ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 173

Systems Group, Department of Computer Science, ETH Zurich

Process Management in a Capability-Based Operating System

by

Razvan-Gabriel Damachi

Supervised by

Prof. Timothy Roscoe, Simon Gerber

March 2017 – September 2017

inf | Informatik
Computer Science

Abstract

Process management is a core responsibility of every operating system. This thesis describes a process state machine for the Barrelfish operating system. The pre- and post-conditions of every state are formally checked using temporal logic formulation. Based on the model, a process subsystem is designed and implemented on top of the distributed, capability-based Barrelfish infrastructure. An authentication protocol between the parties involved in process management is formally specified. Performance of the subsystem is modeled mathematically using queuing theory and compared to empirical measurements. The main benefit of our approach consists in providing a model-checked theoretical basis and a capability-constrained interface for process operations.

Contents

Acknowledgements

Firstly, I would like to thank Prof. Timothy Roscoe, who offered me the chance to work on this project, assessing that it would particularly suit my technical interests. I am also grateful to Simon Gerber and the rest of the Barrelfish team, whose feedback and pointers have greatly helped shape the work presented in this thesis.

Secondly, I would like to thank my girlfriend Yana and my parents for their advice and support, which have been invaluable over the course of the thesis and the Master's programme itself.

Finally, I would like to thank the friends I have made while studying in ETH, who have been with me through both fun times and long working nights.

Chapter 1

Introduction

1.1 Motivation

All operating system flavors feature the notion of a *program in execution*, abstracted into some sort of *actor* which uses system resources to perform its tasks. Conventionally, this actor is referred to as a *process*. Despite the fact that the exact definition of a process varies from one system or specification to another, it universally holds that process creation, destruction and resource accounting should be implicitly abstracted away from the user writing the program code. It thus follows that *process management* falls under the list of fundamental roles any operating system must play. This thesis aims at conceptualizing a formal model for process management in the Barrelfish operating system. Based on the model, we present a distributed process interface, which we implement on top of the current Barrelfish infrastructure.

1.1.1 A process subsystem for Barrelfish

Barrelfish is a capability-based multikernel[1] operating system, designed for the purpose of joining many heterogeneous pieces of hardware. Implicitly, the challenge arising from tailoring a process management subsystem to the likes of Barrelfish is threefold. Firstly, the subsystem should be reasonably well distributed across the arbitrary number of cores running the OS, in order to achieve good balance with respect to the consistency-availability-partition tolerance (CAP) triad. Secondly as much of the implementation as possible should be pushed above the kernel, into the user space. Lastly, the system should make use of the capability infrastructure in Barrelfish – anything that can be implemented using capabilities *should* be implemented using capabilities.

1.2 Objectives

Documenting the Barrelfish approach to processes has so far equated to describing dispatchers, the resources they encapsulate via capabilities and the scheduling policies that bring them into execution. Therefore, while the model behind a Barrelfish dispatcher is fairly well understood, we have yet to thoroughly formalize the requirements and implications of how multiple dispatchers come together to form a *domain*[1].

From the perspective of a conceptual Barrelfish process model formally merging the abstractions of dispatcher and domain, this thesis will provide answers to the following questions:

- What is a *process*?
- What can a process *do*?
- What actions can be performed *on a process* and *who* can perform them?
- What is the *life cycle* of a process?

[1] Over the course of this thesis, the terms *domain* and *process* shall be used interchangeably in the context of Barrelfish. The words hold separate meanings when describing other operating systems in chapters 2 and 3.

- Can a process subsystem benefit from the *TTY subsystem*?

1.3 Contributions

The first contribution of the thesis consists in an abstract process model for Barrelfish, which will establish *what a process is*. The model will also allow to formally specify and model-check pre- and post-conditions for every state that processes can be in. The specification will mathematically guarantee that the operating system controls when a process *starts running*, *stops running* and *releases its capabilities*.

Based on the abstract model, the thesis will present the design and implementation of a process management subsystem for Barrelfish. The subsystem will provide a universal, capability-constrained interface to process operations. In addition, it will facilitate keeping track of past and current processes and their state.

The main contribution of the subsystem resides in a complete authorization model for processes based on capabilities. Specifically, the subsystem will use capabilities to identify the principals involved in process-related operations and to assess what rights they hold. To this end, a formal authentication protocol will be designed and implemented. The protocol will serve two security purposes. Firstly, it will establish secure channels for exchanging process management messages. Secondly, it will help conceal *spawnd* servers from client domains by eliminating their need of registering with the nameserver.

The thesis will then provide a means of comparing the performance of process-related operations on the *bsp* core and on an *app* core. Additionally, we will describe a strategy for bottleneck analysis based on queuing theory.

1.4 Thesis structure

In chapter 2 we will briefly describe processes and process creation techniques in UNIX-based operating systems (mainly FreeBSD), as well as the closest Barrelfish equivalents. We will attempt to answer the thesis questions stated in section 1.2 from the perspective of UNIX processes. The same chapter will introduce two frameworks for formal system modeling, which we will later use to check fundamental properties of our process management solution. Some of the work related to processes in capability-based operating systems will then be surveyed in chapter 3.

We will introduce our process model in chapter 4 and use formal logic to specify and model-check its properties. Subsequently, we will present the process management interface and the authentication protocol in chapter 5. This is also where we will discuss the decisions that shaped the process management subsystem, comparing them to relevant alternatives.

The interface implementation will then be detailed in chapter 6, where we will describe several techniques for improving the system's performance. We will also provide an example use case to help demonstrate how the process management subsystem works.

In chapter 7 we will analyze empirical run time measurements obtained from benchmarking. We will also present the queuing theory models used to investigate the subsystem's bottlenecks.

Finally, chapter 8 will present the thesis conclusions and vectors for future work.

Chapter 2

Background

2.1 UNIX processes

This section describes several fundamental details of the process management model used in UNIX systems. We focus on defining a process and discussing the actions that processes can be subjected to. In addition, we analyze how the rights to perform those actions on a process can be acquired. Exploring these details is important for highlighting the differences between UNIX and the capability-based approach presented later for Barrelfish.

2.1.1 What is a process?

UNIX-based operating systems use *processes* as the design abstraction of a program in execution. The Open Group Base Specifications Issue 7[2] defines a live process as *an address space with one or more threads executing within that address space and the required system resources for those threads.* The FreeBSD operating system gives a thinner definition of the kernel's low-level view of a process as a *task or thread in execution*[3] (below the higher level of POSIX threads), while the address space and resources of a process are referred to as the *process context.*

2.1.2 Creating processes via fork-and-exec

The traditional approach to creating new processes in UNIX systems consists in *fork-and-exec*, wherein firstly the state of the *parent process* is duplicated and secondly the identical copy is replaced by a newly created *child process.* Although the fork-and-exec paradigm is implemented in many a popular operating system nowadays, such as Linux or Mac OS X, it incurs inherent overhead in copying the parent's pages during `fork()`, only to have them dropped and replaced by new pages following the coupled call to `exec()`.

2.1.3 Creating processes via spawning

To improve performance, modern implementations of `fork()` do not copy pages by default, but instead mark them as read-only and lazily copy them later on when the child process attempts a write. This is known as *copy-on-write.* However, copy-on-write implies that virtual addresses of different processes need to be translated to the same physical address, meaning the underlying hardware needs to be capable of performing *dynamic address translation.* In order to work around this limitation, the 6^{th} issue of the specification at [2] introduced the optional *posix_spawn()* extension.

Spawning is another approach to starting new processes, one which differs from the UNIX fork-and-exec model in that it creates a child process directly, without forking an identical copy of the parent first. The paradigm was used by the DOS family of operating systems and has been inherited by Microsoft Windows, where it is implemented as a family of functions which *create and execute a new process*[4]. Spawning is not a perfect stand-in for the more complex fork-and-exec, in which the user has programmatic control over arbitrary operations between the moment when

a copy of the current process is created and when the new image is executed. The technique also lacks the fork-and-exec feature of inherently creating a hierarchical process structure in the system. The specification mentions however that fully duplicating fork-and-exec functionality is not expected *in a simple, fast function with no special hardware requirements*[2].

2.1.4 What can a process do?

At the highest level of abstraction, a UNIX process executes the code of the program it was created to run. More granularly, a process executes one or more *threads*, using system resources allocated to it by the operating system (*e.g.* CPU time, memory, files). A process's threads make use of its address space to perform *computation* and *communication*. In UNIX, the most important implementation building blocks of IPC (*inter-process communication*) consist of signals, pipes, message queues, semaphores, shared memory segments etc, using which processes can exchange data and control information during their lifetime.

Furthermore, UNIX processes can perform actions that potentially result in changing the behavior of other processes' execution. We have already seen that new processes can be created using fork-and-exec (or spawn), wherein the initiator of the process creation action becomes the parent and the recipient of the action becomes the child. Other relevant actions include:

- voluntarily *exiting* execution: typically performed through an *exit syscall*, exiting always returns a *termination status* to the parent process;
- *waiting* for a *descendant* or *group of descendants* to exit execution – the initiator can request resource usage and exit status codes;
- *killing* a process, thereby forcefully causing it to exit execution; killing is done via the *kill syscall*, which posts `SIGKILL` to the victim process.

2.1.5 Who can affect the normal execution of a process?

First and foremost, the operating system itself can choose to forcefully terminate a process. In FreeBSD this is normally the case when the victim has caused a hardware event, such as an illegal instruction[3].

Secondly, other processes can post kill signals via the kill syscall, as long as either the sending and receiving processes have *the same effective user identifier*, or the sender is *superuser*. It thus follows that in UNIX, killing a process is tightly coupled with the concept of *user credentials*. Credentials are an access control tool and are assigned to processes by an external authority, *e.g.* by the filesystem at user login in FreeBSD[3]. Moreover, processes inherit their parent's credentials when they are forked. Credentials can be temporarily augmented (*i.e.* via `setuid`), thus introducing the aforementioned notion of an effective user identifier, which is checked when attempting to kill another process.

