to achieve in the given time frame.

The separation kernel requirements (Section 3.1) were identified to drive the design of separation kernel model. They are a landmarks against which the previous research and resources, kernel design and necessary contributions are evaluated. Separation kernel architecture (Section 4.7) addressed these requirements with corresponding design decisions.

The limitations of dissertation scope mean that not everything can be done. To satisfy all kernel requirements, the design and model need to be extended with configuration support, audit and deployment. Limited project time (May - September 2009) and lack of experience in formal modelling and proofs required to make a choice: either extend the model and abandon formal modelling, or continue with the mechanisation and formal modelling of the existing kernel. Model requirements ([MReq-A1-A4], Section 3.2) put mechanisation as one of the top priorities, therefore it was decided to defer full separation kernel model for future research.

A second set of requirements come from the pilot project (Section 3.2). They define what is expected out of a successful project. Based on these requirements, literature and problem analysis, a superset of requirements, declared the goals of the project have been identified (Section 3.4).

Again, dissertation scope required to limit the goals to a feasible amount. Goals [G-8] (separation kernel completeness) and [G-9] (refinement) were discarded because of their scale and prerequisite of correct model.

Goals [G-1] and [G-6] represent the base requirements for the project. The selected separation kernel components need to be mechanised and everything must be proved. This is comparable to the work completed for another dissertation within the same pilot project [44]. Furthermore, the remaining goals were marked as highly desirable and should be attempted in the case of success. Thus, if goals [G-1]-[G-7] and [G-10] have been addressed, we believe the objectives (Section 1.2) of this work have been achieved.

All selected components within the scope have been mechanised. A detailed analysis of the original model [12, Ch. 5] revealed a number of problems, which have been corrected and alternatives have been proposed, where applicable. Also, interesting kernel component properties have been identified and verified. All mechanisation proofs have been discharged and the model has been verified within the dissertation scope (see Remark 9.1). For all the models and proofs, refer to Appendices E and F.

Moreover, the performed formal modelling produced a number of reusable lemmas and design decisions. An original Transaction design pattern has been developed to achieve transactional properties within composite operations in Z. Finally, a full formal specification with justification, changes and comparison to original has been prepared to be curated within the Verified Software Repository.

These achievements show that all project goals set up at the beginning ([G-1]-[G-7] and [G-10]) have been reached successfully. In the following sections, the project evaluation continues with detailed analysis of the mentioned results, requirement satisfaction and overall experience.
9.2 Mechanisation

Basic types, hardware abstraction, process table, process queues and the scheduler components were translated from the book [12] to the formal Z specification in \texttt{LATEX}. Then the specification was parsed and analysed using Z/Eves. Note: the process was actually incremental and each paragraph was translated/verified one-by-one.

Z/Eves syntax and type checking facilities revealed a number of errors in [12]. The common ones are:

- Copy/paste mistakes and typos - variable names were incorrect or inconsistent;
- Invalid declaration order - a construct needs to be defined before used;
- Missing declarations - some declarations were missing altogether and needed to be defined/deduced;
- Invalid use of Z operators - some operators were selected inappropriately;
- Missing schema inclusion - variables from schemas, which have not been referenced, were used in predicates;
- Inappropriate promotion structure - promotion use for scheduler in the original specification had inadequate structure.

The formal specification in [12] shows good domain expertise, however Z notation carries a number of structural, typography and other simple mistakes, which have been spotted and corrected.

After these errors have been corrected, the full Established Strategy verification was applied (Section 4.3):

- Proofs of generated domain checks revealed missing predicates in a number of schemas;
- Each axiomatic definition was preceded with a global consistency check and proved;
- Initialisation theorems were defined for all state initialisation operations;
- Preconditions were calculated and proved for each operation. The precondition calculation requires a large scale effort in proof. Also, precondition proofs revealed inconsistencies in composite operations, where Craig [12] misused conjunction ($\land$) and schema composition ($\circ$) operators.

Analysis of total (interface) operations revealed two common mistakes in the original specification:

- Missing precondition invariants for error cases - Craig [12] defined preconditions for success operations only. The error signals were added using disjunction ($\lor$) without any invariants. This allows errors to be signalled for all states when operation is called (see Section 4.2.2 for more information);
- Unaccounted error cases - additional error cases in original specification were missing for certain conditions.

The total operations have been corrected by adding necessary preconditions for error operations and introducing new error cases where necessary.

Full details on formal specification changes are given as comments accompanying the specification (Appendix E). All proofs in the mechanisation have been completed. Refer to Section 9.5 for discussion about completed proofs.

The evaluation of mechanisation state shows that the goals [G-1] and [G-6] have been fully satisfied.

9.3 Formal modelling

Having mechanised the formal specification, the formal model and its semantics have been examined with a critical view. The invariants and design decisions were analysed to find their intent and the actual meaning of Z statements, then comparing them with appropriate requirements, possible implementations and invariant implications.

This lead to numerous improvements in the formal model. A number of mistakes and shortcomings have been found. Furthermore, the specification was improved with appropriate invariants and security measures. The important changes are outlined here:

- Improvement of basic types, namely non-emptiness of $TSS$, distinctive kill and context switch interrupts, correct process identifier ranges;
- Correction of invariants about device number uniqueness in $PTab$;
- Addition of necessary invariants for new $UPID$ generation, guaranteeing unused and unique identifiers;
- Identification and removal of redundant error cases;
• Identification of new preconditions for operations to achieve correct behaviour - mechanisation and precondition calculation ensured a valid behaviour, however further analysis has shown that additional preconditions are required to ensure a correct behaviour according to operation’s intent;

• Full initialisation of all process attributes, when a new user process is allocated in PTAB;

• Correction of idle process allocation - Craig [12] tried modelling idle process allocation as a “lightweight” process, which could not be supported by PTAB invariants;

• Proper process deletion, which ensures that information of remaining processes is kept when one process is deleted;

• Adaptation of secure process queue from [21], instead of a compromised one in the original;

• Correct process queue initialisation, which does not restrict underlying process table (correction of a such assumption in [21]);

• New security invariants in the scheduler about queued processes and current/idle process references;

• Removal of redundant reference to previous process in the scheduler;

• Fully constrained composite operations - original model left a large number of variables undefined when constructing complex operations using schema composition;

• Modelling transactional properties for composite scheduling operations - if one of the composed operations fails, the whole operation fails.

The detailed analysis of the formal model lead to substantial improvements, while following the model architecture and separation kernel requirements. Furthermore, the Z specification style was greatly improved in style by selecting proper Z operators and techniques to represent the model, simplifying operational style in [12] and removal of redundant invariants:

• Evolution of process table state schema PTAB involved removing duplicate predicates, using relational image instead of set comprehension and changing complex predicates to partial injection definitions;

• Union operator usage instead of relational override in user process allocation represents the intent better;

• Construction of complex operations using composition was changed to simpler conjunctions where applicable;

• Promotion technique was abandoned in scheduler model in favour of schema inclusion with conjunction. Note that the scheduler model had been completely mechanised using promotion, but the technique application there was not appropriate, therefore the whole scheduler has been remodelled using schema inclusion (conjunction).

The relationship with the original specification was maintained by proving equivalence and refinement theorems on changed schemas, where applicable (Section 4.3.5).

Formal modelling and analysis was performed to an even larger extent, by investigating interesting properties of the components and proving them. Furthermore, in cases where Craig’s abstraction level seemed too low (specific algorithms provided), more abstract models have been discussed.

Finally, the formal specification layout and presentation have been upgraded, following the naming conventions and Established Strategy requirements (Sections 4.2 and 4.4).

Details on the performed modelling, changes and justifications are provided in Chapters 5-8 and Appendix E.

The performed formal modelling satisfied the following goals: [G-2], [G-3], [G-4] and [G-7].

9.4 Contributions & reusability

While a complete formal model of separation kernel was beyond the scope of the dissertation, the project has produced significant contributions:

• Formal models of separation kernel process table, process queue and a round-robin scheduler. They have been modelled and verified with a certain level of independence from other components. Components have clearly defined total interface operations, increasing their reusability in other projects. Furthermore, a number of component properties have been identified and proved;

• General lemmas about injective functions, sequences and relationship domain/range - these lemmas can be directly reused in other projects;
• **Transaction** design pattern - an original design pattern has been developed to achieve *atomicity* property when total operations are joined using schema composition. Composite operations are frequently used in complex formal models, therefore the Transaction pattern is a significant contribution by this project;

• Kernel modelling decisions - the problems found and fixed, as well as general modelling decisions (Section 9.3) can benefit further research on this or similar kernels;

• Full separation kernel requirements and design - the performed analysis covered a large amount of material beyond the scope of the dissertation. It can serve as a reference for further research on separation kernel models.

To be curated within Verified Software Repository, the formal specification must be verified, demonstrate quality, reliability and maintainability properties. The specification has closely followed *Established Strategy* requirements for structure and verification. Identification and verification of interesting model properties established further model validation. The reasoning and verification capabilities provided by formal methods mean that testing is not mandatory for such models, because formal proofs guarantee the examined properties.

Finally, the produced formal Z specifications are using \LaTeX notation, a standard in industry and academia. Set naming, layout and structure conventions have been followed to improve the presentation. Because of full model mechanisation, it is easy to extend the model with further research or reuse parts of the model in similar projects. Also, all justification, changes and descriptions are provided together with the model in Z \LaTeX specification (Appendix E).

The analysis of contributions and project reusability show that goals [G-5] and [G-10] have been satisfied. This completes the project evaluation against the goals set during problem analysis and shows that all goals necessary for the complete success have been reached.

The project evaluation is then continued with analysis of performed proofs and comparison of project statistics with similar work.

### 9.5 Proofs & benchmarks

The project evaluation highlighted mechanisation and modelling achievements. The associated proofs verify the correctness of target models.

Proving theorems in formal methods application is one of the most difficult and time-consuming tasks. Literature review (Chapter 2) revealed that the main obstacle in verifying kernels is the difficulty of encountered proofs.

Before analysing proofs and project benchmarks, it is necessary to put them in a context. Project statistics are calculated in a similar style to Mondex pilot project [23]. Then they can be compared with benchmarks available from Mondex and a similar simple kernel mechanisation [44] projects. Simple kernel mechanisation project gives excellent comparisons, because it is constrained within the same time frame and builds on [12].

Table 9.1 presents a summary of paragraphs (schemas, definitions, etc.) in the formal specification.

<table>
<thead>
<tr>
<th>Table 9.1: Summary of paragraphs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Given sets</strong></td>
</tr>
<tr>
<td><strong>Free types</strong></td>
</tr>
<tr>
<td><strong>Axiomatic definitions</strong></td>
</tr>
<tr>
<td><strong>Abbreviations</strong></td>
</tr>
<tr>
<td><strong>Schemas</strong></td>
</tr>
<tr>
<td><strong>Operation signature schemas</strong></td>
</tr>
<tr>
<td><strong>Total number of paragraphs</strong></td>
</tr>
</tbody>
</table>

The majority of the 263 paragraphs are Z schemas. Note that almost half of them are operation signature schemas, which are used to prove calculated preconditions. The number of paragraphs somewhat exceeds Mondex specification (199 paragraphs, though unclear whether signature schemas are used). Furthermore, this formal specification is 34% larger than simple kernel’s, indicating project success in scope size.

The proofs are analysed in a similar way. Table 9.2 summarises the theorems and proofs in separation kernel model. The verification of created model requires 254 proofs, out of which 36% are precondition proofs. Furthermore, 35% of all proofs are Z/Eves automation rules or lemmas. These theorems are defined to facilitate other proofs. Comparison with other projects allows making interesting assumptions. The ratio of “helper” theorems in Mondex is 44%, while
Table 9.2: Summary of proofs.

<table>
<thead>
<tr>
<th>Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumption rules</td>
<td>33</td>
</tr>
<tr>
<td>Forward rules</td>
<td>9</td>
</tr>
<tr>
<td>Rewrite rules</td>
<td>34</td>
</tr>
<tr>
<td>Lemmas</td>
<td>12</td>
</tr>
<tr>
<td>Theorems</td>
<td>143</td>
</tr>
<tr>
<td>Global consistency</td>
<td>6</td>
</tr>
<tr>
<td>Initialisation</td>
<td>9</td>
</tr>
<tr>
<td>Equivalence</td>
<td>24</td>
</tr>
<tr>
<td>Precondition</td>
<td>91</td>
</tr>
<tr>
<td>Domain checks</td>
<td>23</td>
</tr>
</tbody>
</table>

**Total number of proofs** 254

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proved</td>
<td>253</td>
</tr>
<tr>
<td>Unproved</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 9.3: Summary of proof steps.

<table>
<thead>
<tr>
<th>Step</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>44</td>
</tr>
<tr>
<td>cases</td>
<td>52</td>
</tr>
<tr>
<td>equality substitute</td>
<td>6</td>
</tr>
<tr>
<td>instantiate</td>
<td>97</td>
</tr>
<tr>
<td>invoke</td>
<td>201</td>
</tr>
<tr>
<td>next</td>
<td>96</td>
</tr>
<tr>
<td>prenex</td>
<td>14</td>
</tr>
<tr>
<td>prove</td>
<td>224</td>
</tr>
<tr>
<td>prove by reduce</td>
<td>99</td>
</tr>
<tr>
<td>rearrange</td>
<td>19</td>
</tr>
<tr>
<td>reduce</td>
<td>41</td>
</tr>
<tr>
<td>rewrite</td>
<td>166</td>
</tr>
<tr>
<td>simplify</td>
<td>28</td>
</tr>
<tr>
<td>split</td>
<td>75</td>
</tr>
<tr>
<td>use</td>
<td>147</td>
</tr>
<tr>
<td>with disabled</td>
<td>180</td>
</tr>
<tr>
<td>with enabled</td>
<td>40</td>
</tr>
</tbody>
</table>

**Total number of commands** 1309

Liu [44] has 21%. These ratios may imply difficulty of proofs in each model. A higher number of lemmas required to prove target theorems suggest that the proofs were more difficult.

Comparison of project sizes shows that this project is 20% smaller in proof size than Mondex and 60% larger than mechanisation of the simple kernel [44].

Table 9.2 shows that there are three incomplete proofs. One of them, namely \(\text{IPQCatPIDIsInIseqPID}\) cannot be proved due to limitations of Z/Eves theorem prover.

**Theorem** rule IPQCatPIDIsInIseqPID \[X\]

\[
\forall A : P \ X \cdot \forall s : \text{iseq } A ; x : A \mid x \notin \text{ran } s \cdot s \cdot (x) \in \text{iseq } A
\]

The lemma states that for the injective sequence concatenation to remain injective, the added element must not belong to the sequence already. While such definition produces a sequence without duplicates, this cannot be proved using Z/Eves due to its internal representations of sequences as LIST lists. If we were to change the toolkit to model sequence displays using Z as in the other mathematical operators, this problem would disappear.

This lemma has been created by Freitas [22] to complete process queue proofs. Since the process queue was adapted to suit the separation kernel, the same theorem was used to prove the properties of injective sequences in this kernel. Refer to Section 7.5 and [22] for more information.

**Remark 9.1. (Unfinished proofs)** The two other unfinished proofs are precondition proofs of transactional suspend operations RequeueUserProcessTrans and RequeueDeviceProcessTrans. Note that the proofs are very similar, as only the variable names are different. Therefore it can be assumed that only one proof is incomplete, as the other one would be proved immediately with the success of the first one.

These theorems have not been proved because of the lack of time, as the Transaction pattern has been identified and developed by the very end of the project.

This should not diminish confidence in the pattern, as it has been proved successfully (Appendix C). The proof difficulty arises from the complexity of RequeueUserProcess operation, as it is composed of a large number of smaller total operations using disjunction and schema composition.

Furthermore, the transactional operations are the last in the dependency chain and no other models and proofs depend on them. Also, the preconditions of original composite operations RequeueUserProcess and RequeueDeviceProcess have been proved successfully.

Based on precondition proofs for Transaction pattern example (Appendix C) and verified preconditions of operations...
RequeueUserProcess and RequeueDeviceProcess, I can confidently state that these preconditions can be proved with the current model, it is just a matter of allocated time.

Also, with these operations being at the end of dependency chain, the remaining model has been verified and its mechanisation (up to transactional operations) is complete.

All other proofs in the model have been completed.

The final statistics are presented in Table 9.3. It lists the proof commands used to complete the proofs in question. 23% of the listed commands (apply, split, used and with enabled) require creative steps of various levels of difficulty. The split command is used to guide the prover in specified paths to achieve goal. apply, used and with enabled commands involve choosing an appropriate lemma or rule (either defined in the project or from Z/Eves mathematical toolkit [58]) to transform the proof goal [46]. Using these commands requires a good proof plan and precise application of available lemmas. The ratio of creative steps is similar to other projects.

One of the biggest challenges when proving theorems about complex operations has been the size of expanded proof goals, notably in scheduler component. Solving this requires a good proof plan and knowing what exactly needs to be expanded to be enough for the prover to complete. Commands invoke and with disabled (when combined with expanding proof commands) are used to expand specific schemas and constitute almost 30% of all commands, showing that such guidance has been often used during the proof.

Finally, a quantified comparison shows a disparity between the projects. Even though this project has 74% more proofs steps than [44], it is 71% smaller than Mondex. This suggests that Mondex proofs have been more difficult and therefore required more steps. Also, the level of automation is higher in this (40% of commands compared with 27% in Mondex), therefore less manual proof steps have been required.

The provided statistics allow to conclude that the proof effort in this project has been really strong. The proof difficulty may have been lower than in Mondex, however the absence of experience in using Z/Eves and performing formal proofs needs to be noted. Furthermore, the project goals have been reached and all mechanisation theorems have been proved (note Remark 9.1).

9.6 Limitations

The largest limitations of the separation kernel model (but not of the project) are caused by the scope of dissertation. Full formal model requires a much larger effort than available resources. To complete the separation kernel model, it is necessary to perform similar formal modelling to remaining components identified during the design (Section 4.7). Furthermore, the model needs to be extended to fully satisfy separation kernel requirements. Finally, refinement needs to be performed down to code level to have a working verified kernel. All this work is beyond the dissertation scope as established in problem analysis (Chapter 3).

Some compromises have been done for basic types and hardware, where only mechanisation has been done (mostly) because hardware, interrupts and other types are out of dissertation scope.

Finally, the level of abstraction used by Craig [12] is sometimes too low, where implementation details (algorithms) are modelled. It is noted in the formal specification where abstraction can be achieved.

9.7 Development process & experiences

The selected Established Strategy development process has shown to be an excellent choice for the separation kernel model. The requirements for formal specification structure and necessary proofs ensure that all necessary parts of the model have been defined and verified.

I have exercised the iterative nature of the process heavily, going back to earlier parts to apply corrections spotted later. A usual way of modelling can be described as design a particular schema, write its Z specification, verify its properties, go back to design if verification failed or produced unexpected/inadequate results, write and verify again. Such design/verification process has been applied in throughout whole formal specification, therefore the justification and verification is given together with the design and model, as it seems natural.

Furthermore, there exists a learning curve with formal modelling, large scale Z application and especially proving theorems. The learning curve of Z/Eves theorem prover is about 2 weeks for basic proofs and 4 weeks to attempt difficult proofs successfully. Also, learning rules and theorems available in Z/Eves toolkit [58] is necessary. Because of this (and the initial lack of experience), a lot of substantial changes have been done in the later parts of the project. The initial mechanisation results were analysed with critical approach (see Section 9.3) and a number of important changes or even remodelling was done. Changes in the process table caused a ripple through the specification, as both
the model and proofs of the process queue and the scheduler needed updating. This was expected, as it corresponds to the chosen iterative development process.

I have developed the Transaction design pattern as a solution to problems found in composite operations. A critical analysis of composite scheduler operations has shown that if one operation fails, other are executed, which leaves the kernel in an intermediate (albeit correct) state. I have found a partial solution (early failure handling) in the literature, however there were no suggestions how to handle failure of the last operation in composition. Therefore I have developed the pattern as original research.

9.8 Summary

This chapter evaluated the project results and concluded that all its goals have been achieved. The produced formal model is fully mechanised and verified. Various improvements resolve semantic problems in the original specification and upgrade model security and assurance properties. The project contributes to the Verified Software Repository as a case study on separation kernel modelling, as well as by providing new kernel data types, general lemmas and design patterns.

The discussion on performed proofs has outlined the proof process and effort. Project benchmarks have lead to conclusion that the project size in definitions and proofs exceeds that of a similar project on simple kernel mechanisation.

In the following chapter, the project is concluded with discussion on the achievements in general and proposals as to how this project can be extended.
Chapter 10

Conclusions and Future Work

In this final chapter the project is brought to a close. The project achievements are analysed in a broader context than in evaluation (Chapter 9), drawing conclusions about project’s place and contributions within the Grand Challenge. Furthermore, a number of ways how the project can be extended and applied in the future are examined. Finally, a supplement to the project - prototype tool for Z/Eves integration with modern platform - is described.

10.1 Achievements in broader context

The Grand Challenge in Verified Software is a long-term initiative and each pilot project contributes towards the eventual goal. Different pilot projects help us explore the best approaches to modelling various application domains. The verification of each model yields different challenges, solutions of which contribute for the next project.

This project has proved the benefits of Verified Software Repository and contributions from previous projects. A number of general lemmas have been successfully reused to prove properties about separation kernel model and new lemmas, which in turn contribute back to the Repository. Working in the same domain as other researchers, allowed reuse of the domain-specific data types. The process queue has been defined for the simple kernel [21] but was adapted to the separation kernel as well without significant effort.

The project extended data types collection in the domain of operating system (OS) kernels: the produced process table differs significantly from the simple kernel’s in a way that it contains processes of distinct types. This means that a number of operations can be reused for both processes, while others are specific to each type. The design decisions and verification experience for such process table can facilitate further projects, which require multi-type data structures.

Furthermore, the round-robin scheduler with distinct process queues may not be such frequent in other applications, but it spurred very interesting and useful findings about composite operations. The newly developed Transaction design pattern for Z solves atomicity problems when total operations are composed. The complex models eventually reuse a number of total operations from their components, which need to be composed using schema composition (5). If there is a requirement that such operations cannot leave the system in intermediate state, Transaction pattern can be reused to define atomic operations.

The improvements in the formal specification in comparison to the original [12] are significant. Mechanisation and critical analysis allowed corrections on a number of predicates and introductions of new ones, making the model more defined and secure. A number of findings can be directly applied to similar data structures in the simple kernel, achieving similar improvements.

The preparation work for the project was extensive and provided results beyond the scope of the dissertation. The examined separation kernel requirements or full architectural design of separation kernel model have been addressed in this project, however it is also very useful for future work on the separation kernel. The next research can benefit from the findings that have been identified during this project.

The project has been very successful in terms of results and contributions. Furthermore, the experience gained during this period of time about formal modelling and proofs is very valuable, increasing my interest in formal methods application and research beyond this dissertation.

10.2 Future Work

This project makes up only a part of full effort necessary to complete formal model of separation kernel. In problem analysis (Chapter 3), the project scope was limited several times, leaving a lot of work for further research.

The following overview outlines the significant work that can be continued from this project.
Finish top-level model. The project was limited to basic types, process table, process queue and the scheduler. There are a number of significant work left to mechanise and improve remaining separation kernel components. The messaging system (MsgQ) is a good exercise on Promotion technique, while the memory abstraction and Storage Pool has quite algorithmic invariants. A more challenging modelling exercise is modelling a proper interrupt handling system and defining complete hardware abstraction.

Upgrade error handling. The error handling system in current model has been defined by Craig [12] as “paranoid”, halting the kernel on each error. A correct error handling system may investigate each error and make actions to counter the arising problems.

Complete model to satisfy SKPP. The NSA Separation Kernel Protection Profile (SKPP) [63] yielded a number of requirements for separation kernel (Section 3.1). The model design did not address all of them, therefore additional work is necessary to extend the model for configuration, deployment and other support.

Refine model to implementation. One of the main objectives of the Verified OS Kernels pilot project, refinement must be performed after the model is complete. This is necessary because otherwise a change in the model will require changes in refinement, therefore additional proofs. Different approaches can be taken during the refinement. A classic refinement applies the refinement calculus [76] to achieve corresponding implementation. Additionally, a tool chain can be investigated for doing refinement:

\[
\text{Z/Eves} \rightarrow \text{ZRC-Refine/Gabriel} \rightarrow \text{Spec#-Boogie/PL}
\]

Refinement can also benefit from commercial products, such as INFORMED design method for SPARK [1] or Perfect Developer [51]. In the latter case, it is necessary to define the translation from Z to Perfect and working implementation code can be generated using the tool.

Produce abstract model. The separation kernel as defined by Craig [12] carries a number of specific algorithms and other implementation details. This means that the level of abstraction is lower than it can be. The lower abstraction level models have more details and therefore are harder to reason about. This is a problem if the reasoning does not need the specific algorithms and an abstract model would have suited better. Thus the model can be abstracted into higher level. Note that the abstraction is usually easier than the refinement, as details no longer needed.

These are the main future goals for the separation kernel. It is necessary to note that some of them, for instance the refinement, are large and need to be split into smaller tasks if allocated to projects of similar scope to this.

10.3 Improving Z/Eves

As a supplement to this project, I have developed a prototype Eclipse [15] plug-in for Z/Eves. Z/Eves has certain limitations (Section 4.6) when working with either provided interface. Frequent crashes in its Python UI and, in particular, lack of proper copy/paste functionality caused frustration when trying to work on the project. Furthermore, the formal specification was developed as a number of *.tex files with Z written using LATEX notation. Importing/exporting the specification is very cumbersome when using the Z/Eves with Python UI.

The text interface is limited even more in Windows operating system, because of poor capabilities of the cmd.exe command-line. The provided integration with emacs caused totally different problems, because I am used to Eclipse-like environments and different shortcuts and process in emacs was stopping me more than actually allowing to work with Z/Eves.

To overcome these obstacles, I decided to integrate Z/Eves text interface with Eclipse platform. Afterwards I expanded it with additional features. The developed plug-in provides the following improvements for Z/Eves and Z modelling in general within Eclipse:

Access to Z/Eves text interface. A new external tool configuration has been developed to launch Z/Eves within Eclipse Console view. This allowed interaction with the text interface using text copy/paste functionality from anywhere (Eclipse and other tools). Furthermore, Eclipse controls for console can be used for Z/Eves as well, such as pinning console window, locking the scroll or clearing it altogether.

Z LATEX preview. Z/Eves text interface outputs proof goals using Z LATEX notation. With little experience reading such output, it is extremely hard to understand such predicates and expressions. I have created a Z Console view, which transforms the Z LATEX notation into corresponding UTF symbols. The console shows the transformation for the currently selected text, therefore it is easy to check both proofs and the created formal specification - it is only needed to select the interesting part of the text.
Z/Eves error reporting. When Z/Eves encounters an error in text interface, it is output to the console. A problem arises when a large quantity of commands is run in batch mode (e.g. reading a specification file). The error is reported when encountered and then the prover continues, thus it is very easy to loose the error and keep wondering what is the problem. The plug-in addresses this problem by opening an error dialog when such error is encountered.

Entered command list. Entering repetitive commands by hand can take a lot of effort. I have developed a Z Commands view, which captures the entered commands and keeps them in a list. The commands can be selected and copied in the list; or double-clicked to execute the command in Z/Eves immediately.

Z outline. Large Z specifications quickly become hard to navigate, when a specific schema or theorem needs to be found. I have extended the Texlipse plug-in for Eclipse within my plug-in to display Z schemas, theorems and definitions in an Eclipse Outline view. This allows quick navigation by selecting target paragraph in the outline. The plug-in is still a very rough prototype, however it has been very useful during the execution of this project. Furthermore, it was unofficially demonstrated in SBMF09 conference [60] and Dagstuhl Seminar 09381 [13] to a great reception.

I believe that this tool can evolve into a proper Z/Eves integration with Eclipse. Eclipse is one of the leading platforms in tool integration. For instance, I have typeset the formal specification and this report using Texlipse plugin for Eclipse [70], which provides an environment to typeset LaTeX documents in Eclipse platform. Woodcock et al. [78] notes that formal methods tool integration into existing tools for design, programming or static analysis is one of the goals towards automated verification.

Furthermore, the plug-in can be easily extended to provide aids for mechanisation by automatically generating proof obligations, which need to be proved, thus ensuring a better verification of formal specification, since less work is done manually. Another possible extension is providing project statistics, calculating definitions, proofs and commands as in Section 9.5. These extensions would take Z/Eves user interface to a modern platform.

The plug-in is called lt.velykis.zext (Z extensions) and is delivered as an electronic supplement to this thesis. Note that it is still in prototype stage and does not have any documentation, however that was not the intent of this project. I hope that I will find time to improve it to a good state or at least give the source code for dedicated people to maintain.
Bibliography


Appendix A

Figures
Figure A.1: Separation kernel architecture
Appendix B

Operation Partitioning Example

This example illustrates construction of operations by joining smaller operations and possible pitfalls when incomplete operations are totalled using disjunction.

B.1 Formal specification

Consider a state schema with 3 number variables and no invariants.

\[
\begin{align*}
\text{Numbers} & \triangleq x, y, z : \mathbb{Z}
\end{align*}
\]

Assume that we want to have an operation, which adjust these variables in a certain way. Using operation partitioning, we will instead define 3 separate operations for each variable, which can then be combined to a larger operation.

\[
\begin{align*}
\text{IncX} & \triangleq [\Delta \text{Numbers} \mid x' = x + 1] \\
\text{DoubleY} & \triangleq [\Delta \text{Numbers} \mid y' = y \ast 2] \\
\text{Plus5Z} & \triangleq [\Delta \text{Numbers} \mid z' = z + 5]
\end{align*}
\]

Let’s define an operation, which incorporates 2 of smaller increments and adds an arbitrary precondition, in order to produce a total operation with error cases afterwards.

\[
\begin{align*}
\text{UpdateNumbers0} \triangleq \\
\text{IncX} \\
\text{DoubleY} \\
x < 10
\end{align*}
\]

Then we define the error case to keep the state unchanged.

\[
\begin{align*}
\text{ErrGreater10} & \triangleq [\Xi \text{Numbers} \mid x \geq 10]
\end{align*}
\]

Finally, following the \text{Delta/Xi : disjoin errors} design pattern, we create the total operation:

\[
\text{UpdateNumbers} \triangleq \text{UpdateNumbers0} \lor \text{ErrGreater10}
\]

Now, we have an operation \text{UpdateNumbers} that changes the values of 2 variables within our \text{Numbers} schema. However, the remaining variable \(z\) is left unconstrained, therefore after this operation, its value can be anything as long as it satisfies the state invariant.

However, if the variable is important, its undefinedness is undesirable, because anything can happen to the variable. In this case, a correct solution is to complete the operation by setting values to all variables within state schema. However, a pitfall in this situation is to try and complete the operation with a conjunction over the totalled operation, e.g.

\[
\text{UpdateAllWrong} \triangleq \text{UpdateNumbers} \land \text{Plus5Z}
\]

When a total operation \text{UpdateNumbers} is expanded, the conjunction is distributive over disjunction and we get

\[
\text{UpdateAllWrongDisjoint} \equiv (\text{UpdateNumbers0} \land \text{Plus5Z}) \lor (\text{ErrGreater10} \land \text{Plus5Z})
\]
While the “success” case is correct, we get a \( \text{false} \) in the error case, because of “leave unchanged” \( (\exists \text{Numbers}) \) policy in \( \text{ErrGreater10} \):

\[
(z' = z \land z' = z + 5) \Rightarrow \text{false}
\]

This results in removal of error case from the total operation \( (P \lor \text{false} \equiv P) \), thus rendering the operation as no longer being total.

We formulate this fact as a theorem and prove it.

**Theorem tUpdateAllWrongNotTotal**

\[
\forall \text{Numbers} \mid x \geq 10 \bullet \text{UpdateAllWrong} \Rightarrow \text{false}
\]

The correct way is then to have a complete operation over all variables within the state schema, before using it in a disjunction with error cases, or within a schema composition. In our case, we replace \( \text{UpdateNumbers} \) with

\[
\text{UpdateAll} \cong (\text{UpdateNumbers0} \land \text{Plus5Z}) \lor \text{ErrGreater10}
\]

Now we investigate preconditions of the operations.

\[
\text{UpdateNumbersSig} \cong \text{Numbers}
\]

\[
\text{UpdateAllWrongSig} \cong [\text{Numbers} \mid x < 10]
\]

\[
\text{UpdateAllSig} \cong \text{Numbers}
\]

The precondition of \( \text{UpdateAllWrong} \) is not \text{true}, which means that joined operation made the operation no longer total.

**Theorem tUpdateNumbersPre**

\[
\forall \text{UpdateNumbersSig} \bullet \text{pre UpdateNumbers}
\]

**Theorem tUpdateAllWrongPre**

\[
\forall \text{UpdateAllWrongSig} \bullet \text{pre UpdateAllWrong}
\]

**Theorem tUpdateAllPre**

\[
\forall \text{UpdateAllSig} \bullet \text{pre UpdateAll}
\]

### B.2 Proof scripts

**Proof tUpdateAllWrongEquiv**

prove by reduce;

split \( x' = 1 + x \);

prove;

**Proof tUpdateAllWrongNotTotal**

prove by reduce;

**Proof tUpdateNumbersPre**

split \( x < 10 \);

cases;

prove by reduce;

instantiate \( z' = 1 \);

rewrite;

next;

prove by reduce;

next;

98
\textbf{proof}[t\text{Update}\text{AllWrongPre}]
\begin{itemize}
\item prove by reduce;
\end{itemize}

\textbf{proof}[t\text{Update}\text{AllPre}]
\begin{itemize}
\item prove by reduce;
\item split $x < 10$;
\item prove;
\end{itemize}
Appendix C

Transactional Operations Example

This example illustrates schema composition of total operations (with disjoined error cases) and how to achieve that success atomicity, i.e. that successful cases of operations are executed only when both operations succeed. This allows to avoid an occurrence when one operation succeeds while another fails.

C.1 Formal specification

C.1.1 State and component operations

The state schema contains the variables without any invariants.

\[
\begin{array}{l}
  \text{Numbers} \\
  x, y : \mathbb{Z}
\end{array}
\]

We also define an error state to store errors from each operation. We use \text{error}1 for first operation and \text{error}2 for the second.

\[
\text{Status} ::= \text{ok} \mid \text{error}1 \mid \text{error}2
\]

\[
\begin{array}{l}
  \text{ErrState} \\
  \text{err} : \text{Status}
\end{array}
\]

We define a simple operation that increments \(x\) and \(y\) when \(x < 10\).

\[
\begin{array}{l}
  \text{UpdateFirst0} \\
  \Delta \text{Numbers} \\
  \Delta \text{ErrState} \\
  x < 10 \\
  x' = x + 1 \\
  y' = y + 1 \\
  \text{err'} = \text{ok}
\end{array}
\]

We handle the opposite, when \(x \geq 10\), as an error case. It keeps the state unchanged.

\[
\begin{array}{l}
  \text{ErrFirst} \\
  \Xi \text{Numbers} \\
  \Delta \text{ErrState} \\
  x \geq 10 \\
  \text{err'} = \text{error}1
\end{array}
\]

Finally define a total operation for the first increment.

\[
\text{UpdateFirst} \equiv \text{UpdateFirst0} \lor \text{ErrFirst}
\]

In the same fashion, we define a similar operation to increment \(x\) and \(y\) with a different precondition.
C.1.2 Plain composition

Finally define a composed operation which executes the operations in a succession. The operations are joined with
schema composition (\(
\text{\textbf{\textup{\textbullet}}}
\)) because they both have constraints on the same variables. Such composition, if both cases
succeed, increments the variables twice - once for each operation.

\[\text{UpdateFullNoCheck} \equiv \text{UpdateFirst \text{\textbf{\textup{\textbullet}}} UpdateSecond}\]

**theorem** \(t_{\text{UpdateFullNoCheckSuccess}}\)
\[\forall \text{Numbers} \mid x < 10 \land y < 9 \bullet\]
\[\text{UpdateFullNoCheck} \Rightarrow (x' = x + 2 \land y' = y + 2 \land err' = ok)\]

If both operations fail, nothing is changed and error is reported.

**theorem** \(t_{\text{UpdateFullNoCheckFailBoth}}\)
\[\forall \text{Numbers} \mid x \geq 10 \land y \geq 10 \bullet\]
\[\text{UpdateFullNoCheck} \Rightarrow (x' = x \land y' = y \land err' = \text{error2})\]

However, for the state when only one success precondition is satisfied, the composed operation is no longer atomic -
one operation succeeds and changes \textit{Numbers} state, while the other handles an error state and does not change the
state.

We investigate a case when first operation fails.

**theorem** \(t_{\text{UpdateFullNoCheckFailFirst}}\)
\[\forall \text{Numbers} \mid x \geq 10 \land y < 10 \bullet\]
\[\text{UpdateFullNoCheck} \Rightarrow (x' = x + 1 \land y' = y + 1 \land err' = ok)\]

The full operation is not atomic, because the second operation was executed. Even more - it overrides error state and
loses the error. This is clearly a non-desirable result.