2.1.6 What is the life cycle of a process?

In FreeBSD, the first state a process exists in is NEW, which indicates that the process has been created following a fork call and is currently pending resource allocation. When enough resources have been allocated, the process transitions to state NORMAL, which it will remain in after it begins executions and until it terminates. A process can terminate either of its own accord, or following an external signal. Once execution has ended, the state changes to ZOMBIE, indicating that the process is waiting for its resources to be cleaned up and its exit code to be communicated to the parent. When this has happened, the resources have been recycled and the process virtually stops existing. A diagram depicting the process life cycle is presented in Figure 2.1.

Moreover, McKusick et al state that while a process is in state NORMAL, its threads fluctuate between states RUNNABLE (preparing to be executed, or actually executing), SLEEPING (waiting for an event) and STOPPED (by a signal or the parent process)[3].

Lastly, it is important to mention that in FreeBSD (and, largely speaking, in UNIX) the convention is that once a process has finished execution – *i.e.* it has entered state ZOMBIE –

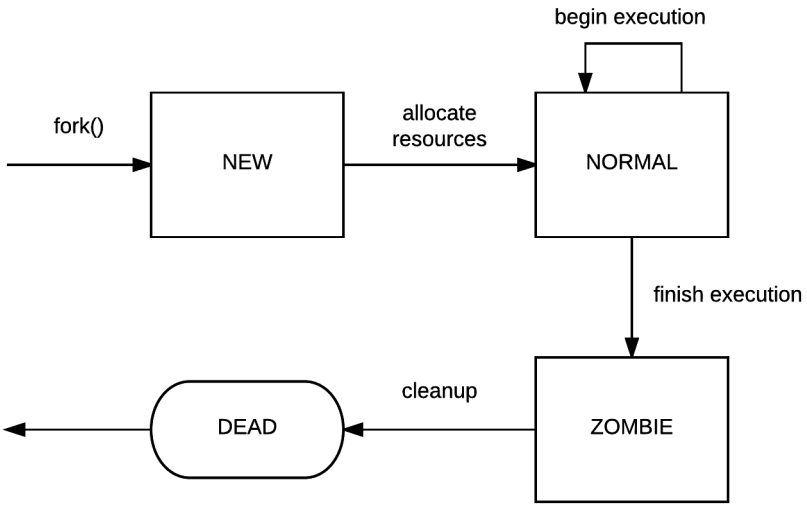


Figure 2.1: Life cycle of a FreeBSD process. Here, state DEAD means that the process has had its resources cleaned up and its exit code communicated to the parent; as far as the OS is concerned, the process has ceased to exist.

its resources will be cleaned up by its parent. However, if some process finishes execution before its children, the responsibility for cleaning up said children's resources is passed to the special *init* process. Among other responsibilities such as helping boot up the system, init *reaps* zombie processes without parents. This constitutes the strategy for ensuring that process resources are not leaked when execution is complete.

2.1.7 Processes and the TTY subsystem

Before talking about the TTY subsystem in UNIX, let us survey a few relevant concepts from the FreeBSD perspective, as presented by McKusick et al[3]:

1. **Process groups** (sometimes referred to as *jobs*) are a means of combining multiple processes for control reasons, such as sending all of them a signal. When created, processes inherit the group of their parent, however the system provides mechanisms using which processes can branch out and create their own groups.

2. **Sessions** are collections of process groups. Like the name suggests, sessions are used to create isolated environments for processes (and groups). An outstanding example is a user's *login shell*, which allows interfacing with the user and which is implemented as the process at the root of a user-dedicated session. The root process is also called the *session leader*.

A **TTY** commonly refers to a (physical or virtual) terminal device. In FreeBSD, a session leader can set up a connection to a TTY driver, action which is referred to as allocating a *controlling terminal*. This way, a user session such as the one started by the login shell can grant its user access to input/output facilities through the terminal device.

The relation between processes and the TTY subsystem can be summed up as follows:

1. Processes are collected into groups;

2. Groups are collected into sessions;

3. Sessions are linked with terminal devices via controlling terminals.

2.2 Barrelfish domains and dispatchers

The Barrelfish Architecture Overview technical note[5] refers to user level applications and servers as *domains*, implemented using one or more *dispatchers*. In Barrelfish, a dispatcher is a scheduler activation implementation, being the closest correspondent to a UNIX process. Every instance of the CPU driver schedules and runs its own dispatchers. Every dispatcher then schedules and runs its own userspace threads for the domain it represents.

Domains can span across multiple cores, *e.g.* in OpenMP applications, by giving the CPU driver instance on each target core a dispatcher for the respective domain. Dispatchers, however, do not migrate between cores.

2.2.1 Spawning in Barrelfish

Barrelfish as a whole is designed with the goal of being highly compatible with a multitude of interconnected heterogeneous devices. This includes commodity hardware many instances of which can be joined together to achieve fast, scale-out architectures. Any implementation decision must try to incur as few hardware requirements as possible. Barrelfish hence prefers the spawn paradigm to fork-and-exec since it is simpler, faster and has no particular hardware requirements.

In addition, implementing fork would raise questions regarding the relation between the child and parent cspaces. The implementation would have to establish which of the parent's capabilities should be copied into the child's cspace and what security implications sharing those capabilities would incur. For instance, if the forked process keeps running the same program as its parent then sharing capabilities might be acceptable. However if a new program is to be run instead (like in the case of fork-and-exec in UNIX), then it might be unsafe to allow it to access the parent's capabilities. Lastly, there the question of when it would be best to replicate the cspace, *i.e.* immediately after forking or later on, through copy-on-write.

2.2.2 Spawning as a service

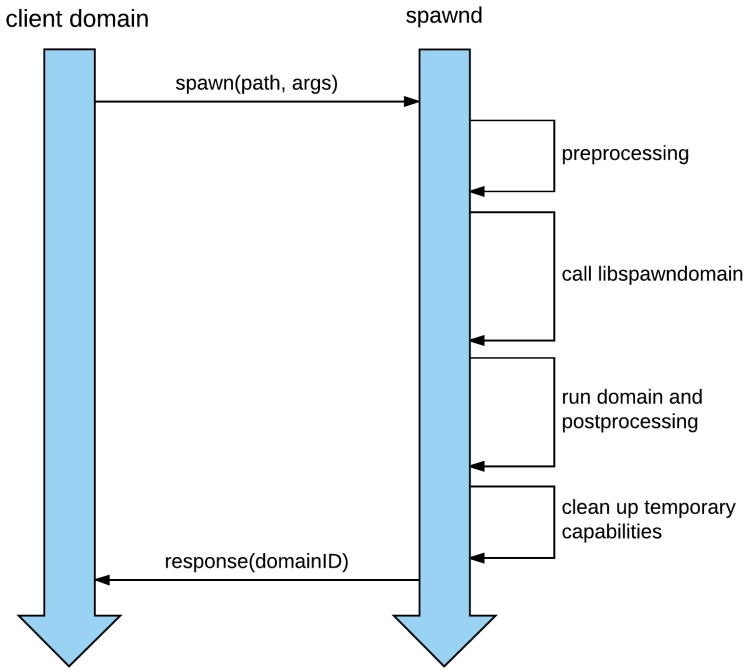
Leveraging the distributed, message-passing nature of Barrelfish, spawning is offered to user domains *as a service*, as opposed to *as a library*. Furthermore, since in Barrelfish every core runs its own instance of the CPU driver and various services in the user space on top of that, the spawning service is provided at a per-core level, *i.e.* every core runs its own server implementing the spawn interface. We refer to such as server as *spawnd* (from spawn daemon). Consequently, the spawn subsystem as of when research for this thesis had begun required that if one wanted to spawn a domain on some arbitrary core in the system, they must do so using the spawnd instance on that core, through remote procedure calls (RPCs). Such an RPC is sketched in Figure 2.2.

It is worth pointing out that this strategy of communicating directly with the spawnd on the core we want to run the new program, albeit simple and fast, implies that every individual spawnd instance will only possess knowledge of the programs it has been personally asked to spawn. In other words, *unless some consensus strategy is designed*, every spawnd server will know the state of the dispatchers it is running, but not that of any *overarching domain*.

In addition, as of when research for the thesis had started, a spawnd instance would assign any new program it spawned a numeric core-local ID, so far called a *domain ID* for convenience (*dispatcher ID* would be more appropriate semantically). The problem with assigning such an ID is twofold: firstly, when core A and core B each spawn their first program, they will both assign it the same initial ID. On propagating this ID to some other core C, the latter will find it non-trivial to distinguish between the program spawned by A and that spawned by B. Secondly, the ID returned is numeric, which makes it trivial for an attacker to forge it and hence impersonate the newly spawned domain. Both these issues will be addressed by the process management solution presented in this thesis.

2.3 Capabilities

Described briefly in the technical note[5], Barrelfish capabilities are the building block of the operating system's access control mechanism. In contrast to the access control lists primarily



centering

Figure 2.2: Flow of a *spawn* RPC (from top to bottom). Here, *libspawndomain* refers to the Barrelfish library that offers functionality for spawning, such as setting up a dispatcher's capability, virtual address space, dispatcher frame etc; *domainID* is a core-local numeric identifier returned for the newly spawned domain.

used in UNIX systems, which, given an object, answer the question of *which entities can access the object*, capabilities shift focus to the subject, answering the question of *what objects a given entity can access*. Capabilities are paramount to implementing a process subsystem in Barrelfish, being used for various purposes such as referencing dispatchers and their resources, as well as inter-dispatcher communication.

2.3.1 Inter-dispatcher communication using capabilities

In Barrelfish, core-local messages and RPCs use *endpoint capabilities* to identify the receiver of a message, or the server implementing the remote procedure. The interconnect infrastructure for messages sent to recipients on the same core as the sender is called *local message passing (LMP)*. Abstractly speaking, given two dispatchers A and B running on the same core, an LMP channel between the two is a unidirectional channel oriented as either $A \to B$ (only B can receive messages on the channel), or $B \to A$ (only A can receive messages on the channel). Consequently, in order for A to send messages to B via some $A \to B$ channel, A needs to hold an endpoint capability for B. On the other hand, B can allocate multiple endpoint capabilities, which can be created by retyping from B's *dispatcher control block (DCB)* capability, and which B can give to arbitrary dispatchers to act as clients in communicating with it.

For messages sent between dispatchers running on different yet cache-coherent cores, the interconnect driver used is called *user-level message passing (UMP)*. While in the case of UMP the implementation is not based on endpoint capabilities, the two protocols share the abstraction that arbitrary dispatchers A and B can be bound through unidirectional channels $A \to B$ and $B \to A$.

2.4 Temporal Logic of Actions

Temporal Logic of Actions (TLA+)[6] is a specification framework for behavioral properties. It can be used to model real-time systems and describe essential conditions and behaviors over the systems' state space, using formal mathematical and logical notation. For example, given two states A and B of some arbitrary system, TLA+ can be applied to formally check whether once in state A, the system will *ever* (or *always*) eventually reach state B.

2.4.1 TLA+ models

A TLA+ model consists of **variables** and **states**. Variables are defined on arbitrary mathematical spaces, such as booleans or natural numbers. A state is a particular assignment of values to variables, *i.e.* a mapping $(v_1, v_2, ...) \mapsto (val_1, val_2, ...)$, where val_1 and val_2 are of the types on which v_1 and v_2 are respectively defined.

A model **transitions** between states when at least one variable changes its value. Any transition from any state A depends only on the values of variables in A (*i.e.* does not depend on values of past states). A **behavior** is a (potentially infinite) sequence of states $S_1, S_2, ...$, starting with a user-defined **initial state**.

2.4.2 Behavior operators

TLA+ includes the following temporal logic operators, described in [6]:

Globally. A formula $\Box P$, where P is a state predicate, is true of a behavior iff P is true in every state of the behavior.
Eventually. $\Diamond F$ is defined to equal $\neg\Box\neg F$. It asserts that F is not always false, which means that F is true at some point in time.

2.4.3 PlusCal

PlusCal is an algorithm language based on TLA+[7]. A PlusCal algorithm can be translated to a TLA+ specification, which is useful when modeling systems that are difficult to formalize using sheer mathematics. PlusCal has a C-based syntax and a Pascal-based one; in this thesis, the C-syntax will be used to describe the domain state model.

2.4.4 TLC model checker

Specifications written in TLA+ or translated from PlusCal can be subjected to model checking using the TLC model checker[8]. TLC will be used to prove fundamental properties of the domain model presented later in the thesis.

2.5 BAN logic

Burrows-Abadi-Needham (BAN) logic[9] is a language for formally specifying authentication protocols. BAN logic focuses on what the participants in a protocol *say*, *see* and *believe* to infer whether consensus is reached with regards to information held by the principles and their communication keys or channels.

2.5.1 Notation

The paper by Burrows, Abadi and Needham[9] presents a set of specification formulae, of which the following are of interest to this thesis:

- P **believes** X: the principal P may act as though X is true;
- P **sees** X: someone has sent a message containing X to P; P can read and repeat X;

- P **said** X: the principal P at some point in time sent a message including the statement X;
- P **controls** X: P has jurisdiction over X, meaning it is an authority over X and should be trusted on the matter of X; for example, in Barrelfish the memory server is trusted with providing clients with valid RAM;
- **fresh**(X): the formula X is fresh, *i.e.* it is still valid during the current run of the authentication protocol;
- $P \overset{K}{\leftrightarrow} Q$: P and Q may use the shared key K to communicate; in addition, K will never be discovered by a principal other than P, Q, or a principal trusted by P or Q.

2.5.2 Deduction rules

Based on the rules mentioned above, the paper[9] states the following deduction rules:

The **message-meaning** rule:

$$\frac{P \textbf{ believes } Q \overset{K}{\leftrightarrow} P, \quad P \textbf{ sees } \{X\}_K}{P \textbf{ believes } Q \textbf{ said } X},$$

reading *if P believes it can communicate with Q through the shared key K and P sees message X encrypted with key K, then P believes that Q said X.*

The **nonce-verification** rule:

$$\frac{P \textbf{ believes fresh}(X), \quad P \textbf{ believes } Q \textbf{ said } X}{P \textbf{ believes } Q \textbf{ believes } X},$$

reading *if P believes X is currently valid and P believes that Q said X, then P believes that Q believes X.*

The **jurisdiction** rule:

$$\frac{P \textbf{ believes } Q \textbf{ controls } X, \quad P \textbf{ believes } Q \textbf{ believes } X}{P \textbf{ believes } X},$$

reading *if P believes that Q has jurisdiction over X and P believes that Q believes X, then P itself believes X.*

Chapter 3

Related Work

This chapter surveys process models used in other capability-based operating systems. We investigate how processes are created, as well as what the closest equivalents to Barrelfish domains and dispatchers might be. Additionally, we assess whether other operating systems feature means of collectively managing threads or processes that run on different cores and share the same virtual address space.

3.1 KeyKOS

KeyKOS is a capability-based, object-oriented, *nano-kernel* operating system that was first used in production in 1983[10]. Originally designed to *solve security, sharing, pricing, reliability and extensibility requirements of commercial computer services in network environments*[11], KeyKOS treats every interacting system component as an *object* which uses *keys* (*i.e.* capabilities) to send messages to other objects.

3.1.1 KeyKOS domains

The *KeyKOS Architecture* paper by Hardy calls the *actors* of KeyKOS *domains*[11], stating that all events in KeyKOS are the result of an action performed by some domain. Domains are said to *interpret* programs, whereby they provide the context that a program needs in order to run, including general-purpose registers, the program's address space and several *key slots*. KeyKOS key slots are much like Barrelfish capability slots in *cnodes*. This implies that in spite of the naming confusion KeyKOS domains correspond in fact best to Barrelfish dispatchers.

3.1.2 Process creation

KeyKOS creates processes by building the address space segment, *obtaining a fresh domain* and inserting the key (capability) to the address segment into the domain's special address segment slot[10]. It is also stated that if two or more domains are to share the same address space in a thread-like paradigm, a key to the same address segment should be given to them all, similarly to how Barrelfish dispatchers within a domain share the same *vspace*.

The architecture papers by Hardy et al[10][11] mention that the system provides *domain factories*, which are special domains that instantiate other domains of a given type. Factories are described in a publishing patent by Hardy[12]: a factory can provide a *requester domain* with the ability to install a program in the factory. Having installed the program, the factory provides the requester with a special key, the invocation of which causes a special domain to be created for the program. This domain allows the requester to use the desired program to process data, without being able to inspect the program itself. Consequently, a factory-created domain acts like a black box which the requesting domain can wait for to process data.

3.1.3 Domain state model

There are three mutually exclusive domain states:

- **running** is the state in which domains execute instructions;
- **available** and **waiting** are states in which domains do not execute instructions.

Transitioning between a domain's states is done through one of the three available invocations – *FORK*, *CALL* or *RETURN* – as follows:

- FORK leaves the domain *running*;
- CALL leaves the domain *waiting*, generating a special *resume key* the invocation of which will put the domain that originally issued the CALL back into the *running* state; basically, CALLs yield execution and wait until the recipient domain has invoked the resume key;
- RETURN leaves the domain in state *available*.

The three inter-domain invocations presented above can be called on two types of keys: *start* and *resume* keys. Invoking a start key queues the invoker until the target domain is in state *available*. This implies that if a domain RETURNs, all domains which were queued up following an invocation of a start key to this domain when it was not available are restarted. Resume keys, on the other hand, only exist for domains which are in the waiting state. A diagram of the state machine described so far is presented in Figure 3.1.

The KeyKOS domain state machine resembles a scheduling algorithm, which reinforces the similarity between KeyKOS domains and Barrelfish dispatchers. Although multiple KeyKOS domains can be combined to simulate a multithreaded environment – much like multiple Barrelfish dispatchers are collected into a domain – there is no mentioning of how the global state of such an environment is modeled based on the state of every individual component.

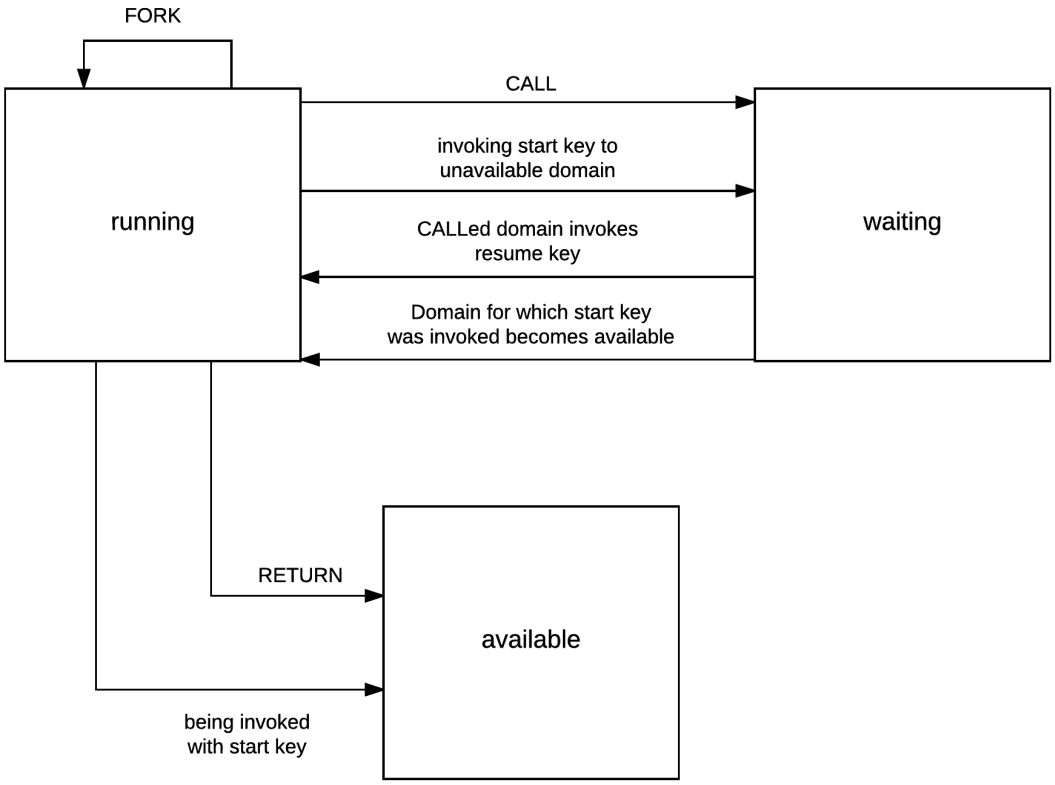


Figure 3.1: KeyKOS domain states and transitions.