C.1.3 Composition with check for first operation failure

We can check whether the first operation has failed by adding a check for system state to allow the second operation
to execute. If the system is not in OK state, the second operation is not executed and error is unchanged.

We define operations to check for system state and propagate error in failure case. We use \(\Delta \text{ErrState}\) to allow it to
be joined with operations that change error.

\[\text{IsOk} \equiv [\Delta \text{ErrState} \mid err = ok]\]
\[\text{ErrNotOk} \equiv [\Xi \text{Numbers}; \Xi \text{ErrState} \mid \neg err = ok]\]
Now we can define composed operation, which checks whether the first operation failed.

\[ \text{UpdateFullCheckFirst} \triangleq \text{UpdateFirst} \circ (\text{IsOk} \land \text{UpdateSecond}) \lor \text{ErrNotOk} \]

We inspect the operation behaviour using similar theorems as before.

**Theorem** \( t_{\text{UpdateFullCheckFirstSuccess}} \)
\[ \forall \text{Numbers} \mid x < 10 \land y < 9 \bullet \text{UpdateFullCheckFirst} \Rightarrow (x' = x + 2 \land y' = y + 2 \land \text{err'} = \text{ok}) \]

**Theorem** \( t_{\text{UpdateFullCheckFirstFailBoth}} \)
\[ \forall \text{Numbers} \mid x \geq 10 \land y \geq 10 \bullet \text{UpdateFullCheckFirst} \Rightarrow (x' = x \land y' = y \land \text{err'} = \text{error}1) \]

**Theorem** \( t_{\text{UpdateFullCheckFirstFailFirst}} \)
\[ \forall \text{Numbers} \mid x \geq 10 \land y < 10 \bullet \text{UpdateFullCheckFirst} \Rightarrow (x' = x \land y' = y \land \text{err'} = \text{error}1) \]

If the first operation fails, the error is preserved to the end, which is the correct behaviour.

Such operation structure is simple and is useful for handling first operation errors in schema composition. It should be used when the second operation always succeeds. However, if the second operation has failure cases, such composition fails.

**Theorem** \( t_{\text{UpdateFullCheckFirstFailSecond}} \)
\[ \forall \text{Numbers} \mid x < 10 \land y \geq 9 \bullet \text{UpdateFullCheckFirst} \Rightarrow (x' = x + 1 \land y' = y + 1 \land \text{err'} = \text{error}2) \]

Even though the error is reported correctly, the state is changed by the first operation.

### C.1.4 Composition within transaction

To prevent first operation from execution if the second will fail, we propose to simulate a transaction of the operations. This means that we investigate the result of the operations and if the composed operation failed, the state is reverted to the original one. Otherwise, the change is approved.

To be able to revert to the initial state in the case of transaction failure, we define a new decorated state \( \text{Numbers''} \), which we use to store initial state. We define operations to backup the initial state into \( \text{Numbers''} \), using theta \( (\theta) \) expression. Then we define an operation to revert it to post-state \( \text{Numbers'} \) in the same manner.

\[
\begin{align*}
\text{NumbersBackup} & \triangleq [\Delta \text{Numbers}; \text{Numbers''} | \theta \text{Numbers} = \theta \text{Numbers''}] \\
\text{NumbersRevert} & \triangleq [\Delta \text{Numbers}; \text{Numbers''} | \theta \text{Numbers'} = \theta \text{Numbers''}]
\end{align*}
\]

Next we define transaction handling operations. They investigate system state and act accordingly. If no error has been produced, the operation results are kept with \( \Xi \) operators.

\[
\begin{align*}
\text{CompleteTransaction}0 & \triangleq [\Xi \text{Numbers}; \Xi \text{ErrState} | \text{err} = \text{ok}] \\
\text{ErrRevertTransaction} & \triangleq [\Delta \text{Numbers}; \Xi \text{ErrState}; \text{NumbersRevert} | \text{err} \neq \text{ok}]
\end{align*}
\]

If the operation failed and system is no longer in OK state, we revert the state to the backed up one using \( \text{NumbersRevert} \). Furthermore, we preserve the error state so that even though the data is reverted, we keep the log of what happened.

\[
\text{ErrRevertTransaction} \triangleq [\Delta \text{Numbers}; \Xi \text{ErrState}; \text{NumbersRevert} | \text{err} \neq \text{ok}]
\]

We disjoin these cases for a total transaction handling operation.

\[
\text{CompleteTransaction} \triangleq \text{CompleteTransaction}0 \lor \text{ErrRevertTransaction}
\]

Now we can define the full transactional operation. We reuse \( \text{UpdateFullCheckFirst} \) definition, because we need error propagation if the first operation fails. The initial state is backed up by \( \text{NumbersBackup} \) and results are handled using \( \text{CompleteTransaction} \).

\[
\begin{align*}
\text{UpdateFullTransaction} & \triangleq \exists \text{Numbers''} \bullet \\
& ( (\text{UpdateFullCheckFirst} \land \text{NumbersBackup}) \circ \text{CompleteTransaction})
\end{align*}
\]
The local intermediate backup state Numbers′′ is existentially quantified to hide it from when the transaction operation is used next.

We verify that the transactional definition handles second operation failure correctly - nothing is changed and error is indicated correctly.

**theorem** tUpdateFullTransactionFailSecond
\[ \forall \text{Numbers} \mid x < 10 \land y \geq 9 \bullet \]
\[ \text{UpdateFullTransaction \Rightarrow (x' = x \land y' = y \land err' = error2)} \]

We verify that the operation succeeds for other cases as well.

**theorem** tUpdateFullTransactionSuccess
\[ \forall \text{Numbers} \mid x < 10 \land y < 9 \bullet \]
\[ \text{UpdateFullTransaction \Rightarrow (x' = x + 2 \land y' = y + 2 \land err' = ok)} \]

**theorem** tUpdateFullTransactionFailBoth
\[ \forall \text{Numbers} \mid x \geq 10 \land y \geq 10 \bullet \]
\[ \text{UpdateFullTransaction \Rightarrow (x' = x \land y' = y \land err' = error1)} \]

**theorem** tUpdateFullTransactionFailFirst
\[ \forall \text{Numbers} \mid x \geq 10 \land y < 10 \bullet \]
\[ \text{UpdateFullTransaction \Rightarrow (x' = x \land y' = y \land err' = error1)} \]

Finally we can formulate a general theorem about transaction results. For any given state of Numbers, the transaction either succeeds with ok signal, or fails and does not change the state.

**theorem** tUpdateFullTransaction
\[ \forall \text{Numbers} \bullet \text{UpdateFullTransaction \Rightarrow (x' = x + 2 \land y' = y + 2 \land err' = ok)} \lor (x' = x \land y' = y \land err' \neq ok) \]

Finally we make sure that the transaction is a total operation - it’s precondition is true.

\[ \text{UpdateFullTransactionSig} \equiv [\text{Numbers}; \text{ErrState}] \]

**theorem** tUpdateFullTransactionPre
\[ \forall \text{UpdateFullTransactionSig} \bullet \text{pre \ UpdateFullTransaction} \]

### C.2 Proof scripts

**proof**[tUpdateFullNoCheckSuccess]
prove by reduce;

**proof**[tUpdateFullNoCheckFailBoth]
prove by reduce;

**proof**[tUpdateFullNoCheckFailFirst]
prove by reduce;

**proof**[tUpdateFullCheckFirstSuccess]
prove by reduce;
\textbf{proof}\texttt{[tUpdateFullCheckFirstFailBoth]}
prove by reduce;

\textbf{proof}\texttt{[tUpdateFullCheckFirstFailFirst]}
prove by reduce;

\textbf{proof}\texttt{[tUpdateFullCheckFirstFailSecond]}
prove by reduce;

\textbf{proof}\texttt{[tUpdateFullTransactionSuccess]}
prove by reduce;

\textbf{proof}\texttt{[tUpdateFullTransactionFailBoth]}
prove by reduce;

\textbf{proof}\texttt{[tUpdateFullTransactionFailFirst]}
prove by reduce;

\textbf{proof}\texttt{[tUpdateFullTransactionFailSecond]}
prove by reduce;

\textbf{proof}\texttt{[tUpdateFullTransaction]}\texttt{[tUpdateFullTransactionPre]}
prove by reduce;
split $x < 10$;
prove;
split $1 + y < 10$;
prove;
Appendix D

General Lemmas

This chapter features general lemmas, which extend the Z/Eves Mathematical Toolkit and simplify further proofs. These lemmas have been defined by Leo Freitas in previous GC pilot projects. The proofs of our lemmas and theorems involve these lemmas, therefore they need to be specified at the beginning.

We have separated these lemmas from the main formal specification to show distinctively that these lemmas have been created and proven by Leo Freitas and give credit where its due.

D.1 General theory lemmas

\textbf{theorem} disabled rule lRanElemType \[ X, Y \]
\[ \forall A : \mathbb{P} \ X; \ B : \mathbb{P} \ Y \bullet \forall R : A \leftrightarrow B \mid (x, y) \in R \bullet y \in B \]

\textbf{theorem} disabled rule lRelWeakening \[ X, Y \]
\[ R \in X \leftrightarrow Y \leftrightarrow R \in \mathbb{P} \ (X \times Y) \land (\forall \text{elem} : R \bullet \text{elem} \in X \times Y) \]

\textbf{theorem} disabled rule lPFunPointIsPfunElem \[ X, Y \]
\[ \forall x : X; \ y : Y; \ f : X \rightarrow Y \mid x \in \text{dom} \ f \land y = fx \bullet (x, y) \in f \]

\textbf{theorem} disabled rule lPFunWeakening \[ X, Y \]
\[ f \in X \rightarrow Y \leftrightarrow f \in X \leftrightarrow Y \land (\forall d : X; \ r1, r2 : Y \mid (d, r1) \in f \land (d, r2) \in f \bullet r1 = r2) \]

\textbf{theorem} disabled rule lPFunWeakeningFresh \[ X, Y \]
\[ f \in X \rightarrow Y \leftrightarrow f \in X \leftrightarrow Y \land (\forall df : X; \ rf1, rf2 : Y \mid (df, rf1) \in f \land (df, rf2) \in f \bullet rf1 = rf2) \]

\textbf{theorem} disabled rule lPInjWeakening \[ X, Y \]
\[ f \in X \rightarrow Y \leftrightarrow f \in X \rightarrow Y \land (\forall r : Y; \ d1, d2 : X \mid (d1, r) \in f \land (d2, r) \in f \bullet d1 = d2) \]

\textbf{theorem} disabled rule lPInjWeakeningFresh \[ X, Y \]
\[ f \in X \rightarrow Y \leftrightarrow f \in X \rightarrow Y \land (\forall rf : Y; \ df1, df2 : X \mid (df1, rf) \in f \land (df2, rf) \in f \bullet df1 = df2) \]

\textbf{theorem} disabled rule lBijectionFinite \[ X, Y \]
\[ \forall A : \mathbb{F} \ X; \ B : \mathbb{P} \ Y \bullet \forall f : A \rightarrow B \bullet f \in A \rightarrow B \land B \in \mathbb{F} \ Y \land \# A = \# B = \# f \]

\textbf{theorem} disabled rule lNonMaximalCardEquiv \[ X \]
\[ \forall A : \mathbb{P} \ X \bullet \forall S : \mathbb{F} \ A \bullet \# S = \#[A] S \]

\textbf{theorem} disabled rule lNonMaxDomEquiv \[ X, Y \]
\[ \forall A : \mathbb{P} \ X; \ B : \mathbb{P} \ Y \bullet \forall R : A \leftrightarrow B \bullet \text{dom}[X, Y] \ R = \text{dom}[A, B] R \]

\textbf{theorem} disabled rule lSeqWeakening \[ X \]
\[ s \in \text{seq} \ X \leftrightarrow s \in N \rightarrow X \land (\exists n : N \bullet \text{dom} \ [Z, X] s = 1 \ldots n) \]

\textbf{theorem} disabled rule lSeqElemType \[ X \]
\[ \forall A : \mathbb{P} \ X \bullet \forall s : \text{seq} \ A \mid e \in s \bullet e \in N_1 \times A \]
\begin{align*}
\text{theorem disabled rule lSeqRanElemType } [X] & \\
\forall A : \mathbb{P} X \cdot \forall s : \text{seq } A \mid (i, y) \in s \cdot y \in A
\end{align*}

\begin{align*}
\text{theorem disabled rule lSeqNonEmptySize } [X] & \\
\forall s : \text{seq } X \mid \neg s = \langle \rangle \cdot 1 \leq \#s
\end{align*}

\begin{align*}
\text{theorem disabled rule lSeqDomEquiv } [X] & \\
\forall s : \text{seq } X \cdot \text{dom } s \setminus \{ \#s \} = 1 \ldots \#s - 1
\end{align*}

\begin{align*}
\text{theorem disabled rule llSeqWeakening } [X] & \\
s \in \text{iseq } X \Leftrightarrow s \in \text{seq } X \wedge s \in \mathbb{N} \hookrightarrow X
\end{align*}

\begin{align*}
\text{theorem disabled rule lSeqCatLastLeftApply } [X] & \\
\forall s, t : \text{seq } X \mid \neg s = \langle \rangle \wedge \neg t = \langle \rangle \cdot (s \triangle t)(\#s) = \text{lasts}
\end{align*}

\begin{align*}
\text{theorem disabled rule lSeqCatHeadRightApply } [X] & \\
\forall s, t : \text{seq } X \mid \neg s = \langle \rangle \wedge \neg t = \langle \rangle \cdot (s \triangle t)(1 + \#s) = \text{head } t
\end{align*}

\begin{align*}
\text{theorem disabled rule lTailElemInDom } [X] & \\
\forall Y : \mathbb{P} X \cdot \forall s : \text{seq } Y \mid \neg s = \langle \rangle \wedge (i, a) \in \text{tail } s \cdot i + 1 \in \text{dom } s
\end{align*}

\begin{align*}
\text{theorem disabled rule lTailSeqIsISeq } [X] & \\
\forall Y : \mathbb{P} X \cdot \forall s : \text{iseq } Y \mid \neg s = \langle \rangle \cdot \text{tail } s \in \text{iseq } Y
\end{align*}

\begin{align*}
\text{theorem disabled rule lTailDom } [X] & \\
\forall s : \text{seq } X \mid \neg s = \langle \rangle \cdot \text{dom } (\text{tail } s) = 1 \ldots -1 + \#s
\end{align*}

\begin{align*}
\text{theorem disabled rule lTailElemInRanTail } [X] & \\
\forall s : \text{seq } X \mid \neg s = \langle \rangle \wedge (i, y) \in \text{tail } s \cdot y \in \text{ran } (\text{tail } s)
\end{align*}

\begin{align*}
\text{theorem disabled rule lTailRanSubset } [X] & \\
\forall s : \text{seq } X \mid \neg s = \langle \rangle \wedge y \in \text{ran } (\text{tail } s) \cdot y \in \text{ran } s
\end{align*}

\begin{align*}
\text{theorem disabled rule lUnitBelongs } & \\
(i, a) \in \langle x \rangle \Leftrightarrow i = 1 \wedge a = x
\end{align*}

\begin{align*}
\text{theorem disabled rule lUnitElemBelongs } & \\
e \in \langle x \rangle \Leftrightarrow e = (1, x)
\end{align*}

\begin{align*}
\text{theorem disabled rule lCatLeftLastEquiv } [X] & \\
\forall r, s, t : \text{seq } X \mid r = s \triangle t \wedge \neg s = \langle \rangle \cdot \text{last } s = r (\#s)
\end{align*}

\begin{align*}
\text{theorem disabled rule lCatRightHeadEquiv } [X] & \\
\forall r, s, t : \text{seq } X \mid r = s \triangle t \wedge \neg t = \langle \rangle \cdot \text{head } t = r (1 + \#s)
\end{align*}

\begin{align*}
\text{theorem disabled rule lSizeCatEquals } [X] & \\
\forall s, t : \text{seq } X \mid r = s \triangle t \cdot \# s + \# t = \# r
\end{align*}

\begin{align*}
\text{theorem disabled rule lDomCat } [X] & \\
\forall s, t : \text{seq } X \cdot \text{dom } (s \triangle t) = 1 \ldots \# s + \# t
\end{align*}
D.2 Arithmetic lemmas

**Theorem** disabled rule lLessNeg
\[
\forall i,j:\mathbb{Z} \mid i \geq j \Rightarrow i < j
\]

**Theorem** disabled rule lGreaterNeg
\[
\forall i,j:\mathbb{Z} \mid i \leq j \Rightarrow i > j
\]

**Theorem** disabled rule lEqNeg
\[
\forall i,j:\mathbb{Z} \mid i > j \Rightarrow i \leq j
\]

**Theorem** disabled rule lGeqNeg
\[
\forall i,j:\mathbb{Z} \mid i < j \Rightarrow i \geq j
\]

**Theorem** rule lLessFlip
\[
\forall i,j:\mathbb{Z} \mid j > i \Rightarrow i < j
\]

**Theorem** disabled rule lGreaterFlip
\[
\forall i,j:\mathbb{Z} \mid j < i \Rightarrow i > j
\]

**Theorem** disabled rule lEqFlip
\[
\forall i,j:\mathbb{Z} \mid j \geq i \Rightarrow i \leq j
\]

**Theorem** disabled rule lGeqFlip
\[
\forall i,j:\mathbb{Z} \mid j \leq i \Rightarrow i \geq j
\]

**Theorem** disabled rule lLessPromote
\[
\forall i,j:\mathbb{Z} \mid i \geq 1 + j \Rightarrow i < j
\]

**Theorem** disabled rule lGreaterPromote
\[
\forall i,j:\mathbb{Z} \mid i \geq 1 + j \Rightarrow i > j
\]
Appendix E

Formal Specification

E.1 General lemmas

This section defines several lemmas, which extend the Z/Eves Mathematical Toolkit and together with Leo Freitas’ lemmas simplify the proofs of the formal specification.

Theorem lInverseIsInj \[[X, Y]\]
\[
\forall A : \mathbb{P} X; B : \mathbb{P} Y \bullet \forall P : A \rightarrow B; Q : B \rightarrow A \bullet
P = Q^\sim \Rightarrow P \in A \Rightarrow B \land Q \in B \Rightarrow A
\]

Theorem disabled rule lOverrideDisjointDom \[[X, Y]\]
\[
\forall A : \mathbb{P} X; B : \mathbb{P} Y \bullet \forall P, Q : A \rightarrow B \mid \text{dom } P \cap \text{dom } Q = \{\} \bullet P \oplus Q = P \cup Q
\]

Theorem disabled rule lNonEmptySeqHasHead \[[X]\]
\[
\forall Y : \mathbb{P} X \bullet \forall s : \text{seq}_{1} Y \bullet \text{head } s \in \text{ran } s
\]

Theorem disabled rule lDisjointSeqHead \[[X]\]
\[
\forall Y : \mathbb{P} X \bullet \forall s, t : \text{seq}_{1} Y \mid \text{ran } s \cap \text{ran } t = \{\} \land \neg s = \{\} \bullet \neg \text{head } s \in \text{ran } t
\]

Theorem disabled rule lHeadRanSubset \[[X]\]
\[
\forall s : \text{seq}_{1} X \mid \neg s = \{\} \land y = \text{head } s \bullet y \in \text{ran } s
\]

Theorem grule gEmptyRan \[[X, Y]\]
\[
\forall A : \mathbb{P} X; B : \mathbb{P} Y \bullet \forall P : A \leftrightarrow B \bullet \text{ran}_{[X, Y]} P = \{\} \Rightarrow P = \{\}
\]

Theorem grule gEmptyDom \[[X, Y]\]
\[
\forall A : \mathbb{P} X; B : \mathbb{P} Y \bullet \forall P : A \leftrightarrow B \bullet \text{dom}_{[X, Y]} P = \{\} \Rightarrow P = \{\}
\]

Theorem disabled rule lElemNotInDomNrres \[[X, Y]\]
\[
\forall P : X \rightarrow Y; R : \mathbb{P} Y \bullet \neg x \in \text{dom } P \Rightarrow \neg x \in \text{dom } (P \triangleright R)
\]

E.2 Basic types and their properties

YESNO ::= yes | no

Theorem tPIDConsistency
\[
\exists \text{minpid, maxpid} : \mathbb{N} \bullet \text{minpid} \leq \text{maxpid}
\]

\[
\text{minpid}, \text{maxpid} : \mathbb{N}
\]

\[
\langle \text{rule lRangePID} \rangle \text{minpid} \leq \text{maxpid}
\]
\[ \text{PID} == \text{minpid} \ldots \text{maxpid} \]

\textbf{theorem} grule gPIDMaxType  
\[ \text{PID} \in \mathbb{P} \mathbb{Z} \]

\textbf{theorem} grule gPIDFinType  
\[ \text{PID} \in \mathbb{F} \mathbb{Z} \]

\textbf{theorem} grule gPIDIsFinset  
\[ \text{PID} \in \mathbb{F} \text{PID} \]

\textbf{theorem} rule lMinpidIsPID  
\[ \text{minpid} \in \text{PID} \]

\textbf{theorem} grule gPIDNotEmpty  
\[ \neg \text{PID} = \{\} \]

\textbf{theorem} tGPIDConsistency  
\[ \exists \text{nullpid} : \mathbb{N} \bullet \forall p : \text{PID} \bullet p < \text{nullpid} \]

\[ \begin{array}{c}
\text{nullpid} : \mathbb{N} \\
\langle \text{rule lRangeGPID} \rangle \forall p : \text{PID} \bullet p < \text{nullpid}
\end{array} \]

\[ \text{GPID} == \{\text{nullpid}\} \cup \text{PID} \]

\textbf{theorem} grule gGPIDMaxType  
\[ \text{GPID} \in \mathbb{P} \mathbb{Z} \]

\textbf{theorem} tNullpidNotPID  
\[ \text{nullpid} \notin \text{PID} \]

\[ \text{UPID} == \mathbb{N} \]

\textbf{theorem} grule gUPIDMaxType  
\[ \text{UPID} \in \mathbb{P} \mathbb{Z} \]

\[ \text{PTYPE} ::= \text{uproc} \mid \text{dproc} \]

Updated to have \( \leq \) instead of \( < \).

\textbf{theorem} tDevNoConsistency  
\[ \exists \text{mindev}, \text{maxdev} : \mathbb{N} \bullet \text{mindev} \leq \text{maxdev} \]

\[ \begin{array}{c}
\text{mindev}, \text{maxdev} : \mathbb{N} \\
\langle \text{rule lRangeDevNo} \rangle \text{mindev} \leq \text{maxdev}
\end{array} \]

\[ \text{DevNo} == \text{mindev} \ldots \text{maxdev} \]

\textbf{theorem} grule gDevNoType  
\[ \text{DevNo} \in \mathbb{P} \mathbb{Z} \]
\textbf{PSTATE} ::= \textit{psterm} \\
| \textit{psrunning} \\
| \textit{psready} \\
| \textit{psdevwait} \\
| \textit{pswtgdev}

\textbf{theorem} \textit{tAddrConsistency} \\
\exists \textit{nulladdr}, \textit{maxaddr} : \mathbb{N} \cdot \textit{nulladdr} = 0 \land \textit{nulladdr} < \textit{maxaddr}

\textit{nulladdr}, \textit{maxaddr} : \mathbb{N} \\
\textlangle disabled rule lRangeNullAddr \textrangle \textit{nulladdr} = 0 \\
\textlangle disabled rule lRangeMaxAddr \textrangle \textit{nulladdr} < \textit{maxaddr}

\textit{Addr} == \textit{nulladdr} .. \textit{maxaddr}

\textbf{theorem} \textit{grule gAddrType} \\
\textit{Addr} \in \mathbb{P} \mathbb{Z}

\textbf{theorem} \textit{grule gNulladdrInAddr} \\
\textit{nulladdr} \in \textit{Addr}

[\textit{PSU}]

[\textit{MSG, MSGDATA}]

\textit{nullmsg} : \textit{MSG} \\
\textit{nullmsgdata} : \textit{MSGDATA}

\textbf{SYSOPCODE} ::= \textit{newuproc} \\
| \textit{suspsel} \\
| \textit{termpid} \\
| \textit{sndmsg} \\
| \textit{gotmsgs} \\
| \textit{gotmsgfromsrc} \\
| \textit{nextmsg} \\
| \textit{nextmsgfromsrc} \\
| \textit{devrequest}

To \textit{SYSSERR} added \textit{alreadyqueued}, \textit{notuserpid}, \textit{notdevicepid}, \textit{badpidcurr}, \textit{badpididle}. 
SYSERR ::= sysok |
    unusedpd |
    pdinuse |
    ptabfull |
    emptyqueue |
    alreadyqueued |
    notuserpid |
    notdevicepid |
    badpidcurr |
    badpididle |
    nospaceinstore |
    blocklocerror |
    badblockaddr |
    msgqfull |
    emptymsgq |
    nomsysfrom |
    calleridmismatch |
    mainstorefull |
    badmsgdest |
    nodevreply |
    baddevnum |
    badcallerid

ErrV
serr : SYSERR

ErrVInit

ErrV' = sysok

SetSysErr

ΔErrV

e? : SYSERR

serr' = e?

SysErr

ΞErrV

e! : SYSERR

e! = serr

SysOk ≡ [SetSysErr | e? = sysok] \ (e?)

SysOkOriginal ≡ (∃ e : SYSERR | e = sysok • SetSysErr[e/e?])

theorem tSysOkEquiv

SysOk ⇔ SysOkOriginal

SysOkExpand

ΔErrV

serr' = sysok
theorem tSysOkExpandEquiv

\[
\begin{align*}
\text{SysOk} & \Leftrightarrow \text{SysOkExpand} \\
\text{IsSysOk} & \equiv \text{ErrV} \\
serr & = \text{sysok}
\end{align*}
\]

E.2.1 Operation precondition signatures

\[
\begin{align*}
\text{SetSysErrSig} & \equiv [\text{ErrV}; e? : \text{SYSERR} | \text{true}] \\
\text{SysErrSig} & \equiv [\text{ErrV} | \text{true}] \\
\text{SysOkSig} & \equiv \text{SysErrSig} \\
\text{IsSysOkSig} & \equiv [\text{ErrV} | \text{serr} = \text{sysok}]
\end{align*}
\]

E.2.2 Operation preconditions

\[
\begin{align*}
\text{theorem tErrVInit} & \equiv \exists \text{ErrV} \bullet \text{ErrVInit} \\
\text{theorem tSetSysErrPre} & \equiv \forall \text{SetSysErrSig} \bullet \text{pre SetSysErr} \\
\text{theorem tSysErrPre} & \equiv \forall \text{SysErrSig} \bullet \text{pre SysErr} \\
\text{theorem tSysOkPre} & \equiv \forall \text{SysOkSig} \bullet \text{pre SysOk} \\
\text{theorem tIsSysOkPre} & \equiv \forall \text{IsSysOkSig} \bullet \text{pre IsSysOk}
\end{align*}
\]

E.3 Hardware considerations

Need to reorder the specification to have the definition of \text{IntNo} before \text{HW}.

We need strict less than, because there are at least 2 distinct interrupt numbers defined.

\[
\begin{align*}
\text{theorem tIntNoConsistency} & \equiv \exists \text{minint}, \text{maxint} : \mathbb{N} \bullet \text{minint} < \text{maxint} \\
\text{IntNo} & = \langle \text{disabled rule lRangeIntNo} \rangle \langle \text{minint} < \text{maxint} \rangle \\
\text{IntNo} & = \text{minint} .. \text{maxint}
\end{align*}
\]

\[
\begin{align*}
\text{theorem grule gIntNoType} & \equiv \text{IntNo} \in \mathbb{P} \mathbb{Z} \\
\text{theorem rule lMinintIsIntNo} & \equiv \text{minint} \in \text{IntNo}
\end{align*}
\]
[TSS0]

\[ TSS : \mathbb{P}_1 \to TSS0 \]

**theorem** ITSSNotEmpty

\[ \exists t : TSS \bullet true \]

Added intno because it's used later and not indicated in original HW specification.

\[
\begin{array}{l}
\text{HW} \\
\text{intno : IntNo}
\end{array}
\]

We do not care about the initial interrupt value - it can be set to anything, because the interrupt handling is done after \text{RaiseInterrupt} operation, which always sets its interrupt value.

\[
\begin{array}{l}
\text{HWinit} \\
\text{HW'}
\end{array}
\]

\[
\begin{array}{l}
\text{RaiseInterrupt} \\
\Delta HW \\
\text{ino? : IntNo} \\
\text{intno'} = \text{ino}?
\end{array}
\]

**theorem** tInterruptsConsistency

\[ \exists \text{killintno}, \text{ctxtswintno} : \text{IntNo} \bullet \sim \text{killintno} = \text{ctxtswintno} \]

\[
\begin{array}{l}
\text{killintno} : \text{IntNo} \\
\text{ctxtswintno} : \text{IntNo}
\end{array}
\]

\[ \langle \text{rule } \text{IDistinctIntNo} \rangle \sim \text{killintno} = \text{ctxtswintno} \]

**theorem** grule gKillintnoType

\[ \text{killintno} \in \mathbb{Z} \]

**theorem** grule gctxtswintnoType

\[ \text{ctxtswintno} \in \mathbb{Z} \]

\[ \text{RaiseKillInterrupt} \equiv [\text{RaiseInterrupt} | \text{ino?} = \text{killintno}] \setminus \text{ino}? \]

\[ \text{RaiseKillInterruptOriginal} \equiv \exists \text{ino} : \text{IntNo} | \text{ino} = \text{killintno} \bullet \text{RaiseInterrupt}[\text{ino}/\text{ino}?] \]

**theorem** tRaiseKillInterruptEquiv

\[ \text{RaiseKillInterrupt} \Leftrightarrow \text{RaiseKillInterruptOriginal} \]

\[ \text{CtxtSw} \equiv [\text{RaiseInterrupt} | \text{ino?} = \text{ctxtswintno}] \setminus \text{ino}? \]

\[ \text{CtxtSwOriginal} \equiv \exists \text{ino} : \text{IntNo} | \text{ino} = \text{ctxtswintno} \bullet \text{RaiseInterrupt}[\text{ino}/\text{ino}?] \]

**theorem** tCtxtSwEquiv

\[ \text{CtxtSw} \Leftrightarrow \text{CtxtSwOriginal} \]

In this specification, all error conditions raise kill interrupt, thus affecting \text{HW} state. However, when total operations are defined, the successful branch only signals \text{SysOk} and does not alter \text{HW} state at all. This causes problems when combined schemas are defined, therefore we need to define an operation to be used together with \text{SysOk}, which keeps \text{HW} in the same state as previously.

\[ \text{RaiseError} \equiv \text{SetSysErr} \land \text{RaiseKillInterrupt} \]

\[ \text{RaiseOk} \equiv [\text{SysOk}; \exists \text{HW}] \]
E.3.1 Operation precondition signatures

\[ \text{RaiseInterruptSig} \equiv [\text{HW}; \text{ino} : \text{IntNo} \mid \text{true}] \]

\[ \text{RaiseNamedInterruptSig} \equiv [\text{HW} \mid \text{true}] \]

\[ \text{RaiseKillInterruptSig} \equiv \text{RaiseNamedInterruptSig} \]

\[ \text{CtxtSwSig} \equiv \text{RaiseNamedInterruptSig} \]

\[ \text{RaiseErrorSig} \equiv [\text{HW}; \text{SetSysErrSig}] \]

\[ \text{RaiseOkSig} \equiv [\text{HW}; \text{SysOkSig}] \]

E.3.2 Operation preconditions

\text{theorem} \; \text{tHWInit}
\exists \; \text{HW}' \; \bullet \; \text{HWInit}

\text{theorem} \; \text{tRaiseInterruptPre}
\forall \; \text{RaiseInterruptSig} \; \bullet \; \text{pre} \; \text{RaiseInterrupt}

\text{theorem} \; \text{tRaiseKillInterruptPre}
\forall \; \text{RaiseKillInterruptSig} \; \bullet \; \text{pre} \; \text{RaiseKillInterrupt}

\text{theorem} \; \text{tCtxtSwPre}
\forall \; \text{CtxtSwSig} \; \bullet \; \text{pre} \; \text{CtxtSw}

\text{theorem} \; \text{tRaiseErrorPre}
\forall \; \text{RaiseErrorSig} \; \bullet \; \text{pre} \; \text{RaiseError}

\text{theorem} \; \text{tRaiseOkPre}
\forall \; \text{RaiseOkSig} \; \bullet \; \text{pre} \; \text{RaiseOk}

E.4 Message queues

These schemas are used in \textit{PTab}, therefore should be declared before.

The message pointer type, \textit{MPtr} is encountered in modelling messaging subsystem of the separation kernel. Craig specifies only \textit{MPtr} \subset \textit{Addr} constraint in the original specification. We specify this data type properly using axiomatic definition and adding Craig’s restriction.

\[ \text{MPtr} \; : \; \mathbb{P} \; \textit{Addr} \]
\[ \langle \text{rule lMPtrDef} \rangle \; \text{MPtr} \subset \textit{Addr} \]

The message queue \textit{MsgQ} contains a sequence \textit{mq} of message pointers \textit{MPtr}, which size is limited to the maximum number of messages \textit{maxMs}.

\[ \text{MsgQ} \]
\[ \text{mq} : \text{seq} \; \text{MPtr} \]
\[ \text{maxMs} : \mathbb{N}_1 \]
\[ \# \; \text{mq} \leq \text{maxMs} \]
Message queue is initialised with an input size and no messages. Initialisation is necessary for process allocation in \( PTab \), because when a new process is created, its message queue is initialised.

\[
\begin{array}{l}
\text{\textit{MsgQInit}} \\
\text{\textit{MsgQ'}} \\
\text{\textit{maxMs? : N}}_1 \\
\text{\textit{maxMs'} = maxMs?} \\
\text{\textit{mq'} = ()}
\end{array}
\]

Prove initialisation theorem for message queue.

\[ \text{\textbf{theorem} tMsgQInit} \] 

\[ \exists \text{\textit{MsgQ'}}; \text{\textit{maxMs? : N}}_1 \bullet \text{\textit{MsgQInit}} \]

### E.5 Memory store segments

The process occupied memory is referenced by indicating code and stack segments. A segment descriptor \( SDesc \) type is defined as a cartesian product of \( Addr \) and natural number set \( \mathbb{N} \).

\[ SDesc == Addr \times \mathbb{N} \]

\[ \text{\textbf{theorem}} \text{\textit{grule gSDescType}} \]

\[ SDesc \in P (\mathbb{Z} \times \mathbb{Z}) \]

Such definition shows that a segment descriptor consists of a tuple: start of segment address of \( Addr \) type and natural number segment size.

The \( SDesc \) data is initialised using a constructor function out of its components. It is necessary for process allocation in \( PTab \), because when a new process is created, its code and data/stack segments are initialised.

\[ \text{\textit{mkSDesc}}: Addr \times \mathbb{N} \rightarrow SDesc \]

\[ \text{(disabled rule lSDescFunDef)} \]

\[ \forall a : Addr; s : \mathbb{N} \bullet \text{\textit{mkSDesc}}(a, s) = (a, s) \]

During the proofs we have encountered several cases, when automation rules are necessary about \( SDesc \).

\[ \text{\textbf{theorem}} \text{\textit{grule gSDescFunRelType}} \]

\[ \text{\textit{mkSDesc}} \in \mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z} \times \mathbb{Z} \]

\[ \text{\textbf{theorem}} \text{\textit{disabled rule lSDescTotalFun}} \]

\[ \forall a : Addr; s : \mathbb{N} \bullet (a, s) \in \text{\textit{dom}} \text{\textit{mkSDesc}} \]

### E.6 Process table

#### E.6.1 State

The process table state schema, \( PTab \), contains kernel’s processes and their related information. The original \( PTab \) schema contains syntax and structuring errors, redundant and missing statements. The schema syntax can be easily simplified for the benefit of both the reader and the person modelling the specification.

We correct the original schema into \( PTabOriginal \) and then update it in small steps, in order to see the full progress. With each change, we show the schema equivalence. Where equivalence is not possible (due to additional predicate), we show a schema refinement. The full \( PTab \) evolution can be shown as:

\[ PTab \Rightarrow PTabv_4 \Leftrightarrow PTabv_3 \Leftrightarrow PTabv_2 \Rightarrow PTabv_1 \Leftrightarrow PTabOriginal \]

This chain shows that most transformations produced an equivalent version of the schema. The schemas together with all justification are provided below.
Original schema

The following changes have been performed to correct the definition of $PTab$ in the original specification:

- Corrected a name error: $devs$ was used in one place instead of $dprocs$. Made all cases to be $dprocs$.

- Predicates ran $extpid = uprocs$ and $pidext = uprocs$ must be added to the conjunction in $\exists devs, uprocs \ldots$ predicate, as the name $uprocs$ exists in the predicate only.