3.2 seL4

seL4 is a microkernel operating system designed to be *a secure, safe, and reliable foundation for systems in a wide variety of application domains*[13]. It features a capability-based access control model similar to that of Barrelfish. The technical note mentions that Barrelfish has *a considerably larger type system and extensions for distributed capability management between cores*[5] than that of seL4.

3.2.1 Threads

In seL4, programs are run by single threads. Like Barrelfish dispatchers, seL4 threads have an associated cspace and vspace *which may be shared with other threads*[13]. Both seL4 threads and Barrelfish dispatchers are scheduled by the systems' respective kernels. However Barrelfish dispatchers further encapsulate their own set of threads for execution. In seL4, different threads can also be scheduled together by means of grouping them into the same *domain*.

3.2.2 Domains

A seL4 domain is a collection of threads which are always scheduled together. A thread *belongs to exactly one domain, and will only run when that domain is active*[13]. Domains in seL4 are significantly different from Barrelfish in that:

- different threads belonging to the same seL4 domain can run different programs;
- threads in a seL4 domain do not implicitly share their vspace;
- threads in a seL4 domain are always scheduled together, whereas Barrelfish dispatchers are scheduled independently, even if they belong to the same domain.

Aside from scheduling decisions, seL4 does not manage threads assembled into a domain together. Moreover, there is no means of collectively managing different threads which share the same vspace (whether part of the same domain or not).

Chapter 4

The Process Model

4.1 Defining a domain

We define a domain to be *a uniquely identifiable set of dispatchers* which share the same vspace with the purpose of running a single program, using one or more cores. For the most part, it will be assumed that the dispatcher set is non-empty, at least while the domain is said to be running, as we will see shortly. The words *uniquely identifiable* refer to the requirement that any entity should be able to tell the difference between any two domains running in the system, *e.g.* even if they both run the same program, on the same set of cores.

Among all dispatchers running under a given domain, we single out the chronologically first dispatcher, which we call the *main* or *root* dispatcher. Specifically, the main dispatcher is the one that runs the program which the domain was created for. Subsequent dispatchers share the role of providing the root with easy access to resources only available to their cores. This decision follows the Barrelfish requirement that any dispatcher can only use capabilities referred to in its own cspace, on its own core. Normally, if a dispatcher wants to obtain a capability referenced in another cspace on a different core, it needs to obtain a copy of that capability via message-passing. However, dispatchers running under the same domain have the advantage of *sharing the same vspace*, making it easier for the root dispatcher to access capabilities in non-root dispatchers' cspaces by writing them to and reading them from the same pages.

4.1.1 Comparison to UNIX

We have previously mentioned in chapter 2 that UNIX processes are characterized by an address space, one or more threads executing within that address space, as well as the resources the threads require[2]. Based on the definition outlined above, Barrelfish domains and UNIX processes could be equated in the following way:

- a Barrelfish domain's dispatchers correspond to a UNIX process's threads;
- the vspace that all dispatchers in a domain share corresponds to the address space of a UNIX process;
- the union over the capability space of all of a domain's dispatchers corresponds to the resources UNIX threads need to execute.

However, abiding by such a perspective incurs confusion. As previously mentioned, the most natural Barrelfish equivalent to a UNIX process is a dispatcher, as it is the minimal entity encapsulating threads and an address space for running a program. We therefore choose to view a domain as a level of abstraction that UNIX lacks and define it solely based on the dispatchers that constitute it.

4.2 Domain properties and operations

We define the following core properties regarding the relation between domains and their dispatchers:

1. Upon creation, any dispatcher belongs to **exactly one domain**. Thereafter, during its existence the dispatcher will belong to the same domain, *i.e.* dispatchers cannot transcend domains.

2. Any running domain has **at least one dispatcher**, the root. When it has no dispatchers left, a domain stops running; based on this, we define the stronger property described in point 3:

3. When a domain's root dispatcher stops running, the domain as a whole stops running. In other words, the root dispatcher's running is a *necessary* condition for the domain's running. This also goes to say that non-root dispatchers depend on the root in the sense that they can only be running for as long as the root is.

4. Destroying a dispatcher other than the root will remove it from the corresponding domain's set of dispatchers. However, the state of the domain as a whole will be unaffected.

These properties help shape what a domain *is* in terms of its dispatchers. Furthermore, they hint at what *can be done to* a domain and it dispatchers. We thus define the following **domain operations**:

- creating a domain amounts to creating at least its main dispatcher;
- in order to destroy a domain, it is *necessary* to destroy its main dispatcher and it is *sufficient* to destroy all of its dispatchers.

Once created, the following *dispatcher operations* can be performed within the boundaries of a single domain:

- creating a new dispatcher; the new dispatcher will be considered non-root;
- destroying a non-root dispatcher, which will not affect the state of the domain;
- destroying the root dispatcher, which will cause all the other dispatchers and the effective domain to be destroyed as well.

4.3 Domain state machine

A dispatcher's state machine is a well understood concept in Barrelfish, however, for the scope of our model, we will only be interested in two simplified dispatcher states: whether it is *in the kernel's run queue* on its given core or not[1]. These states will be used to describe what makes a domain *run* or *stop*. We are not interested in what threads the dispatcher is currently scheduling, or even if it is in enabled or disabled mode.

Using the simplified dispatcher state, an aggregated domain state machine can be built. However, since each of a domain's dispatchers can be in a different state at the same time, we present a projection of the aggregated domain state to an arbitrary core in the system. Any given domain will be in exactly one state per core in the system, yet in one single state globally. For a fixed domain and a fixed core, the resulting states are:

- **nil** is the initial state: the core has no knowledge of the domain, meaning that either the domain has not been created yet, or it has been created but it is running on other cores in the system;
- **run** is the state which says that the domain is currently running on this core;

[1]When a dispatcher is in the kernel's run queue, it is vouched to be eventually brought into execution.

- **stop pending** is the state in which this core has removed its dispatcher for the domain from its run queue;
- **stop** is the state in which all cores have removed their dispatcher for the domain from their run queues;
- **cleanup** is the state which says that the domain's resources have been freed; the domain can persist in this state for historical reasons, *i.e.* if any entity in the system should inquire whether *this* particular domain has ever existed.

These individual per-core states compose the global domain state as follows:

- global state **nil** means that all cores are in state *nil*;
- global state **run** means that at least one core is in state *run* and all other cores are either in state *run* or *nil*;
- global state **stop pending** means that at least one core is in state *stop pending*, but no core is in state *stop* yet;
- state **stop** is always global by definition;
- global state **cleanup** means that all cores are in state *cleanup*.

In order to transition between states, cores need to *decide* to either run, stop or clean up their dispatcher for some domain. For simplicity, the model was devised with the assumption that some third party is *instructing* every core to perform transitions through a simple protocol comprised of the following messages:

- **span** instructs the core to run a dispatcher for the given domain; if the domain has not run on any cores yet, this message can be thought of as *spawn*;
- **stop** instructs the core to stop running its dispatcher for the given domain;
- **allstop** informs a core that all cores in the system have stopped running their dispatchers for the given domain;
- **free resources** instructs the core to clean up the resources the domain used locally.

Two alternatives to the third party-based message-passing protocol were considered when designing the model: the first one involved cores sending messages to other cores asking whether they should run or stop a dispatcher for some domain. However, this option was not pursued because it would have incurred many more messages flying around the system. The second alternative consisted in all cores' writing and reading information concerning domains to and from central storage. Central storage would have been an unnecessary single point of failure and it would have required synchronization, therefore this strategy was dropped too. In addition, it seems unnecessary for cores to go out of their way and inquire what to do for a given domain, when it is the domain that wants to use their resources.

Figure 4.1 shows the domain state machine diagram. Table 4.1 then describes the pre- and post-conditions for each state, from the perspective of a fixed core. The state machine must guarantee that once it has reached its final state, the domain is no longer running on any core and it has released resources on all cores.

4.3.1 Outracing malicious domains

A particularly interesting problem that arose while devising the domain state model focused on malicious domains trying to persist in the system, even if some relevant authority decides they should be stopped. For example, imagine that a domain runs on core 0 and the monitor on core 0 decides it should be stopped. However, before the dispatcher can be dequeued from core 0's run queue, the domain spans onto core 1. Then, before the dispatcher on core 1 can be dequeued, the domain spans onto core 2 and so forth. The domain is hence essentially racing the system trying

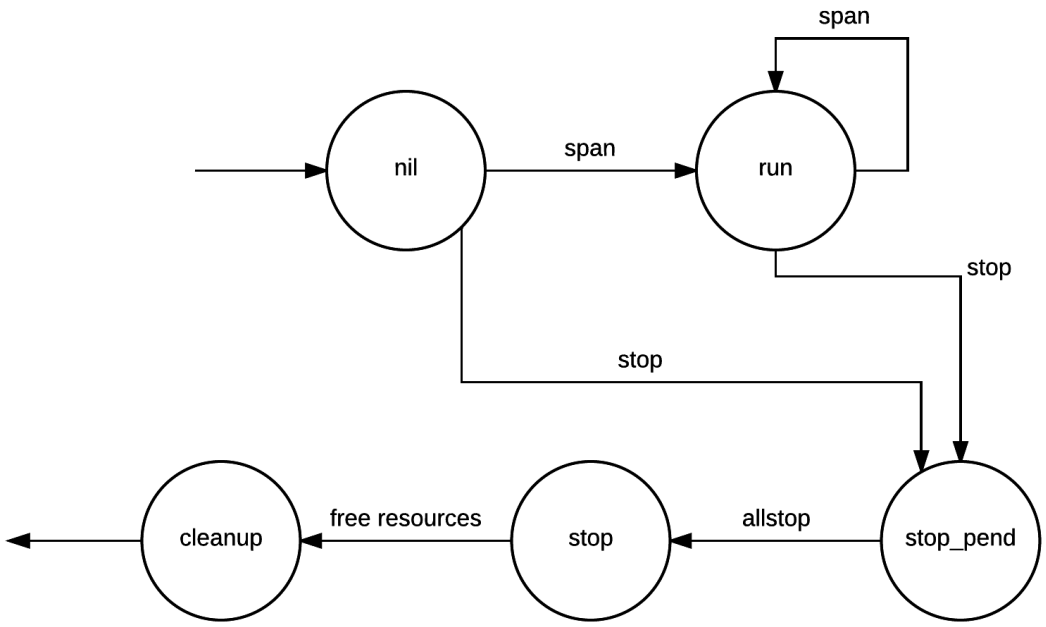


Figure 4.1: The per-core domain state machine. The diagram assumes that once a domain's resources have been freed, it will exist in state *cleanup* for historical reasons; if resource freeing amounted to the domain's been fully erased from the system and its identity recycled, then there would be a *cleanup* to *nil* transition.

State	Requires	Ensures
nil	no message regarding the domain has been received	domain has no dispatcher in this core's run queue, nor is it using the core's resources
run	span message has been received from state *nil* or *run*	dispatcher has been added to the core's run queue and it is using its resources
stop pending	*stop* message has been received	domain's dispatcher has been removed from this core's run queue
stop	all other cores in the system are either in state *stop pending* or *stop*	domain's dispatchers have been removed from *all cores'* run queues
cleanup	*free resources* message has been received from state *stop*	domain's resources have been freed from *all cores*

Table 4.1: Pre- and post-conditions for domain states, from the perspective of an arbitrary domain and an arbitrary core in the system. Requirements are stated in terms of messages received from a trusted third party. Guarantees refer to whether the domain has a dispatcher running on the core and whether it is using the core's resources.

to stop and clean it up. This can be a serious issue, since not only can the domain keep using system resources such as memory and CPU time, it can also carry on manifesting its adversarial behavior (*e.g.* impersonating another principal in the system).

In order to solve this problem, a *stop* message will cause the domain's state to change to *stop pending* even on cores where the domain is not running any dispatchers. In other words, a *stop* message is meant to tell a core that the domain is in the process of being stopped, so the core must remove its dispatcher for the domain if it is currently running one and *no dispatchers for this domain should be run at any point in the future.* This way, when a malicious domain tries to outrun its killing process, it will be denied new dispatchers on any core in the system.

4.4 TLA+ specification

This section presents a TLA+ specification for the domain model, simulating a system running on $N > 0$ cores. An arbitrary domain was fixed for which the simulation walked through all the domain model states. The specification was automatically translated from a PlusCal algorithm. The PlusCal algorithm, the resulting specification and its invariants are given completely in appendix A. We begin by presenting the algorithm and continue by listing the main properties for which the specification was model-checked using TLC.

4.4.1 Variables

The PlusCal algorithm declares the following variables:

- $st : \{1, 2, ..., N\} \to \{nil,\ run,\ stop\_pend,\ stop,\ cleanup\}$ is the domain state projected onto each of the N cores. Initially, $st[i] = nil\ \forall i \in \{1, 2, ..., N\}$.
- $dcb\_rq : \{1, 2, ..., N\} \to \{0, 1\}$, where $\forall i \in \{1, 2, ..., N\}$
$$dcb\_rq[i] = \begin{cases} 1, & \text{if core } i \text{ has a dispatcher for the domain in its run queue,} \\ 0, & \text{otherwise.} \end{cases}$$
Initially, $dcb\_rq[i] = 0\ \forall i \in \{1, 2, ..., N\}$.
- $res : \{1, 2, ..., N\} \to \{0, 1\}$, where $\forall i \in \{1, 2, ..., N\}$
$$res[i] = \begin{cases} 1, & \text{if the domain is using resources on core } i, \\ 0, & \text{otherwise.} \end{cases}$$
Initially, $res[i] = 0\ \forall i \in \{1, 2, ..., N\}$.
- $msg\_in : \{1, 2, ..., N\} \to \{nil,\ span,\ stop,\ allstop,\ free\_res\}$ is the message received by core i from the authoritative third party. Initially, $msg\_in[i] = nil\ \forall i \in \{1, 2, ..., N\}$.
- $msg\_out : \{1, 2, ..., N\} \to \{nil,\ run,\ stop\_pend,\ stop,\ cleanup\}$ is the message with which core i responds to the *msg_in* sent by the authoritative third party. Initially, $msg\_out[i] = nil\ \forall i \in \{1, 2, ..., N\}$.
- $old\_st : \{1, 2, ..., N\} \to \{nil,\ run,\ stop\_pend,\ stop\}$ is the previous state, *i.e.* the state that every core *transitions from* upon receiving a message. Initially, $old\_st[i] = nil\ \forall i \in \{1, 2, ..., N\}$.
- $turn \in \{0, 1, ..., N\}$ denotes the entity which acts in the current round of the simulation, where $\forall i \in \{1, 2, ..., N\}$ core i acts according to the value of $msg\_in[i]$ and $st[i]$ when $turn = i$. The value $turn = 0$ is reserved for the authoritative third party.
- $last\_proc \in \{0, 1, ..., N\}$ denotes the entity which acted in the previous round of the simulation.
- $domain\_state \in \{nil,\ run,\ stop\_pend,\ stop,\ cleanup\}$ is the global domain state, aggregated from $st[i]$ for $i \in \{1, 2, ..., N\}$. Initially, $domain\_state = nil$.

4.4.2 The Manager process

The PlusCal algorithm first defines the *Manager* process, which corresponds to the authoritative third party sending messages to cores $1, 2, ..., N$ instructing them what to do with the domain under modeling. The manager acts when the variable *turn* is set to 0. It infers the local state $st[i]$ of every core i based on the response $msg\_out[i]$ which the core provided to the manager's last message. Based on the inferred $st[i]$, the manager sends a new message to every core i triggering a transition to the next state as per the machine presented in Figure 4.1. Although the process loops indefinitely, once it learns that the domain has reached state *cleanup*, it stops sending messages about it. PlusCal pseudo-code for the process is given in algorithm 1.

```
1  while true do
2      await turn = 0;
3      with i ∈ {1, 2, ..., N} do
4          if msg_out[i] = nil ∨ msg_out[i] = run then
5              with action ∈ {span, stop} do
6                  msg_in[i] := action;
7              end
8          else if msg_out[i] = stop_pend then
9              if ∀i ∈ {1, 2, ..., N}: msg_out[i] = stop_pend ∨ msg_out[i] = stop then
10                 msg_in[i] := allstop;
11             end
12         else
13             if ∀i ∈ {1, 2, ..., N}: msg_out[i] = stop ∨ msg_out[i] = cleanup then
14                 msg_in[i] := free_res;
15             end
16         end
17     end
18     if ∀i ∈ {1, 2, ..., N}: msg_out[i] = nil ∨ msg_out[i] = run then
19         if ∃i ∈ {1, 2, ..., N}: msg_out[i] = run then
20             domain_state := run;
21         end
22     else
23         if ∀i ∈ {1, 2, ..., N}: msg_out[i] = cleanup then
24             domain_state := cleanup;
25         else if ∃i ∈ {1, 2, ..., N}: msg_out[i] = stop then
26             domain_state := stop;
27         else
28             domain_state := stop_pend;
29         end
30     end
31     last_proc := 0;
32     turn := 1;
33 end
```

Algorithm 1: The PlusCal manager process. The **await** instruction on line 2 blocks the process until $turn = 0$. The **with** instruction on lines 3 and 5 causes the simulation to run a separate step for each value specified as parameter. Lines 3-17 infer the local state for every core i from the message it last responded to with $msg\_out[i]$; based on this state, the manager will instruct the core to transition to the next state. Lines 18-30 then aggregate the individual core states into the global domain state.

4.4.3 The Worker process

Similarly to the Manager process, we define a *Worker* process which has N instances, each corresponding to a core the domain can run on. Every worker i acts when the variable *turn* is set to i: it reads the message $msg\_in[i]$ received from the manager and its own state $st[i]$, it decides on the next state to transition to and it communicates this state via $msg\_out[i]$ back to the manager.

The Worker process is described in algorithm 2.

```
1  while true do
2      await turn = self;
3      old_st[self] := st[self];
4      if msg_in[self] = span then
5          if st[self] = nil ∨ st[self] = run then
6              st[self] := run;
7              dcb_rq[self] := 1;
8              res[self] := 1;
9              msg_out[self] := run;
10         end
11     else if msg_in[self] = stop then
12         if st[self] = nil ∨ st[self] = run then
13             st[self] := stop_pend;
14             dcb_rq[self] := 0;
15             msg_out[self] := stop_pend;
16         end
17     else if msg_in[self] = allstop then
18         if st[self] = stop_pend then
19             st[self] := stop;
20             msg_out[self] := stop;
21         end
22     else
23         if st[self] = stop then
24             st[self] := cleanup;
25             res[self] := 0;
26             msg_out[self] = cleanup;
27         end
28     end
29     last_proc := self;
30     turn := (self + 1)mod(N + 1);
31 end
```

Algorithm 2: The PlusCal worker process. An instance of the algorithm is run for every worker i for $i \in \{1, 2, ..., N\}$. The variable *self* denotes the index of the currently running worker.

4.4.4 Specifying pre- and post-conditions

The PlusCal code was compiled into a TLA+ specification on top of which several *invariants* were defined. The invariants formally restated the pre- and post-conditions mentioned in Table 4.1. Model checking was run over them in order to guarantee that the conditions hold. We continue by listing the invariant formulae and explaining them and why they hold:

State nil

1. Pre-condition:

$$(last\_proc > 0 \land st[last\_proc] = nil) \Rightarrow msg\_in[last\_proc] = nil,$$

reading *if the last process to act is a worker and its local state is nil, then it must have not received any (non-nil) message.* Therefore, the right-hand side is a *necessary* condition for the left-hand side.