- Predicates $used = dom \ state$, $used = dom \ ptype$ and $used = dom \ tss$ are concerned with the whole $used$ set. They do not depend on either $dprocs$ or $uprocs$, therefore there is no need to specify them in $\exists devs, uprocs \ldots$ predicate. Moreover, $used = dom \ ptype$ is required outside $\exists \ldots$ predicate to prove schema’s domain check.

- Added $free : \mathbb{P} \ PID \ | \ free = PID \ \setminus \ used$ definition to schema, because it was declared outside the schema in the original specification, which is a syntax error.

- Removed $devrqs$ variable from original specification. The variable is not used anywhere in the specification. The variable to possibly store device requests (trying to decode the name), it is very likely had been obsoleted by $devmsg$, which is also used when device request is made. Craig has made a number of typos in the original specification and this is very likely to be one of them.

The original process table schema with these corrections is shown as $PTab_{Original}$.

<table>
<thead>
<tr>
<th>$PTab_{Original}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nextupid : UPID$</td>
</tr>
<tr>
<td>$extpid : UPID \rightarrow PID$</td>
</tr>
<tr>
<td>$pidext : PID \rightarrow UPID$</td>
</tr>
<tr>
<td>$used, free : \mathbb{F} \ PID$</td>
</tr>
<tr>
<td>$tss : PID \rightarrow TSS$</td>
</tr>
<tr>
<td>$devmap : DevNo \rightarrow PID$</td>
</tr>
<tr>
<td>$state : PID \rightarrow PSTATE$</td>
</tr>
<tr>
<td>$ptype : PID \rightarrow PTYPE$</td>
</tr>
<tr>
<td>$msgq : PID \rightarrow \mathbb{M}sgQ$</td>
</tr>
<tr>
<td>$devmsg : PID \rightarrow (GPID \times MSG)$</td>
</tr>
<tr>
<td>$devrpy : PID \rightarrow MSG$</td>
</tr>
<tr>
<td>$cdseg : PID \rightarrow SDesc$</td>
</tr>
<tr>
<td>$dsseg : PID \rightarrow SDesc$</td>
</tr>
</tbody>
</table>

$free = PID \ \setminus \ used$

$used = dom \ state = dom \ ptype = dom \ tss$

$\exists dprocs, uprocs : \mathbb{F} \ PID \ |
\hspace{1cm} dprocs = \{p : PID \ | p \in used \land ptype(p) = dproc\} \land$

$\hspace{1cm} uprocs = \{p : PID \ | p \in used \land ptype(p) \neq dproc\} \bullet$

$\hspace{1cm} dprocs = ran \ devmap = dom \ devmsg = dom \ devrpy \land$

$\hspace{1cm} uprocs = dom \ cdseg = dom \ dsseg = dom \ msgq = dom \ pidext = ran \ extpid$

$\hspace{1cm} pidext = extpid$\~$

$\forall d : DevNo \bullet d \in dom \ devmap \Rightarrow (\exists_1 p : PID \bullet p = devmap(d))$

Redundant predicate about device numbers

The last predicate in original schema $PTab_{Original}$ is redundant, because its meaning simply repeats the same that is defined by the meaning of a function - that for each element in function domain there exists only one value.

We prove this in $lDevmapPFunImpliesUnique$ below. This means that we can remove the predicate altogether in schema $PTab_1$. We prove that the predicate was redundant by proving equivalence in $tPTab_{Original\_1Equiv}$. 

116
We need the following assumption rules for the automation of proof.

\textbf{theorem} grule gDevmapRelType
\begin{align*}
devmap & \in DevNo \mapsto PID \Rightarrow devmap \in Z \leftrightarrow Z
\end{align*}

\textbf{theorem} grule gDevmapPFunType
\begin{align*}
devmap & \in DevNo \mapsto PID \Rightarrow devmap \in Z \mapsto Z
\end{align*}

We prove that the redundant predicate can be implied from the partial function definition.

\textbf{theorem} disabled rule lDevmapPFunImpliesUnique
\begin{align*}
devmap & \in DevNo \mapsto PID \Rightarrow \\
(\forall d : DevNo \mid d \in \text{dom} \ devmap \bullet (\exists_1 p : PID \bullet p = devmap(d)))
\end{align*}

We prove that the updated schema is equivalent to the original one.

\textbf{theorem} tPTabOriginalv1Equiv
\begin{align*}
PTabOriginal & \Leftrightarrow PTabv1
\end{align*}

\textbf{Device number restrictions}

We think that Craig made a mistake in the original predicate about device numbers, which was removed as redundant. We assume that he actually wanted to indicate that each device process has a unique device number.

This intention can be deduced from the device number allocation in operations \textit{InitDeviceNum} and \textit{NewDeviceProcess} in the original specification. We can see there that the devmap variable is updated only in the cases when the device number \( d : DevNo \mid d \notin \text{dom} \ devmap \).

Therefore in schema \textit{PTabv2} we introduce a new predicate, which is very similar to the original one, but instead enforces that no 2 device numbers can point to the same process - for each device process \( PID \) there is a unique DevNo.
The introduction of new predicate means that the new $PTabv2$ schema is not equivalent to the previous evolution $PTabv1$. However, we can prove implication $PTabv2 \Rightarrow PTabv1$ to show that the new schema is a refinement of the previous one.

**Theorem** $tPTabv1v2Refinement$

$PTabv2 \Rightarrow PTabv1$

### Partial injection function

Next we show that the introduced predicate about unique device number for each device process identifier can be written as a partial injection ($\Rightarrow \mapsto$) function. We remove the predicate and change definition to $devmap : DevNo \mapsto PID$. Then we prove the equivalence between schemas.

$PTabv3$

<table>
<thead>
<tr>
<th>nextupid : UPID</th>
<th>extpid : UPID $\Rightarrow$ PID</th>
<th>pidext : PID $\Rightarrow$ UPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>used, free : $\mathbb{F}$ PID</td>
<td>tss : PID $\Rightarrow$ TSS</td>
<td>devmap : DevNo $\Rightarrow$ PID</td>
</tr>
<tr>
<td>state : PID $\Rightarrow$ PSTATE</td>
<td>ptype : PID $\Rightarrow$ PTYPE</td>
<td>msgq : PID $\Rightarrow$ MsgQ</td>
</tr>
<tr>
<td>devmsg : PID $\Rightarrow$ (GPID $\times$ MSG)</td>
<td>devrpy : PID $\Rightarrow$ MSG</td>
<td>cdseg : PID $\Rightarrow$ SDesc</td>
</tr>
<tr>
<td>dsseg : PID $\Rightarrow$ SDesc</td>
<td>free = PID $\setminus$ used</td>
<td>used = dom state = dom ptype = dom tss</td>
</tr>
<tr>
<td>$\exists dprocs, uprocs : F PID$</td>
<td>$\exists dprocs = { p : PID \mid p \in used \land ptype(p) = dproc } \land$</td>
<td>$\exists dprocs, uprocs : F PID$</td>
</tr>
</tbody>
</table>
| $dprocs = \{ p : PID \mid p \in used \land ptype(p) \neq dproc \} \bullet$ | $uprocs = \{ p : PID \mid p \in used \land ptype(p) \neq dproc \} \bullet$ | $dprocs = \{ p : PID \mid p \in used \land ptype(p) = dproc \} \land$
| $dprocs = \text{ran devmap} = \text{dom devmsg} = \text{dom devrpy} \land$ | $uprocs = \text{dom cdseg} = \text{dom dsseg} = \text{dom msgq} = \text{dom pidext} = \text{ran extpid}$ | $dprocs = \text{ran devmap} = \text{dom devmsg} = \text{dom devrpy} \land$
| $\exists p : PID \land p \in \text{ran devmap} \Rightarrow (\exists d : \text{DevNo} \mid d \in \text{dom devmap} \land p = \text{devmap}(d)) \bullet$ | pidext = extpid$^{-}$ | $\exists p : PID \land p \in \text{ran devmap} \Rightarrow (\exists d : \text{DevNo} \mid d \in \text{dom devmap} \land p = \text{devmap}(d)) \bullet$
We show the equivalence between \textit{devmap} uniqueness predicate and partial injection function by proving implications to both sides.

\begin{definition}
\textbf{lDevmapUniqueImpliesPInj}
\end{definition}
\begin{align*}
\text{devmap} \in \text{DevNo} \rightarrow \text{PID} & \land \\
(\forall \ p : \text{PID} \bullet \ p \in \text{ran} \ \text{devmap} \Rightarrow (\exists \ d : \text{DevNo} \mid d \in \text{dom} \ \text{devmap} \bullet p = \text{devmap}(d))) & \Rightarrow \\
\text{devmap} \in \text{DevNo} \rightarrow \text{PID}
\end{align*}

\begin{definition}
\textbf{lDevmapPInjImpliesUnique}
\end{definition}
\begin{align*}
\text{devmap} \in \text{DevNo} \rightarrow \text{PID} & \Rightarrow \\
(\forall \ p : \text{PID} \bullet p \in \text{ran} \ \text{devmap} & \Rightarrow (\exists \ d : \text{DevNo} \mid d \in \text{dom} \ \text{devmap} \bullet p = \text{devmap}(d)))
\end{align*}

\begin{definition}
\textbf{tPTabv2v3Equiv}
\end{definition}
\begin{align*}
\text{PTabv2} & \Leftrightarrow \text{PTabv3}
\end{align*}

\textbf{Relational image}

We can simplify the definition of \textit{dprocs} and \textit{uprocs} by using relational image instead of set comprehension. Set comprehension is difficult to reason about, while relational image contains a number of available rules and theorems.

We specify device/user process sets using relational image of process type on inverse \textit{ptype} function, e.g.

\begin{align*}
dprocs & = \text{ptype}^{-1}(\{\text{dproc}\}) \\
\end{align*}

Together with \textit{used} = \text{dom} \ \text{ptype}, the definition is equivalent to the set comprehension one. Thus in \textit{PTabv3} we replace set comprehension with appropriate relational image expressions.

Moreover, in the same \textit{PTab} evolution we remove predicate \textit{dom pidext = uprocs}, because it is redundant. The inversion \textit{pidext = extpid} and predicate \textit{ran extpid = uprocs} already enforces that.

\begin{align*}
\text{PTabv4}
\end{align*}

\begin{align*}
nextrpid & : \text{UPID} \\
\text{extpid} & : \text{UPID} \rightarrow \text{PID} \\
\text{pidext} & : \text{PID} \rightarrow \text{UPID} \\
\text{used}, \text{free} & : \text{F \ PID} \\
\text{tss} & : \text{PID} \rightarrow \text{TSS} \\
\text{devmap} & : \text{DevNo} \rightarrow \text{PID} \\
\text{state} & : \text{PID} \rightarrow \text{PSTATE} \\
\text{ptype} & : \text{PID} \rightarrow \text{PTYPE} \\
\text{msgq} & : \text{PID} \rightarrow \text{MsgQ} \\
\text{devmsg} & : \text{PID} \rightarrow (\text{GPID} \times \text{MSG}) \\
\text{devrpy} & : \text{PID} \rightarrow \text{MSG} \\
\text{cdseg} & : \text{PID} \rightarrow \text{SDesc} \\
\text{dsseg} & : \text{PID} \rightarrow \text{SDesc} \\
\text{free} & = \text{PID} \setminus \text{used} \\
\text{used} & = \text{dom} \ \text{state} = \text{dom} \ \text{ptype} = \text{dom} \ \text{tss} \\
\exists \ \text{dprocs}, \text{uprocs} : \text{F \ PID} & \mid \\
\text{dprocs} & = \text{ptype}^{-1}(\{\text{type}\}) \land \\
\text{uprocs} & = \text{ptype}^{-1}(\{\text{uproc}\}) \\
\text{dprocs} & = \text{ran} \ \text{devmap} = \text{dom} \ \text{devmsg} = \text{dom} \ \text{devrpy} \land \\
\text{uprocs} & = \text{dom} \ \text{cdseg} = \text{dom} \ \text{dsseg} = \text{dom} \ \text{msgq} = \text{ran} \ \text{extpid} \\
\text{pidext} & = \text{extpid}^{-1}
\end{align*}

We prove that relational image set is equal to the set comprehension.

\begin{definition}
\textbf{lPtypeSetComprehension}
\end{definition}
\begin{align*}
\forall \ \text{ptype} : \text{PID} \rightarrow \text{PTYPE}; \ \text{type} : \text{PTYPE} & \bullet \\
\text{ptype}^{-1}(\{\text{type}\}) & = \{p : \text{PID} \mid p \in \text{dom} \ \text{ptype} \land \text{ptype}(p) = \text{type}\}
lPtypeNotDproc guides the prover with reasoning about the distinctiveness of PTYPE values.

\[ \forall \text{ptype} : \text{PID} \rightarrow \text{PTYPE}; \ p : \text{PID} | p \in \text{dom ptype} \bullet \]
\[ \text{ptype}(p) \neq \text{dproc} \Leftrightarrow \text{ptype}(p) = \text{uproc} \]

The prover has problems to easily reason about expression in set comprehension - we have to guide it to show that set of not device processes is the same as set of user processes.

\[ \forall \text{ptype} : \text{PID} \rightarrow \text{PTYPE} \bullet \]
\[ \{ p : \text{PID} | p \in \text{dom ptype} \land \text{ptype}(p) \neq \text{dproc} \} = \]
\[ \{ p : \text{PID} | p \in \text{dom ptype} \land \text{ptype}(p) = \text{uproc} \} \]

\[ \text{PTabv3v4Equiv} \]
\[ \text{PTabv3} \Leftrightarrow \text{PTabv4} \]

Final schema

In the final update of PTab, we add a new predicate

\[ \forall u : \text{UPID} | u \geq \text{nextupid} \bullet u \notin \text{dom extpid} \]

It is necessary for the reasoning about new user process, when we need to ensure that each user process PID has a unique external UPID identifier. Thus we constrain the schema that all identifiers, which are larger or equal to nextupid are unused.

\[ \exists \text{dprocs}, \text{uprocs} : \mathbb{F} \text{PID}| \]
\[ \text{dprocs} = \text{ptype}^{-1}(\{\text{dproc}\}) \land \]
\[ \text{uprocs} = \text{ptype}^{-1}(\{\text{uproc}\}) \bullet \]
\[ \text{dprocs} = \text{ran devmap} = \text{dom devmsg} = \text{dom devrpy} \land \]
\[ \text{uprocs} = \text{dom cdseg} = \text{dom dsseg} = \text{dom msgq} = \text{ran extpid} \]
\[ \text{pidext} = \text{extpid}^{-1} \]
\[ \forall u : \text{UPID} | u \geq \text{nextupid} \bullet u \notin \text{dom extpid} \]

\[ \text{tPTabv4v5Refinement} \]
\[ \text{PTab} \Rightarrow \text{PTabv4} \]

E.6.2 Interesting properties

We have proved several interesting properties about process table PTab.
Injective \textit{extpid} and \textit{pidext} functions

Because of inverse relationship between \textit{extpid} and \textit{pidext} functions, we can show that they are injective.

\begin{verbatim}
theorem tPTabExtpidPinj
\forall PTab \bullet extpid \in UPID \Rightarrow P \land pidext \in PID \Rightarrow UPID
\end{verbatim}

Disjoint user and device processes

We can prove that user and device process sets are disjoint within \textit{PTab} - no process can be both user and device process. To formulate that we use \textit{dprocs} and \textit{uprocs} as relational images (from \textit{PTab} definition).

\begin{verbatim}
theorem tPTabDprocsUprocsDisjoint
\forall PTab \bullet disjoint \langle ptype \sim (\{ dproc \}), ptype \sim (\{ uproc \}) \rangle
\end{verbatim}

Next \textit{UPID} is available

We can show that the next available external identifier is not yet used. This is defined as a forward rule but is a good property to show separately.

\begin{verbatim}
theorem disabled frule fPTabNextupidUnused
PTab \Rightarrow \neg nextupid \in dom extpid
\end{verbatim}

E.6.3 Automation

\begin{verbatim}
theorem frule fNextupidType
\forall PTab \bullet nextupid \in Z
\end{verbatim}

\begin{verbatim}
theorem frule fNextupidUPIDType
nextupid \in UPID \Rightarrow nextupid \in Z
\end{verbatim}

\begin{verbatim}
theorem frule fPTabDprocsPowerUsed
\forall PTab \bullet ptype \sim (\{ dproc \}) \in P \ used
\end{verbatim}

\begin{verbatim}
theorem frule fPTabUprocsPowerUsed
\forall PTab \bullet ptype \sim (\{ uproc \}) \in P \ used
\end{verbatim}

\begin{verbatim}
theorem frule fPTabDomPidext
\forall PTab \bullet dom pidext = ptype \sim (\{ uproc \})
\end{verbatim}

\begin{verbatim}
theorem lPTabPidextExtpid
PTab \Rightarrow pidext = extpid \sim
\end{verbatim}

\begin{verbatim}
theorem disabled rule lDomExtpidSubsetUsed
\forall PTab \bullet ran extpid \subseteq used
\end{verbatim}

\begin{verbatim}
theorem grule gPTabUsedPIDInDomPtype
\forall PTab; p? : PID | p? \in used \bullet p? \in dom ptype
\end{verbatim}

\begin{verbatim}
theorem grule gPTabUsedPIDInDomState
\forall PTab; p? : PID | p? \in used \bullet p? \in dom state
\end{verbatim}

\begin{verbatim}
theorem frule fDevmsgMaxType
\forall PTab \bullet devmsg \in P (Z \times (Z \times MSG))
\end{verbatim}

121
\textbf{theorem} frule fCdsegMaxType
\begin{align*}
\forall P\text{Tab} \bullet \text{cdseg} \in \mathbb{P} (\mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z}))
\end{align*}

\textbf{theorem} frule fDssegMaxType
\begin{align*}
\forall P\text{Tab} \bullet \text{dsseg} \in \mathbb{P} (\mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z}))
\end{align*}

\textbf{theorem} grule gPTabUnknownPIDDproc
\begin{align*}
\forall P\text{Tab}; P : \text{PID} \mid \neg p \in \text{used} \bullet \neg p \in \text{ptype} \sim (\{\text{dproc}\} \cup P)
\end{align*}

\textbf{theorem} disabled rule lPTabUnusedUPIDs
\begin{align*}
\forall P\text{Tab}; u : \text{UPID} \mid u \geq \text{nextupid} \bullet \neg u \in \text{dom extpid}
\end{align*}

\textbf{theorem} disabled rule lPtypeCupImageDifferent
\begin{align*}
\forall \text{ptype} : \text{PID} \mapsto \text{PTYPE}; p : \text{PID}; t, u : \text{PTYPE} \mid \neg p \in \text{dom ptype} \land \neg t = u \bullet \\
(\text{ptype} \sim (\{d\}) \cup \{(t, p)\}) \cup \{u\} = (\text{ptype} \sim (\{u\}) \cup \{(t, p)\})
\end{align*}

\textbf{theorem} disabled rule lPtypeCupImageSame
\begin{align*}
\forall \text{ptype} : \text{PID} \mapsto \text{PTYPE}; p : \text{PID}; t : \text{PTYPE} \mid \neg p \in \text{dom ptype} \bullet \\
(\text{ptype} \sim (\{d\}) \cup \{(t, p)\}) \cup \{u\} = (\text{ptype} \sim (\{u\}) \cup \{(t, p)\})
\end{align*}

\textbf{theorem} grule gPTabEmptyDprocs
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{ptype} \sim (\{\text{dproc}\}) = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyUprocs
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{ptype} \sim (\{\text{uproc}\}) = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyExtpid
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{extpid} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyState
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{state} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyTss
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{tss} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyPtype
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{ptype} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyMsgq
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{msgq} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyCdseg
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{cdseg} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyDsseg
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{dsseg} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyDevmap
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{devmap} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyDevmsg
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{devmsg} = \{}
\end{align*}

\textbf{theorem} grule gPTabEmptyDevrpy
\begin{align*}
\forall P\text{Tab} \bullet \text{used} = \{} \Rightarrow \text{devrpy} = \{}
\end{align*}
E.6.4 Error cases

The error schemata for error cases in $PTab$ operations are defined with the reuse of $RaiseError$ operation.

$RaiseErrUnusedPD$ error occurs when an invalid process identifier (PID) is referenced.

$$RaiseErrUnusedPD \equiv [RaiseError \mid e? = unusedpd] \setminus (e?)$$

$RaiseErrPTabFull$ signals that all available process identifiers in the system are in use.

$$RaiseErrPTabFull \equiv [RaiseError \mid e? = ptabfull] \setminus (e?)$$

The original contains $RaiseErrPDInUse$ error case (with error value $pdinuse$), which is redundant. This error case is not used in separation kernel specification and is carried from the simple kernel specification. Moreover, the error case is redundant in the simple kernel specification as well, because it handles exactly the same case as $RaiseErrPTabFull$. Thus we removed the $RaiseErrPDInUse$ error case from this specification.

Equivalence with original error signals

We show that the new style of specifying error cases is equivalent to the original one.

$$RaiseErrUnusedPDOriginal \equiv (\exists e : SYSERR \mid e = unusedpd \bullet SetSysErr[e/e?] \land RaiseKillInterrupt)$$

**theorem tRaiseErrUnusedPDEquiv**

$$RaiseErrUnusedPD \iff RaiseErrUnusedPDOriginal$$

$$RaiseErrPTabFullOriginal \equiv (\exists e : SYSERR \mid e = ptabfull \bullet SetSysErr[e/e?] \land RaiseKillInterrupt)$$

**theorem tRaiseErrPTabFullEquiv**

$$RaiseErrPTabFull \iff RaiseErrPTabFullOriginal$$

E.6.5 Operations

A number of simple operations from original specification are used here as they were defined.

Initialisation

<table>
<thead>
<tr>
<th>$PTabInit$</th>
<th>$PTab'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$used' = \emptyset$</td>
<td></td>
</tr>
<tr>
<td>$nextupid' = 1$</td>
<td></td>
</tr>
</tbody>
</table>

Query operations

| $UsedPID$ | $\Xi PTab$ |
| $p? : PID$ |
| $p? \in used$ |

| $GotFreePIIDs$ | $\Xi PTab$ |
| $used \subseteq PID$ |
We show that predicate in \textit{GotFreePIDs} actually means that there are free \textit{PIDs}, since it is equivalent to free $\neq \emptyset$.

\textbf{theorem} \ tPTabGotFreePIDsEquiv
\begin{equation*}
\forall P\mathit{Tab} \bullet \ used \subset \ PID \Leftrightarrow \ free \neq \emptyset
\end{equation*}

We added a precondition $u \in \text{dom} \ extpid$, which is necessary for \textit{PIDforUPID} to ensure the function is called within its domain, thus we allow this operation for valid identifiers only.

\begin{align*}
\text{PIDforUPID} \\
\subseteq P\mathit{Tab} \\
\exists u \in \text{dom} \ extpid \\
\ni \\
\text{ProcType} \quad \text{UsedPID} \\
pt! : \text{PTYPE} \\
pt! = \text{ptype}(p?)
\end{align*}

A precondition $p \in \text{used}$ was added to ensure domain checks for \textit{ProcType} and \textit{ProcState}. We reused \textit{UsedPID} for that purpose.

\begin{align*}
\text{ProcState} \\
\text{UsedPID} \\
st! : \text{PSTATE} \\
st! = \text{state}(p?)
\end{align*}

\textbf{Process allocation}

The process allocation operations are defined in small reusable operations, following Craig’s style to some extent. However, where it is not appropriate, we deviate towards simpler expression, showing the equivalence to Craig’s originals, where applicable. Otherwise, justification is provided to explain the choice.

The new process identifier \textit{PID} is allocated non-deterministically.

\begin{align*}
\text{AllocPID} \\
\Delta P\mathit{Tab} \\
p! : \text{PID} \\
\ni \\
\text{used}' = \text{used} \cup \{ p! \}
\end{align*}

New external identifier for user processes \textit{UPID} is allocated following a predefined algorithm.

\begin{align*}
\text{AllocUPID} \\
\Delta P\mathit{Tab} \\
u! : \text{UPID} \\
u! = \text{nextupid} \\
\text{nextupid}' = \text{nextupid} + 1
\end{align*}

We added precondition predicates to \textit{AddProcUPID} and \textit{AddProcUPIDOriginal}, to satisfy partial injection constraints and to disallow identifier reuse. Technically, using relational override ($\oplus$), it would be possible to reuse identifiers directly, by setting new \textit{PID} for existing \textit{UPID}. This approach would need unnecessary cumbersome associated
predicates. As the operation is not used in such way, we restrict its usage with the new predicates, in order to get a clear operation intention.

Based on the added preconditions, we can use union instead of relational override, as it better communicates the operation intent. The change of operator is proved using equivalence.

\[
\text{AddProcUPID} \quad \Delta \text{PTab} \\
p? : \text{PID} \\
u? : \text{UPID} \\
\begin{align*}
u? & \notin \text{dom } \text{extpid} \\
p? & \notin \text{ran } \text{extpid} \\
\text{extpid}' &= \text{extpid} \cup \{u? \mapsto p?\}
\end{align*}
\]

\[
\text{AddProcUPIDOriginal} \quad \Delta \text{PTab} \\
p? : \text{PID} \\
u? : \text{UPID} \\
\begin{align*}
u? & \notin \text{dom } \text{extpid} \\
p? & \notin \text{ran } \text{extpid} \\
\text{extpid}' &= \text{extpid} \oplus \{(u? \mapsto p?)\}
\end{align*}
\]

The following theorems are used to prove equivalence and \text{AddProcUPIDOriginal} precondition proof (for research intent only, to show how operations with relational override can be transformed into union-based ones).

\text{theorem} disabled rule \text{lExtpidOplusIsCup} \\
\forall \text{PTab}; p : \text{PID}; u : \text{UPID} | u \notin \text{dom } \text{extpid} \\
\quad \text{extpid} \oplus \{(u, p)\} = \text{extpid} \cup \{(u, p)\}

\text{theorem} disabled rule \text{lExtpidInvCup} \\
\forall \text{PTab}; p : \text{PID}; u : \text{UPID} | u \notin \text{dom } \text{extpid} \\
\quad \{(p, u)\} \cup \text{extpid}\sim = \text{pidext} \cup \{(p, u)\}

\text{theorem} \text{tAddProcUPIDEquiv} \\
\text{AddProcUPID} \Leftrightarrow \text{AddProcUPIDOriginal}

When all three allocation operation are joined in \text{NewUPIDForProcessOriginal}, the preconditions in \text{AddProcUPID} are implied by \(p!\) and \(u!\) allocation statements. Also, we wanted to reduce the unnecessary verbosity and introduce a simpler allocation operation. The equivalence between both versions is proved.

We added a missing a \(p! : \text{PID}\) statement, which was a typo in the original specification.

\[
\text{NewUPIDForProcess} \quad \Delta \text{PTab} \\
p! : \text{PID} \\
u! : \text{UPID} \\
\begin{align*}p! & \notin \text{used} \\
\text{used'} &= \text{used} \cup \{p!\} \\
u! &= \text{nextupid} \\
\text{nextupid'} &= \text{nextupid} + 1 \\
\text{extpid'} &= \text{extpid} \cup \{u! \mapsto p!\}
\end{align*}
\]

\[
\text{NewUPIDForProcessOriginal} \equiv \text{AllocPID} \land \text{AllocUPID} \land \text{AddProcUPID}[p!/p?, u!/u?]
\]

\text{theorem} \text{tNewUPIDForProcessEquiv} \\
\text{NewUPIDForProcess} \Leftrightarrow \text{NewUPIDForProcessOriginal}

125
Originally, Craig modelled AddPD operation by allowing to specify process type for the operation, thus possibly wanting to create a certain type of process. However that was wrong, because UPID allocation is strictly for user processes. We prove a theorem to verify that new process allocation using NewUPIDForProcess implies uproc process type.

**theorem** tNewUPIDIsUser

\[ \text{NewUPIDForProcess} \Rightarrow \text{ptype'} p! = \text{uproc} \]

The SetProcType operation sets process type. As a process type cannot be changed after initialisation, this operation is used for new processes only.

| \[ \text{SetProcType} \]
| \[ \Delta PTab \]
| \[ p? : \text{PID} \]
| \[ pt? : \text{PTYPE} \]
| \[ \text{ptype}' = \text{ptype} \cup \{ p? \mapsto pt\} \]

Originally Craig modelled process allocation operations as partial ones. He only specified properties of interest, leaving the rest undefined. When operation is then made total with error cases (e.g. AddPD), it can no longer be joined with other operations to complete setting state variables.

Due to this error, we have corrected the specification and implemented full information allocation for AddPD0 operation. All variables in PTab are set to definite values.

The AddPDesc operation is reusable for both types of processes. It initialises process state to psready, as all processes start as being ready for scheduling. Furthermore, it assigns process its allocated TSS.

| \[ \text{AddPDesc} \]
| \[ \Delta PTab \]
| \[ p? : \text{PID} \]
| \[ tss? : \text{TSS} \]
| \[ \text{state}' = \text{state} \cup \{ p? \mapsto \text{psready} \} \]
| \[ \text{tss}' = \text{tss} \cup \{ p? \mapsto \text{tss}\} \]

We initialise user process message queue using the promotion pattern.

| \[ \text{PromotePTabMNew} \]
| \[ \Delta PTab \]
| \[ \text{MsgQ}' \]
| \[ p? : \text{PID} \]
| \[ \text{msgq}' = \text{msgq} \cup \{ p? \mapsto \theta \text{MsgQ}' \} \]

\[ \text{NewUProcMsgQ} \equiv \exists \text{MsgQ}' \bullet \text{PromotePTabMNew} \land \text{MsgQInit} \]

We show the expanded version of the promoted operation.

| \[ \text{NewUProcMsgQExpand} \]
| \[ \Delta PTab \]
| \[ p? : \text{PID} \]
| \[ \text{maxMs}? : \mathbb{N}_1 \]
| \[ \text{msgq}' = \text{msgq} \cup \{ p? \mapsto \theta \text{MsgQ}'[\text{maxMs} := \text{maxMs}?, \text{mq} := \langle \rangle] \} \]

**theorem** tNewUProcMsgQEquiv

\[ \text{NewUProcMsgQ} \Leftrightarrow \text{NewUProcMsgQExpand} \]
We have created an operation to allocate user process specific information in `NewUProcInfo`. It builds on the shared information allocation `AddPDesc`. Also, it reuses memory segment allocation from Craig’s specification. We have decided not to make them as separate operations, because this is the only place where they are used. Furthermore, it incorporates the promotion-based message queue initialisation, sets process type as `uproc` and makes sure that the device-related information is not changed.

\[
\begin{align*}
\text{NewUProcInfo} & \quad \text{AddPDesc} \\
& \quad \text{NewUProcMsgQ} \\
& \quad \text{cdAddr}? : \text{Addr}; \text{cdSize}? : N \\
& \quad \text{dsAddr}? : \text{Addr}; \text{dsSize}? : N \\
\end{align*}
\]

\[
\begin{align*}
\text{ptype'} = \text{ptype} \cup \{ p? \mapsto \text{uproc} \} \\
\text{cdseg'} = \text{cdseg} \cup \{ p? \mapsto \text{mkSDesc} (\text{cdAddr}? , \text{cdSize}?) \} \\
\text{dsseg'} = \text{dsseg} \cup \{ p? \mapsto \text{mkSDesc} (\text{dsAddr}? , \text{dsSize}?) \} \\
\text{devmap'} = \text{devmap} \\
\text{devmsg'} = \text{devmsg} \\
\text{devrpy'} = \text{devrpy}
\end{align*}
\]

We have combined the process and its information allocation in `AddPD0`. Furthermore, precondition to check that new process allocation is possible, was added.

\[
\begin{align*}
\text{AddPD0} & \quad \text{NewUPIDForProcess} \\
& \quad \text{NewUProcInfo}[p!/p?] \\
\text{used} \subseteq \text{PID}
\end{align*}
\]

We have adapted the original specification style as well, when updating the completeness of operations. Craig uses a verbose style to check allocation precondition with `GotFreePIDs`. In our version, we avoid unnecessary existential quantifier of schema composition (\(\exists\)).

\[
\text{AddPD0Original} \equiv \text{GotFreePIDs} \downarrow (\text{NewUPIDForProcess} \land \text{NewUProcInfo}[p!/p?])
\]

We prove the equivalence between the operations.

**Theorem** \(\text{tAddPD0Equiv}\)

\[
\text{AddPD0} \iff \text{AddPD0Original}
\]

The `used \subseteq \text{PID}` precondition of `AddPD0` (coming from `GotFreePIDs`) and \(\exists p : \text{PID} \bullet p \notin \text{used}\) of `NewUPIDForProcess` are equivalent, as shown by `lExistFreePIDs`.

\[
\text{theorem} \quad \text{lExistFreePIDs}
\]

\[
\forall \text{used} : \mathbb{P} \quad \text{PID} \bullet (\exists p : \text{PID} \mid p \notin \text{used} \bullet \text{true}) \iff \text{used} \subseteq \text{PID}
\]

The equivalence means that we can leave only a single precondition `used \subseteq \text{PID}` for the operation. This equivalence was not spotted originally by Craig and he modelled an additional error operation `pdinuse` to indicate when \(\exists p : \text{PID} \bullet p \notin \text{used}\) is not satisfied. Because of this reason, we removed the unnecessary precondition and the error case.

The error cases need to have explicitly stated preconditions, when they happen. In simple cases, it is a negation of “correct” operation precondition. This was not practiced by Craig, thus in all cases we need to explicitly specify error preconditions.

`ErrPTabFull` defines a condition when a process table is full (no more identifiers are available for allocation). We specify this error case to perform no change on `PTab` and raise a corresponding error. The error case is created to be reusable and therefore does not have a homogenous interface with `AddPD0`.

\[
\begin{align*}
\text{ErrPTabFull} & \quad \exists \text{PTab} \\
& \quad \text{RaiseErrPTabFull} \\
& \quad \neg \text{used} \subseteq \text{PID}
\end{align*}
\]
Then we can define the total operation for user process allocation \( AddPD \):

\[
AddPD \triangleq (AddPD0 \land \text{RaiseOk}) \lor \text{ErrPTabFull}
\]

For the clarity, and following Craig’s style, we provide the expanded version of \( AddPD \). However, as \( \text{ErrPTabFull} \) has different interface than \( AddPD0 \), we had to perform some adjustments in the equality proof.

The expanded version has a single set of input/output variables. They apply to both success and error case branches. To make the interface of \( AddPD \) uniform for both cases, we introduced \( AddPDInterface \), which only defines input/output variables for the \( AddPD0 \).

Theorem \( tAddPDEquiv \)

\[
(AddPD \land AddPDInterface) \leftrightarrow AddPDExpand
\]

Furthermore, we prove an interesting property as defined by Craig. It shows that when operation is successful, the output identifier is not a free \( PID \). Note that this was adapted for \( AddPD0 \), instead of \( AddPD \) in Craig’s specification, because in the error case \( p! \) is undefined.

Theorem \( tAddPD0NotFree \)

\[
AddPD0 \Rightarrow p! \notin \text{free}'
\]
Idle process allocation

An idle process is used by the scheduler, to run when no other processes are running. Originally, Craig modelled it as a “lightweight” process, which would have only a part of attributes. For example, it was defined as a user process without an external UPID. Craig argued that we can decide on its type.

However, further analysis proved that we cannot define such process, because of the constraints in PTab. The idle process has to be a user process, because its code/data segments are allocated further in the specification. This also requires for it to have an external UPID.

Furthermore, as updated AddPD allocates a process with all variables set, the idle process allocation operation is actually exactly the same as AddPD. We redefine the operation with a new name and renamed output variable.

\[ AddIdleProcess \equiv AddPD[ip]/p! \]

We do not provide an expanded version of the operation, as it is exactly the same as AddPDExand.

Process deletion

Process deletion operation removes the process from all variables and ensures remaining information stays the same. Originally Craig modelled process deletion as simply removing process from used and extpid set. However, such approach leaves the remaining variables in undefined state and we corrected to specify outcome of all variables.

\[ \begin{align*}
    FreePID \\
    \Delta PTab \\
    p? : \text{PID} \\
    \text{nextupid}' = \text{nextupid} \\
    \text{used}' = \text{used} \setminus \{p\} \\
    \text{tss}' = \{p\} \setminus \text{tss} \\
    \text{state}' = \{p\} \setminus \text{state} \\
    \text{ptype}' = \{p\} \setminus \text{ptype} \\
    \text{extpid}' = \text{extpid} \setminus \{p\} \\
    \text{msgq}' = \{p\} \setminus \text{msgq} \\
    \text{cdseg}' = \{p\} \setminus \text{cdseg} \\
    \text{dsseg}' = \{p\} \setminus \text{dsseg} \\
    \text{devmap}' = \text{devmap} \setminus \{p\} \\
    \text{devmsg}' = \{p\} \setminus \text{devmsg} \\
    \text{devrpy}' = \{p\} \setminus \text{devrpy}
\end{align*} \]

FreePID does not distinguish between process types when removing a process. For variables of other process type, where \( p? \) does not belong, domain/range restriction does not introduce any change.