2. Post-condition:

$$(last\_proc > 0 \land st[last\_proc] = nil) \Rightarrow dcb\_rq[last\_proc] = 0,$$

reading *if the last process to act is a worker and its local state is nil, then it has no dispatcher on its run queue.* Therefore, the left-hand side is a *sufficient* condition for the right-hand

side. Of course, if the left-hand side member was always *false*, the implication would always be *true*. To formally check that the left-hand side does actually happen, an additional check was performed, presented below:

3. Validation:
$$(last\_proc > 0 \wedge st[last\_proc] = nil) \Rightarrow FALSE,$$
the failure of which proves that the left-hand side does indeed happen, thus the pre- and post-conditions for state *nil* hold for every individual core. Finally, let us now aggregate the individual core states into the domain state:

4. Aggregate domain state:
$$last\_proc = 0 \Rightarrow (domain\_state = nil \Leftrightarrow \forall i \in 1..N : st[i] = nil),$$
reading *if the last process to act is the manager, then the domain is in state nil iff all cores are in state nil*. The left-hand side is required here because the domain state will only be accurately updated straight after the manager acts.

State run

1. Pre-condition:
$$\begin{aligned}(last\_proc > 0 \wedge st[last\_proc] = run) \Rightarrow \quad & msg\_in[last\_proc] = span \\ & \wedge (old\_st[last\_proc] = nil \vee old\_st[last\_proc] = run),\end{aligned}$$
reading *if the last process was a worker and its state is run, then it must have received a span message and its previous state must have been either nil or run*; the right-hand side is a *necessary* condition for the left-hand side.

2. Post-condition:
$$(last\_proc > 0 \wedge st[last\_proc] = run) \Rightarrow dcb\_rq[last\_proc] = 1,$$
reading *if the last process was a worker and its state is run, then its dispatcher is on the run queue*; the left-hand side is a *sufficient* condition for the right-hand side.

3. Validation:
$$(last\_proc > 0 \wedge st[last\_proc] = run) \Rightarrow FALSE$$

4. Aggregate domain state:
$$\begin{aligned}last\_proc = 0 \Rightarrow (domain\_state = run \Leftrightarrow \quad & ((\exists i \in 1..N : st[i] = run) \\ & \wedge (\forall i \in 1..N : st[i] = nil \vee st[i] = run))),\end{aligned}$$
reading *if the last process was the manager, then the domain is in state run iff at least one core is in state run and all other cores are in state nil*.

State stop pending

1. Pre-condition:
$$(last\_proc > 0 \wedge st[last\_proc] = stop\_pend) \Rightarrow msg\_in[last\_proc] = stop,$$
reading *if the last process was a worker and its state is stop pending, then it must have received a stop message*; the right-hand side is a *necessary* condition for the left-hand side.

2. Post-condition:
$$(last\_proc > 0 \wedge st[last\_proc] = stop\_pend) \Rightarrow dcb\_rq[last\_proc] = 0,$$
reading *if the last process was a worker and its state is stop pending, then its dispatcher is not on the run queue*; the left-hand side is a *sufficient* condition for the right-hand side.

3. Validation:
$$(last\_proc > 0 \wedge st[last\_proc] = stop\_pend) \Rightarrow FALSE$$

4. Aggregate domain state:
$$last\_proc = 0 \Rightarrow (domain\_state = stop\_pend \Leftrightarrow \quad (\exists i \in 1..N : st[i] = stop\_pend) \\ \wedge \neg(\exists i \in 1..N : st[i] = stop)),$$
reading *if the last process was the manager, then the domain is in state stop pending iff at least one core is in state stop pending and no core has reached state stop.* The statement follows from the requirement of state *stop* that all cores reach *stop pending.*

State stop

1. Pre-condition:
$$(\exists i \in 1..N\ st[i] = stop) \Rightarrow (\forall i \in 1..N\ \neg(st[i] = nil \vee st[i] = run)),$$
reading *if there is a worker in state stop, then all workers are neither in state nil nor run*; the right-hand side is a *necessary* condition for the left-hand side. The right-hand side is equivalent to saying that all workers are in one of three states: *stop pending*, *stop* or *cleanup.* The reason for including state *cleanup* too is that the manager sends the *free resources* message when all workers have reached state *stop*, hence if a worker is in state *stop*, some others might have already transitioned to *cleanup.*

2. Post-condition:
$$(\exists i \in 1..N\ st[i] = stop) \Rightarrow (\forall i \in 1..N\ dcb\_rq[i] = 0),$$
reading *if there is a worker in state stop, then no workers have dispatchers on their run queues*; the left-hand side is a *sufficient* condition for the right-hand side.

3. Validation:
$$(\exists i \in 1..N\ st[i] = stop) \Rightarrow FALSE$$

4. Aggregate domain state:
$$last\_proc = 0 \Rightarrow (domain\_state = stop \Leftrightarrow (\exists i \in 1..N : st[i] = stop)),$$
reading *if the last process was the manager, then the domain is in state stop iff at least one core is in state stop.* Note that one core's being in state *stop* is sufficient for the whole domain to be in state *stop* since any given core will only enter *stop* when all cores have reached *stop pending*, *i.e.* all dispatchers have been dequeued.

State cleanup

1. Pre-condition:
$$(\exists i \in 1..N\ st[i] = cleanup) \Rightarrow (\forall i \in 1..N\ st[i] = stop \vee st[i] = cleanup),$$
reading *if there is a worker in state cleanup, then all workers are either in state stop or cleanup*; the right-hand side is a *necessary* condition for the left-hand side.

2. Post-condition:
$$(\forall i \in 1..N\ st[i] = cleanup) \Rightarrow (\forall i \in 1..N\ res[i] = 0),$$
reading *if all workers are in state cleanup, then the domain is not using resources on any core anymore*; the left-hand side is a *sufficient* condition for the right-hand side.

3. Validation:
$$(\forall i \in 1..N\ st[i] = cleanup) \Rightarrow FALSE$$

4. Aggregate domain state:
$$last\_proc = 0 \Rightarrow (domain\_state = cleanup \Leftrightarrow (\forall i \in 1..N : st[i] = cleanup)),$$
reading *if the last process was the manager, then the domain is in state cleanup iff all cores are in state cleanup.*

4.4.5 Specifying the race guard

The transition from *nil* to *stop pending* meant to prevent rogue domains from outracing the killing process was also formally written in TLA+. Specifically, we will analyze the following statement: *"a core will eventually leave state nil without ever adding a dispatcher to its run queue iff it eventually transitions from nil to stop pending through a stop message"*. Since it might be hard to grasp at first, let us isolate each member of the equivalence:

1. *a core will eventually leave state nil without ever adding a dispatcher to its run queue* means that no matter what messages the core receives, it will never run a dispatcher for the domain. In other words, the core will discard *span* messages from the malicious domain. In TLA+ notation and for a fixed core i, the statement is a conjunction between the formulae $\Diamond\neg(st[i] = nil)$ (eventually, the state will not be *nil*) and $\Box(dcb\_rq[i] = 0)$ (globally, no dispatcher is in the run queue), yielding:

$$\Diamond\neg(st[i] = nil) \wedge \Box(dcb\_rq[i] = 0) \tag{4.1}$$

2. *a core eventually transitions from nil to stop pending through a stop message* describes the state machine transition and can be written in TLA+ using the variables $old\_st[i]$, $st[i]$ and $msg\_in[i]$ as:

$$\Diamond(old\_st[i] = nil \wedge st[i] = stop\_pend \wedge msg\_in[i] = stop) \tag{4.2}$$

From Equation 4.1 and Equation 4.2 we derive the race guard formula:

$$\begin{aligned} \forall i \in 1..N : \quad & \Diamond\neg(st[i] = nil) \wedge \Box(dcb\_rq[i] = 0) \\ & \Leftrightarrow \Diamond(old\_st[i] = nil \wedge st[i] = stop\_pend \wedge msg\_in[i] = stop) \end{aligned} \tag{4.3}$$

Equation 4.3 can be checked with TLC by breaking the equivalence into a double implication, namely:

$$\begin{aligned} \forall i \in 1..N : \quad & \Diamond\neg(st[i] = nil) \wedge \Box(dcb\_rq[i] = 0) \\ & \Rightarrow \Diamond(old\_st[i] = nil \wedge st[i] = stop\_pend \wedge msg\_in[i] = stop) \end{aligned} \tag{4.4}$$

and:

$$\begin{aligned} \forall i \in 1..N : \quad & \Diamond(old\_st[i] = nil \wedge st[i] = stop\_pend \wedge msg\_in[i] = stop) \\ & \Rightarrow \Diamond\neg(st[i] = nil) \wedge \Box(dcb\_rq[i] = 0) \end{aligned} \tag{4.5}$$

Lastly, a sanity check is performed to ensure that the left-hand side members are not always *false*. For Equation 4.4 we check for the failure of:

$$\forall i \in 1..N : \Diamond\neg(st[i] = nil) \wedge \Box(dcb\_rq[i] = 0) \Rightarrow FALSE$$

and for Equation 4.5:

$$\forall i \in 1..N : \Diamond(old\_st[i] = nil \wedge st[i] = stop\_pend \wedge msg\_in[i] = stop) \Rightarrow FALSE$$

Chapter 5

System Design

This chapter focuses on building a process management subsystem for Barrelfish based on the formal model presented in chapter 4. We describe the system components, their interface and interaction. Design decisions and their implications are motivated in terms of the formal model. An authentication protocol between the principals in the system is devised and formalized using BAN logic.

5.1 Functional requirements

Firstly, let us look back at the domain definition and core properties established by the model in chapter 4. Applying those properties in the context of a Barrelfish subsystem yields the requirements listed in Table 5.1.

Moreover, based on the domain and dispatcher operations formulated by the model, the process management subsystem should provide means for:

- Creating a domain, which produces a new uniquely identifiable capability. It suffices to create one dispatcher (the main one), which should be given a program to run.
- Creating a new dispatcher for a domain, *i.e. spanning*. The new dispatcher will not run a new program, but persist only to serve the main dispatcher by providing capabilities using the shared virtual address space.
- Destroying a dispatcher, which requires performing *at least* two steps:
    1. Revoking and destroying the DCB, thereby de-scheduling it;

Domain model property	Design requirement
Domains are uniquely identifiable.	Every domain instance in the system should be identified by a *unique capability*.
Any domain has at least the *root dispatcher*; creating a domain amounts to creating its root dispatcher.	The domain capability should be assigned no sooner than the first *dispatcher control block (DCB)* is created.
Any dispatcher belongs to exactly one domain.	Spanning to a new core needs to link the new dispatcher to exactly one domain capability.
Any domain stops running when its root dispatcher stops running.	When the main dispatcher *exits* or *is killed*, all dispatchers belonging to the domain must be dequeued from the run queues on all cores.