Technically, the precondition of FreePID is true, which means that it can be executed for any \( p? \) value. However, for unused PID values, no change would be done. To avoid this and have a stricter specification, we treat such execution as an error. Therefore, we introduce a precondition about known process identifier: \( p? \in \text{used} \).

\[ \begin{align*}
    DelPD0 \\
    FreePID \\
    p? \in \text{used}
\end{align*} \]

To handle case when an unknown PID is referenced, we introduce an error case ErrUnusedPD.

\[ \begin{align*}
    ErrUnusedPD \\
    \Xi PTab \\
    \text{RaiseErrUnusedPD} \\
    p? : \text{PID} \\
    p? \notin \text{used}
\end{align*} \]
Now we can define the total operation to delete a process $DelPD$:

$$DelPD \equiv (DelPD0 \land \text{RaiseOk}) \lor \text{ErrUnusedPD}$$

Craig had equivalent operations $DelProcUPID$ and $DelExtPD$ defined to clear external $UPID$ value. As this action is handled by $FreePID$, we no longer need these operations and therefore remove them.

Finally, an operation to delete all processes is defined as in Craig’s specification. It is used to clear all processes at once.

<table>
<thead>
<tr>
<th>$DeleteAllProcesses$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta PTab$</td>
</tr>
<tr>
<td>$\text{used}' = \emptyset$</td>
</tr>
</tbody>
</table>

When this operation is used, the $PTab$ state invariant requires that all other variables are also cleared. We can show this as an example for a single case using $tDeleteAllExtpid$.

**Theorem** $tDeleteAllExtpid$

$DeleteAllProcesses \Rightarrow extpid' = \emptyset$

**Process state operations**

The process state can be changed during kernel operation. Originally, Craig defined $SetProcState$ as a partial update operation, which does constrain the remaining $PTab$ variables. Examining how the operation is used, we have decided to require that the remaining $PTab$ variables do not change. This means that when a process state is changed, no other process information is changed.

The “everything else stays the same” intent is modelled using schema variable hiding on $\Xi PTab$. This keeps the $\Xi PTab$ mapping for all variables except the hidden one.

$$PTabChangeState \equiv \Xi PTab \setminus (\text{state})$$

<table>
<thead>
<tr>
<th>$SetProcState$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta PTab$</td>
</tr>
<tr>
<td>$PTabChangeState$</td>
</tr>
<tr>
<td>$p? : \text{PID}$</td>
</tr>
<tr>
<td>$st? : \text{PSTATE}$</td>
</tr>
<tr>
<td>$\text{state}' = \text{state} \oplus {p? \mapsto st?}$</td>
</tr>
</tbody>
</table>

Now we can define shortcut operations to set specific process states. We use the hiding style as introduced first in error cases. The equivalence with original style proofs are provided further.

$$SetStateToReady \equiv [SetProcState \mid st? = \text{psready}] \setminus (st?)$$

Following Craig’s style, we provide the expanded operation.

<table>
<thead>
<tr>
<th>$SetStateToReadyExpand$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta PTab$</td>
</tr>
<tr>
<td>$p? : \text{PID}$</td>
</tr>
<tr>
<td>$\text{state}' = \text{state} \oplus {p? \mapsto \text{psready}}$</td>
</tr>
<tr>
<td>$\text{nextupid}' = \text{nextupid}$</td>
</tr>
<tr>
<td>$\text{used}' = \text{used}$</td>
</tr>
<tr>
<td>$\text{tss}' = \text{tss}$</td>
</tr>
<tr>
<td>$\text{ptype}' = \text{ptype}$</td>
</tr>
<tr>
<td>$\text{extpid}' = \text{extpid}$</td>
</tr>
<tr>
<td>$\text{msgq}' = \text{msgq}$</td>
</tr>
<tr>
<td>$\text{cdseg}' = \text{cdseg}$</td>
</tr>
<tr>
<td>$\text{dsseg}' = \text{dsseg}$</td>
</tr>
<tr>
<td>$\text{devmap}' = \text{devmap}$</td>
</tr>
<tr>
<td>$\text{devmsg}' = \text{devmsg}$</td>
</tr>
<tr>
<td>$\text{devrpy}' = \text{devrpy}$</td>
</tr>
</tbody>
</table>
\textbf{theorem} tSetStateToReadyEquiv
\[
\text{SetStateToReady} \Leftrightarrow \text{SetStateToReadyExpand}
\]
\[
\text{SetStateToRunning} \equiv [\text{SetProcState} \mid \text{st}? = \text{psrunning}] \setminus \text{(st?)}
\]
\[
\text{SetStateToTerminated} \equiv [\text{SetProcState} \mid \text{st}? = \text{pterm}] \setminus \text{(st?)}
\]

Further we provide the original style of operations and equivalence proofs.
\[
\text{SetStateToReadyOriginal} \equiv
\exists \text{st} : \text{PSTATE} \mid \text{st} = \text{psready} \bullet \text{SetProcState[st/st?]}\]

\textbf{theorem} tSetStateToReadyOrigEquiv
\[
\text{SetStateToRunning} \Leftrightarrow \text{SetStateToRunningOriginal}
\]
\[
\text{SetStateToRunningOriginal} \equiv
\exists \text{st} : \text{PSTATE} \mid \text{st} = \text{psrunning} \bullet \text{SetProcState[st/st?]}\]

\textbf{theorem} tSetStateToRunningOrigEquiv
\[
\text{SetStateToTerminated} \Leftrightarrow \text{SetStateToTerminatedOriginal}
\]
\[
\text{SetStateToTerminatedOriginal} \equiv
\exists \text{st} : \text{PSTATE} \mid \text{st} = \text{pterm} \bullet \text{SetProcState[st/st?]}\]

\textbf{E.6.6 Operation precondition signatures}
\[
\begin{align*}
\text{RaiseErrSig} & \equiv [\text{ErrV}; \text{RaiseKillInterruptSig} \mid \text{true}] \\
\text{RaiseErrUnusedPDSig} & \equiv \text{RaiseErrSig} \\
\text{RaiseErrPTabFullSig} & \equiv \text{RaiseErrSig} \\
\text{PTabOpSig} & \equiv [\text{PTab}; \text{p}? : \text{PID}] \\
\text{UsedPIDSig} & \equiv [\text{PTabOpSig} \mid \text{p}? \in \text{used}] \\
\text{GotFreePIDsSig} & \equiv [\text{PTab} \mid \text{used} \subset \text{PID}] \\
\text{PIDforUPIDSig} & \equiv [\text{PTab}; \text{u}? : \text{UPID} \mid \text{u}? \in \text{dom extpid}] \\
\text{ProcTypeSig} & \equiv \text{UsedPIDSig} \\
\text{ProcStateSig} & \equiv \text{UsedPIDSig} \\
\text{NewPIDSig} & \equiv [\text{PTab} \mid \exists \text{p} : \text{PID} \bullet \text{p} \notin \text{used}] \\
\end{align*}
\]

As the \textit{AllocPID} operation is shared between both user and device processes, we show the preconditions for both types. Thus we define different signatures for each type and prove 2 precondition theorems.
\[
\text{AllocPIDDprocSig} \equiv [\text{NewPIDSig} \mid \exists \text{d} : \text{DevNo} \bullet \text{d} \in \text{dom devmap}]
\]
AllocPIDUprocSig \equiv \text{NewPIDSig}

AllocUPIDSig \equiv \text{PTab}

NewProcessSig \equiv [\text{PTabOpSig} \mid p? \notin \text{used}]

AddProcUPIDSig \equiv [\text{NewProcessSig}; u? : \text{UPID} \mid u? \geq \text{nextupid}]

NewUPIDForProcessSig \equiv \text{NewPIDSig}

\text{NewDProcSig} \equiv [\text{NewProcessSig} \mid \exists \, d : \text{DevNa} \cdot \neg d \in \text{dom devmap}]

\text{NewUProcSig} \equiv \text{NewProcessSig}

Again, show 2 different preconditions for \text{SetProcType}.

\text{SetDProcTypeSig} \equiv [\text{NewDProcSig}; pt? : \text{PTYPE} \mid pt? = \text{dproc}]

\text{SetUProcTypeSig} \equiv [\text{NewUProcSig}; pt? : \text{PTYPE} \mid pt? = \text{uproc}]

\text{AddPDesc} \text{ has 2 preconditions.}

\text{AddPDescDprocSig} \equiv [\text{NewDProcSig}; \text{tss}? : \text{TSS} \mid \text{true}]

\text{AddPDescUprocSig} \equiv [\text{NewUProcSig}; \text{tss}? : \text{TSS} \mid \text{true}]

\text{PromotePTabMNewSig} \equiv \text{NewUProcSig}

\text{NewUProcMsgQSig} \equiv [\text{NewUProcSig}; \text{maxMs}? : \text{N}_1 \mid \text{true}]

\text{NewUProcInfoSig} \equiv [\text{AddPDescUprocSig}; \text{maxMs}? : \text{N}_1;
\text{cdAddr}? : \text{Addr}; \text{cdSize}? : \text{N};
\text{dsAddr}? : \text{Addr}; \text{dsSize}? : \text{N} \mid \text{true}]

\text{AddPD0Sig} \equiv [\text{PTab};
\text{tss}? : \text{TSS}; \text{maxMs}? : \text{N}_1;
\text{cdAddr}? : \text{Addr}; \text{cdSize}? : \text{N};
\text{dsAddr}? : \text{Addr}; \text{dsSize}? : \text{N} \mid \text{used} \subset \text{PID}]

\text{PTabTotalSig} \equiv [\text{PTab}; \text{RaiseErrSig}]

\text{PTabOpTotalSig} \equiv [\text{PTabOpSig}; \text{RaiseErrSig}]

\text{ErrPTabFullSig} \equiv [\text{PTabTotalSig} \mid \neg \text{used} \subset \text{PID}]

\text{AddPSig} \equiv [\text{PTabTotalSig};
\text{tss}? : \text{TSS}; \text{maxMs}? : \text{N}_1;
\text{cdAddr}? : \text{Addr}; \text{cdSize}? : \text{N};
\text{dsAddr}? : \text{Addr}; \text{dsSize}? : \text{N}]

\text{AddIdleProcessSig} \equiv \text{AddPSig}
FreePIDSig ≡ PTabOpSig

DelPDOSig ≡ [PTabOpSig | p? ∈ used]

ErrUnusedPIDSig ≡ [PTabOpTotalSig | p? /∈ used]

DelPDSig ≡ PTabOpTotalSig

DeleteAllProcessesSig ≡ PTab

SetProcStateSig ≡ [UsedPIDSig; st? : PSTATE | true]

SetStateToReadySig ≡ UsedPIDSig

SetStateToRunningSig ≡ UsedPIDSig

SetStateToTerminatedSig ≡ UsedPIDSig

E.6.7 Operation preconditions

**theorem** tRaiseErrUnusedPDPre
∀ RaiseErrUnusedPDSig • pre RaiseErrUnusedPD

**theorem** tRaiseErrPTabFullPre
∀ RaiseErrPTabFullSig • pre RaiseErrPTabFull

**theorem** tPTabInit
∃ PTab′ • PTabInit

**theorem** tUsedPIDPre
∀ UsedPIDSig • pre UsedPID

**theorem** tGotFreePIDsPre
∀ GotFreePIDsSig • pre GotFreePIDs

**theorem** tPIDforUPIDPre
∀ PIDforUPIDSig • pre PIDforUPID

**theorem** tProcTypePre
∀ ProcTypeSig • pre ProcType

**theorem** tProcStatePre
∀ ProcStateSig • pre ProcState

We simplify subsequent allocation proofs by defining theorems for proving common parts. All allocation precondition proofs are taken to state defined using lAllocPIDDproc and lAllocPIDUproc. Then these proofs are reused to complete precondition proofs.

**theorem** lAllocPIDDproc
∀ PTab; p : PID; t : TSS; d : DevNo; st : PSTATE | p /∈ used ∧ ¬ d ∈ dom devmap •
PTab[devmap := devmap ∪ \{(d, p)\},
devmsg := devmsg ∪ \{(p, (nullpid, nullmsg))\},
devrpy := devrpy ∪ \{(p, nullmsg)\},
free := free \{p\},
ptype := ptype ∪ \{(p, dproc)\},
state := state ∪ \{(p, st)\},
tss := tss ∪ \{(p, t)\},
used := used ∪ \{p\}]

133
The following theorems (\(t\text{AllocPIDDprocPre}\) and \(t\text{AllocPIDUprocPre}\)) both prove two alternative preconditions of \(\text{AllocPID}\), when it is used to allocate Device or User processes.

\begin{align*}
\text{theorem} & \ t\text{AllocPIDDprocPre} \\
& \forall \text{AllocPIDDprocSig} \ \bullet \ \text{pre \ AllocPID}
\end{align*}

\begin{align*}
\text{theorem} & \ t\text{AllocPIDUprocPre} \\
& \forall \text{AllocPIDUprocSig} \ \bullet \ \text{pre \ AllocPID}
\end{align*}

\begin{align*}
\text{theorem} & \ t\text{AllocUPIDPre} \\
& \forall \text{AllocUPIDSig} \ \bullet \ \text{pre \ AllocUPID}
\end{align*}

\begin{align*}
\text{theorem} & \ t\text{AddProcUPIDPre} \\
& \forall \text{AddProcUPIDSig} \ \bullet \ \text{pre \ AddProcUPID}
\end{align*}

We have also proved a precondition of the original specification, in order to show how the proof is transformed to the similar case as \(t\text{AddProcUPIDPre}\).

\begin{align*}
\text{theorem} & \ t\text{AddProcUPIDOriginalPre} \\
& \forall \text{AddProcUPIDSig} \ \bullet \ \text{pre \ AddProcUPIDOriginal}
\end{align*}

\begin{align*}
\text{theorem} & \ t\text{NewUPIDForProcessPre} \\
& \forall \text{NewUPIDForProcessSig} \ \bullet \ \text{pre \ NewUPIDForProcess}
\end{align*}

Again, two different preconditions are proved for \(\text{SetProcType}\).

\begin{align*}
\text{theorem} & \ t\text{SetDProcTypePre} \\
& \forall \text{SetDProcTypeSig} \ \bullet \ \text{pre \ SetProcType}
\end{align*}

\begin{align*}
\text{theorem} & \ t\text{SetUProcTypePre} \\
& \forall \text{SetUProcTypeSig} \ \bullet \ \text{pre \ SetProcType}
\end{align*}

\(\text{AddPDesc}\) also has 2 different preconditions.

\begin{align*}
\text{theorem} & \ t\text{AddPDescDprocPre} \\
& \forall \text{AddPDescDprocSig} \ \bullet \ \text{pre \ AddPDesc}
\end{align*}

\begin{align*}
\text{theorem} & \ t\text{AddPDescUprocPre} \\
& \forall \text{AddPDescUprocSig} \ \bullet \ \text{pre \ AddPDesc}
\end{align*}

\begin{align*}
\text{theorem} & \ t\text{PromotePTabMNewPre} \\
& \forall \text{PromotePTabMNewSig} \ \bullet \ \text{pre \ PromotePTabMNew}
\end{align*}

\begin{align*}
\text{theorem} & \ t\text{NewUProcMsgQPre} \\
& \forall \text{NewUProcMsgQSsig} \ \bullet \ \text{pre \ NewUProcMsgQ}
\end{align*}
The process deletion proofs are simplified using \( \text{tFreePID} \) theorem.

\textbf{theorem} \( \text{tFreePID} \)
\( \forall P\text{Tab}; \ p? : \text{PID} \bullet \)
\quad \( P\text{Tab}[c\text{dseg} := \{p\?\} \triangleq c\text{dseg}, \)
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 

\textbf{theorem} \( \text{tFreePIDPre} \)
\( \forall \text{FreePIDSig} \bullet \text{pre} \ \text{FreePID} \)

\textbf{theorem} \( \text{tDelPD0Pre} \)
\( \forall \text{DelPD0Sig} \bullet \text{pre} \ \text{DelPD0} \)

\textbf{theorem} \( \text{tErrUnusedPDPre} \)
\( \forall \text{ErrUnusedPDSig} \bullet \text{pre} \ \text{ErrUnusedPD} \)

\textbf{theorem} \( \text{tDelPDPre} \)
\( \forall \text{DelPDSig} \bullet \text{pre} \ \text{DelPD} \)

\textbf{theorem} \( \text{tDeleteAllProcessesPre} \)
\( \forall \text{DeleteAllProcessesSig} \bullet \text{pre} \ \text{DeleteAllProcesses} \)

\textbf{theorem} \( \text{tSetProcStatePre} \)
\( \forall \text{SetProcStateSig} \bullet \text{pre} \ \text{SetProcState} \)

\textbf{theorem} \( \text{tSetStateToReadyPre} \)
\( \forall \text{SetStateToReadySig} \bullet \text{pre} \ \text{SetStateToReady} \)

\textbf{theorem} \( \text{tSetStateToRunningPre} \)
\( \forall \text{SetStateToRunningSig} \bullet \text{pre} \ \text{SetStateToRunning} \)

\textbf{theorem} \( \text{tSetStateToTerminatedPre} \)
\( \forall \text{SetStateToTerminatedSig} \bullet \text{pre} \ \text{SetStateToTerminated} \)
E.7 Process queue

The process queue definition and proof structure is taken from the work by Leo Freitas on defining corresponding process queue data structure for simple kernel. The definitions and proofs have been adapted to use the process table \( PTab \) of separation kernel.

Note that Leo Freitas models the process queue using a simpler specification style in comparison to Craig’s original verbose specification. We use a similar style as Leo Freitas.

E.7.1 State

Although Craig suggests using a simplified process queue version with simple sequence and no relation to \( PTab \), we chose to use full definition as in the simple kernel, because we are reusing the definitions and proofs.

\[
\begin{align*}
\text{PQueue} & \subseteq \text{PTab} \\
\text{procs} & : \text{iseq} \ PID \\
\text{ran } \text{procs} & \subseteq \text{used}
\end{align*}
\]

We model \( \text{PQueue} \) as an injective sequence with the requirement that all processes in the queue are known to the kernel (are used in \( PTab \)). These were exact requirements as suggested by Craig.

E.7.2 Automation

\[
\begin{align*}
\text{theorem} \text{ rule lPIDInPQPIDInUsed} & \forall \text{PQueue} ; \ p : \text{PID} \pmb{\bullet} \ p \in \text{ran } \text{procs} \Rightarrow p \in \text{used} \\
\text{theorem} \text{ rule lPQHeadIsInPID} & \forall \text{procs} : \text{iseq} \ PID | \text{procs} \neq () \pmb{\bullet} \text{head } \text{procs} \in \text{PID} \\
\text{theorem} \text{ rule lPQTailIsInIseqPID} & \forall \text{procs} : \text{iseq} \ PID | \text{procs} \neq () \pmb{\bullet} \text{tail } \text{procs} \in \text{iseq } \text{PID}
\end{align*}
\]

The following theorem is not fully proved by Leo Freitas due to limitations of \( Z/Eves \) toolkit about sequence concatenation.

\[
\begin{align*}
\text{theorem} \text{ rule lPQCatPIDIsInIseqPID } [X] & \forall A : P X \pmb{\bullet} \forall s : \text{iseq } A ; \ x : A | \neg x \in \text{ran } s \pmb{\bullet} s \oslash (x) \in \text{iseq } A
\end{align*}
\]

E.7.3 Error cases

We use the common variable hiding style to define error cases, as we do for all instantiations of specific variable values.

\[
\text{RaiseErrEmptyQueue} \equiv [\text{RaiseError} | e? = \text{emptyqueue}] \setminus (e?)
\]

The \( \text{alreadyqueued} \) error case was added by Leo Freitas to handle cases when the added \( \text{PID} \) is already in the queue. This is required by the injective sequence used for \( \text{PQueue} \).

\[
\text{RaiseErrAlreadyQueued} \equiv [\text{RaiseError} | e? = \text{alreadyqueued}] \setminus (e?)
\]

E.7.4 Operations

Initialisation

Leo Freitas suggests that Craig’s initialisation is wrong and adds \( PTabInit \) to \( \text{PQueue} \) initialisation. This is too restrictive because of how \( \text{PQueue} \) is afterwards used in the scheduler - it must be initialised with an existing \( PTab \).

\[
\begin{align*}
\text{PQueueInit} & \subseteq \text{PQueue} \\
\text{procs}' & = ()
\end{align*}
\]
Query operations

Leo Freitas models process queue operations to leave underlying process table unchanged. That is, the queue can be updated, however process information and attributes must not change while doing so. This is ensured by $\Xi PTab$ predicate in $PQueueOp$.

```
PQueueOp
\Delta PQueue
p : PID
\Xi PTab
```

For non-empty queues, the first element in the queue (head of the queue) can be queried for.

```
HeadPQueue0
\Xi PQueue
p! : PID
procs ≠ \langle \rangle
p! = head procs
```

The first element exists and therefore can be queried only if the query is not empty. For the opposite case we define an error operation to signal that error. In $ErrEmptyQueue$ definition we reuse $EmptyPQueue$ operation and supplement it with error signal.

```
ErrEmptyQueue ≡ EmptyPQueue \land RaiseErrEmptyQueue
```

We do not try to achieve homogeneity between $ErrEmptyQueue$ and $HeadPQueue0$ for the same reasons as for operations in $PTab$.

```
HeadPQueue ≡ (HeadPQueue0 \land RaiseOk) \lor ErrEmptyQueue
```

Enqueue operations

A process is enqueued by adding it to the end of the process queue. Note that $PQueue$ state invariants create the precondition of this operation that $p?$ is known to $PTab$ and not yet enqueued.

```
EnqueuePQueue0.
PQueueOp[p?/p]
procs' = procs \cup \langle p? \rangle
```

Next we define error operations to handle cases when preconditions of $EnqueuePQueue0$ are not satisfied. We have already defined the error operation for unused process identifiers in $PTab$, therefore we reuse it here with additional requirement that $PQueue$ does not change.

```
ErrUnusedPDPQueue
ErrUnusedPD
\Xi PQueue
```

```
ErrAlreadyQueued
\Xi PQueue
PQueueOp[p?/p]
RaiseErrAlreadyQueued
p? \in \text{ran} procs
```
We define the total operation to enqueue PID in the process queue in the usual fashion of disjoining execution cases.

\[ EnqueuePQueue \equiv (EnqueuePQueue0 \land RaiseOk) \lor \text{ErrUnusedPDPQueue} \lor \text{ErrAlreadyQueued} \]

**Dequequeue operations**

The first process in the process queue can be dequeued - removed from the queue and returned via output variable.

\[
\begin{align*}
\text{DequeuePQueue0} & \\
PQueueOp[p! / p] & \\
\text{procs} \neq () & \\
p! = \text{head } \text{procs} & \\
\text{procs}' = \text{tail } \text{procs} &
\end{align*}
\]

A dequeue operation is only possible when there is something to dequeue - the queue is not empty. We use the already defined \text{ErrEmptyQueue} error operation for this case.

\[ DequeuePQueue \equiv (DequeuePQueue0 \land RaiseOk) \lor \text{ErrEmptyQueue} \]

**E.7.5 Operation precondition signatures**

\[
\begin{align*}
\text{RaiseErrEmptyQueue} & \equiv \text{RaiseErrSig} \\
\text{RaiseErrAlreadyQueued} & \equiv \text{RaiseErrSig} \\
\text{PQueueOp} & \equiv [\text{PQueue}; \ p? : \text{PID}] \\
\text{EmptyPQueue} & \equiv [\text{PQueue} \mid \text{procs} = ()] \\
\text{PQueueTotal} & \equiv [\text{PQueue}; \text{RaiseErrSig}] \\
\text{PQueueOpTotal} & \equiv [\text{PQueueOp}; \text{RaiseErrSig}] \\
\text{PQueueNonEmpty} & \equiv [\text{PQueue} \mid \text{procs} \neq ()] \\
\text{HeadPQueue0} & \equiv \text{PQueueNonEmpty} \\
\text{HeadPQueue} & \equiv \text{PQueueTotal} \\
\text{ErrEmptyQueue} & \equiv [\text{EmptyPQueue} ; \text{RaiseErrSig}] \\
\text{EnqueuePQueue0} & \equiv [\text{PQueueOp} \mid p? \in \text{used} \land p? \notin \text{ran } \text{procs}] \\
\text{ErrUnusedPDPQueue} & \equiv [\text{PQueueOpTotal} \mid p? \notin \text{used}] \\
\text{ErrAlreadyQueued} & \equiv [\text{PQueueOpTotal} \mid p? \in \text{ran } \text{procs}] \\
\text{EnqueuePQueue} & \equiv \text{PQueueOpTotal} \\
\text{DequeuePQueue0} & \equiv \text{PQueueNonEmpty} \\
\text{DequeuePQueue} & \equiv \text{PQueueTotal}
\end{align*}
\]
E.7.6 Operation preconditions

**Theorem tPQueueInit**

\[ \exists PQueue' \cdot PQueueInit \]

**Theorem tRaiseErrEmptyQueuePre**

\[ \forall RaiseErrEmptyQueueSig \cdot \text{pre } RaiseErrEmptyQueue \]

**Theorem tRaiseErrAlreadyQueuedPre**

\[ \forall RaiseErrAlreadyQueuedSig \cdot \text{pre } RaiseErrAlreadyQueued \]

**Theorem tPQueueOpPre**

\[ \forall PQueueOpSig \cdot \text{pre } PQueueOp[p?] \]

**Theorem tEmptyPQueuePre**

\[ \forall EmptyPQueueSig \cdot \text{pre } EmptyPQueue \]

**Theorem tHeadPQueue0Pre**

\[ \forall HeadPQueue0Sig \cdot \text{pre } HeadPQueue0 \]

**Theorem tErrEmptyQueuePre**

\[ \forall ErrEmptyQueueSig \cdot \text{pre } ErrEmptyQueue \]

**Theorem tHeadPQueuePre**

\[ \forall HeadPQueueSig \cdot \text{pre } HeadPQueue \]

**Theorem tEnqueuePQueue0Pre**

\[ \forall EnqueuePQueue0Sig \cdot \text{pre } EnqueuePQueue0 \]

**Theorem tErrUnusedPDPQueuePre**

\[ \forall ErrUnusedPDPQueueSig \cdot \text{pre } ErrUnusedPDPQueue \]

**Theorem tErrAlreadyQueuedPre**

\[ \forall ErrAlreadyQueuedSig \cdot \text{pre } ErrAlreadyQueued \]

**Theorem tEnqueuePQueuePre**

\[ \forall EnqueuePQueueSig \cdot \text{pre } EnqueuePQueue \]

**Theorem tDequeuePQueue0Pre**

\[ \forall DequeuePQueue0Sig \cdot \text{pre } DequeuePQueue0 \]

**Theorem tDequeuePQueuePre**

\[ \forall DequeuePQueueSig \cdot \text{pre } DequeuePQueue \]

E.8 Device queue

A device queue is just another instance of process queue. We define it to rename `PQueue` variables so that both cases can be used together in the scheduler without variable overlap because of matching names.

E.8.1 State

\[ DeviceQueue \doteq PQueue[devs/procs] \]
E.8.2 Operations

We define just the top-level operations of the queue, as only these are necessary and used further in the specification.

\[ DeviceQueueInit \triangleq PQueueInit[devs'/procs'] \]

\[ EnqueueDeviceQueue \triangleq EnqueuePQueue[devs/procs, devs'/procs'] \]

\[ DequeueDeviceQueue \triangleq DequeuePQueue[devs/procs, devs'/procs'] \]

We prove the initialisation and precondition theorems for these operations. In all cases, we reuse proofs of corresponding renamed \( PQueue \) operations. However, we do these preconditions again instead of relying on rename operations for reusability. If the device queue is changed in the future to be different than process queue, these theorems can be used to ensure that the data structure is defined correctly.

E.8.3 Operation precondition signatures

\[ EnqueueDeviceQueueSig \triangleq EnqueuePQueueSig[devs/procs] \]

\[ DequeueDeviceQueueSig \triangleq DequeuePQueueSig[devs/procs] \]

E.8.4 Operation preconditions

\[ \textbf{theorem} \; \text{tDeviceQueueInit} \]
\[ \exists DeviceQueue' \; \bullet \; DeviceQueueInit \]

\[ \textbf{theorem} \; \text{tEnqueueDeviceQueuePre} \]
\[ \forall EnqueueDeviceQueueSig \; \bullet \; \text{pre} \; \text{EnqueueDeviceQueue} \]

\[ \textbf{theorem} \; \text{tDequeueDeviceQueuePre} \]
\[ \forall DequeueDeviceQueueSig \; \bullet \; \text{pre} \; \text{DequeueDeviceQueue} \]

E.9 Scheduler

The scheduler in separation kernel is a non-preemptive round-robin scheduler. Furthermore, it has basic priority support, with device processes having priority over user processes. When no device/user processes exist, an idle process is run.

E.9.1 State

The scheduler employs 2 process queues - \( PQueue \) (represented as \( procs \) sequence variable) for user processes and \( DeviceQueue \) (\( devs \) variable) for devices. Furthermore, the scheduler contains references to current (\( curr \)) and idle (\( ipid \)) processes.

Originally, Craig modelled a reference to previously run process \( prev \). After careful examination, we found that it is not being used anywhere throughout the specification (only set - never read). Also, we did not find any real use for the reference in general kernel design.

Craig tries to use the \( prev \) variable (as well as \( curr \) in the same context) to reason about the scheduler in the corresponding simple kernel. However, his statements that \( prev \in \text{used} \lor \text{state}(prev) = \text{pterm} \) are invalid in the context of \( PTab \). He states that a process is either used or terminated, however when process is terminated and removed from \( used \), its attribute information such as state is deleted as well, thus \( \text{state}(prev) \) value is undefined. This further contributes to the redundancy and subsequent removal of \( prev \) reference.
Originally, Craig had no predicates on the scheduler. This caused large preconditions for operations interacting with process queues. Also, the restrictions on the variables were implied verbally (e.g. that one of the queues is for devices) but not modelled formally.

We have added state invariants to require that both the running and idle processes are known within the kernel. We have explicitly stated that \( \text{procs} \) is for user processes (of type \( \text{uproc} \)) and \( \text{devs} \) - for device processes (\( \text{dproc} \)).

Also, we have created a derived variable \( \text{queued} \), which represents processes from both queues. We require that neither running (current) nor idle process can be queued.

### E.9.2 Interesting properties

We show that no process can be enqueued in both queues - they are disjoint.

**Theorem** disabled rule lSchedSeparateQueues
\[
\forall Sched \bullet \text{ran} \ \text{procs} \cap \text{ran} \ \text{devs} = \emptyset
\]

We show that deadlock cannot occur in the scheduler by enqueuing all processes, because the proper subset assures that there always is a known process that is not in the queues.

**Theorem** tSchedQueuedSubsetUsed
\[
\forall Sched \bullet \text{queued} \subset \text{used}
\]

### E.9.3 Automation

**Theorem** disabled rule lISeqPIDHeadNotInTail
\[
\forall \text{procs} : \text{iseq} \ \text{PID} \mid \neg \text{procs} = \langle \rangle \bullet
\neg \text{head} \ \text{procs} \in \text{ran} \ (\text{tail} \ \text{procs})
\]

**Theorem** disabled rule lISeqPIDNotInTail
\[
\forall \text{procs} : \text{iseq} \ \text{PID} ; p : \text{PID} \mid \neg \text{procs} = \langle \rangle \land \neg p \in \text{ran} \ \text{procs} \bullet
\neg p \in \text{ran} \ (\text{tail} \ \text{procs})
\]

**Theorem** disabled rule lISeqPIDDisjointHead
\[
\forall \text{procs}, \text{devs} : \text{iseq} \ \text{PID} \mid \neg \text{devs} = \langle \rangle \land \text{ran} \ \text{devs} \cap \text{ran} \ \text{procs} = \{ \} \bullet
\neg \text{head} \ \text{devs} \in \text{ran} \ \text{procs}
\]

**Theorem** disabled rule lISeqPIDNotInHead
\[
\forall \text{procs} : \text{iseq} \ \text{PID} ; p : \text{PID} \mid \neg \text{procs} = \langle \rangle \land \neg p \in \text{ran} \ \text{procs} \bullet
\neg \text{head} \ \text{procs} = p
\]

**Theorem** disabled rule lIPIDDifferentPType
\[
\forall \text{ptype} : \text{PID} \rightarrow \text{PTYPE} ; \text{devs} : \text{P} \ \text{PID} ; p : \text{PID} ; \text{dtype} : \text{PTYPE} \mid
\text{devs} \in \text{P} \ (\text{ptype} \sim (\{ \text{dtype} \} \}) \land \neg \text{ptype} \ p = \text{dtype} \bullet \neg p \in \text{devs}
\]

The following theorems are special cases of \( \text{lIPIDDifferentPType} \) in order to be applied in large proofs without problems.

**Theorem** disabled rule lIPIDDifferentPTypeUproc
\[
\forall \text{ptype} : \text{PID} \rightarrow \text{PTYPE} ; \text{devs} : \text{P} \ \text{PID} ; p : \text{PID} \mid
\text{devs} \in \text{P} \ (\text{ptype} \sim (\{ \text{dproc} \} \}) \land \text{ptype} \ p = \text{uproc} \bullet \neg p \in \text{devs}
\]
theorem disabled rule IPIDDifferentPTypedProc
\( \forall \text{ptype} : \text{PID} \mapsto \text{PTYPE}; \text{procs} : \mathbb{P} \text{PID}; \text{p} : \text{PID} \mid \text{procs} \in \mathbb{P} (\text{ptype} \sim (\{\text{uproc}\}) \land \text{ptype} \text{p} = \text{dproc} \bullet \neg \text{p} \in \text{procs} \)

E.9.4 Error cases

We have added all following error cases, as they are coming from the added state invariants. Originally separation kernel scheduler had not defined error cases.

Errors\ notuserpid and notdevicepid are used when process of incorrect type is being added to a certain queue.

\[
\text{RaiseErrNotUserPID} \equiv \left[ \text{RaiseError} \mid e = \text{notuserpid} \right] \setminus (e?)
\]

\[
\text{RaiseErrNotDevicePID} \equiv \left[ \text{RaiseError} \mid e = \text{notdevicepid} \right] \setminus (e?)
\]

Errors badpidcurr and badpididle are used when the currently running or idle process is being added to a process queue.

\[
\text{RaiseErrBadPIDCurr} \equiv \left[ \text{RaiseError} \mid e = \text{badpidcurr} \right] \setminus (e?)
\]

\[
\text{RaiseErrBadPIDIdle} \equiv \left[ \text{RaiseError} \mid e = \text{badpididle} \right] \setminus (e?)
\]

E.9.5 Operations

Craig attempted to model process queues within \( \text{Sched} \) using promotion. His specification has a lot of syntax and type-checking mistakes because of such approach, as he applied the promotion erroneously. We could have corrected the mistakes in promotion, however decided to abandon this technique for scheduler. Promotion is not applicable here, because we would need to define 2 specific (not based on an identifier) promotion schemas and handle a large number of existential quantifiers in the proofs. Instead, we reuse process queues and their operations directly using conjunction (inclusion in scheduler schemas).

Initialisation

We have corrected the initialisation schema to use queue initialisation directly, without promotion. Furthermore, we changed the initial schema to contain the idle process as current. The original specification had minpid as initial value of curr. This came from the assumption that the idle process is assigned minpid identifier.

The actual specification has non-deterministic process allocation, therefore it is not stated that idle process has minpid as its PID value. Therefore, original scheduler initialisation required used to have at least 2 processes, \( p? \) and minpid, unless \( p? = \text{minpid} \). This is too restrictive as a single existing process should be enough to initialise scheduler - the idle process.

\[
\begin{align*}
\text{SchedInit} \\
\text{Sched}' \\
\text{PQueueInit} \\
\text{DeviceQueueInit} \\
p' : \text{PID} \\
\text{ipid}' = p' \\
\text{curr}' = \text{ipid}'
\end{align*}
\]

This scheduler initialisation shows that having PTabInit inside PQueueInit is too restrictive. The idle process needs to be allocated before the initialisation of SchedInit, therefore PTab must have been initialised before. The initialisation within PQueueInit results in SchedInit causing inconsistent specification.

We define the full initialisation of both process table and scheduler, including allocation of idle process.

\[
\text{PTabFullInit} \equiv \text{PTabInit} \land \text{ErrV'} \land \text{HW'}
\]

\[
\text{SchedPTabInit} \equiv \text{PTabFullInit} \land (\text{AddIdleProcess} \land \text{SchedInit[ip!/p?]}) \setminus (ip!, a!)
\]

Note that ErrV' and HW' are not restricted, because this initialisation requires allocated hardware-related information (memory segments) and therefore these schemas are likely to be initialised before.
We provide an expanded version of $SchedPTabInit$ to show that it is actually a complex initialisation operation (defining only post-variables), which satisfies $Sched$ state invariant.

\[
\text{\textbf{SchedPTabInitExpand}}
\]

\begin{align*}
\text{Sched}' \\
\text{ErrV}' \\
\text{HW}' \\
\text{tss? : TSS} \\
\text{maxMs? : } \mathbb{N}_1 \\
\text{cdAddr? : Addr; cdSize? : } \mathbb{N} \\
\text{dsAddr? : Addr; dsSize? : } \mathbb{N} \\
\text{used'} = \{\text{ipid}'\} \\
\text{nextupid'} = 2 \\
\text{extpid'} = \{1 \mapsto \text{ipid}'\} \\
\text{state'} = \{\text{ipid}' \mapsto \text{psready}\} \\
\text{tss'} = \{\text{ipid}' \mapsto \text{tss}'\} \\
\text{ptype'} = \{\text{ipid}' \mapsto \text{uproc}\} \\
\text{msgq'} = \{\text{ipid}' \mapsto \theta\text{MsgQ}[\text{maxMs} := \text{maxMs'}, \text{mq} := \emptyset]\} \\
\text{cdseg'} = \{\text{ipid}' \mapsto (\text{cdAddr'}, \text{cdSize'})\} \\
\text{dsseg'} = \{\text{ipid}' \mapsto (\text{dsAddr'}, \text{dsSize'})\} \\
\text{devmap'} = \emptyset \\
\text{devmsg'} = \emptyset \\
\text{devrpy'} = \emptyset \\
\text{serr'} = \text{sysok} \\
\text{curr'} = \text{ipid'} \\
\text{procs'} = \emptyset \\
\text{devs'} = \emptyset
\end{align*}

\textbf{theorem} \ tSchedPTabInitEquiv \\
$SchedPTabInit \iff SchedPTabInitExpand$

\textbf{Query and update operations}

\[
\textbf{IdleProcess}
\]

\begin{align*}
\exists \text{Sched} \\
p!' : \text{PID} \\
p! = \text{ipid}
\end{align*}

\[
\textbf{RunningProcess}
\]

\begin{align*}
\exists \text{Sched} \\
p!' : \text{PID} \\
p! = \text{curr}
\end{align*}

The current process is set using $UpdateCurrentProcess$.

\[
\textbf{UpdateCurrentProcess}
\]

\begin{align*}
\Delta \text{Sched} \\
p'? : \text{PID} \\
curr' = p'
\end{align*}

The \textit{prev} variable in $Sched$ schema was removed, therefore operations such as $SetPreviousProcess$ no longer apply. Its removal made $UpdateCurrentProcess$ exactly the same as $SetRunningProcess$. Because of this, to avoid maintenance of unneeded schemas, only $UpdateCurrentProcess$ operation is defined.
Enqueue operations

We have abandoned use of promotion for scheduler because it is not applicable here and was used incorrectly. We model operations that Craig originally defined using promotion by including corresponding queue operations with conjunction. Furthermore, we define them to keep remaining $Sched$ variables constrained, e.g., when a user process is enqueued, the device queue and running/idle processes should not change.

In $EnqueueUserSched0$ we reuse $EnqueuePQueue$ operation, which is a total operation and has its own failure cases. In the $EnqueueUserSched0$ operation, we add predicates to keep remaining $Sched$ variables unchanged.

$$ EnqueueUserSched0 \begin{array}{l} \Delta Sched \\ EnqueuePQueue \\ curr' = curr \\ ipid' = ipid \\ devs' = devs \end{array} $$

$$ EnqueueDeviceSched0 \begin{array}{l} \Delta Sched \\ EnqueueDeviceQueue \\ curr' = curr \\ ipid' = ipid \\ procs' = procs \end{array} $$

The state invariants of $Sched$ require that the enqueued process is of correct type and is neither idle process nor the currently running one. All other cases are handled as errors with the following error operations. In case of an error, scheduler state is not changed.

$$ NotUserPID0 \begin{array}{l} \Xi Sched \\ p? : PID \end{array} \\ p? \in used \\ \neg ptype p? = uproc $$

$$ NotDevicePID0 \begin{array}{l} \Xi Sched \\ p? : PID \end{array} \\ p? \in used \\ \neg ptype p? = dproc $$

We define error operations to signal corresponding errors.

$$ ErrNotUserPID \equiv (NotUserPID0 \land RaiseErrNotUserPID) $$

$$ ErrNotDevicePID \equiv (NotDevicePID0 \land RaiseErrNotDevicePID) $$

$$ ErrBadPIDCurr \equiv (RunningProcess[p?/p!] \land RaiseErrBadPIDCurr) $$

$$ ErrBadPIDIdle \equiv (IdleProcess[p?/p!] \land RaiseErrBadPIDIdle) $$

Note that in $EnqueueUserSched$ we cannot add the OK signal ($RaiseOk$), because $EnqueuePQueue$ already contains several error cases and using conjunction on them with $RaiseOk$ will produce a false state.

$$ EnqueueUserSched \equiv EnqueueUserSched0 \lor ErrNotUserPID \lor ErrBadPIDCurr \lor ErrBadPIDIdle $$

$$ EnqueueDeviceSched \equiv EnqueueDeviceSched0 \lor ErrNotDevicePID \lor ErrBadPIDCurr \lor ErrBadPIDIdle $$
Ready operations

The full operation of readying user process in the scheduler involves adding it to the queue and setting its state to ready. Since queue management does not change process table and setting process state requires to update PTab, we use schema composition to join the operations. Both enqueue and state operations are complete (constrain all variables within PTab), the composition is fully defined and does not involve non-determinism.

$EnqueueUserSched$ has a number of error cases, which are handled and $EnqueueUserSched$ is therefore a total operation. For all error cases, the error message is set in $ErrV$ schema, $serr$ variable. So, even if the operation fails, the result of $EnqueueUserSched$ is true. In order to ensure that process state is changed only when enqueuing succeeded, we need to check that the operation $SetStateToReadySched$ is executed only when system is in $sysok$ state - checked by $IsSysOk$. So, the whole composed operation $MakeReady$ will succeed only when both composed operations succeed.

The situation when the first part of the operation $EnqueueUserSched$ fails is handled by $ErrSysFailPTab$. In this case no change within process table is not done and error is preserved.

$$\begin{align*}
\Xi PTab \\
\Xi ErrV \\
\neg serr = sysok
\end{align*}$$

Now we define the full operation to make a user process ready within the scheduler (enqueue and set state to ready).

$MakeReady \equiv EnqueueUserSched \cup (IsSysOk \land SetStateToReady) \lor ErrSysFailPTab$

The $SetStateToReady$ operation has a precondition $p? \in used$, however the case when $p? \notin used$ is handled by $EnqueueUserSched$. This order of operations is convenient, because we do not need to have additional handling for case $p? \notin used$.

Craig claims that additional checks compromise the performance, therefore advocates for operations without these checks. This statement is based on wrong ideas about the application of formal methods and is not true.

We will not provide a full expansion of $MakeReady$, because it contains a lot of failure branches and the full expansion would be huge and not clear. So instead we will expand only the OK case in $MakeReadyOkExpand$.

$$\begin{align*}
\Delta Sched \\
\Xi HW \\
\Delta ErrV \\
PTabChangeState \\
p? : PID \\
state' = state \oplus \{p? \mapsto psready\} \\
curr' = curr \\
appid' = appid \\
procs' = procs \smallsetminus \{p?\} \\
deves' = deves \\
serr' = sysok
\end{align*}$$

theorem tMakeReadyOkEquiv

$$(MakeReady \land serr' = sysok) \leftrightarrow MakeReadyOkExpand$$

Also, as an example of error cases, we expand one of the failure cases defined in $EnqueueUserSched$. We show that no data is changed in case of failure (e.g. when enqueued process is already queued).
\textbf{Theorem} tMakeReadyAlreadyQueuedEquiv

\[(\text{MakeReady} \land \text{serr}' = \text{alreadyqueued}) \iff \text{MakeReadyAlreadyQueuedExpand}\]

The \textit{ReadyDeviceProcess} is analogous to the \textit{MakeReady} operations, so all justification and examples apply here directly.

\[\text{ReadyDeviceProcess} \equiv \text{EnqueueDeviceSched} \quad ((\text{IsSysOk} \land \text{SetStateToReady}) \lor \text{ErrSysFailPTab})\]

\textbf{Schedule operations}

The actual scheduling algorithm is simple - the devices have a priority over user processes. When no processes are queued, idle process is scheduled.

We define an operation to run idle process. It does not change the rest of scheduler.

Since \textit{RunIdleProcess} operation does not affect process queues and we do not use \textit{PQueue} operations, we do not need to use composition with state changing operation \textit{SetStateToRunning} and instead can join operations with conjunction.

\[
\begin{align*}
\text{RunIdleProcess} \\
\Delta \text{Sched} \\
\text{SetStateToRunning}[\text{ipid}/p?] \\
\text{curr}' = \text{ipid} \\
\text{ipid}' = \text{ipid} \\
\text{procs}' = \text{procs} \\
\text{devs}' = \text{devs}
\end{align*}
\]

Craig defines operation \textit{SKMakeUnready} to remove a running process from the scheduler, however it is actually obsoleted by suspend operation \textit{RequeueUserProcess} further in the specification, therefore we chose not to specify the original operation.

Next we define \textit{Run\langle Process\rangle\_Next} operations for idle, user and device processes. They contain scheduling instructions based on queue emptiness. Then they are joined with disjunction to form the full scheduling operation.

We need to make \textit{RunIdle\_Next0} homogenous with other scheduling operations therefore we add \textit{RaiseOk} to set \textit{sysok} state and keep interrupt unchanged.

\[
\begin{align*}
\text{RunIdle\_Next0} \\
\text{RunIdleProcess} \\
\text{RaiseOk} \\
\text{devs} = \langle \rangle \\
\text{procs} = \langle \rangle
\end{align*}
\]

Idle process is run only if both queues are empty.

User process is scheduled when there are processes in user queue and device queue is empty, thus allowing device processes to run first, if they are queued. User process is scheduled by removing it from the queue by reusing \textit{DequeuePQueue} from \textit{PQueue} definition, and setting it as the current running process. The scheduled process is also output in \text{p}!.
We can prove that because of the scheduling rules, this operation will always succeed and leave system in OK state.

**Theorem** tSchedUserNext0AlwaysOk
\[
\forall \text{SchedUserNext0} \cdot \text{serr}' = \text{sysok}
\]

We define RunUserNext0 as a composition, because SchedUserNext0 only manages the queue and does not change PTab, while SetStateToRunning affects state variable within PTab. In this composition we do not need to check that the system is in sysok state, because we have proven that is always true in tSchedUserNext0AlwaysOk.

\[
\text{RunUserNext0} \equiv (\text{SchedUserNext0}[n/p!] \circ \text{SetStateToRunning}[n/p?]) \setminus (n)
\]

Both composed operations constrain all variables within Sched and PTab.

We also provide an expanded version of running user process. Note PTabChangeState usage to constrain all variables within PTab except state.

**Theorem** tRunUserNext0Equiv
\[
\text{RunUserNext0} \Leftrightarrow \text{RunUserNext0Expand}
\]

The device process is scheduled when there are device processes in the device queue. The operation definition is similar to user’s.

**Theorem** tSchedDeviceNext0AlwaysOk
\[
\forall \text{SchedDeviceNext0} \cdot \text{serr}' = \text{sysok}
\]
\[ \text{RunDeviceNext}^0 \equiv (\text{SchedDeviceNext}^0[n/p!] \parallel \text{SetStateToRunning}[n/p?]) \setminus (n) \]

<table>
<thead>
<tr>
<th>RunDeviceNext0Expand</th>
<th>(\Delta \text{Sched} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Xi \text{HW} )</td>
<td>(\Delta \text{ErrV} )</td>
</tr>
<tr>
<td>(\text{PTabChangeState} )</td>
<td>(\text{devs} \neq \emptyset )</td>
</tr>
<tr>
<td>(\text{curr}' = \text{head \ devs} )</td>
<td>(\text{ipid}' = \text{ipid} )</td>
</tr>
<tr>
<td>(\text{procs}' = \text{procs} )</td>
<td>(\text{devs}' = \text{tail \ devs} )</td>
</tr>
<tr>
<td>(\text{state}' = \text{state} \oplus {\text{curr}' \mapsto \text{psrunning}} )</td>
<td>(\text{serr}' = \text{sysok} )</td>
</tr>
</tbody>
</table>

**Theorem** \(\text{tRunDeviceNext0Equiv} \)

\[ \text{RunDeviceNext}^0 \Leftrightarrow \text{RunDeviceNext}^0 \parallel \text{Expand} \]

For the full specification of \(\text{SchedNext} \), we combine all scheduling operations and also add the context switch. Originally, \(\text{CtxtSw} \) was added to each individual scheduling operation, however with composition distribution over disjunction, the \(\text{CtxtSw} \) operation can be moved outside.

\(\text{SchedNext} \equiv (\text{RunIdleNext}^0 \lor \text{RunUserNext}^0 \lor \text{RunDeviceNext}^0) \parallel \text{CtxtSw} \)

The \(\text{SchedNext} \) operation has no preconditions, as seen in \(\text{SchedNextSig} \) signature. Moreover, it has no failure cases, as the failures appearing from dequeuing are solved by the scheduling algorithm. This statement is also proved in \(\text{tSchedNextAlwaysOk} \) theorem, which shows that for all runs of \(\text{SchedNext} \), the system is always in \text{OK} state.

**Theorem** \(\text{tSchedNextAlwaysOk} \)

\[ \forall \text{SchedNext} \bullet \text{serr}' = \text{sysok} \]

**Suspend operations**

The \(\text{RequeueUserProcess} \) has a strange name and thus it is not entirely clear what this operation does. In fact, it is afterwards used in user process suspension operation and makes the major part of that operation. However, the \(\text{RequeueUserProcess} \) operation does not require that the requeued process be the running one (which is removed in the scheduling next process).

The \(\text{RequeueUserProcess} \) operation involves a schema composition, however since \(\text{SchedNext} \) always leaves the system in \text{OK} state, as show in \(\text{tSchedNextAlwaysOk} \), we do not need to check that system is \text{OK} in composition with \(\text{MakeReady} \) and do not need additional error case handling.

\(\text{RequeueUserProcess} \equiv (\text{SchedNext} \parallel \text{MakeReady}) \)

Originally, Craig defined the operation by composing two total operations. This means that first the next process is scheduled and afterwards the target process is enqueued. \(\text{MakeReady} \) operation has several error cases, therefore it may fail for certain system state and input values. However, the scheduling operation \(\text{SchedNext} \) is always executed successfully. Therefore the operation is not atomic - it is possible that one operation succeeds while another one fails.

Moreover, the previously applied checking that system is in \text{OK} state cannot be employed here. This order of operations is required to reuse the operation for suspension of current process. Enqueueing current process will always produce an error case, therefore we need to set new running process using \(\text{SchedNext} \).

To achieve atomicity, we employ the defined \text{Transaction} design pattern, which allows to roll back the transaction if any of the operations fail.
\[ Sched_{Backup} \triangleq \{ \Delta Sched; \; Sched'' \mid \theta Sched = \theta Sched'' \} \]

\[ Sched_{Revert} \triangleq \{ \Delta Sched'; \; Sched'' \mid \theta Sched' = \theta Sched'' \} \]

\[ Sched_{Trans0} \triangleq \{ \Xi Sched; \; \Xi ErrV; \; \Xi HW \mid serr = sysok \} \]

\[ ErrSched_{RevertTrans} \triangleq \{ \Delta Sched; \; \Xi ErrV; \; \Xi HW; \; Sched_{Revert} \mid serr \neq sysok \} \]

\[ Sched_{Trans} \triangleq Sched_{Trans0} \lor ErrSched_{RevertTrans} \]

\[ RequeueUserProcess_{Trans} \triangleq \exists \ Sched'' \bullet ((RequeueUserProcess \land Sched_{Backup}) \triangleright Sched_{Trans}) \]

We do not provide the expanded version of RequeueUserProcess or RequeueUserProcess_{Trans}, because it would be humongous. Since Sched_{Next} leaves the system in OK state, all error cases in this composite operation come from MakeReady. As witnessed before, MakeReady has 5 error cases, therefore RequeueUserProcess also has the same 5 error cases plus 1 success case. However, when combined with 3 execution branches in Sched_{Next}, we get a total of \(3 \times 6 = 18\) branches. This means that to show just OK case of RequeueUserProcess, we need an expansion schema with 3 branches of Sched_{Next}.

The requeue (suspend) operation for devices is analogous to that for user processes.

\[ RequeueDevice_{Process} \triangleq (Sched_{Next} \triangleright ReadyDevice_{Process}) \]

\[ RequeueDevice_{ProcessTrans} \triangleq \exists \ Sched'' \bullet ((RequeueDevice_{Process} \land Sched_{Backup}) \triangleright Sched_{Trans}) \]

### E.9.6 Operation precondition signatures

\[ RaiseErrNotUserPIDSig \triangleq RaiseErrSig \]

\[ RaiseErrNotDevicePIDSig \triangleq RaiseErrSig \]

\[ RaiseErrBadPIDCurrSig \triangleq RaiseErrSig \]

\[ RaiseErrBadPIDIdleSig \triangleq RaiseErrSig \]

\[ IdleProcessSig \triangleq Sched \]

\[ RunningProcessSig \triangleq Sched \]

\[ SchedOpSig \triangleq [Sched; \; p? : PID] \]

\[ UpdateCurrentProcessSig \triangleq [SchedOpSig \mid p? \in used \land p? \notin queued] \]

\[ SchedTotalSig \triangleq [Sched; \; RaiseErrSig] \]

\[ SchedOpTotalSig \triangleq [SchedOpSig; \; RaiseErrSig] \]

\[ EnqueueUserSched0Sig \triangleq [SchedOpTotalSig \mid \neg \text{ptype } p? = uproc \land p? \notin \{curr, ipid\}] \]

\[ EnqueueDeviceSched0Sig \triangleq [SchedOpTotalSig \mid \neg \text{ptype } p? = dproc \land p? \notin \{curr, ipid\}] \]

\[ ErrNotUserPIDSig \triangleq [SchedOpTotalSig \mid p? \in used \land \neg \text{ptype } p? = uproc] \]

\[ ErrNotDevicePIDSig \triangleq [SchedOpTotalSig \mid p? \in used \land \neg \text{ptype } p? = dproc] \]
ErrBadPIDCurrSig ≜ [SchedOpTotalSig | p? = curr]

ErrBadPIDIdleSig ≜ [SchedOpTotalSig | p? = ipid]

EnqueueUserSchedSig ≜ SchedOpTotalSig

EnqueueDeviceSchedSig ≜ SchedOpTotalSig

ErrSysFailPTabSig ≜ [PTab; ErrV | ¬ serr = sysok]

MakeReadySig ≜ SchedOpTotalSig

ReadyDeviceProcessSig ≜ SchedOpTotalSig

RunIdleProcessSig ≜ Sched

RunIdleNext0Sig ≜ [SchedTotalSig | devs = () ∧ procs = ()]

SchedUserNext0Sig ≜ [SchedTotalSig | devs = () ∧ procs ≠ ()]

RunUserNext0Sig ≜ [SchedTotalSig | devs = () ∧ procs ≠ ()]

SchedDeviceNext0Sig ≜ [SchedTotalSig | devs ≠ ()]

RunDeviceNext0Sig ≜ [SchedTotalSig | devs ≠ ()]

SchedNextSig ≜ SchedTotalSig

RequeueUserProcessSig ≜ SchedOpTotalSig

SchedBackupSig ≜ [Sched; Sched" | θSched = θSched"]

SchedRevertSig ≜ [Sched; Sched"]

SchedTrans0Sig ≜ [SchedTotalSig | serr = sysok]

ErrSchedRevertTransSig ≜ [SchedTotalSig; Sched" | serr ≠ sysok]

SchedTransSig ≜ [SchedTotalSig; Sched"]

RequeueUserProcessTransSig ≜ SchedOpTotalSig

RequeueDeviceProcessSig ≜ SchedOpTotalSig

RequeueDeviceProcessTransSig ≜ SchedOpTotalSig
E.9.7 Operation preconditions

**Theorem tSchedInit**
\[ \exists Sched'; p? : PID \bullet SchedInit \]

**Theorem tPTabFullInit**
\[ \exists PTab'; ErrV'; HW' \bullet PTabFullInit \]

**Theorem tSchedPTabInit**
\[ \exists Sched'; ErrV'; HW'; tss? : TSS; maxMs? : N_1; cdAddr? : Addr; cdSize? : N; dsAddr? : Addr; dsSize? : N \bullet SchedPTabInit \]

**Theorem tRaiseErrNotUserPIDPre**
\[ \forall RaiseErrNotUserPIDSig \bullet \text{pre} \ RaiseErrNotUserPID \]

**Theorem tRaiseErrNotDevicePIDPre**
\[ \forall RaiseErrNotDevicePIDSig \bullet \text{pre} \ RaiseErrNotDevicePID \]

**Theorem tRaiseErrBadPIDCurrPre**
\[ \forall RaiseErrBadPIDCurrSig \bullet \text{pre} \ RaiseErrBadPIDCurr \]

**Theorem tRaiseErrBadPIDIdlePre**
\[ \forall RaiseErrBadPIDIdleSig \bullet \text{pre} \ RaiseErrBadPIDIdle \]

**Theorem tIdleProcessPre**
\[ \forall IdleProcessSig \bullet \text{pre} \ IdleProcess \]

**Theorem tRunningProcessPre**
\[ \forall RunningProcessSig \bullet \text{pre} \ RunningProcess \]

**Theorem tUpdateCurrentProcessPre**
\[ \forall UpdateCurrentProcessSig \bullet \text{pre} \ UpdateCurrentProcess \]

**Theorem tEnqueueUserSched0Pre**
\[ \forall EnqueueUserSched0Sig \bullet \text{pre} \ EnqueueUserSched0 \]

**Theorem tEnqueueDeviceSched0Pre**
\[ \forall EnqueueDeviceSched0Sig \bullet \text{pre} \ EnqueueDeviceSched0 \]

**Theorem tErrNotUserPIDPre**
\[ \forall ErrNotUserPIDSig \bullet \text{pre} \ ErrNotUserPID \]

**Theorem tErrNotDevicePIDPre**
\[ \forall ErrNotDevicePIDSig \bullet \text{pre} \ ErrNotDevicePID \]

**Theorem tErrBadPIDCurrPre**
\[ \forall ErrBadPIDCurrSig \bullet \text{pre} \ ErrBadPIDCurr \]

**Theorem tErrBadPIDIdlePre**
\[ \forall ErrBadPIDIdleSig \bullet \text{pre} \ ErrBadPIDIdle \]

**Theorem tEnqueueUserSchedPre**
\[ \forall EnqueueUserSchedSig \bullet \text{pre} \ EnqueueUserSched \]
\begin{align*}
\text{theorem} \quad & t\text{EnqueueDeviceSchedPre} \\
& \forall \text{EnqueueDeviceSchedSig} \bullet \text{pre EnqueueDeviceSched} \\
\text{theorem} \quad & t\text{ErrSysFailPTabPre} \\
& \forall \text{ErrSysFailPTabSig} \bullet \text{pre ErrSysFailPTab} \\
\end{align*}

Theorems \text{lMakeReadyPreSysOk} and \text{lMakeReadyPreSysFail} are used to simplify the proof (to break it down into several cases and prove separately) of \text{tMakeReadyPre}.

\begin{align*}
\text{theorem} \quad & \text{lMakeReadyPreSysOk} \\
& \forall \text{SchedOpSig}: \text{RaiseErrSig}; \text{EnqueueUserSched} \mid \text{serr}' = \text{sysok} \bullet \\
& \quad \text{EnqueueUserSched}[\text{cmsg}/\text{cmsg}', \text{devmap}/\text{devmap}', \text{devmsg}/\text{devmsg}', \text{devrpy}/\text{devrpy}', \\
& \quad \quad \text{dsseg}/\text{dsseg}', \text{extpid}/\text{extpid}', \text{free}/\text{free}', \\
& \quad \quad \text{msgq}/\text{msgq}', \text{nextupid}/\text{nextupid}', \text{pidext}/\text{pidext}', \text{ptype}/\text{ptype}', \\
& \quad \quad \text{state}/\text{state}', \text{tss}/\text{tss}', \text{used}/\text{used}'] \land \\
& \quad ((\text{IsSysOk}[\text{serr}'/\text{serr}] \land \text{SetStateToReady}[\text{state}' := \text{state}' \oplus \{(p?, \text{psready})\}]) \\
& \quad \quad \lor \text{ErrSysFailPTab}[\text{serr}'/\text{serr}] = \text{state}' \oplus \{(p?, \text{psready})\}) \\
\text{theorem} \quad & \text{lMakeReadyPreSysFail} \\
& \forall \text{SchedOpSig}: \text{RaiseErrSig}; \text{EnqueueUserSched} \mid \neg \text{serr}' = \text{sysok} \bullet \\
& \quad \text{EnqueueUserSched}[\text{cmsg}/\text{cmsg}', \text{devmap}/\text{devmap}', \text{devmsg}/\text{devmsg}', \text{devrpy}/\text{devrpy}', \\
& \quad \quad \text{dsseg}/\text{dsseg}', \text{extpid}/\text{extpid}', \text{free}/\text{free}', \\
& \quad \quad \text{msgq}/\text{msgq}', \text{nextupid}/\text{nextupid}', \text{pidext}/\text{pidext}', \text{ptype}/\text{ptype}', \\
& \quad \quad \text{state}/\text{state}', \text{tss}/\text{tss}', \text{used}/\text{used}'] \land \\
& \quad ((\text{IsSysOk}[\text{serr}'/\text{serr}] \land \text{SetStateToReady}) \\
& \quad \quad \lor \text{ErrSysFailPTab}[\text{serr}'/\text{serr}] = \text{state}' \oplus \{(p?, \text{psready})\}) \\
\text{theorem} \quad & \text{tMakeReadyPre} \\
& \forall \text{MakeReadySig} \bullet \text{pre MakeReady} \\
\text{theorem} \quad & \text{lReadyDeviceProcessPreSysOk} \\
& \forall \text{SchedOpSig}: \text{RaiseErrSig}; \text{EnqueueDeviceSched} \mid \text{serr}' = \text{sysok} \bullet \\
& \quad \text{EnqueueDeviceSched}[\text{cmsg}/\text{cmsg}', \text{devmap}/\text{devmap}', \text{devmsg}/\text{devmsg}', \text{devrpy}/\text{devrpy}', \\
& \quad \quad \text{dsseg}/\text{dsseg}', \text{extpid}/\text{extpid}', \text{free}/\text{free}', \\
& \quad \quad \text{msgq}/\text{msgq}', \text{nextupid}/\text{nextupid}', \text{pidext}/\text{pidext}', \text{ptype}/\text{ptype}', \\
& \quad \quad \text{state}/\text{state}', \text{tss}/\text{tss}', \text{used}/\text{used}'] \land \\
& \quad ((\text{IsSysOk}[\text{serr}'/\text{serr}] \land \text{SetStateToReady}[\text{state}' := \text{state}' \oplus \{(p?, \text{psready})\}]) \\
& \quad \quad \lor \text{ErrSysFailPTab}[\text{serr}'/\text{serr}] = \text{state}' \oplus \{(p?, \text{psready})\}) \\
\text{theorem} \quad & \text{lReadyDeviceProcessPreSysFail} \\
& \forall \text{SchedOpSig}: \text{RaiseErrSig}; \text{EnqueueDeviceSched} \mid \neg \text{serr}' = \text{sysok} \bullet \\
& \quad \text{EnqueueDeviceSched}[\text{cmsg}/\text{cmsg}', \text{devmap}/\text{devmap}', \text{devmsg}/\text{devmsg}', \text{devrpy}/\text{devrpy}', \\
& \quad \quad \text{dsseg}/\text{dsseg}', \text{extpid}/\text{extpid}', \text{free}/\text{free}', \\
& \quad \quad \text{msgq}/\text{msgq}', \text{nextupid}/\text{nextupid}', \text{pidext}/\text{pidext}', \text{ptype}/\text{ptype}', \\
& \quad \quad \text{state}/\text{state}', \text{tss}/\text{tss}', \text{used}/\text{used}'] \land \\
& \quad ((\text{IsSysOk}[\text{serr}'/\text{serr}] \land \text{SetStateToReady}) \\
& \quad \quad \lor \text{ErrSysFailPTab}[\text{serr}'/\text{serr}] = \text{state}' \oplus \{(p?, \text{psready})\}) \\
\text{theorem} \quad & \text{tReadyDeviceProcessPre} \\
& \forall \text{ReadyDeviceProcessSig} \bullet \text{pre ReadyDeviceProcess} \\
\text{theorem} \quad & \text{tRunIdleProcessPre} \\
& \forall \text{RunIdleProcessSig} \bullet \text{pre RunIdleProcess} \\
\text{theorem} \quad & \text{tRunIdleNext0Pre} \\
& \forall \text{RunIdleNext0Sig} \bullet \text{pre RunIdleNext0}
\textbf{theorem} \( t\text{SchedUserNext}0\) \( \text{Pre} \)
\( \forall \text{SchedUserNext}0\) \( \text{Sig} \bullet \text{pre} \ SchedUserNext0 \)

\textbf{theorem} \( t\text{RunUserNext}0\) \( \text{Pre} \)
\( \forall \text{RunUserNext}0\) \( \text{Sig} \bullet \text{pre} \ RunUserNext0 \)

\textbf{theorem} \( t\text{SchedDeviceNext}0\) \( \text{Pre} \)
\( \forall \text{SchedDeviceNext}0\) \( \text{Sig} \bullet \text{pre} \ SchedDeviceNext0 \)

\textbf{theorem} \( t\text{RunDeviceNext}0\) \( \text{Pre} \)
\( \forall \text{RunDeviceNext}0\) \( \text{Sig} \bullet \text{pre} \ RunDeviceNext0 \)

\textbf{theorem} \( t\text{RunUserNext0IntnoUnchanged} \)
\( \forall \text{RunUserNext0} \mid \text{procs} \neq \{\} \bullet \text{intno}' = \text{intno} \)

\textbf{theorem} \( t\text{RunDeviceNext0IntnoUnchanged} \)
\( \forall \text{RunDeviceNext0} \mid \text{devs} \neq \{\} \bullet \text{intno}' = \text{intno} \)

\textbf{theorem} \( t\text{SchedNextPre} \)
\( \forall \text{SchedNext} \) \( \text{Sig} \bullet \text{pre} \ SchedNext \)

\textbf{theorem} \( t\text{SchedNextExistsPost} \)
\( \forall \text{SchedNext} \) \( \bullet \text{Sched}' \land HW' \)

\textbf{theorem} \( t\text{RequeueUserProcessPre} \)
\( \forall \text{RequeueUserProcess} \) \( \text{Sig} \bullet \text{pre} \ RequeueUserProcess \)

\textbf{theorem} \( t\text{SchedBackupPre} \)
\( \forall \text{SchedBackup} \) \( \text{Sig} \bullet \text{pre} \ SchedBackup \)

\textbf{theorem} \( t\text{SchedRevertPre} \)
\( \forall \text{SchedRevert} \) \( \text{Sig} \bullet \text{pre} \ SchedRevert \)

\textbf{theorem} \( t\text{SchedTrans0Pre} \)
\( \forall \text{SchedTrans0} \) \( \text{Sig} \bullet \text{pre} \ SchedTrans0 \)

\textbf{theorem} \( t\text{ErrSchedRevertTransPre} \)
\( \forall \text{ErrSchedRevertTrans} \) \( \text{Sig} \bullet \text{pre} \ ErrSchedRevertTrans \)

\textbf{theorem} \( t\text{SchedTransPre} \)
\( \forall \text{SchedTrans} \) \( \text{Sig} \bullet \text{pre} \ SchedTrans \)

\textbf{frule} \( f\text{EnqueueUserSched0ExistsPost} \)
\( \text{EnqueueUserSched0} \Rightarrow \text{Sched}' \)

\textbf{frule} \( f\text{ErrNotUserPIDExistsPost} \)
\( \text{ErrNotUserPID} \Rightarrow \text{Sched}' \)

\textbf{frule} \( f\text{ErrBadPIDCurrExistsPost} \)
\( \text{ErrBadPIDCurr} \Rightarrow \text{Sched}' \)
\begin{verbatim}
theorem frule fErrBadPIDIdleExistsPost
  ErrBadPIDIdle ⇒ Sched'

theorem lEnqueueUserSchedExistsPost
  ∀ EnqueueUserSched • Sched'

theorem frule fMakeReadyExistsPost
  MakeReady ⇒ Sched'

theorem lRequeueUserProcessExistsPost
  ∀ RequeueUserProcess • Sched'

theorem frule fEnqueueUserSched0ExistsPostHW
  EnqueueUserSched0 ⇒ HW'

theorem frule fErrNotUserPIDExistsPostHW
  ErrNotUserPID ⇒ HW'

theorem frule fErrBadPIDCurrExistsPostHW
  ErrBadPIDCurr ⇒ HW'

theorem frule fErrBadPIDIdleExistsPostHW
  ErrBadPIDIdle ⇒ HW'

theorem lEnqueueUserSchedExistsPostHW
  ∀ EnqueueUserSched • HW'

theorem frule fMakeReadyExistsPostHW
  MakeReady ⇒ HW'

theorem lRequeueUserProcessExistsPostHW
  ∀ RequeueUserProcess • HW'

RequeueUserProcessTransPre1 ≡ RequeueUserProcess ∧ SchedBackup

theorem tRequeueUserProcessTransPre1
  ∀ RequeueUserProcessSig; SchedBackupSig • pre RequeueUserProcessTransPre1

RequeueUserProcessTransPre2 ≡ (RequeueUserProcess ∧ SchedBackup) ⇒ SchedTrans

theorem tRequeueUserProcessTransPre2
  ∀ RequeueUserProcessSig; SchedBackupSig • pre RequeueUserProcessTransPre2

theorem tRequeueDeviceProcessPre
  ∀ RequeueDeviceProcessSig • pre RequeueDeviceProcess

theorem frule fEnqueueDeviceSched0ExistsPost
  EnqueueDeviceSched0 ⇒ Sched'
\end{verbatim}
\textbf{theorem} frule fErrNotDevicePIDExistsPost  
\texttt{ErrNotDevicePID} $\Rightarrow$ \texttt{Sched'}

\textbf{theorem} lEnqueueDeviceSchedExistsPost  
$\forall$ \texttt{EnqueueDeviceSched} $\bullet$ \texttt{Sched'}

\textbf{theorem} frule fReadyDeviceProcessExistsPost  
\texttt{ReadyDeviceProcess} $\Rightarrow$ \texttt{Sched'}

\textbf{theorem} lRequeueDeviceProcessExistsPost  
$\forall$ \texttt{RequeueDeviceProcess} $\bullet$ \texttt{Sched'}

\textbf{theorem} frule fEnqueueDeviceSched0ExistsPostHW  
\texttt{EnqueueDeviceSched0} $\Rightarrow$ \texttt{HW'}

\textbf{theorem} frule fErrNotDevicePIDExistsPostHW  
\texttt{ErrNotDevicePID} $\Rightarrow$ \texttt{HW'}

\textbf{theorem} lEnqueueDeviceSchedExistsPostHW  
$\forall$ \texttt{EnqueueDeviceSched} $\bullet$ \texttt{HW'}

\textbf{theorem} frule fReadyDeviceProcessExistsPostHW  
\texttt{ReadyDeviceProcess} $\Rightarrow$ \texttt{HW'}

\textbf{theorem} lRequeueDeviceProcessExistsPostHW  
$\forall$ \texttt{RequeueDeviceProcess} $\bullet$ \texttt{HW'}

\begin{align*}
\text{RequeueDeviceProcessTransPre1} & \equiv \text{RequeueDeviceProcess} \land \text{SchedBackup} \\
\textbf{theorem} \ t\text{RequeueDeviceProcessTransPre1} & \\
& \forall \texttt{RequeueDeviceProcessSig}; \texttt{SchedBackupSig} \bullet \text{pre} \ \texttt{RequeueDeviceProcessTransPre1} \\
\text{RequeueDeviceProcessTransPre2} & \equiv (\text{RequeueDeviceProcess} \land \text{SchedBackup}) \triangleright \text{SchedTrans} \\
\textbf{theorem} \ t\text{RequeueDeviceProcessTransPre2} & \\
& \forall \texttt{RequeueDeviceProcessSig}; \texttt{SchedBackupSig} \bullet \text{pre} \ \texttt{RequeueDeviceProcessTransPre2} \\
\textbf{theorem} \ t\text{RequeueDeviceProcessTransPre} & \\
& \forall \texttt{RequeueDeviceProcessTransSig} \bullet \text{pre} \ \texttt{RequeueDeviceProcessTrans}
\end{align*}
Appendix F

Proof Scripts

F.1 General lemmas proof scripts

\textbf{proof}[\textit{lInverseIsInj}]
   \texttt{invoke \texttt{\_ \_ \_ \_ \_ \_};}
   \texttt{with enabled (lPFunWeakening) prove;}

\textbf{proof}[\textit{lOverrideDisjointDom}]
   \texttt{apply oplusDef;}
   \texttt{apply ndresNothing;}
   \texttt{rewrite;}