Table 5.1: Process management requirements based on the model properties presented in chapter 4.

2. Revoking and destroying the dispatcher's cspace, releasing all capabilities it references and thus freeing all memory it privately holds.

- Destroying a domain, which implies destroying all its dispatchers.

In addition, inspired from UNIX process management, the system should be able to achieve the following:

- Allowing domains to voluntarily *exit* execution, *e.g.* when they reach the end of their `main()` function.
- Allowing domains to *wait* for the termination of other domains.
- Subject to certain restrictions, allowing domains to *kill* other domains.

5.1.1 Security implications of killing a domain

Killing domains is a sensitive matter due to its security implications. The process management system needs to ensure that domains cannot kill arbitrary domains without having the *right* to do so. In UNIX, this is controlled through user credentials by means of *sessions*, namely processes can kill other processes if they belong in the same session. Intuitively, this approach is based on the assumption that it is fine for a user to kill their own processes, as long as they do not interfere with those of other users.

Barrelfish possesses a similar concept of sessions, denoting a hierarchy of domains started by the same terminal shell. While the process management subsystem could adopt and adapt sessions to control domain operations, we have chosen not to pursue this path for the scope of this thesis for two reasons. Firstly, in Barrelfish sessions are more closely dependent on the TTY subsystem. Sessions are created by the program *angler* specifically for launching the *fish* shell. Consequently, they do not hold meaning outside the scope of a TTY shell (for example, domains started by the monitor have no concept of session).

Secondly, it has been established that domains need to be uniquely identifiable across the system and that a suitable way to achieve this consists in using capabilities. We can hence devise a different kill protocol: a domain can kill *any domain for which it has a capability.* For a given domain, the set of domains which can kill it is thus more tightly under its supervision, since it can choose whom to share its capability with. This approach provides more control over a domain's descendants compared to UNIX, as a domain will not be killable by domains it creates. However, a domain will be granted identifying capabilities for all domains which it spawns.

We can now state a partial answer to the thesis questions defined in chapter 1: *a domain can be killed by the domain that spawned it, as well as any other domain that holds its identifying capability.*

5.2 Implementing as a service versus as a library

The next step towards designing a process management solution consists in choosing the most suitable implementation manner. In terms of interfacing with the user, two paradigms were of interest: process management *as a service* or *as a library.* We continue by discussing the advantages and disadvantages of each approach.

5.2.1 Process management as a service

Among the benefits provided by a service-based solution, we list:

- The domain or domains implementing the service can act as the trusted third party sending domain-related messages to all cores in the system, as assumed by the model in chapter 4.
- The domain or domains implementing the service can control domain-related operations by requiring clients to provide capabilities with their requests (*e.g.* for killing other domains).

- Spawning is currently offered as a service; a process management server could leverage this by acting as an intermediary between spawnd and its clients. The advantage of such an interposition is twofold: firstly, the process manager can assess whether a client is allowed to be served by a specific spawnd instance, which is something required when domains are to be restricted from spanning to new cores as per the race problem presented in chapter 4. Additionally, registering spawnd instances with the process manager eliminates their need to register with the nameserver, meaning they cannot be reached by any domain in the system following a simple lookup. If spawnd could be reached following a name lookup, then it would need to personally inquire about the client domain and establish whether its request is valid (*e.g.* if the domain is not currently being killed elsewhere in the system). Such an inquiry could be made either by asking all other spawnds if they believe the domain should be served, or by reading from some authorized location. The former option would be computationally expensive due to the high number of messages, while the latter would be more cumbersome than having the process manager simply forward valid requests to spawnd.

- The process management server or servers can act as storage for domain metadata. For a given domain, such metadata includes its global state and the cores it runs on. This information can be useful for resource accounting, as well as facilitating an implementation of *wait*.

- The process manager can ensure that cleanup is performed after domains are stopped. In addition, it can hold a reference to the identifying capability of every domain it creates (or intermediates the creation of), which can be used to stop the domain prematurely and clean up after it should the system ever demand it.

Conversely, implementing as a service could have the following shortcomings:

- The *consistency-availability-partitioning (CAP)* problem. On one hand, if there is only one server implementing the process management service in the system, then some *availability* concerns need be addressed, such as *how many different clients it can serve at once* and *what happens if the server domain exits unexpectedly*. On the other hand, if the system runs multiple instances of the process manager, then a coherence protocol will be necessary to guarantee *consistency*. Lastly there is the problem of *partitioning*: if there are multiple servers then how do they *relate* to each other, say, *if one or more of them should fail?*

- A server implementing the process management interface would be an essential principal in the system, therefore its identity needs to be verifiable. The system should hence provide an *authentication protocol* to ensure that the process manager is not impersonated by malicious domains.

5.2.2 Process management as a library

The advantages of a purely library-based approach include:

- If domains did not need to engage in RPCs for domain-related operations, then availability might not be a problem (ideally, any domain could call library functions independently of other domains). The problem of partitioning would not apply either, since there would be no network of process management servers to partition in the first place.

- Library calls can still include capability checks even when there is no server dedicated to sanitize them.

Lastly, the library-based solution would have the following disadvantages:

- Consistency would be difficult to achieve, since library calls do not implicitly involve any sentient entity that could track the state of different domains (like a server would). For example, if a domain wished to run a dispatcher on a new core, in order to make sure it can proceed to do so the library would need to read the domain state from somewhere, such as the SKB. However, involving the SKB incurs communication overhead and potentially even availability problems.

- There would be no trusted authority to hold identifying capabilities for spawned domains. That is, the *spawner* would be the one responsible for killing the *spawnee* and cleaning up after it (or a UNIX-like domain hierarchy could be established for killing and cleanup). This would make it easier for rogue domains to conceal their spawnees' capabilities thereby making them difficult to kill. A potential solution consists in having spawnd hold identifying capabilities so that it could be instructed by the system to kill specific domains when deemed necessary.

Weighing in the benefits and shortcomings of each option, the service approach was deemed more suitable for the scope of this thesis. The remainder of this chapter will describe how the resulting process management service leverages the advantages presented and attempts to circumvent the disadvantages.

5.3 Survey of service implementation options

With the process management as-a-service concept in mind, the next step consisted in narrowing down the available service implementation options. Two main overarching questions motivated the research presented in this section: *where should the global domain state be stored?* and *what are the implications with respect to the consistency-availability-partitioning problem?*

5.3.1 Single dedicated server

The first solution consists in implementing the process management service through one single dedicated server (*i.e.* in a single domain). In this context, *dedicated* means that the server should implement one single interface and that interface should only be implemented by this server. Such a solution is appealing because it is the simplest and least cumbersome, making it the most suitable for testing the formal model devised in chapter 4. The global domain state question can be answered by using the server domain's address space as storage. However, there are two main drawbacks:

1. The single server is a single point of failure. This is particularly troublesome if the process management service is interposed between clients and the spawn service, since if the server domain fails no user space entity will be able to perform domain-related operations.

2. The server needs to be given some degree of *privilege*, *i.e.* trust with regards to managing domain state for all (or most) domains in the system. Most importantly, it needs to be trusted to properly clean up or initiate the cleanup of domain resources.

The first point could be circumvented to some extent if the service is carefully designed so that it could eventually scale out to multiple instances. The second point could be addressed by leveraging on the Barrelfish capability system so that the service contract reads along the lines of *you can do this if you hold the capability for it*, rather than *you can do this if I decide that you can.*

5.3.2 Multiple dedicated servers

The previous approach can be naturally extended by distributing the process management service to multiple servers running on different cores. On one hand, this solves the single point of failure problem and improves service availability. On the other though, a distributed service poses the following questions:

- Where do the servers store the global domain state?
- How should they achieve consistency with respect to the global domain state?
- How should they perform load balancing between multiple clients?

A primitive solution would rely on storing domain state for all domains every process management server. Servers could engage in an all-to-all message passing protocol: whenever the state of a domain is modified on a server, that server sends a message with the updated state to every other server. The receiver of such a message would update its version of the domain state, thereby achieving consistency. Of course, such a solution is undesirably complex in terms of messages exchanged. In addition, it does not tackle the load balancing problem.

A more elegant solution could be built on top of a leader election protocol. Assuming the process management servers can elect a leader, the leader can provide storage for the global domain state either in its own address space, or by delegating some other entity which it trusts. Load balancing can also be performed by the leader, either statically (*e.g.* round-robin) or dynamically (*e.g.* by choosing the server with the fewest requests assigned). This way, both the availability and consistency problems would be addressed. Tolerance to partitioning can then be achieved by designing a heartbeat protocol, where servers occasionally check if other servers are still online: if a non-leader server fails, the leader can remove it from its load balancing policy; if the leader fails, the other servers can run a new round of the election algorithm to designate a new leader.

5.3.3 Implementing in spawnd

An alternative to the dedicated server approach is having the existing spawnd servers implement the process management service interface in addition to the spawn interface. As Barrelfish runs one spawnd instance per core, this strategy could also benefit from a leader election protocol for load balancing and storing the global domain state.

The advantage of this solution amounts to potentially fewer message passes being necessary for domain-related operations. Specifically, client requests would not need to go through a dedicated process management gateway before reaching spawnd servers, as these would act as gateway themselves.

The disadvantages concentrate around two main ideas: firstly, every spawnd instance would have to perform considerably more work by itself – it might be desirable to offload the extra process management tasks to a separate domain. Secondly, the question of *how many spawnd servers should actually implement the process management interface* arises. It might not be necessary that all of them do: for a system running N cores it could be experimentally determined that $K < N$ process management instances can efficiently handle requests coming from all N cores. However, the follow-up question is *if not all spawnd instances offer process management services, then which of them do?* A subset of cores could be chosen, for example, based on how much work every core's spawnd – or even the core as a whole – does.

5.3.4 Implementing in the monitor

Based on the idea elaborated above, a similar solution can be devised where the process management interface is implemented by the monitors. This approach differs from the spawnd-based one in that:

1. The monitors already naturally communicate with each other via the URPC-based *intermon channels*. In practice, this could be an advantage because the monitors on two different cores already need to communicate when arbitrary user space domains on those cores want to establish a channel.