\textbf{proof}[\textit{lNonEmptySeqHasHead}]
   \texttt{rewrite;}

\texttt{156}
proof\[l\text{SeqHeadNotInTail}\]
apply headDef to expression head\[X\] s;
use lSeqNonEmptySize\[X\][s := s];
rearrange;
rewrite;
split \(1 \in \text{dom}[Z, X]\) s;
cases;
prove;
next;
apply inDom to predicate \(1 \in \text{dom}[Z, X]\) s;
prenex;
use pairInFunction\([Z, Y]\)[f := s, x := 1, y := y];
rearrange;
rewrite;
rearrange;
simplify;
equality substitute s 1;
apply inRan to predicate \(y \in \text{ran}[Z, X]\) (tail\[X\] s);
rewrite;
apply tailDef to expression tail\[X\] s;
rewrite;
rearrange;
prenex;
with disabled (inRange) rewrite;
equality substitute;
split s 1 = s (1 + x);
simplify;
use pinjApplicationsEqual\([Z, X]\)[A := N, B := Y, f := s, x := 1, y := 1 + x];
rearrange;
rewrite;
rearrange;
rewrite;
next;

proof\[l\text{DisjointSeqHead}\]
simplify;
use lNonEmptySeqHasHead\[X]\];
split head\[X\] s \(\in\) ran \([Z, X]\) s;
prove;
split head\[X\] s \(\in\) ran \([Z, X]\) t;
prove;
apply extensionality to predicate ran \([Z, X]\) s \(\cap\) \([X]\) ran \([Z, X]\) t = \{\};
prove;
instantiate x == head\[X\] s;
rewrite;

proof\[l\text{HeadRanSubset}\]
with enabled (headDef, inRanFunction) rewrite;
instantiate x == 1;
with enabled (lSeqNonEmptySize) rewrite;
proof\[gEmptyRan\]
apply extensionality to predicate \( \text{ran}[X, Y, P] = \emptyset \);
prove;
split \( P = \emptyset \);
prove;
use nonEmptySetHasMember\[S := P\];
prove;
\( x \in X \times Y \);
cases;
prove;
next;
prove;
next;
\( x_0 == x.2 \);
apply \( \text{inRan} \);
prove;
\( x_0 == x.1 \);
prove;

proof\[gEmptyDom\]
apply extensionality to predicate \( \text{dom}[X, Y, P] = \emptyset \);
prove;
split \( P = \emptyset \);
prove;
use nonEmptySetHasMember\[S := P\];
prove;
\( x \in X \times Y \);
cases;
prove;
next;
prove;
next;
\( x_0 == x.1 \);
apply \( \text{inDom} \);
prove;
\( y == x.2 \);
prove;

proof\[lElemNotInDomNrres\]
prove;
apply \( \text{inDom} \);
prove;
\( y_0 == y \);
rewrite;

F.2 Basic Types Proof Scripts

proof\[tPIDConsistency\]
\( \text{minpid} == 1 \), \( \text{maxpid} == 9999 \);
rewrite;

proof\[gPIDMaxType\]
reduce;
proof [gPIDFinType]
  reduce;

proof [gPIDIsFinset]
  reduce;
  use minpid $declaration;
  use maxpid $declaration;
  apply inNat;
  rewrite;

proof [gPIDNotEmpty]
  apply extensionality;
  prove;
  instantiate \( x = \text{minpid} \);
  prove;

proof [MinpidIsPID]
  use minpid $declaration;
  apply inNat;
  reduce;

proof [tGPIDConsistency]
  instantiate \( \text{nullpid} = \text{maxpid} + 1 \);
  prove by reduce;
  use maxpid $declaration;
  apply inNat;
  simplify;

proof [gGPIDMaxType]
  invoke GPID;
  rewrite;

proof [tNullpidNotPID]
  invoke PID;
  use maxpid $declaration;
  apply inNat;
  use lRangeGPID \( p := \text{maxpid} \);
  invoke PID;
  prove;

proof [gUPIDMaxType]
  reduce;

proof [tDevNoConsistency]
  instantiate \( \text{mindev} = 1, \text{maxdev} = 9999 \);
  rewrite;

159
**F.3 Hardware Issues Proof Scripts**

```
proof[gDevNoType]
   reduce;
```

```
proof[tAddrConsistency]
   instantiate maxaddr == 9999;
   rewrite;
```

```
proof[gAddrType]
   reduce;
```

```
proof[gNulladdrInAddr]
   invoke Addr;
   use lRangeMaxAddr;
   prove;
```

```
proof[tSysOkExpandEquiv]
   prove by reduce;
```

```
proof[tSysOkEquiv]
   prove by reduce;
```

```
proof[tErrVInit]
   prove by reduce;
```

```
proof[tSetSysErrPre]
   reduce;
```

```
proof[tSysErrPre]
   reduce;
```

```
proof[tSysOkPre]
   reduce;
```

```
proof[tIsSysOkPre]
   reduce;
```

```
proof[tIntNoConsistency]
   instantiate minint == 1, maxint == 9999;
   rewrite;
```

proof[\text{gIntNoType}]
reduce;

proof[\text{MinIntIsIntNo}]
use minint$\$\$declaration;
apply inNat;
reduce;
use lRangeIntNo;
rewrite;

proof[\text{TSSNotEmpty}]
use TSS$\$\$declaration;
apply inPower1;
use nonEmptySetHasMember[S := TSS];
prenex;
instantiate t == x;
simplify;

proof[\text{tInterruptsConsistency}]
instantiate killintno == minint, ctxtswintno == maxint;
reduce;
use lRangeIntNo;
reduce;

proof[\text{gKillintnoType}]
use killintno$\$\$declaration;
reduce;

proof[\text{gctxtswintnoType}]
use ctxtswintno$\$\$declaration;
reduce;

proof[\text{tHWInit}]
reduce;
instantiate intno' == minint;
reduce;
use minint$\$\$declaration;
apply inNat;
use lRangeIntNo;
rewrite;

proof[\text{tRaiseInterruptPre}]
with disabled (IntNo) reduce;

proof[\text{tRaiseKillInterruptPre}]
with disabled (IntNo) reduce;
proof\[tCtxtSwPre\]
  with disabled (IntNo) reduce;

proof\[tRaiseErrorPre\]
  with disabled (IntNo) reduce;

proof\[tRaiseOkPre\]
  with disabled (IntNo) reduce;

proof\[tRaiseKillInterruptEquiv\]
  with disabled (IntNo) prove by reduce;

proof\[tCtxtSwEquiv\]
  with disabled (IntNo) prove by reduce;

F.4 Message queue proof scripts

proof\[MsgQ\$domainCheck\]
  rewrite;

proof\[tMsgQInit\]
  instantiate maxMs? == 1;
  prove by reduce;

F.5 Memory store segments proof scripts

proof\[gSDescType\]
  invoke SDesc;
  rewrite;

proof\[mkSDesc\$domainCheck\]
  reduce;

proof\[gSDescFunRelType\]
  use mkSDesc\$declaration;
  invoke (_ → _);
  invoke (_ → _);
  invoke SDesc;
  invoke (_ ↔ _);
  prove;
  apply inPower;
  prove;
  instantiate e_0 == e;
  apply inCross2;
  prove;
proof[ISDescTotalFun]
simplify;
  use mkISdesc$declaration;
  invoke (\_ \rightarrow \_);
prove;

F.6 Process table proof scripts
proof[tRaiseErrUnusedPDEquiv]
  with disabled (IntNo) prove by reduce;

proof[tRaiseErrPTabFullEquiv]
  with disabled (IntNo) prove by reduce;

proof[tRaiseErrUnusedPDPre]
  with disabled (IntNo) reduce;

proof[tRaiseErrPTabFullPre]
  with disabled (IntNo) reduce;

proof[PTabOriginal$domainCheck]
prove;

proof[PTabv1$domainCheck]
prove;

proof[gDevmapRelType]
reduce;

proof[gDevmapPFunType]
reduce;

proof[lDevmapPFunImpliesUnique]
rewrite;
  apply inDom to predicate d \in dom devmap;
rewrite;
  prenex;
  use l6ranElemType[Z, Z][A := DevNa, B := PID, R := devmap, x := d, y := y];
  use pairInFunction[Z, Z][f := devmap, x := d, y := y];
  rearrange;
rewrite;

proof[lPTabOriginalv1Equiv]
invoke PTabOriginal;
invoke PTabv1;
  with enabled (lDevmapPFunImpliesUnique) rewrite;
proof[PTabv2$domainCheck]
prove;

proof[PTabv1v2Refinement]
invoke PTabv2;
invoke PTabv1;
rewrite;

proof[PTabv3$domainCheck]
prove;

proof[DevmapUniqueImpliesPInj]
apply lPInjWeakening to predicate devmap ∈ DevNo → PID;
simplify;
prenex;
rewrite;
use memberFirstInDom[Z, Z][R := devmap, x := d1, y := r];
use memberSecondInRan[Z, Z][R := devmap, x := d1, y := r];
use pairInFunction[DevNo, PID][f := devmap, x := d1, y := r];
rarrange;
rewrite;
prenex;
rewrite;
use memberFirstInDom[Z, Z][R := devmap, x := d2, y := r];
rarrange;
rewrite;
prenex;
rewrite;
use pairInFunction[DevNo, PID][f := devmap, x := d2, y := r];
rarrange;
rewrite;
prenex;
rewrite;
use pinjApplicationsEqual[Z, Z][A := DevNo, B := PID, f := devmap, x := x, y := y];
rarrange;
rewrite;

proof[DevmapPInjImpliesUnique]
apply inRanFunction to predicate p ∈ ran devmap;
prenex;
rewrite;
prenex;
rewrite;
prenex;
use pinjApplicationsEqual[Z, Z][A := DevNo, B := PID, f := devmap, x := x, y := y];
rarrange;
rewrite;
\textbf{proof}[tPTabv2v3Equiv]
split PTabv2;
rewrite;
cases;
rewrite;
next;
split PTabv3;
rewrite;
next;
cases;
invoke PTabv2;
invoke PTabv3;
use lDevmapUniqueImpliesPInj;
rearrange;
simplify;
next;
split PTabv3;
rewrite;
next;
cases;
invoke PTabv2;
invoke PTabv3;
prenex;
use lDevmapPInjImpliesUnique;
prove;
next;

\textbf{proof}[lPtypeSetComprehension]
apply extensionality to predicate
\begin{align*}
\text{ptype} \sim \emptyset & \{ \{ \text{type} \} \} \\
= & \{ p_0 : \text{PID} \\
& \mid p_0 \in \text{dom} \text{ptype} \\
& \land \text{ptype} p_0 = \text{type} \};
\end{align*}
prove;

\textbf{proof}[lPtypeNotDproc]
use \text{PTYPE}$\in\text{member}[x \in \text{ptype} p]$;
use applyInRanPfun[\text{Z}, \text{PTYPE}][A := \text{PID}, B := \text{PTYPE}, f := \text{ptype}, a := p]$;
prove;

\textbf{proof}[lPtypeNotDprocSetEqual]
apply extensionality;
prove;
use lPtypeNotDproc[x/p];
rewrite;
proof[tPTabv3v4Equiv]
  split PTabv3;
  rewrite;
  cases;
  rewrite;
  next;
  split PTabv4;
  rewrite;
  next;
  cases;
  invoke PTabv3;
  invoke PTabv4;
  rewrite;
  equality substitute;
  use lPtypeSetComprehension[type := dproc];
  use lPtypeSetComprehension[type := uproc];
  split dom state = dom ptype;
  simplify;
  equality substitute dom state;
  with enabled (lPtypeNotDprocSetEqual) rewrite;
  next;
  invoke PTabv3;
  invoke PTabv4;
  rewrite;
  equality substitute;
  split dom state = dom ptype;
  simplify;
  equality substitute dom state;
  use lPtypeSetComprehension[type := dproc];
  use lPtypeSetComprehension[type := uproc];
  with enabled (lPtypeNotDprocSetEqual) prove;
  next;

proof[tPTabv4v5Refinement]
  invoke PTab;
  invoke PTabv4;
  prove;

proof[tPTabExtpidPinj]
  invoke PTab;
  use lInverseIsInj[Z, Z][A := UPID, B := PID, P := extpid, Q := pidext];
  prove;

proof[tPTabDprocsUprocsDisjoint]
  invoke PTab;
  with enabled (disjointCat) rewrite;
  apply extensionality to predicate
    (ptype ~ (\{ uproc \}) \cap (ptype ~ (\{ dproc \})) = {});
  prenex;
  rewrite;

166
proof\[PTab\text{Nextupid\_Unused}\]
  invoke PTab;
  instantiate \( u \) == nextupid;
  invoke UPID;
  rewrite;

proof\[\text{Nextupid\_Type}\]
  invoke PTab;
  rewrite;

proof\[\text{Nextupid\_UPID\_Type}\]
  rewrite;

proof\[\text{PTab\_Dprocs\_Power\_Used}\]
  invoke PTab;
  apply inPower;
  prove;

proof\[\text{PTab\_Uprocs\_Power\_Used}\]
  invoke PTab;
  apply inPower;
  prove;

proof\[\text{PTab\_Dom\_Pid\_ext}\]
  invoke PTab;
  prove;

proof\[\text{PTab\_Pid\_ext\_Ext\_pid}\]
  invoke PTab;
  prove;

proof\[\text{Dom\_Ext\_pid\_Subset\_Used}\]
  invoke PTab;
  prove;
  apply inPower;
  prenex;
  apply inImage;
  prove;
  use memberFirstInDom\[\mathbb{Z}, \text{PTYPE}\][R := ptype, x := e, y := uproc];
  prove;

proof\[\text{gPTab\_Used\_PID\_In\_Dom\_P\_type}\]
  invoke PTab;
  simplify;

proof\[\text{gPTab\_Used\_PID\_In\_Dom\_State}\]
  invoke PTab;
  simplify;
proof[fDevmsgMaxType]
  reduce;

proof[fCdsegMaxType]
  reduce;

proof[fDssegMaxType]
  reduce;

proof[gPTabUnknownPIDDproc]
  invoke PTab;
  rewrite;

proof[lPTabUnusedUPIDs]
  invoke PTab;
  instantiate u_0 == u;
  rewrite;

proof[lExistFreePIDs]
  prove;
  apply extensionality;
  prove;
  instantiate p == y;
  simplify;

proof[lPTabGotFreePIDsEquiv]
  invoke PTab;
  with enabled (diffSuperset) prove;
  apply extensionality to predicate PID \ dom state = {};
  rewrite;
  apply extensionality to predicate dom state = PID;
  prenex;
  rewrite;
  instantiate x == y;
  rewrite;
proof[tAddProcUPIDEquiv]
  split AddProcUPID;
  rewrite;
  cases;
  rewrite;
  next;
  split AddProcUPIDOriginal;
  rewrite;
  next;
  cases;
  invoke AddProcUPID;
  invoke AddProcUPIDOriginal;
  with enabled (lExtPidOplusIsCup) prove;
  next;
  invoke AddProcUPID;
  invoke AddProcUPIDOriginal;
  with enabled (lExtPidOplusIsCup) prove;
  next;

proof[tNewUPIDForProcessEquiv]
  with disabled (PTab, PID, UPID) invoke;
  prove;
  invoke PTab;
  instantiate u == nextupid;
  prove;

proof[tNewUPIDIsUser]
  with disabled (PID, UPID) invoke;
  prove;
  apply extensionality to predicate
  \[ ptype' \sim (\{ uproc \} \cup (ptype \sim (\{ uproc \} \cup))) \];
  instantiate y == p!;
  rewrite;

proof[tNewUProcInfo$domainCheck]
  with enabled (lSDescTotalFun) rewrite;

proof[tNewUProcMsgQEQequiv]
  with disabled (PTab, PID) prove by reduce;

proof[tAddPD0Equiv]
  split AddPD0;
  rewrite;
  cases;
  rewrite;
  next;
  split AddPD0Original;
  rewrite;
  next;
  cases;
  with disabled (NewUPIDForProcess, PTab, PID) prove by reduce;
  next;
  with disabled (NewUPIDForProcess, PTab, PID) prove by reduce;
  next;
proof[tAddPDEquiv]
  split AddPD \land AddPDInterface;
  rewrite;
  cases;
  rewrite;
  next;
  split AddPDExpand;
  rewrite;
  next;
  cases;
  with disabled (PTab, PID, DevNo, UPID, IntNo, Addr) prove by reduce;
  split used = PID;
  with enabled (lSDescFunDef) prove;
  next;
  with disabled (PTab, PID, DevNo, UPID, IntNo, Addr) prove by reduce;
  split used = PID;
  with enabled (lSDescFunDef) prove;
  next;

proof[tAddPD0NotFree]
  invoke AddPD0;
  invoke NewUPIDForProcess;
  invoke \Delta PTab;
  invoke PTab;
  prove;

proof[PIDforUPID$domainCheck]
  prove;

proof[tDeleteAllExtpid]
  invoke DeleteAllProcesses;
  invoke \Delta PTab;
  use lDomExtpidSubsetUsed;
  prove;
  with disabled (PID, UPID, DevNo, Addr) invoke;
  prove;
  use gEmptyRan[\mathbb{Z}, \mathbb{Z}][A := UPID, B := PID, P := extpid'];
  prove;
  split ran extpid' = {};
  cases;
  prove;
  next;
  prove;
  apply extensionality to predicate ran extpid' = {};
  prove;
  next;
proof\[tSetStateToReadyEquiv\]
    split SetStateToReady;
    rewrite;
    cases;
    rewrite;
    next;
    split SetStateToReadyExpand;
    rewrite;
    next;
    cases;
    with disabled (PTab, PID, UPID, DevNo, Addr) prove by reduce;
    next;
    with disabled (PTab, PID, UPID, DevNo, Addr) prove by reduce;
    invoke PTab;
    rewrite;
    next;

proof\[tSetStateToReadyOrigEquiv\]
    with disabled (SetProcState) prove by reduce;

proof\[tSetStateToRunningOrigEquiv\]
    with disabled (SetProcState) prove by reduce;

proof\[tSetStateToTerminatedOrigEquiv\]
    with disabled (SetProcState) prove by reduce;

proof[ProcType\$domainCheck]
    invoke;
    simplify;

proof[ProcState\$domainCheck]
    invoke;
    simplify;

proof[tPTabInit]
    reduce;
    instantiate
    extpid' == {},
    pidext' == {},
    free' == PID,
    tss' == {},
    devmap' == {},
    state' == {},
    ptype' == {},
    msgq' == {},
    devmsg' == {},
    devrpy' == {},
    cdseg' == {},
    dsseg' == {};
    invoke PTab;
    invoke UPID;
    rewrite;
proof[\text{tUsedPIDPre}] with disabled (PTab, PID, DevNo, UPID) prove by reduce;

proof[\text{tGotFreePIDsPre}] with disabled (PTab, PID, DevNo, UPID) prove by reduce;

proof[\text{LPtypeCupImageDifferent}] apply imageDef; rewrite;

proof[\text{LPtypeCupImageSame}] apply imageDef; rewrite;

proof[\text{gPTabEmptyDprocs}] invoke PTab; prove;
apply extensionality to predicate ptype ∼ ( \{ dproc \} ) = {};
prove;

proof[\text{gPTabEmptyUprocs}] invoke PTab; prove;
apply extensionality to predicate ptype ∼ ( \{ uproc \} ) = {};
prove;

proof[\text{gPTabEmptyExtpid}] use gPTabEmptyUprocs with disabled (gPTabEmptyUprocs) prove;
invoke PTab;
use gEmptyRan[Z, Z] [A := UPID, B := PID, P := extpid];
prove;

proof[\text{gPTabEmptyState}] invoke PTab;
use gEmptyDom[Z, PSTATE] [A := PID, B := PSTATE, P := state];
prove;

proof[\text{gPTabEmptyTss}] invoke PTab;
use gEmptyDom[Z, TSS] [A := PID, B := TSS, P := tss];
prove;

proof[\text{gPTabEmptyPtype}] invoke PTab;
use gEmptyDom[Z, PTYPE] [A := PID, B := PTYPE, P := ptype];
prove;
proof[\textit{gPTabEmptyMsgq}]
  use \textit{gPTabEmptyUprocs};
  with disabled (\textit{gPTabEmptyUprocs}) prove;
  invoke \textit{PTab};
  use \textit{gEmptyDom}[\mathbb{Z}, \langle | maxMs : \mathbb{Z}; \ mq : \mathbb{P} (\mathbb{Z} \times \mathbb{Z}) | \rangle]
  \[ A := \text{PID}, B := \text{MsgQ}, P := \text{msgq}; \]
  prove;

proof[\textit{gPTabEmptyCdseg}]
  use \textit{gPTabEmptyUprocs};
  with disabled (\textit{gPTabEmptyUprocs}) prove;
  invoke \textit{PTab};
  use \textit{gEmptyDom}[\mathbb{Z}, \mathbb{Z} \times \mathbb{Z}][A := \text{PID}, B := \text{SDesc}, P := \text{cdseg}];
  prove;

proof[\textit{gPTabEmptyDsseg}]
  use \textit{gPTabEmptyUprocs};
  with disabled (\textit{gPTabEmptyUprocs}) prove;
  invoke \textit{PTab};
  use \textit{gEmptyDom}[\mathbb{Z}, \mathbb{Z} \times \mathbb{Z}][A := \text{PID}, B := \text{SDesc}, P := \text{dsseg}];
  prove;

proof[\textit{gPTabEmptyDevmap}]
  use \textit{gPTabEmptyDprocs};
  with disabled (\textit{gPTabEmptyDprocs}) prove;
  invoke \textit{PTab};
  use \textit{gEmptyRan}[\mathbb{Z}, \mathbb{Z}][A := \text{DevNo}, B := \text{PID}, P := \text{devmap}];
  prove;

proof[\textit{gPTabEmptyDevmsg}]
  use \textit{gPTabEmptyDprocs};
  with disabled (\textit{gPTabEmptyDprocs}) prove;
  invoke \textit{PTab};
  use \textit{gEmptyDom}[\mathbb{Z}, \mathbb{Z} \times \mathbb{MSG}][A := \text{PID}, B := \text{GPID} \times \text{MSG}, P := \text{devmsg}];
  prove;

proof[\textit{gPTabEmptyDevrpy}]
  use \textit{gPTabEmptyDprocs};
  with disabled (\textit{gPTabEmptyDprocs}) prove;
  invoke \textit{PTab};
  use \textit{gEmptyDom}[\mathbb{Z}, \mathbb{MSG}][A := \text{PID}, B := \text{MSG}, P := \text{devrpy}];
  prove;

proof[\textit{AllocPIDDproc}]
  invoke \textit{PTab};
  invoke \textit{GPID};
  with enabled (\textit{cupInPinj}, \textit{lPtypeCupImageDifferent}, \textit{lPtypeCupImageSame}) prove;
  instantiate \( u_0 \) == \( u \);
  prove;

173
proof[AllocPIDUproc]
invoke PTab;
with enabled (cupInPinj, lPtypeCupImageDifferent, lPtypeCupImageSame) prove;
with disabled (PID, Addr) prove by reduce;
instantiate u_1 == u;
prove;
instantiate u_1 == u_0;
prove;

proof[tAllocPIDDprocPre]
use lTSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
instantiate
p! == p,
nextupid' == nextupid,
free' == free \ {p},
tss' == tss \ {(p, t)},
state' == state \ {(p, psready)},
ptype' == ptype \ {(p, dproc)},
extpid' == extpid,
pidext' == pidext,
msgq' == msgq,
cdseq' == cdseg,
dsseq' == dsseq,
devidmap' == devmap \ {(d, p)},
devidmsg' == devmsg \ {(p, (nullpid, nullmsg))},
devidrpy' == devrpy \ {(p, nullmsg)};
rewrite;
use lAllocPIDDproc[st := psready];
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

proof[tAllocPIDUprocPre]
use lTSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
instantiate
p! == p,
nextupid' == nextupid + 1,
free' == free \ {p},
tss' == tss \ {(p, t)},
state' == state \ {(p, psready)},
ptype' == ptype \ {(p, uproc)},
extpid' == extpid \ {(nextupid, p)},
(pidext' == pidext \ {(p, nextupid)},
msgq' == msgq \ {(p, θ MsgQ mq : { }, maxMs := 1)}},
cdseq' == cdseq \ {(p, (nulladdr, 0))},
dsseq' == dsseq \ {(p, (nulladdr, 0))},
devidmap' == devmap,
devidmsg' == devmsg,
devidrpy' == devrpy;
rewrite;
use lAllocPIDUproc[st := psready, u := nextupid, maxMs := 1,
cdAddr := nulladdr, cdSize := 0, dsAddr := nulladdr, dsSize := 0];
prove;
proof[AllocUPIDPre]

with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

  instantiate
    used' == used,
    free' == free,
    tss' == tss,
    state' == state,
    ptype' == ptype,
    extpid' == extpid,
    pidext' == pidext,
    msgq' == msgq,
    cdseg' == cdseg,
    dsseg' == dsseg,
    devmap' == devmap,
    devmsg' == devmsg,
    devrpy' == devrpy;

  prove by reduce;

proof[ExtpidOplusIsCup]

use
  lOverrideDisjointDom[Z, Z][A := UPID, B := PID, P := extpid, Q := \{(u, p)\}];

prove;

proof[ExtpidInvCup]

use lPTabPidextExtpid;

split pidext = extpid ~;

prove;

proof[AddProcUPIDPre]

use ITSSNotEmpty;

with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

  instantiate
    nextupid' == u? + 1,
    used' == used \cup \{p?\},
    free' == free \setminus \{p?\},
    tss' == tss \cup \{(p?, 0)\},
    state' == state \cup \{(p?, psready)\},
    ptype' == ptype \cup \{(p?, uproc)\},
    pidext' == pidext \cup \{(p?, u?)\},
    msgq' == msgq \cup \{(p?, 0) MsgQ[msg := \langle \rangle, maxMs := 1]\},
    cdseg' == cdseg \cup \{(p?, (nulladdr, 0))\},
    dsseg' == dsseg \cup \{(p?, (nulladdr, 0))\},
    devmap' == devmap,
    devmsg' == devmsg,
    devrpy' == devrpy;

  rewrite;

use lAllocPIDUproc[p := p?, st := psready, u := u?, maxMs := 1, cdAddr := nulladdr, cdSize := 0, dsAddr := nulladdr, dsSize := 0];

prove;

invoke PTab;

  instantiate u == u?;

prove;
proof[tAddProcUPIDOriginalPre]
use lTSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

 instantiate
  nextupid' == u? + 1,
  used' == used ∪ {p?},
  free' == free \ {p?},
  tss' == tss ∪ {(p?, t)},
  state' == state ∪ {(p?, psready)},
  ptype' == ptype ∪ {(p?, uproc)},
  pidext' == (extpid ⊕ {(a?, p?)})~,
  msgq' == msgq ∪ {(p?, θ MsgQ[mq := ( ), maxMs := 1]}},
  cdseg' == cdseg ∪ {(p?, (nulladdr, 0))},
  dsseg' == dsseg ∪ {(p?, (nulladdr, 0))},
  devmap' == devmap,
  devmsg' == devmsg,
  devrpy' == devrpy;
with enabled
  (lPTabUnusedUPIDs, lExtPIDOpplusIsCup, lExtPIDInvCup) rewrite;
use lAllocPIDUproc[p := p?, st := psready, u := u?, maxMs := 1,
  cdAddr := nulladdr, cdSize := 0, dsAddr := nulladdr, dsSize := 0];
prove;
invoke PTab;
prove;

proof[tNewUPIDForProcessPre]
use lTSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

 instantiate
  p! == p,
  free' == free \ {p},
  tss' == tss ∪ {(p, t)},
  state' == state ∪ {(p, psready)},
  ptype' == ptype ∪ {(p, uproc)},
  pidext' == pidext ∪ {(p, nextupid)},
  msgq' == msgq ∪ {(p, θ MsgQ[mq := ( ), maxMs := 1]}},
  cdseg' == cdseg ∪ {(p, (nulladdr, 0))},
  dsseg' == dsseg ∪ {(p, (nulladdr, 0))},
  devmap' == devmap,
  devmsg' == devmsg,
  devrpy' == devrpy;
rewrite;
use lAllocPIDUproc[st := psready, u := nextupid, maxMs := 1,
  cdAddr := nulladdr, cdSize := 0, dsAddr := nulladdr, dsSize := 0];
prove;


proof\{tSetDProcTypePre\}
use lTSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
instantiate
\(nextupid' = nextupid\),
\(used' = used \cup \{p?\}\),
\(free' = free \setminus \{p?\}\),
\(tss' = tss \cup \{(p?, t)\}\),
\(state' = state \cup \{(p?, psready)\}\),
\(extpid' = extpid\),
\(pidext' = pidext\),
\(msgq' = msgq\),
\(cdseg' = cdseg\),
\(dsseg' = dsseg\),
devmap' = devmap \cup \{(d, p?)\},
devmsg' = devmsg \cup \{(p?, (nullpid, nullmsg))\},
devrpy' = devrpy \cup \{(p?, nullmsg)\};
rewrite;
use lAllocPIDDproc[p := p?, st := psready];
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

proof\{tSetUProcTypePre\}
use lTSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
instantiate
\(nextupid' = nextupid + 1\),
\(used' = used \cup \{p?\}\),
\(free' = free \setminus \{p?\}\),
\(tss' = tss \cup \{(p?, 1)\}\),
\(state' = state \cup \{(p?, psready)\}\),
\(extpid' = extpid \cup \{(nextupid, p?)\}\),
\(pidext' = pidext \cup \{(p?, nextupid)\}\),
\(msgq' = msgq \cup \{(p?, \theta \text{MsgQ}\{mq := \{\}, maxMs := 1\})\}\),
\(cdseg' = cdseg \cup \{(p?, (nulladdr, 0))\}\),
\(dsseg' = dsseg \cup \{(p?, (nulladdr, 0))\}\),
devmap' = devmap,
devmsg' = devmsg,
devrpy' = devrpy;
rewrite;
use lAllocPIDUproc[p := p?, st := psready, u := nextupid, maxMs := 1,
cdAddr := nulladdr, cdSize := 0, dsAddr := nulladdr, dsSize := 0];
prove;
proof\[\text{tAddPDdescDprocPre}\]
use lTSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

instantiate
nextupid' == nextupid + 1,
used' == used \cup \{p\},
free' == free \setminus \{p\},
ptype' == ptype \cup \{(p?, dproc)\},
extpid' == extpid,
pidxt' == pidxt,
msgq' == msgq,
cdseg' == cdseg,
dsseg' == dsseg,
devmap' == devmap \cup \{(d, p?)\},
devmsg' == devmsg \cup \{(p?, (nullpid, nullmsg))\},
devrpy' == devrpy \cup \{(p?, nullmsg)\};
rewrite;
use lAllocPIDDproc[p := p?, st := psready, t := tss?];
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

proof\[\text{tAddPDdescUprocPre}\]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;

instantiate
nextupid' == nextupid,
used' == used \cup \{p\},
free' == free \setminus \{p\},
ptype' == ptype \cup \{(p?, uproc)\},
extpid' == extpid \cup \{(nextupid, p?)\},
pidxt' == pidxt \cup \{(p?, nextupid)\},
msgq' == msgq \cup \{(p?, \theta \MsgQ\{mq := (\langle\rangle, maxMs := 1)\})\},
cdseg' == cdseg \cup \{(p?, nulladdr, 0)\},
dsseg' == dsseg \cup \{(p?, nulladdr, 0)\},
devmap' == devmap,
devmsg' == devmsg,
devrpy' == devrpy;
rewrite;
use lAllocPIDUproc[p := p?, st := psready, u := nextupid, t := tss?, maxMs := 1, cdAddr := nulladdr, cdSize := 0, dsAddr := nulladdr, dsSize := 0];
prove;
proof[PromotePTabMNewPre]
use ITSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
instantiate
nextupid' == nextupid + 1,
used' == used \ {p?},
free' == free \ {p?},
tss' == tss \ \{p?, t\},
ptype' == ptype \ \{(p?, uproc)\},
state' == state \ \{(p?, psready)\},
extpid' == extpid \ \{(nextupid, p?)\},
pidext' == pidext \ \{(p?, nextupid)\},
cdseg' == cdseg \ \{(p?, (nulladdr, 0))\},
dseg' == dseg \ \{(p?, (nulladdr, 0))\},
tss' == tss \ \{(p?, t)\},
ptype' == ptype \ \{(p?, uproc)\},
state' == state \ \{(p?, psready)\},
extpid' == extpid \ \{(nextupid, p?)\},
pidext' == pidext \ \{(p?, nextupid)\},
cdseg' == cdseg \ \{(p?, (nulladdr, 0))\},
dseg' == dseg \ \{(p?, (nulladdr, 0))\},
devmap' == devmap,
devmsg' == devmsg,
devrpy' == devrpy,
maxMs' == maxMs,
mq' == \langle \rangle;
rewrite;
use lAllocPIDUproc[p := p?, st := psready, u := nextupid, maxMs := maxMs?,
   cdAddr := nulladdr, cdSize := 0, dsAddr := nulladdr, dsSize := 0];
prove;

proof[NewUProcMsgQPre]
use ITSSNotEmpty;
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
instantiate
nextupid' == nextupid + 1,
used' == used \ {p?},
free' == free \ {p?},
tss' == tss \ \{p?, t\},
ptype' == ptype \ \{(p?, uproc)\},
state' == state \ \{(p?, psready)\},
extpid' == extpid \ \{(nextupid, p?)\},
pidext' == pidext \ \{(p?, nextupid)\},
cdseg' == cdseg \ \{(p?, (nulladdr, 0))\},
dseg' == dseg \ \{(p?, (nulladdr, 0))\},
tss' == tss \ \{(p?, t)\},
ptype' == ptype \ \{(p?, uproc)\},
state' == state \ \{(p?, psready)\},
extpid' == extpid \ \{(nextupid, p?)\},
pidext' == pidext \ \{(p?, nextupid)\},
cdseg' == cdseg \ \{(p?, (nulladdr, 0))\},
dseg' == dseg \ \{(p?, (nulladdr, 0))\},
devmap' == devmap,
devmsg' == devmsg,
devrpy' == devrpy,
maxMs' == maxMs,
mq' == \langle \rangle;
rewrite;
use lAllocPIDUproc[p := p?, st := psready, u := nextupid, maxMs := maxMs?,
   cdAddr := nulladdr, cdSize := 0, dsAddr := nulladdr, dsSize := 0];
prove;
proof[\text{tNewUProcInfoPre}]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
  instantiate
  nextupid' == nextupid + 1,
  used' == used \cup \{p?\},
  free' == free \setminus \{p?\},
  extpid' == extpid \cup \{(nextupid, p?)\},
  pidext' == pidext \cup \{(p?, nextupid)\};
rewrite;
use lAllocPIDUproc[p := p?, st := psready, u := nextupid, 
  t := tss?, maxMs := maxMs?,
  cdAddr := cdAddr?, cdSize := cdSize?,
  dsAddr := dsAddr?, dsSize := dsSize?];
with enabled (lSDescFunDef) prove;

proof[\text{tAddPD0Pre}]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
use lExistFreePIDs;
prove;
  instantiate
  p! == p,
  free' == free \setminus \{p\},
  pidext' == pidext \cup \{(p, nextupid)\};
rewrite;
use lAllocPIDUproc[st := psready, u := nextupid, 
  t := tss?, maxMs := maxMs?,
  cdAddr := cdAddr?, cdSize := cdSize?,
  dsAddr := dsAddr?, dsSize := dsSize?];
with enabled (lSDescFunDef) prove;

proof[\text{tErrPTabFullPre}]
with disabled (PTab, PID, DevNo, UPID, IntNo) prove by reduce;
proof[tAddPDPre]
  invoke AddPDSig;
  split used ⊂ PID;
  cases;
  use tAddPD0Pre;
  with disabled (AddPD0, PTab, PID, IntNo, Addr, ErrPTabFull) prove by reduce;
  instantiate
    nextupid_0' == nextupid',
    used_0' == used',
    free_0' == free',
    tss_0' == tss',
    ptype_0' == ptype',
    state_0' == state',
    extpid_0' == extpid',
    pidext_0' == pidext',
    msgq_0' == msgq',
    cdseg_0' == cdseg',
    dsseg_0' == dsseg',
    devmap_0' == devmap',
    devmsg_0' == devmsg',
    devrpy_0' == devrpy',
    p_0! == p!,
    u_0! == u!,
    intno' == intno,
    serr' == sysok;
  rewrite;
  next;
  use tErrPTabFullPre;
  with disabled (AddPD0, PTab, PID, IntNo, Addr, ErrPTabFull) prove by reduce;
  instantiate
    nextupid_0' == nextupid',
    used_0' == used',
    free_0' == free',
    tss_0' == tss',
    ptype_0' == ptype',
    state_0' == state',
    extpid_0' == extpid',
    pidext_0' == pidext',
    msgq_0' == msgq',
    cdseg_0' == cdseg',
    dsseg_0' == dsseg',
    devmap_0' == devmap',
    devmsg_0' == devmsg',
    devrpy_0' == devrpy',
    intno_0' == intno',
    serr_0' == serr',
    p! == minpid,
    u! == 1;
  rewrite;
  next;
\textbf{proof[\texttt{tAddIdleProcessPre}]}
\begin{verbatim}
invoke AddIdleProcessSig;
invoke AddIdleProcess;
use \texttt{tAddPDPre};
prove;
  instantiate
  nextupid_0' == nextupid',
  used_0' == used',
  free_0' == free',
  tss_0' == tss',
  ptype_0' == ptype',
  state_0' == state',
  extpid_0' == extpid',
  pidext_0' == pidext',
  msgq_0' == msgq',
  cdseg_0' == cdseg',
  dsseg_0' == dsseg',
  devmap_0' == devmap',
  devmsg_0' == devmsg',
  devrpy_0' == devrpy',
  ip! == p!,
  u_0! == u!,
  intno_0' == intno',
  serr_0' == serr';
rewrite;
\end{verbatim}

\textbf{proof[\texttt{tPIDforUPIDPre}]}
\begin{verbatim}
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
invoke PTab;
prove;
use applyInRanPfun[Z, Z][A := UPID, B := PID, f := extpid, a := u?];
prove;
\end{verbatim}

\textbf{proof[\texttt{lFreePID}]}
\begin{verbatim}
invoke PTab;
invoke PID;
invoke SDesc;
with enabled (disjointCat) prove;
apply extensionality to predicate
  \{ p? \} \cup (PID \setminus \text{dom state})
  = PID \setminus (\text{dom state} \setminus \{ p? \});
prove;
split x = p?;
rewrite;
instantiate u_0 == u;
with enabled (lElemNotInDomNrres) prove;
\end{verbatim}

\textbf{proof[\texttt{tFreePIDPre}]}
\begin{verbatim}
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
instantiate
  free' == free \cup \{ p? \},
  pidext' == \{ p? \} \ltimes pidext;
rewrite;
use lFreePID;
rewrite;
\end{verbatim}

\textbf{proof[\texttt{lFreePIDPre}]}
\begin{verbatim}
\end{verbatim}
\textbf{proof}[tDelPD0Pre]
\begin{itemize}
\item with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
\item instantiate
\item \(\text{free'} \equiv \text{free} \cup \{p?\}\),
\item \(\text{pidext'} \equiv \{p?\} \triangleleft \text{pidext}\);
\item rewrite;
\item use lFreePID;
\item rewrite;
\end{itemize}

\textbf{proof}[tErrUnusedPDPre]
\begin{itemize}
\item with disabled (PTab, PID, DevNo, UPID, Addr, IntNo) prove by reduce;
\end{itemize}
proof[tDelPDPre]
  invoke DelPDSig;
  split p? ∈ used;
  cases;
  use tDelPD0Pre:
  with disabled (DelPD0, PTab, PID, IntNo, Addr, ErrUnusedPD) prove by reduce;
  instantiate
  nextupid_0' == nextupid',
  used_0' == used',
  free_0' == free',
  tss_0' == tss',
  ptype_0' == ptype',
  state_0' == state',
  extpid_0' == extpid',
  pidext_0' == pidext',
  msgq_0' == msgq',
  cdseg_0' == cdseg',
  dsseg_0' == dsseg',
  devmap_0' == devmap',
  devmsg_0' == devmsg',
  devrpy_0' == devrpy',
  intno_0' == intno,
  serr_0' == sysok;
  rewrite;
  next;
  use tErrUnusedPDPre:
  with disabled (DelPD0, PTab, PID, IntNo, Addr, ErrUnusedPD) prove by reduce;
  instantiate
  nextupid_0' == nextupid',
  used_0' == used',
  free_0' == free',
  tss_0' == tss',
  ptype_0' == ptype',
  state_0' == state',
  extpid_0' == extpid',
  pidext_0' == pidext',
  msgq_0' == msgq',
  cdseg_0' == cdseg',
  dsseg_0' == dsseg',
  devmap_0' == devmap',
  devmsg_0' == devmsg',
  devrpy_0' == devrpy',
  intno_0' == intno,
  serr_0' == serr';
  rewrite;
  next;

•
proof[tDeleteAllProcessesPre]
with disabled (PID, GPID, UPID, Addr, DevNo, MsgQ, SDesc) prove by reduce;
instantiate
nextupid' == nextupid,
tss' == {};
state' == {};
ptype' == {};
extpid' == {};
msgq' == {};
cdseg' == {};
dsseg' == {};
devmap' == {};
devmsg' == {};
devrpy' == {}
rewrite;

proof[tProcTypePre]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
use applyInRanPfun[Z, PTYPE][A := PID, B := PTYPE, f := ptype, a := p?];
prove;

proof[tProcStatePre]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
use applyInRanPfun[Z, PSTATE][A := PID, B := PSTATE, f := state, a := p?];
prove;

proof[tSetProcStatePre]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
invoke PTab;
with enabled (cupSubsetLeft) rewrite;

proof[tSetStateToReadyPre]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
invoke PTab;
with enabled (cupSubsetLeft) rewrite;

proof[tSetStateToRunningPre]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
invoke PTab;
with enabled (cupSubsetLeft) rewrite;

proof[tSetStateToTerminatedPre]
with disabled (PTab, PID, DevNo, UPID, Addr) prove by reduce;
invoke PTab;
with enabled (cupSubsetLeft) rewrite;

F.7 Process queue proof scripts

proof[lPIDInPQPIDInUsed]
with disabled (PID) prove by reduce;
proof[IPQHeadIsInPID]
  with disabled (PID) prove by reduce;

proof[IPQTailIsInIseqPID]
  use lTailISeqIsIseq[Y := PID, s := procs];
  rearrange;
  rewrite;

proof[HeadPQueue0$domainCheck]
  rewrite;

proof[EnqueuePQueue0$domainCheck]
  rewrite;

proof[DequeuePQueue0$domainCheck]
  rewrite;

As expected the proof for initialisation is trivial, and keeps faithful to \textit{PTabInit} precondition proof.

proof[tPQueueInit]
  reduce;
  instantiate
    nextupid' == 1,
    used' == {},
    extpid' == {},
    pidext' == {},
    free' == PID,
    tss' == {},
    devmap' == {},
    state' == {},
    ptype' == {},
    msgq' == {},
    devmsg' == {},
    devrpy' == {},
    cdseg' == {},
    dsseg' == {};
  invoke PQueue;
  invoke PTab;
  invoke UPID;
  rewrite;

With the rightly laid out specifications, the proofs are very simple indeed, when compared with the previous versions.

proof[tRaiseErrEmptyQueuePre]
  with disabled (IntNo) reduce;

proof[tRaiseErrAlreadyQueuedPre]
  with disabled (IntNo) reduce;
proof [\text{tEmptyPQueuePre}]
  with disabled (PTab, PID, IntNo) prove by reduce;

proof [\text{nPQueueOpPre}]
  with disabled (PTab, PID) prove by reduce;
  instantiate procs' == procs;
  rewrite;

proof [\text{tHeadPQueue0Pre}]
  with disabled (PTab, PID) prove by reduce;

proof [\text{tHeadPQueuePre}]
  split procs \neq \phi;
  cases;
  use tHeadPQueue0Pre;
  with disabled (DequeuePQueue0, ErrEmptyQueue, PTab, PID, IntNo) prove by reduce;
  instantiate
    nextupid_0' == nextupid', used_0' == used', free_0' == free',
    tss_0' == tss', ptype_0' == ptype', state_0' == state',
    extpid_0' == extpid', pidext_0' == pidext',
    msgq_0' == msgq', cdseg_0' == cdseg', dsseg_0' == dsseg',
    devmap_0' == devmap', devmsg_0' == devmsg', devrpy_0' == devrpy',
    intno' == intno, serv' == sysok,
    procs_0' == procs', p_0'! == p!;
  rewrite;
  next;
  use tErrEmptyQueuePre;
  with disabled (DequeuePQueue0, ErrEmptyQueue, PTab, PID, IntNo) prove by reduce;
  instantiate
    nextupid_0' == nextupid', used_0' == used', free_0' == free',
    tss_0' == tss', ptype_0' == ptype', state_0' == state',
    extpid_0' == extpid', pidext_0' == pidext',
    msgq_0' == msgq', cdseg_0' == cdseg', dsseg_0' == dsseg',
    devmap_0' == devmap', devmsg_0' == devmsg', devrpy_0' == devrpy',
    intno' == intno', serv_0' == serv',
    procs_0' == procs', p_0'! == minpid;
  rewrite;
  next;

proof [\text{tEnqueuePQueue0Pre}]
  with disabled (PTab, PID) prove by reduce;

proof [\text{tErrUnusedPDPQueuePre}]
  with disabled (PTab, PID, IntNo) prove by reduce;

proof [\text{tErrAlreadyQueuedPre}]
  with disabled (PTab, PID, IntNo) prove by reduce;
proof[\text{tEnqueuePQueuePre}]
\begin{align*}
\text{split } p? &\in \text{used} \land p? \notin \text{ran} \text{procs}; \\
\text{cases}; \\
\text{use } \text{tEnqueuePQueue0Pre}; \\
\text{with disabled } (\text{EnqueuePQueue0}, \text{ErrUnusedPDPQueue}, \text{ErrAlreadyQueued}, \text{PTab}, \text{PID}, \text{IntNo}) \text{ prove by reduce}; \\
\text{instantiante}
\begin{align*}
\text{nextupid}_0' &= \text{nextupid}', \text{used}_0' &= \text{used}', \text{free}_0' &= \text{free}', \\
\text{tss}_0' &= \text{tss}', \text{ptype}_0' &= \text{ptype}', \text{state}_0' &= \text{state}', \\
\text{extpid}_0' &= \text{extpid}', \text{pidext}_0' &= \text{pidext}', \\
\text{msgq}_0' &= \text{msgq}', \text{cdseg}_0' &= \text{cdseg}', \text{dsseg}_0' &= \text{dsseg}', \\
\text{devmap}_0' &= \text{devmap}', \text{devmsg}_0' &= \text{devmsg}', \text{devrpy}_0' &= \text{devrpy}', \\
\text{intro}_0' &= \text{intro}', \text{serr}_0' &= \text{serr}', \\
\text{procs}_0' &= \text{procs}'; \\
\text{rewrite}; \\
\text{next}; \\
\text{split } p? &\in \text{ran} \text{procs}; \\
\text{cases}; \\
\text{use } \text{tErrAlreadyQueuedPre}; \\
\text{with disabled } (\text{EnqueuePQueue0}, \text{ErrUnusedPDPQueue}, \text{ErrAlreadyQueued}, \text{PTab}, \text{PID}, \text{IntNo}) \text{ prove by reduce}; \\
\text{instantiante}
\begin{align*}
\text{nextupid}_0' &= \text{nextupid}', \text{used}_0' &= \text{used}', \text{free}_0' &= \text{free}', \\
\text{tss}_0' &= \text{tss}', \text{ptype}_0' &= \text{ptype}', \text{state}_0' &= \text{state}', \\
\text{extpid}_0' &= \text{extpid}', \text{pidext}_0' &= \text{pidext}', \\
\text{msgq}_0' &= \text{msgq}', \text{cdseg}_0' &= \text{cdseg}', \text{dsseg}_0' &= \text{dsseg}', \\
\text{devmap}_0' &= \text{devmap}', \text{devmsg}_0' &= \text{devmsg}', \text{devrpy}_0' &= \text{devrpy}', \\
\text{intro}_0' &= \text{intro}', \text{serr}_0' &= \text{serr}', \\
\text{procs}_0' &= \text{procs}'; \\
\text{rewrite}; \\
\text{next}; \\
\text{use } \text{tErrUnusedPDPQueuePre}; \\
\text{with disabled } (\text{EnqueuePQueue0}, \text{ErrUnusedPDPQueue}, \text{ErrAlreadyQueued}, \text{PTab}, \text{PID}, \text{IntNo}) \text{ prove by reduce}; \\
\text{instantiante}
\begin{align*}
\text{nextupid}_0' &= \text{nextupid}', \text{used}_0' &= \text{used}', \text{free}_0' &= \text{free}', \\
\text{tss}_0' &= \text{tss}', \text{ptype}_0' &= \text{ptype}', \text{state}_0' &= \text{state}', \\
\text{extpid}_0' &= \text{extpid}', \text{pidext}_0' &= \text{pidext}', \\
\text{msgq}_0' &= \text{msgq}', \text{cdseg}_0' &= \text{cdseg}', \text{dsseg}_0' &= \text{dsseg}', \\
\text{devmap}_0' &= \text{devmap}', \text{devmsg}_0' &= \text{devmsg}', \text{devrpy}_0' &= \text{devrpy}', \\
\text{intro}_0' &= \text{intro}', \text{serr}_0' &= \text{serr}', \\
\text{procs}_0' &= \text{procs}'; \\
\text{rewrite}; \\
\text{next}; \\
\end{align*}
\end{align*}
\end{proof}

proof[\text{tDequeuePQueue0Pre}]
\begin{align*}
\text{with disabled } (\text{PTab}, \text{PID}) \text{ prove by reduce};
\end{align*}
proof[tDequeuePQueuePre]

split procs ≠ (');

cases;
use tDequeuePQueue0Pre;
with disabled (DequeuePQueue0, ErrEmptyQueue, PTab, PID, IntNo) prove by reduce;

instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext',
msgq_0' == msgq', cdseg_0' == cdseg', dsseg_0' == dsseg',
devmap_0' == devmap', devmsg_0' == devmsg', devrpy_0' == devrpy',
intno' == intno, serr_0' == sysok,
procs_0' == procs', p_0! == p!

rewrite

next;

use tErrEmptyQueuePre;
with disabled (DequeuePQueue0, ErrEmptyQueue, PTab, PID, IntNo) prove by reduce;

instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext',
msgq_0' == msgq', cdseg_0' == cdseg', dsseg_0' == dsseg',
devmap_0' == devmap', devmsg_0' == devmsg', devrpy_0' == devrpy',
intno' == intno, serr_0' == sysok,
procs_0' == procs', p_0! == p!

rewrite

next;

F.8 Device queue proof scripts

proof[tDeviceQueueInit]
native DeviceQueueInit;
use tPQueueInit;
prove;

instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext',
msgq_0' == msgq', cdseg_0' == cdseg', dsseg_0' == dsseg',
devmap_0' == devmap', devmsg_0' == devmsg', devrpy_0' == devrpy',
intno_0' == intno, serr_0' == sysok,
procs_0' == procs', p_0! == minpid

rewrite

next;


\textbf{proof[tEnqueueDeviceQueuePre]}  
\textit{invoke EnqueueDeviceQueueSig;}  
\textit{invoke EnqueueDeviceQueue;}  
\textit{use tEnqueuePQueuePre[devs/procs];}  
\textit{prove;}  
\textit{instantiate}  
\texttt{nextupid\_0'} == nextupid', \texttt{used\_0'} == used', \texttt{free\_0'} == free',  
\texttt{tss\_0'} == tss', \texttt{ptype\_0'} == ptype', \texttt{state\_0'} == state',  
\texttt{extpid\_0'} == extpid', \texttt{pidevt\_0'} == pidevt',  
\texttt{msgq\_0'} == msgq', \texttt{cdseg\_0'} == cdseg', \texttt{dsseg\_0'} == dsseg',  
\texttt{devmap\_0'} == devmap', \texttt{devmsg\_0'} == devmsg', \texttt{devrpy\_0'} == devrpy',  
\texttt{intro\_0'} == intro', \texttt{serr\_0'} == serr',  
\texttt{devs'} == \texttt{procs'};  
\textit{rewrite;}  
\textbf{prove[tDequeueDeviceQueuePre]}  
\textit{invoke DequeueDeviceQueueSig;}  
\textit{invoke DequeueDeviceQueue;}  
\textit{use tDequeuePQueuePre[devs/procs];}  
\textit{prove;}  
\textit{instantiate}  
\texttt{nextupid\_0'} == nextupid', \texttt{used\_0'} == used', \texttt{free\_0'} == free',  
\texttt{tss\_0'} == tss', \texttt{ptype\_0'} == ptype', \texttt{state\_0'} == state',  
\texttt{extpid\_0'} == extpid', \texttt{pidevt\_0'} == pidevt',  
\texttt{msgq\_0'} == msgq', \texttt{cdseg\_0'} == cdseg', \texttt{dsseg\_0'} == dsseg',  
\texttt{devmap\_0'} == devmap', \texttt{devmsg\_0'} == devmsg', \texttt{devrpy\_0'} == devrpy',  
\texttt{intro\_0'} == intro', \texttt{serr\_0'} == serr',  
\texttt{devs'} == \texttt{procs'};  
\textit{rewrite;}  
\textbf{F.9 Scheduler proof scripts}  
\textbf{proof[tSchedSeparateQueues]}  
\textit{with disabled (PTab, PID) prove by reduce;}  
\textit{use tPTabDprocsUprocsDisjoint;}  
\textit{with enabled (disjointCat) prove;}  
\textit{apply extensionality to predicate ran devs \cap ran procs = \{\};}  
\textit{prove;}  
\textit{use inSubset[Y := ran devs, Z := ptype \sim \{dproc\}];}  
\textit{use inSubset[Y := ran procs, Z := ptype \sim \{uproc\}];}  
\textit{prove;}  
\textbf{proof[tSchedQueuedSubsetUsed]}  
\textit{with disabled (PTab, PID) prove by reduce;}  
\textbf{proof[lISeqPIDHeadNotInTail]}  
\textit{use lISeqHeadNotInTail[Z][Y := PID, s := procs];}  
\textit{prove;}  
\textbf{proof[lISeqPIDNotInTail]}  
\textit{use lTailRanSubset[Z][s := procs, y := p];}  
\textit{prove;
proof{lISeqPIDDisjointHead}
  use lDisjointSeqHead[Z][Y := PID, s := devs, t := procs];
  prove;

proof{lISeqPIDNotInHead}
  use lHeadRanSubset[Z][s := procs, y := p];
  prove;

proof{lPIDDifferentPType}
  apply inPower to predicate
  devs ∈ P (ptype ∼ { atype });
  instantiate e == p;
  rewrite;

proof{lPIDDifferentPTypeUproc}
  with enabled (lPIDDifferentPType) prove;

proof{lPIDDifferentPTypeDproc}
  with enabled (lPIDDifferentPType) prove;
proof[tSchedPTabInitEquiv]
  split SchedPTabInit;
  rewrite;
  cases;
  rewrite;
  next;
  split SchedPTabInitExpand;
  rewrite;
  next;
  cases;
  invoke SchedPTabInit; invoke SchedPTabInitExpand;
  invoke AddIdleProcess; invoke AddPD; invoke ErrPTabFull;
  invoke PTabFullInit; invoke PTabInit;
  prove;
  with disabled (PTab, PID, UPID, DevNo, Addr, IntNo) invoke;
  with enabled (lSDescFunDef) prove;
  next;
  invoke SchedPTabInit; invoke SchedPTabInitExpand;
  invoke AddIdleProcess; invoke AddPD; invoke ErrPTabFull;
  invoke PTabFullInit; invoke PTabInit;
  prove;
  invoke Sched;
  invoke DeviceQueue;
  prove;
  instantiate
  extpid == {},
  pidext == {},
  free == PID,
  tss == {},
  devmap == {},
  state == {},
  ptype == {},
  msgq == {},
  devmsg == {},
  devrpy == {},
  cdseg == {},
  dsseg == {},
  serr == sysok,
  intno == intno';
  prove;
  with disabled (PTab, PID, UPID, DevNo, Addr, IntNo) invoke;
  with enabled (lSDescFunDef) prove;
  invoke MsgQ;
  invoke PTab;
  prove;
  next;

proof[NotUserPID0$domainCheck]
  with disabled (PTab, PID) invoke;
  prove;

proof[NotDevicePID0$domainCheck]
  with disabled (PTab, PID) invoke;
  prove;
proof[MakeReadyOkEquiv]
split MakeReady \land serr' = sysok;
rewrite;
cases;
rewrite;
next;
split MakeReadyOkExpand;
rewrite;
next;
cases;
with disabled (EnqueuePQueue0, Sched, PQueue, PTab, PID, IntNo) invoke;
prove;
with disabled (PTab, PID, IntNo) invoke;
prove;
next;
with disabled (EnqueuePQueue0, Sched, PQueue, PTab, PID, IntNo) invoke;
prove;
with disabled (PTab, PID, IntNo) invoke;
prove;
next;

proof[MakeReadyOkExpand$domainCheck]
rewrite;

proof[MakeReadyAlreadyQueuedEquiv]
split MakeReady \land serr' = alreadyqueued;
rewrite;
cases;
rewrite;
next;
split MakeReadyAlreadyQueuedExpand;
rewrite;
next;
cases;
split serr' = alreadyqueued;
rewrite;
with disabled (EnqueuePQueue0, Sched, PQueue, PTab, PID, IntNo) invoke;
rewrite;
invoke Sched;
simplify;
next;
with disabled (EnqueuePQueue0, Sched, PQueue, PTab, PID, IntNo) invoke;
prove;
next;

proof[tSchedUserNext0AlwaysOk]
with disabled (PTab, PID, IntNo) invoke;
prove;

proof[RunUserNext0Expand$domainCheck]
rewrite;
proof[tRunUserNext0Equiv]
  split RunUserNext0;
  rewrite;
  cases;
  rewrite;
  next;
  split RunUserNext0Expand;
  rewrite;
  next;
  cases;
  with disabled (PTab, PID, IntNo) invoke;
  prove;
  next;
  with disabled (PTab, PID, IntNo) invoke;
  prove;
  next;

proof[tSchedDeviceNext0AlwaysOk]
  with disabled (PTab, PID, IntNo) invoke;
  prove;

proof[RunDeviceNext0Expand$domainCheck]
  rewrite;

proof[tRunDeviceNext0Equiv]
  split RunDeviceNext0;
  rewrite;
  cases;
  rewrite;
  next;
  split RunDeviceNext0Expand;
  rewrite;
  next;
  cases;
  with disabled (PTab, PID, IntNo) invoke;
  prove;
  next;
  with disabled (PTab, PID, IntNo) invoke;
  prove;
  next;
proof\{SchedNextAlwaysOk\}
with disabled (RunIdleNext0, RunUserNext0, RunDeviceNext0, PTab, PID, IntNo) prove by reduce;
split devs = { };
cases;
split procs = { };
cases;
with disabled (RunIdleNext0, DequeuePQueue, SetStateToRunning, Sched, PTab, PID, IntNo) prove by reduce;
with disabled (RunIdleProcess, PID, IntNo) prove by reduce;
next;
with disabled (RunIdleProcess, RunUserNext0, DequeuePQueue, SetStateToRunning, Sched, PTab, PID, IntNo) prove by reduce;
with disabled (DequeuePQueue0, SetStateToRunning, Sched, PQueue, PTab, PID, IntNo) prove by reduce;
next;
with disabled (RunIdleProcess, RunDeviceNext0, DequeuePQueue, SetStateToRunning, Sched, PTab, PID, IntNo) prove by reduce;
with disabled (DequeuePQueue0, SetStateToRunning, Sched, PQueue, PTab, PID, IntNo) prove by reduce;
next;

proof\{EnqueueUserSched0Sig\$domainCheck\}
rewrite;

proof\{EnqueueDeviceSched0Sig\$domainCheck\}
rewrite;

proof\{ErrNotUserPIDSig\$domainCheck\}
rewrite;

proof\{ErrNotDevicePIDSig\$domainCheck\}
rewrite;

195
proof[tSchedInit]
use tTSSNotEmpty;
with disabled (PID) prove by reduce;
instantiate
nextupid' == 2,
used' == {minpid},
free' == PID \ {minpid},
tss' == [{minpid, t}],
state' == { {minpid, psready} },
ptype' == { {minpid, uproc} },
extpid' == { 1, minpid },
pidext' == { {minpid, 1} },
msgq' == { {minpid, θ MsgQ[mq := (), maxMs := 1] } },
cdseg' == { {minpid, (nulladdr, 0)} },
dsseg' == { {minpid, (nulladdr, 0)} },
devmap' == {},
devmsg' == {},
devrpy' == {},
curr' == minpid,
queued' == {};
with disabled (PTab, PID) invoke;
prove;
with disabled (PID, Addr) reduce;

proof[tPTabFullInit]
use tPTabInit;
with disabled (PID, Addr, IntNo) prove by reduce;
instantiate
free_0' == free',
tss_0' == tss',
ptype_0' == ptype',
state_0' == state',
extpid_0' == extpid',
pidext_0' == pidext',
msgq_0' == msgq',
cdseg_0' == cdseg',
dsseg_0' == dsseg',
devmap_0' == devmap',
devmsg_0' == devmsg',
devrpy_0' == devrpy',
intrno' == minint,
serr' == sysok;
with disabled (PID, Addr, IntNo) prove by reduce;
proof[tSchedPTabInit]
  use ITSSNotEmpty;
  with disabled (PTab, PID, Addr, IntNo) prove by reduce;
  instantiate
  free' == PID \ {minpid},
  pidext' == {(minpid, 1)},
  intno' == minint,
  queued' == {},
  curr' == minpid,
  maxMs? == 1,
  tss? == t,
  cdAddr? == nulladdr, cdSize? == 0,
  dsAddr? == nulladdr, dsSize? == 0;
  with disabled (PTab, PID, Addr, IntNo) prove by reduce;
  instantiate
  extpid == {},
  pidext == {},
  free == PID,
  tss == {},
  devmap == {},
  state == {},
  ptype == {},
  msgq == {},
  devmsg == {},
  devrpy == {},
  cdseg == {},
  dsseg == {},
  serr == sysok;
prove;
  invoke PTab;
  with enabled (lSDescFunDef) prove;
  invoke MsgQ;
  invoke UPID;
  invoke SDesc;
prove;

proof[tRaiseErrNotUserPIDPre]
  with disabled (IntNo) reduce;

proof[tRaiseErrNotDevicePIDPre]
  with disabled (IntNo) reduce;

proof[tRaiseErrBadPIDCurrPre]
  with disabled (IntNo) reduce;

proof[tRaiseErrBadPIDIdlePre]
  with disabled (IntNo) reduce;

proof[tIdleProcessPre]
  with disabled (PTab, PID) prove by reduce;
proof[tRunningProcessPre]
with disabled (PTab, PID) prove by reduce;

proof[tUpdateCurrentProcessPre]
with disabled (PTab, PID) prove by reduce;
instantiate
nextupid' == nextupid,
used' == used,
free' == free,
tss' == tss,
ptype' == ptype,
state' == state,
exitpid' == exitpid,
pidext' == pidext,
msgq' == msgq,
cdseg' == cdseg,
dsseg' == dsseg,
devmap' == devmap,
devmsg' == devmsg,
devpy' == devpy,
procs' == procs,
devs' == devs,
pid' == pid;
rewrite;

proof[tEnqueueUserSched0Pre]
use tEnqueuePQueuePre;
with disabled (EnqueuePQueue, PQueue, PID, IntNo) invoke;
rearrange;
simplify;
prenex;
rewrite;

instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
exitpid_0' == exitpid', pidext_0' == pidext', msgq_0' == msgq',
cdseg_0' == cdseg', dsseg_0' == dsseg', devmap_0' == devmap',
devmsg_0' == devmsg', devpy_0' == devpy',
intno_0' == intno', serr_0' == serr',
procs_0' == procs';
rewrite;
with disabled (PTab, PID, IntNo) invoke;

split PTab' ∧ used' = used ∧ ptype' = ptype;
rewrite;
split procs = procs';
cases;
prove;
next;
prove;
next;
proof[tEnqueueDeviceSched0Pre]
use tEnqueueDeviceQueuePre;
with disabled (EnqueuePQueue, PQueue, PID, IntNo) invoke;
rearrange;
simplify;
renex;
rewrite;
 instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidxext_0' == pidxext', msgq_0' == msgq',
cdseg_0' == cdseg', dsseg_0' == dsseg', devmap_0' == devmap',
devmsg_0' == devmsg', devrpy_0' == devrpy',
intno_0' == intno', serr_0' == serr',
devs_0' == devs';
rewrite;
with disabled (PTab, PID, IntNo) invoke;
split PTab' ∧ used' = used ∧ ptype' = ptype;
rewrite;
split devs = devs';
cases;
prove;
next;
prove;
next;

proof[tErrNotUserPIDPre]
with disabled (PTab, PID, IntNo) invoke;
prove;

proof[tErrNotDevicePIDPre]
with disabled (PTab, PID, IntNo) invoke;
prove;

proof[tErrBadPIDCurrPre]
with disabled (PTab, PID, IntNo) invoke;
prove;

proof[tErrBadPIDIdlePre]
with disabled (PTab, PID, IntNo) invoke;
prove;
\textbf{proof} \{ EnqueueUserSchedPre \}

\begin{align*}
\text{split } p? &= curr; \\
\text{cases; } \\
\text{use } t\text{ErrBadPIDCurrPre; } \\
\text{with disabled} \\
\text{(EnqueueUserSched0, } \text{ErrNotUserPID, } \text{ErrBadPIDCurr, } \\
\text{ErrBadPIDIdle, } \text{PQueue, } \text{PTab, } \text{PID, } \text{IntNo}) \text{ invoke; } \\
\text{prove; } \\
\text{instantiate} \\
\text{nextupid}_0' &= \text{nextupid}', \text{used}_0' = \text{used}', \text{free}_0' = \text{free}', \\
\text{tss}_{0'} &= \text{tss}', \text{ptype}_{0'} = \text{ptype}', \text{state}_{0'} = \text{state}', \\
\text{extpid}_{0'} &= \text{extpid}', \text{pidxt}_{0'} = \text{pidxt}', \text{msgq}_{0'} = \text{msgq}', \\
\text{cdseg}_{0'} &= \text{cdseg}', \text{dsseg}_{0'} = \text{dsseg}', \text{devmap}_{0'} = \text{devmap}', \\
\text{devmsg}_{0'} &= \text{devmsg}', \text{devepy}_{0'} = \text{devepy}', \\
\text{introo}_{0'} &= \text{intro}', \text{srv}_{0'} = \text{srv}', \\
\text{curr}_{0'} &= \text{curr}', \text{devs}_{0'} = \text{devs}', \text{ipid}_{0'} = \text{ipid}', \\
\text{procs}_{0'} &= \text{procs}', \text{queued}_{0'} = \text{queued}'; \\
\text{prove; } \\
\text{next; } \\
\text{split } p? &= \text{ipid}; \\
\text{cases; } \\
\text{use } t\text{ErrBadPIDIdlePre; } \\
\text{with disabled} \\
\text{(EnqueueUserSched0, } \text{ErrNotUserPID, } \text{ErrBadPIDCurr, } \\
\text{ErrBadPIDIdle, } \text{PQueue, } \text{PTab, } \text{PID, } \text{IntNo}) \text{ invoke; } \\
\text{prove; } \\
\text{instantiate} \\
\text{nextupid}_0' &= \text{nextupid}', \text{used}_0' = \text{used}', \text{free}_0' = \text{free}', \\
\text{tss}_{0'} &= \text{tss}', \text{ptype}_{0'} = \text{ptype}', \text{state}_{0'} = \text{state}', \\
\text{extpid}_{0'} &= \text{extpid}', \text{pidxt}_{0'} = \text{pidxt}', \text{msgq}_{0'} = \text{msgq}', \\
\text{cdseg}_{0'} &= \text{cdseg}', \text{dsseg}_{0'} = \text{dsseg}', \text{devmap}_{0'} = \text{devmap}', \\
\text{devmsg}_{0'} &= \text{devmsg}', \text{devepy}_{0'} = \text{devepy}', \\
\text{introo}_{0'} &= \text{intro}', \text{srv}_{0'} = \text{srv}', \\
\text{curr}_{0'} &= \text{curr}', \text{devs}_{0'} = \text{devs}', \text{ipid}_{0'} = \text{ipid}', \\
\text{procs}_{0'} &= \text{procs}', \text{queued}_{0'} = \text{queued}'; \\
\text{prove; } \\
\text{next; } \\
\text{split } p? \in \text{used } \land \neg \text{ptype } p? = \text{uproc; } \\
\text{cases; } \\
\text{use } t\text{ErrNotUserPIDPre; } \\
\text{with disabled} \\
\text{(EnqueueUserSched0, } \text{ErrNotUserPID, } \text{ErrBadPIDCurr, } \\
\text{ErrBadPIDIdle, } \text{PQueue, } \text{PTab, } \text{PID, } \text{IntNo}) \text{ invoke; } \\
\text{prove; } \\
\text{instantiate} \\
\text{nextupid}_0' &= \text{nextupid}', \text{used}_0' = \text{used}', \text{free}_0' = \text{free}', \\
\text{tss}_{0'} &= \text{tss}', \text{ptype}_{0'} = \text{ptype}', \text{state}_{0'} = \text{state}', \\
\text{extpid}_{0'} &= \text{extpid}', \text{pidxt}_{0'} = \text{pidxt}', \text{msgq}_{0'} = \text{msgq}', \\
\text{cdseg}_{0'} &= \text{cdseg}', \text{dsseg}_{0'} = \text{dsseg}', \text{devmap}_{0'} = \text{devmap}', \\
\text{devmsg}_{0'} &= \text{devmsg}', \text{devepy}_{0'} = \text{devepy}', \\
\text{introo}_{0'} &= \text{intro}', \text{srv}_{0'} = \text{srv}', \\
\text{curr}_{0'} &= \text{curr}', \text{devs}_{0'} = \text{devs}', \text{ipid}_{0'} = \text{ipid}', \\
\text{procs}_{0'} &= \text{procs}', \text{queued}_{0'} = \text{queued}'; \\
\text{prove; } \\
\text{next; } \\
\text{use } t\text{EnqueueUserSched0Pre; } \\
\text{with disabled} \\
\text{(EnqueueUserSched0, } \text{ErrNotUserPID, } \text{ErrBadPIDCurr, } \\
\text{ErrBadPIDIdle, } \text{PQueue, } \text{PTab, } \text{PID, } \text{IntNo}) \text{ invoke; } \\
\text{prove; } \\
\text{instantiate} \\
\text{nextupid}_0' &= \text{nextupid}', \text{used}_0' = \text{used}', \text{free}_0' = \text{free}', \\
\text{tss}_{0'} &= \text{tss}', \text{ptype}_{0'} = \text{ptype}', \text{state}_{0'} = \text{state}', \\
\text{extpid}_{0'} &= \text{extpid}', \text{pidxt}_{0'} = \text{pidxt}', \text{msgq}_{0'} = \text{msgq}', \\
\text{cdseg}_{0'} &= \text{cdseg}', \text{dsseg}_{0'} = \text{dsseg}', \text{devmap}_{0'} = \text{devmap}', \\
\text{devmsg}_{0'} &= \text{devmsg}', \text{devepy}_{0'} = \text{devepy}', \\
\text{introo}_{0'} &= \text{intro}', \text{srv}_{0'} = \text{srv}', \\
\text{curr}_{0'} &= \text{curr}', \text{devs}_{0'} = \text{devs}', \text{ipid}_{0'} = \text{ipid}', \\
\text{procs}_{0'} &= \text{procs}', \text{queued}_{0'} = \text{queued}'; \\
\text{prove; } \\
\end{align*}
proof[EnqueueDeviceSchedPre]
split p? = curr;
cases;
use tErrBadPIDCurrPre;
with disabled
(EnqueueDeviceSched0, ErrNotDevicePID, ErrBadPIDCurr,
ErrBadPIDIdle, PQueue, PTab, PID, IntNo) invoke;
prove:
instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
rsseg_0' == rsseg', dssq_0' == dssq', devmap_0' == devmap',
devmsg_0' == devmsg', devryp_0' == devryp',
intno_0' == intno', serr_0' == serr',
curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
procs_0' == procs', queued_0' == queued';
prove;
next;
split p? = ipid;
cases;
use tErrBadPIDIdlePre;
with disabled
(EnqueueDeviceSched0, ErrNotDevicePID, ErrBadPIDCurr,
ErrBadPIDIdle, PQueue, PTab, PID, IntNo) invoke;
prove:
instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
rsseg_0' == rsseg', dssq_0' == dssq', devmap_0' == devmap',
devmsg_0' == devmsg', devryp_0' == devryp',
intno_0' == intno', serr_0' == serr',
curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
procs_0' == procs', queued_0' == queued';
prove;
next;
split p? ∈ used ∧ ¬ ptype p? = dproc;
cases;
use tErrNotDevicePIDPre;
with disabled
(EnqueueDeviceSched0, ErrNotDevicePID, ErrBadPIDCurr,
ErrBadPIDIdle, PQueue, PTab, PID, IntNo) invoke;
prove:
instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
rsseg_0' == rsseg', dssq_0' == dssq', devmap_0' == devmap',
devmsg_0' == devmsg', devryp_0' == devryp',
intno_0' == intno', serr_0' == serr',
curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
procs_0' == procs', queued_0' == queued';
prove;
next;
use tEnqueueDeviceSchedPre0;
with disabled
(EnqueueDeviceSched0, ErrNotDevicePID, ErrBadPIDCurr,
ErrBadPIDIdle, PQueue, PTab, PID, IntNo) invoke;
prove:
instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
rsseg_0' == rsseg', dssq_0' == dssq', devmap_0' == devmap',
devmsg_0' == devmsg', devryp_0' == devryp',
intno_0' == intno', serr_0' == serr',
curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
procs_0' == procs', queued_0' == queued';
prove;
proof[ErrSysFailPTabPre]
with disabled (PTab, PID) prove by reduce;

proof[MakeReadyPreSysOk]
with disabled (EnqueuePQueue0, Sched, PQueue, PTab, PID, IntNo) invoke;
prove;
with disabled (PTab, PID, IntNo) invoke;
prove;
invoke PTab;
with enabled (cupSubsetLeft) rewrite;

proof[MakeReadyPreSysFail]
split ≜ Sched;
cases;
with disabled (EnqueuePQueue0, Sched, PQueue, PTab, PID, IntNo) invoke;
prove;
split serr' ≜ badpididle;
simplify;
split serr' ≜ badpidcurr;
simplify;
split serr' ≜ notuserpid;
simplify;
split serr' ≜ alreadyqueued;
simplify;
next;
with disabled (EnqueuePQueue0, PQueue, PTab, PID, IntNo) invoke;
prove;
next;
proof[MakeReadyPre]
  use tEnqueueUserSchedPre;
  invoke MakeReady;
  prove:
    instantiate
    nextupid_0 == nextupid', used_0' == used', free_0' == free',
    tss_0' == tss', ptype_0' == ptype',
    extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
    cdseg_0' == cdseg', dsseg_0' == dsseg', devmap_0' == devmap',
    devmsg_0' == devmsg', devrpy_0' == devrpy',
    intno_0' == intno', serr_0' == serr',
    curr_0' == curr', dev_0' == dev', ipid_0' == ipid',
    procs_0' == procs', queued_0' == queued',
    nextupid_0 == nextupid, used_0 == used, free_0 == free,
    tss_0 == tss, ptype_0 == ptype, state_0 == state,
    extpid_0 == extpid, pidext_0 == pidext, msgq_0 == msgq,
    cdseg_0 == cdseg, dsseg_0 == dsseg, devmap_0 == devmap,
    devmsg_0 == devmsg, devrpy_0 == devrpy,
    serr_0 == serr';
  prove;
  invoke EnqueueUserSchedSig; invoke MakeReadySig;
  prove;
  split serr' = sysok;
  cases;
  instantiate state_0' == state' + {p? ↦ psready};
  prove;
  use lMakeReadyPreSysOk;
  prove;
  next;
  rewrite;
  instantiate state_0' == state';
  prove;
  use lMakeReadyPreSysFail;
  rearrange;
  simplify;
  invoke IsSysOk;
  rewrite;
  next;

proof[ReadyDeviceProcessPreSysOk]
  with disabled (EnqueuePQueue0, Sched, PQueue, PTab, PID, IntNo) invoke;
  prove;
  with disabled (PTab, PID, IntNo) invoke;
  prove;
  invoke PTab;
  with enabled (cupSubsetLeft) rewrite;
proof[ProofDeviceProcessPreSysFail]
split \Xi Sched;
cases;
    with disabled (EnqueuePQueue0, Sched, PQueue, PTab, PID, IntNo) invoke;
prove;
    split serr' = badpididle;
simplify;
    split serr' = badpidcurr;
simplify;
    split serr' = notdevicepid;
simplify;
    split serr' = alreadyqueued;
simplify;
next;
    with disabled (EnqueuePQueue0, PQueue, PTab, PID, IntNo) invoke;
prove;
next;

proof[ProofDeviceProcessPre]
use tEnqueueDeviceSchedPre;
invoke ReadyDeviceProcess;
prove;
instantiate
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype',
extpid_0' == extpid', pext_0' == pidext', msgq_0' == msgq',
cdseg_0' == cdseg', dsseg_0' == dsseg', devmap_0' == devmap',
devmsg_0' == devmsg', devrpy_0' == devrpy',
into_0' == into', serr_0' == serr',
curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
procs_0' == procs', queued_0' == queued',
nextupid_0 == nextupid, used_0 == used, free_0 == free,
tss_0 == tss, ptype_0 == ptype, state_0 == state,
nextupid_0 == nextupid, pext_0 == pidext, msgq_0 == msgq,
cdseg_0 == cdseg, dsseg_0 == dsseg, devmap_0 == devmap,
devmsg_0 == devmsg, devrpy_0 == devrpy,
serr_0 == serr';
prove;
invoke EnqueueDeviceSchedSig; invoke ReadyDeviceProcessSig;
prove;
split serr' = sysok;
cases;
instantiate state_0' == state' \oplus \{ p? \mapsto psready \};
prove;
use tReadyDeviceProcessPreSysOk;
prove;
next;
rewrite;
instantiate state_0' == state';
prove;
use tReadyDeviceProcessPreSysFail;
rearrange;
simplify;
invoke IsSysOk;
rewrite;
next;
proof [tRunIdleProcessPre]  
  with disabled (PTab, PID) invoke;  
  prove;  
  invoke PTab;  
  with enabled (cupSubsetLeft) rewrite;  

proof [tRunIdleNext0Pre]  
  with disabled (PTab, PID, IntNo) invoke;  
  prove;  
  invoke PTab;  
  with enabled (cupSubsetLeft) rewrite;  

proof [tSchedUserNext0Pre]  
  with disabled (PTab, PID, IntNo) invoke;  
  with enabled (lISeqPIDHeadNotInTail, lISeqPIDNotInTail) prove;  

proof [tRunUserNext0Pre]  
  with disabled (PTab, PID, IntNo) invoke;  
  with enabled (lISeqPIDHeadNotInTail, lISeqPIDNotInTail) prove;  
  invoke PTab;  
  with enabled (cupSubsetLeft) rewrite;  

proof [tSchedDeviceNext0Pre]  
  use lSchedSeparateQueues;  
  with disabled (PTab, PID, IntNo) invoke;  
  with enabled (lISeqPIDHeadNotInTail, lISeqPIDNotInTail, lISeqPIDDisjointHead) prove;  

proof [tRunDeviceNext0Pre]  
  use lSchedSeparateQueues;  
  with disabled (PTab, PID, IntNo) invoke;  
  with enabled (lISeqPIDHeadNotInTail, lISeqPIDNotInTail, lISeqPIDDisjointHead) prove;  
  invoke PTab;  
  with enabled (cupSubsetLeft) rewrite;  

proof [lRunUserNext0IntnoUnchanged]  
  with disabled  
  (DequeuePQueue0, SetStateToRunning, Sched, PQueue, PTab, PID, IntNo) invoke;  
  prove;  

proof [lRunDeviceNext0IntnoUnchanged]  
  with disabled  
  (DequeuePQueue0, SetStateToRunning, Sched, PQueue, PTab, PID, IntNo) invoke;  
  prove;
proof[tSchedNextPre]
split devs = ( );
cases;
split procs = ( );
cases;
use \texttt{tRunIdleNext0Prev};
with disabled
(RunIdleNext0, RunUserNext0, RunDeviceNext0, PTab, PID, IntNo) prove by reduce;
instance
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
cdseg_0' == cdseg', dsseg_0' == dsseg', devmap_0' == devmap',
devmsg_0' == devmsg', devpy_0' == devpy',
\texttt{inno}_0 == \texttt{inno}', serr_0' == serr',
curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
procs_0' == procs', queued_0' == queued';
prove;
next;
use \texttt{tRunUserNext0Prev};
with disabled
(RunIdleNext0, RunUserNext0, RunDeviceNext0, PTab, PID, IntNo) prove by reduce;
instance
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
cdseg_0' == cdseg', dsseg_0' == dsseg', devmap_0' == devmap',
devmsg_0' == devmsg', devpy_0' == devpy',
\texttt{inno}_0 == \texttt{inno}', serr_0' == serr',
curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
procs_0' == procs', queued_0' == queued';
prove;
use \texttt{lRunUserNext0IntnoUnchanged};
rearrange;
rewrite;
next;
use \texttt{lRunDeviceNext0Prev};
with disabled
(RunIdleNext0, RunUserNext0, RunDeviceNext0, PTab, PID, IntNo) prove by reduce;
instance
nextupid_0' == nextupid', used_0' == used', free_0' == free',
tss_0' == tss', ptype_0' == ptype', state_0' == state',
extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
cdseg_0' == cdseg', dsseg_0' == dsseg', devmap_0' == devmap',
devmsg_0' == devmsg', devpy_0' == devpy',
\texttt{inno}_0 == \texttt{inno}', serr_0' == serr',
curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
procs_0' == procs', queued_0' == queued';
prove;
use \texttt{lRunDeviceNext0IntnoUnchanged};
rearrange;
rewrite;
next;
\textbf{proof}[\texttt{SchedNextExistsPost}] 
\begin{verbatim}
proof[\texttt{SchedNextExistsPost}]
split devs = ( );
cases;
split procs = ( );
cases;
with disabled (Sched, DequeuePQueue, SetStateToRunning, PTab, PID, IntNo) invoke;
prove;
next;
with disabled (Sched, DequeuePQueue, SetStateToRunning, PTab, PID, IntNo) invoke;
prove;
with disabled (DequeuePQueue, PTab, PID, IntNo) invoke;
prove;
next;
with disabled (Sched, DequeuePQueue, SetStateToRunning, PTab, PID, IntNo) invoke;
prove;
with disabled (DequeuePQueue, PTab, PID, IntNo) invoke;
prove;
next;
\end{verbatim}

\textbf{proof}[\texttt{RequeueUserProcessPre}] 
\begin{verbatim}
proof[\texttt{RequeueUserProcessPre}]
use \texttt{tSchedNextPre}:
with disabled (RequeueUserProcess, SchedNext, PTab, PID, IntNo) invoke;
prove;
use \texttt{lSchedNextExistsPost}:
use \texttt{tMakeReadyPre}[cdseg'/cdseg, curr'/curr',
devid'/devid, devmsg'/devmsg, devrpy'/devrpy, devsp'/devsp, dsseg'/dsseg, extpid'/extpid, free'/free,'
intno'/intno, ipid'/ipid, msgq'/msgq, nextupid'/nextupid, pidevt'/pidevt, procs'/procs, ptype'/ptype,
queued'/queued, serr'/serr, state'/state, tss'/tss, used'/used];
with disabled
(RequeueUserProcess, SchedNext, MakeReady, PTab, PID, IntNo) invoke;
prove;
invoke RequeueUserProcess;
\end{verbatim}
\textbf{proof}\texttt{[RequeueDeviceProcessPre]}

\begin{verbatim}
use \texttt{tSchedNextPre};
with disabled (RequeueDeviceProcess, SchedNext, PTab, PID, IntNo) invoke;
prove;
use \texttt{tSchedNextExistsPost};
use \texttt{tReadyDeviceProcessPre[cmdseg'/cdseg, curr'/curr, devmap'/devmap, devmsg'/devmsg, devrpy'/devrpy, devs'/devs, dsseg'/dsseg, extpid'/extpid, free'/free, intno'/intno, ipid'/ipid, msgq'/msgq, nextupid'/nextupid, pidext'/pidext, procs'/procs, ptype'/ptype, queued'/queued, serr'/serr, state'/state, tss'/tss, used'/used]);
with disabled
(RequeueDeviceProcess, SchedNext, ReadyDeviceProcess, PTab, PID, IntNo) invoke;
prove;
invoke \texttt{RequeueDeviceProcess};
\end{verbatim}

\texttt{instantiate}

nextupid\_0 == nextupid', used\_0 == used', free\_0 == free',
tss\_0 == tss', ptype\_0 == ptype', state\_0 == state',

extpid\_0 == extpid', pidext\_0 == pidext', msgq\_0 == msgq',

cdseg\_0 == cdseg', dsseg\_0 == dsseg', devmap\_0 == devmap',

devmsg\_0 == devmsg', devrpy\_0 == devrpy',

intno\_0 == intno', serr\_0 == serr',

curr\_0 == curr', devs\_0 == devs', ipid\_0 == ipid',

procs\_0 == procs', queued\_0 == queued',

nextupid\_1' == nextupid\_0', used\_1' == used\_0', free\_1' == free\_0',
tss\_1' == tss\_0', ptype\_1' == ptype\_0', state\_1' == state\_0',

extpid\_1' == extpid\_0', pidext\_1' == pidext\_0', msgq\_1' == msgq\_0',

cdseg\_1' == cdseg\_0', dsseg\_1' == dsseg\_0', devmap\_1' == devmap\_0',

devmsg\_1' == devmsg\_0', devrpy\_1' == devrpy\_0',

intno\_1' == intno\_0', serr\_1' == serr\_0',

curr\_1' == curr\_0', devs\_1' == devs\_0', ipid\_1' == ipid\_0',

procs\_1' == procs\_0', queued\_1' == queued\_0';
prove;

\end{verbatim}
proof[tSchedBackupPre]
with disabled (Sched) prove by reduce;
  instantiate
cdseg' == cdseg,
curr' == curr,
devmap' == devmap,
devmsg' == devmsg,
devrpy' == devrpy,
devs' == devs,
dssseg' == dssseg,
extpid' == extpid,
free' == free,
igid' ==igid,
msgq' == msgq,
extuupid' == extuupid,
proc' == proc,
ptype' == ptype,
queued' == queued,
state' == state,
tss' == tss,
used' == used;
prove;

proof[tSchedRevertPre]
with disabled (Sched) prove by reduce;

proof[tSchedTrans0Pre]
with disabled (Sched, ErrV, HW) prove by reduce;

proof[tErrSchedRevertTransPre]
with disabled (Sched, ErrV, HW) prove by reduce;

proof[tSchedTransPre]
with disabled (Sched, ErrV, HW) prove by reduce;
split serr = sysok;
prove;

proof[fEnqueueUserSched0ExistsPost]
invoke EnqueueUserSched0;
invoke \Delta Sched;
rewrite;

proof[fErrNotUserPIDExistsPost]
invoke ErrNotUserPID;
invoke NotUserPID0;
invoke \Xi Sched;
rewrite;
proof ![ErrBadPIDCurrExistsPost]
  invoke ErrBadPIDCurr;
  invoke RunningProcess;
  invoke \(\Xi\) Sched;
  rewrite;

proof ![ErrBadPIDIdleExistsPost]
  invoke ErrBadPIDIdle;
  invoke IdleProcess;
  invoke \(\Xi\) Sched;
  rewrite;

proof ![EnqueueUserSchedExistsPost]
  invoke EnqueueUserSched;
  split EnqueueUserSched0;
  cases;
  prove;
  next;
  split ErrNotUserPID;
  prove;
  split ErrBadPIDCurr;
  prove;
  next;

\[210\]
proof[\text{MakeReadyExistsPost}] 
\quad invoke \text{MakeReady}; 
\quad prove; 
\quad use 
\quad \text{lEnqueueUserSchedExistsPost[cdseg}_0/\text{cdseg}', \text{devmap}_0/\text{devmap}', \text{devmsg}_0/\text{devmsg}', \text{devrpy}_0/\text{devrpy}', \text{dsseg}_0/\text{dsseg}', \text{extpid}_0/\text{extpid}', \text{free}_0/\text{free}', \text{msgq}_0/\text{msgq}', \text{nexttupid}_0/\text{nexttupid}', \text{pidext}_0/\text{pidext}', \text{ptype}_0/\text{ptype}', \text{serr}_0/\text{serr}', \text{state}_0/\text{state}', \text{tss}_0/\text{tss}', \text{used}_0/\text{used}]; 
\quad prove; 
\quad invoke \text{Sched}; 
\quad invoke \text{DeviceQueue}; 
\quad prove; 
\quad invoke \text{setStateToReady}; 
\quad invoke \text{ErrSysFailPTab}; 
\quad invoke \text{IsSysOk}; 
\quad split \text{serr}_0 = \text{sysok}; 
\quad cases; 
\quad prove; 
\quad invoke \text{SetProcState}; 
\quad invoke \text{PTabChangeState}; 
\quad prove; 
\quad invoke \Xi \text{PTab}; 
\quad prove; 
\quad invoke \text{PQueue}; 
\quad prove; 
\quad next; 
\quad prove; 
\quad invoke \Xi \text{PTab}; 
\quad invoke \text{PQueue}; 
\quad prove; 
\quad next; 
\quad ■

proof[\text{RequeueUserProcessExistsPost}] 
\quad invoke \text{RequeueUserProcess}; 
\quad ■

proof[\text{EnqueueUserSched0ExistsPostHW}] 
\quad invoke \text{EnqueueUserSched0}; 
\quad invoke \text{EnqueuePQueue}; 
\quad invoke \text{RaiseOk}; 
\quad invoke \Xi \text{HW}; 
\quad split \text{HW}[\text{intno}]/\text{intno}]; 
\quad prove; 
\quad with disabled (\text{Sched, PQueue, PTab, PID, ErrV, HW}) invoke; 
\quad prove; 
\quad ■

proof[\text{ErrNotUserPIDExistsPostHW}] 
\quad with disabled (\text{Sched, PTab, PID, ErrV, HW}) invoke; 
\quad prove; 
\quad ■
proof\([fErrBadPIDCurrExistsPostHW]\\)
with disabled (Sched, PTab, PID, ErrV, HW) invoke;
prove;

proof\([fErrBadPIDIdleExistsPostHW]\\)
with disabled (Sched, PTab, PID, ErrV, HW) invoke;
prove;

proof\([lEnqueueUserSchedExistsPostHW]\\)
invoke EnqueueUserSched;
split EnqueueUserSched0;
cases;
prove;
next;
split ErrNotUserPID;
prove;
split ErrBadPIDCurr;
prove;
next;

proof\([fMakeReadyExistsPostHW]\\)
invoke MakeReady;
prove;
use
\([lEnqueueUserSchedExistsPostHW]\)[cdseg\_0]/cdseg', \(devmap\_0)/devmap',
\(devmsg\_0)/devmsg', \(deerrpy\_0)/deerrpy',
\(dsseg\_0)/dsseg', \(extpid\_0)/extpid',
\(free\_0)/free', \(msgq\_0)/msgq',
\(nextupid\_0)/nextupid', \(pidext\_0)/pidext',
\(ptype\_0)/ptype', \(serr\_0)/serr',
\(state\_0)/state', \(tss\_0)/tss',
\(used\_0)/used'\];
prove;

proof\([lRequeueUserProcessExistsPostHW]\\)
invoke RequeueUserProcess;
prove;
proof[tRequeueUserProcessTransPre1]
  invoke RequeueUserProcessTransPre1;
  use tRequeueUserProcessPre;
  prove;
  instantiate
  nextupid_0' == nextupid', used_0' == used', free_0' == free',
  tss_0' == tss', ptype_0' == ptype', state_0' == state',
  extpid_0' == extpid', pidext_0' == pidext', msgq_0' == msgq',
  cdseg_0' == cdseg', dsseg_0' == dsseg', devmap_0' == devmap',
  devmsg_0' == devmsg', devrpy_0' == devrpy',
  intno_0' == intno', serr_0' == serr',
  curr_0' == curr', devs_0' == devs', ipid_0' == ipid',
  procs_0' == procs', queued_0' == queued';
prove;
invoke SchedBackup;
invoke SchedBackupSig;
prove;
invoke Δ Sched;
prove;
use lRequeueUserProcessExistsPost;
prove;
proof[tRequeueUserProcessTransPre2]
invoke RequeueUserProcessTransPre2;
use tRequeueUserProcessTransPre1;
prove:
use tSchedTransPre[cdseg'/cdseg, curr'/curr,
devmap'/devmap, devmsg'/devmsg,
devrpy'/devrpy,
devs'/devs, dsseg'/dsseg,
extpid'/extpid, free'/free,
intno'/intno, ipid'/ipid, msgq'/msgq,
nextupid'/nextupid, pidext'/pidext,
procs'/procs, ptype'/ptype,
queued'/queued, serr'/serr,
state'/state, tss'/tss, used'/used];
invoke SchedTransSig;
invoke RequeueUserProcessSig;
invoke SchedBackupSig;
invoke SchedTotalSig;
invoke SchedOpTotalSig;
invoke RequeueUserProcessTransPre1;
prove;
invoke RaiseErrSig;
invoke ErrV;
invoke RaiseKillInterruptSig;
invoke RaiseNamedInterruptSig;
use lRequeueUserProcessExistsPostHW;
prove;
 instantiate
nextupid_0 == nextupid', used_0 == used', free_0 == free',
tss_0 == tss', ptype_0 == ptype', state_0 == state',
extpid_0 == extpid', pidext_0 == pidext', msgq_0 == msgq',
cdseg_0 == cdseg', dsseg_0 == dsseg', devmap_0 == devmap',
devmsg_0 == devmsg', devrpy_0 == devrpy',
intno_0 == intno', serr_0 == serr',
curr_0 == curr', devs_0 == devs', ipid_0 == ipid',
procs_0 == procs', queued_0 == queued',
nextupid_1' == nextupid_0', used_1' == used_0', free_1' == free_0',
tss_1' == tss_0, ptype_1' == ptype_0', state_1' == state_0',
extpid_1' == extpid_0', pidext_1' == pidext_0', msgq_1' == msgq_0',
cdseg_1' == cdseg_0, dsseg_1' == dsseg_0, devmap_1' == devmap_0',
devmsg_1' == devmsg_0, devrpy_1' == devrpy_0',
intno_1' == intno_0, serr_1' == serr_0',
curr_1' == curr_0, devs_1' == devs_0, ipid_1' == ipid_0',
procs_1' == procs_0, queued_1' == queued_0';
prove;
\textbf{proof[tRequeueUserProcessTransPre]}
\begin{align*}
\text{invoke } & \text{tRequeueUserProcessTransSig;} \\
\text{invoke } & \text{tRequeueUserProcessTrans;} \\
\text{split } & \exists \text{ Sched}'' \land \theta \text{ Sched}'' = \theta \text{ Sched}; \\
\text{cases;} \\
\text{prenex;} \\
\text{use } & \text{tRequeueUserProcessTransPre2;} \\
\text{invoke } & \text{tRequeueUserProcessSig;} \\
\text{invoke } & \text{tRequeueUserProcessTransPre2;} \\
\text{prove;} \\
\text{prove;} \\
\text{instantiate} \\
\text{nextupid}_1 &= \text{nextupid}_0, \text{used}_1 = \text{used}_0, \text{free}_1 = \text{free}_0, \\
\text{tss}_1 &= \text{tss}_0, \text{ptype}_1 = \text{ptype}_0, \text{state}_1 = \text{state}_0, \\
\text{extpid}_1 &= \text{extpid}_0, \text{pidext}_1 = \text{pidext}_0, \text{msgq}_1 = \text{msgq}_0, \\
\text{cdseg}_1 &= \text{cdseg}_0, \text{dsseg}_1 = \text{dsseg}_0, \text{devmap}_1 = \text{devmap}_0, \\
\text{devmsg}_1 &= \text{devmsg}_0, \text{devrpy}_1 = \text{devrpy}_0, \\
\text{intno}_1 &= \text{intno}_0, \text{serr}_1 = \text{serr}_0, \\
\text{curr}_1 &= \text{curr}_0, \text{devs}_1 = \text{devs}_0, \text{ipid}_1 = \text{ipid}_0, \\
\text{procs}_1 &= \text{procs}_0, \text{queued}_1 = \text{queued}_0, \\
\text{nextupid}''_0 &= \text{nextupid}''_0, \text{used}''_0 = \text{used}''_0, \text{free}''_0 = \text{free}''_0, \\
\text{tss}''_0 &= \text{tss}''_0, \text{ptype}''_0 = \text{ptype}''_0, \text{state}''_0 = \text{state}''_0, \\
\text{extpid}''_0 &= \text{extpid}''_0, \text{pidext}''_0 = \text{pidext}''_0, \text{msgq}''_0 = \text{msgq}''_0, \\
\text{cdseg}''_0 &= \text{cdseg}''_0, \text{dsseg}''_0 = \text{dsseg}''_0, \text{devmap}''_0 = \text{devmap}''_0, \\
\text{devmsg}''_0 &= \text{devmsg}''_0, \text{devrpy}''_0 = \text{devrpy}''_0, \\
\text{curr}''_0 &= \text{curr}''_0, \text{devs}''_0 = \text{devs}''_0, \text{ipid}''_0 = \text{ipid}''_0, \\
\text{procs}''_0 &= \text{procs}''_0, \text{queued}''_0 = \text{queued}''_0, \\
\text{nextupid}''_0 &= \text{nextupid}''_0, \text{used}''_0 = \text{used}''_0, \text{free}''_0 = \text{free}''_0, \\
\text{tss}''_0 &= \text{tss}''_0, \text{ptype}''_0 = \text{ptype}''_0, \text{state}''_0 = \text{state}''_0, \\
\text{extpid}''_0 &= \text{extpid}''_0, \text{pidext}''_0 = \text{pidext}''_0, \text{msgq}''_0 = \text{msgq}''_0, \\
\text{cdseg}''_0 &= \text{cdseg}''_0, \text{dsseg}''_0 = \text{dsseg}''_0, \text{devmap}''_0 = \text{devmap}''_0, \\
\text{devmsg}''_0 &= \text{devmsg}''_0, \text{devrpy}''_0 = \text{devrpy}''_0, \\
\text{intno}''_0 &= \text{intno}''_0, \text{serr}''_0 = \text{serr}''_0, \\
\text{curr}''_0 &= \text{curr}''_0, \text{devs}''_0 = \text{devs}''_0, \text{ipid}''_0 = \text{ipid}''_0, \\
\text{procs}''_0 &= \text{procs}''_0, \text{queued}''_0 = \text{queued}''_0; \\
\text{prove;} \\
\text{prove;} \\
\text{next;} \\
\text{next;} \\
\end{align*}

\textbf{proof[EnqueueDeviceSched0ExistsPost]}
\begin{align*}
\text{invoke } & \text{EnqueueDeviceSched0;} \\
\text{invoke } & \text{EnqueueDeviceSched0;} \\
\text{rewrite;} \\
\text{\textbf{\textbullet}} \\
\end{align*}

\textbf{proof[ErrNotDevicePIDExistsPost]}
\begin{align*}
\text{invoke } & \text{ErrNotDevicePID;} \\
\text{invoke } & \text{NotDevicePID0;} \\
\text{invoke } & \text{EnqueueDeviceSched0;} \\
\text{rewrite;} \\
\text{\textbf{\textbullet}} \\
\end{align*}

215
**proof**[lEnqueueDeviceSchedExistsPost]
 invoke EnqueueDeviceSched;
split EnqueueDeviceSched0;
cases;
 prove;
next;
split ErrNotDevicePID;
 prove;
split ErrBadPIDCurr;
 prove;
next;

**proof**[fReadyDeviceProcessExistsPost]
 invoke ReadyDeviceProcess;
 prove;
use

!EnqueueDeviceSchedExistsPost[cdseg_0/cdseg', devmap_0/devmap', devmsg_0/devmsg', devrpy_0/devrpy', dsseg_0/dsseg', extpid_0/extpid', free_0/free', msgq_0/msgq', nextupid_0/nextupid', pidext_0/pidext', ptype_0/ptype', serr_0/serr', state_0/state', tss_0/tss', used_0/used'];

 prove;
 invoke Sched;
 invoke DeviceQueue;
 prove;
 invoke ErrStateToReady;
 invoke ErrSysFailPTab;
 invoke IsSysOk;
split serr_0 = sysok;
cases;
 prove;
 invoke SetProcState;
 invoke PTabChangeState;
 prove;
 invoke Ξ PTab;
 prove;
 invoke PQueue;
 prove;
next;
 prove;
 invoke Ξ PTab;
 invoke PQueue;
 prove;
next;

**proof**[lRequeueDeviceProcessExistsPost]
 invoke RequeueDeviceProcess;
 prove;

proof[fEnqueueDeviceSched0ExistsPostHW]  
invoke EnqueueDeviceSched0;  
invoke EnqueueDeviceQueue;  
invoke EnqueuePQueue;  
invoke RaiseOk;  
split HW[intno'/intno];  
prove;  
with disabled (Sched, PQueue, PTab, PID, ErrV, HW) invoke;  
prove;  

proof[fErrNotDevicePIDExistsPostHW]  
with disabled (Sched, PTab, PID, ErrV, HW) invoke;  
prove;  

proof[lEnqueueDeviceSchedExistsPostHW]  
invoke EnqueueDeviceSched;  
split EnqueueDeviceSched0;  
cases;  
prove;  
next;  
split ErrNotDevicePID;  
prove;  
split ErrBadPIDCurr;  
prove;  
next;  

proof[fReadyDeviceProcessExistsPostHW]  
invoke ReadyDeviceProcess;  
prove;  
use  
lEnqueueDeviceSchedExistsPostHW|cdseg_0/cdseg', devmap_0/devmap',  
devmsg_0/devmsg', devrpy_0/devrpy',  
dseg_0/dseg', extpid_0/extpid',  
free_0/free', msgq_0/msgq',  
extupid_0/nextupid', pidext_0/pidext',  
potype_0/potype', serr_0/serr',  
state_0/state', tss_0/tss',  
used_0/used');  
prove;  

proof[lRequeueDeviceProcessExistsPostHW]  
invoke RequeueDeviceProcess;  
prove;
\textbf{proof}[tRequeueDeviceProcessTransPre1] 
\texttt{invoke RequeueDeviceProcessTransPre1;} 
\texttt{use tRequeueDeviceProcessPre;} 
\texttt{prove;} 
\texttt{instantiate} 
\texttt{nextupid\_0' == nextupid', used\_0' == used', free\_0' == free',} 
\texttt{tss\_0' == tss', ptype\_0' == ptype', state\_0' == state',} 
\texttt{extpid\_0' == extpid', pidext\_0' == pidext', msgq\_0' == msgq',} 
\texttt{cdseg\_0' == cdseg', dsseg\_0' == dsseg', devmap\_0' == devmap',} 
\texttt{devmsg\_0' == devmsg', devrpy\_0' == devrpy',} 
\texttt{intno\_0' == intno', serr\_0' == serr',} 
\texttt{curt\_0' == curt', devs\_0' == devs', ipid\_0' == ipid',} 
\texttt{procs\_0' == procs', queued\_0' == queued';} 
\texttt{prove;} 
\texttt{invoke SchedBackup;} 
\texttt{invoke SchedBackupSig;} 
\texttt{prove;} 
\texttt{invoke \Delta Sched;} 
\texttt{prove;} 
\texttt{use lRequeueDeviceProcessExistsPost;} 
\texttt{prove;}
proof[tRequeueDeviceProcessTransPre2]
    invoke RequeueDeviceProcessTransPre2;
    use tRequeueDeviceProcessTransPre1;
    prove;
    use tSchedTransPre[cdseg' / cdseg, curr' / curr,
    devmap' / devmap, devmsg' / devmsg,
    devrpy' / devrpy,
    devs' / devs, dsseg' / dsseg,
    extpid' / extpid, free' / free,
    intno' / intno, ipid' / ipid, msgq' / msgq,
    nextupid' / nextupid, pidext' / pidext,
    procs' / procs, ptype' / ptype,
    queued' / queued, serr' / serr,
    state' / state, tss' / tss, used' / used];
    invoke SchedTransSig;
    invoke RequeueDeviceProcessSig;
    invoke SchedBackupSig;
    invoke SchedTotalSig;
    invoke SchedOpTotalSig;
    invoke SchedOpSig;
    invoke RequeueDeviceProcessTransPre1;
    prove;
    invoke RaiseErrSig;
    invoke ErrV;
    invoke RaiseKillInterruptSig;
    invoke RaiseNamedInterruptSig;
    use lRequeueDeviceProcessExistsPostHW;
    prove;
    instantiate
    nextupid_0 == nextupid, used_0 == used, free_0 == free',
    tss_0 == tss', ptype_0 == ptype', state_0 == state',
    extpid_0 == extpid', pidext_0 == pidext', msgq_0 == msgq',
    cdseg_0 == cdseg', dsseg_0 == dsseg', devmap_0 == devmap',
    devmsg_0 == devmsg', devrpy_0 == devrpy',
    intno_0 == intno', serr_0 == serr',
    curr_0 == curr', devs_0 == devs', ipid_0 == ipid',
    procs_0 == procs', queued_0 == queued',
    nextupid_1' == nextupid_0', used_1' == used_0', free_1' == free_0',
    tss_1' == tss_0', ptype_1' == ptype_0', state_1' == state_0',
    extpid_1' == extpid_0', pidext_1' == pidext_0', msgq_1' == msgq_0',
    cdseg_1' == cdseg_0', dsseg_1' == dsseg_0', devmap_1' == devmap_0',
    devmsg_1' == devmsg_0', devrpy_1' == devrpy_0',
    intno_1' == intno_0', serr_1' == serr_0',
    curr_1' == curr_0', devs_1' == devs_0', ipid_1' == ipid_0',
    procs_1' == procs_0', queued_1' == queued_0';
    prove;
\textbf{proof}[tRequeueDeviceProcessTransPre]

\begin{itemize}
  \item \texttt{invoke RequeueDeviceProcessTransSig;}
  \item \texttt{invoke RequeueDeviceProcessTrans;}
  \item \texttt{split} $3$ \texttt{Sched''} $\cdot$ \texttt{Sched''} $=$ $\emptyset$ \texttt{Sched;}
  \item \texttt{cases;}
  \item \texttt{prener;}
  \item \texttt{use tRequeueDeviceProcessTransPre2;}
  \item \texttt{invoke RequeueDeviceProcessSig;}
  \item \texttt{invoke SchedBackupSig;}
\end{itemize}

\texttt{prove;}

\begin{itemize}
  \item \texttt{invoke RequeueDeviceProcessTransPre2;}
  \item \texttt{prove;}
\end{itemize}

\texttt{instantiate
\begin{align*}
  \texttt{nextupid}_1 & \equiv \texttt{nextupid}_0, \texttt{used}_1 \equiv \texttt{used}_0, \texttt{free}_1 \equiv \texttt{free}_0, \\
  \texttt{tss}_1 & \equiv \texttt{tss}_0, \texttt{ptype}_1 \equiv \texttt{ptype}_0, \texttt{state}_1 \equiv \texttt{state}_0, \\
  \texttt{extpid}_1 & \equiv \texttt{extpid}_0, \texttt{pidext}_1 \equiv \texttt{pidext}_0, \texttt{msgq}_1 \equiv \texttt{msgq}_0, \\
  \texttt{cdseg}_1 & \equiv \texttt{cdseg}_0, \texttt{dsseg}_1 \equiv \texttt{dsseg}_0, \texttt{devmap}_1 \equiv \texttt{devmap}_0, \\
  \texttt{devmsg}_1 & \equiv \texttt{devmsg}_0, \texttt{devrpy}_1 \equiv \texttt{devrpy}_0, \\
  \texttt{intno}_1 & \equiv \texttt{intno}_0, \texttt{serr}_1 \equiv \texttt{serr}_0, \\
  \texttt{curr}_1 & \equiv \texttt{curr}_0, \texttt{devs}_1 \equiv \texttt{devs}_0, \texttt{ipid}_1 \equiv \texttt{ipid}_0, \\
  \texttt{procs}_1 & \equiv \texttt{procs}_0, \texttt{queued}_1 \equiv \texttt{queued}_0, \\
  \texttt{nextupid}_0'' & \equiv \texttt{nextupid}_0', \texttt{used}_0'' \equiv \texttt{used}_0', \texttt{free}_0'' \equiv \texttt{free}_0', \\
  \texttt{tss}_0'' & \equiv \texttt{tss}_0', \texttt{ptype}_0'' \equiv \texttt{ptype}_0', \texttt{state}_0'' \equiv \texttt{state}_0', \\
  \texttt{extpid}_0'' & \equiv \texttt{extpid}_0', \texttt{pidext}_0'' \equiv \texttt{pidext}_0', \texttt{msgq}_0'' \equiv \texttt{msgq}_0', \\
  \texttt{cdseg}_0'' & \equiv \texttt{cdseg}_0', \texttt{dsseg}_0'' \equiv \texttt{dsseg}_0', \texttt{devmap}_0'' \equiv \texttt{devmap}_0', \\
  \texttt{devmsg}_0'' & \equiv \texttt{devmsg}_0', \texttt{devrpy}_0'' \equiv \texttt{devrpy}_0', \\
  \texttt{intno}_0'' & \equiv \texttt{intno}_0', \texttt{serr}_0'' \equiv \texttt{serr}_0', \\
  \texttt{curr}_0'' & \equiv \texttt{curr}_0', \texttt{devs}_0'' \equiv \texttt{devs}_0', \texttt{ipid}_0'' \equiv \texttt{ipid}_0', \\
  \texttt{procs}_0'' & \equiv \texttt{procs}_0', \texttt{queued}_0'' \equiv \texttt{queued}_0', \\
\end{align*}

\texttt{prove;}

\begin{itemize}
  \item \texttt{next;}
  \item \texttt{prove;}
  \item \texttt{next;}
\end{itemize}