2. Instead of adding to the workload of spawnd, the monitor would now have to perform the additional process management tasks. Such a change is arguably even less desirable, as the monitor is perhaps the most critical user space domain in Barrelfish, offering the richest interface already.

5.3.5 Other approaches

Two other notable perspectives on storing global domain state have been considered for this thesis. The first one would require every domain to hold its state in its *first dispatcher*. For example, the DCB can be extended to encapsulate the overarching domain's state. The main difference from

all of the previously mentioned options is that whichever server domain implements the process management interface, that domain only forwards messages between the client and the spawn backend, without tracking the client's state. Instead, computing the state is offloaded to the client domain, which needs to achieve coherence between its dispatchers. On one hand, this results in a lighter process management interface. However, concerns arise when domains are trusted with managing their own state, as a malicious domain could conceal running dispatchers and prevent them from being killed, or pretend to perform cleanup while still leaking system resources.

Building on top of the previous idea, the Barrelfish capability mechanisms can be leveraged to achieve domain state coherence, instead of asking the domain to cohere with all its dispatchers. An example of such an existing mechanism is `cap_revoke`, which deletes all copies of a capability in the system but the one held by the caller. In theory, the capability system could be extended to ensure the propagation of domain state updates, as well as killing and cleanup. For instance, when the first DCB of some domain is destroyed, the capability system could equate that DCB to all other DCBs in the system belonging to the same domain and destroy them all, thereby stopping the domain on all cores. The solution is conceptually elegant, although in practice it would likely result in extra functionality added both to the monitor and the kernel. This might be more difficult to implement and debug. In addition, having no dedicated entity to track domain state means the state machine presented in chapter 4 can be more difficult to implement and prove correct in practice.

Comparing what the different options presented throughout this section have to offer, the *multiple dedicated servers* approach was deemed most suitable in theory. However, for the first iteration of the process management subsystem presented in this thesis, we decided to implement the *single dedicated server* solution instead. The motivation was that the extra layer of complexity added by the multiple-server coherence protocol would distract from the goal of implementing the model devised in chapter 4, given the research time frame for the thesis. The single-server option can eventually be out-scaled to the multiple-server one, however we will not focus on that for the remainder of this chapter.

Summing up the design choices made so far, the process management subsystem will:

- provide functionality for creating new domains (spawning), creating new dispatchers for existing domains (spanning), killing domains, exiting voluntarily and waiting for other domains to finish execution;
- be offered as a service interposed between clients and the spawnd backend;
- store state for domains it manages in a single server, which will be run on the bootstrap core for convenience and without loss of generality.

5.4 A matter of authentication

We have stated that the process manager will forward requests to spawnd servers so that they can avoid publicly registering with the nameservice. A bidirectional authentication problem follows:

1. How can the process manager discover all spawnd servers?
2. How can spawnd tell when requests are coming from the process manager?

5.4.1 The spawnd discovery protocol

To answer the first question, we present an authentication protocol that allows the process manager to discover spawnd servers running in the system. In this context, *discovering a spawnd* amounts to discovering its iref.

For simplicity, let us first investigate the case where the system only runs on one core – that is, everything is on core 0. The protocol makes the following assumptions (the domain X on core Y will be referred to as $X.Y$):

1. the monitor on core 0 starts proc_mgmt.0 before it starts spawnd.0;

2. monitor.0 provides proc_mgmt.0 with an LMP endpoint to itself; both domains know that proc_mgmt.0 can thus send messages to monitor.0;

3. monitor.0 provides spawnd.0 with an LMP endpoint to itself; both domains know that spawnd.0 can thus send messages to monitor.0;

4. proc_mgmt.0 trusts that everything monitor.0 says is true; in particular, it trusts that monitor.0 can provide it with the correct iref for spawnd.0.

To satisfy assumption 1, the monitor simply chooses to launch the process manager before spawnd. Assumptions 2 and 3 hold because the monitor allocates a channel to itself for every domain it spawns. Assumption 4 is reasonable because user space domains generally trust the monitor. The protocol then executes the following steps:

1. At startup, the process manager allocates a special endpoint for the monitor, then sends the endpoint capability to the monitor using the monitor endpoint it has been created with. Consequently, any message received on this endpoint will be treated as if it is coming from the monitor.

2. The monitor establishes a channel to the process manager using the endpoint it received in step 1.

3. At startup, spawnd sends its iref to the monitor privately using the monitor endpoint it has been created with.

4. The monitor receives the iref from spawnd and sends it to the process manager using the channel it created in step 2.

5. The process manager receives the iref on the endpoint it allocated for the monitor in step 1. The process manager knows the message is coming from the monitor and, since it trusts the monitor on the matter, it now believes it holds the correct iref for spawnd.0.

The result at the end of the protocol run is that the process manager holds an iref to something which claims to be spawnd. Since the process manager trusts the monitor, it believes that the server the iref is for is indeed spawnd, hence it proceeds to open a client connection to it. In other words, spawnd has successfully authenticated with the process manager. The protocol is illustrated in Figure 5.1.

5.4.2 Multi-core extension

To extend the discovery protocol to a multi-core scenario, let us identify the step which makes it inherently single-core. So far, after monitor.0 launched spawnd.0, it sent its iref to proc_mgmt.0 through the dedicated LMP endpoint the latter had allocated. Suppose the system boots another core; let it be core 1. Like in the case of core 0, monitor.1 is responsible for starting spawnd.1 and providing it with a monitor binding. However, monitor.1 can not use a special LMP endpoint to communicate the iref for spawnd.1 to proc_mgmt.0, since the two domains run on different cores.

In order to circumvent this, we leverage the intermon channel between monitor.1 and monitor.0, together with the fact that monitor.0 is already authenticated with proc_mgmt.0 using the dedicated LMP endpoint. All that needs to be done is send the iref for spawnd.1 through the intermon channel, resulting in the following steps:

1. On core 0, proc_mgmt.0 and monitor.0 authenticate like in the single-core case.

2. At startup, spawnd.1 sends its iref to monitor.1 privately using the monitor endpoint it has been created with.

3. The iref is received by monitor.1 and forwarded to monitor.0 using the intermon channel.

4. The iref is further received by monitor.0, which sends it to proc_mgmt.0 through the channel created in step 1.

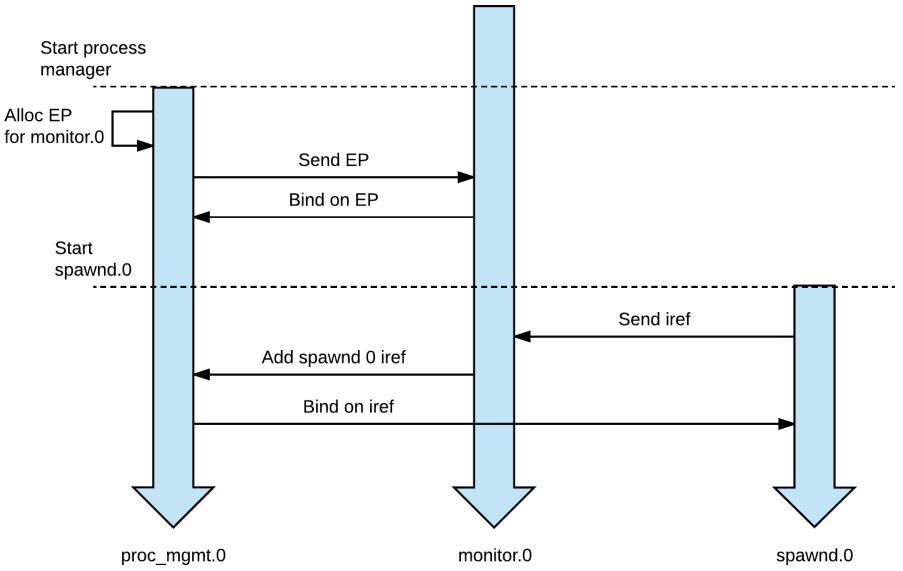


Figure 5.1: Single-core discovery protocol. Time flows vertically from top to bottom. Dotted lines mark the moments when the monitor spawns the process manager and spawnd respectively. At the end of the protocol run, the process manager holds a client channel to spawnd.

5. The process manager receives the iref and believes it is for spawnd.1, since it trusts monitor.0. The process manager hence opens a client connection to spawnd.1.

An additional assumption necessary for the extension is that *the monitors trust each other*, which is normally the case with all inter-core functionality in Barrelfish. The protocol extension is illustrated in Figure 5.2.

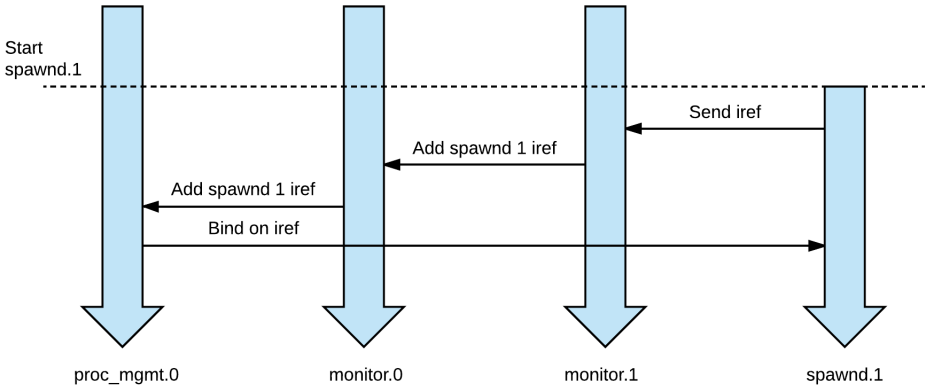


Figure 5.2: Multi-core extension for the discovery protocol. Time flows vertically from top to bottom. The dotted line marks the launching of spawnd.1, which is assumed to happen after monitor.0 and proc_mgmt.0 have already run the local authentication steps presented in Figure 5.1. At the end of the protocol run, the process manager holds a client channel for spawnd.1 in addition to that for spawnd.0.

5.4.3 Formalization

The discovery protocol can be formally stated using BAN logic. For brevity, we only give equations for the single-core version – the multi-core extension builds on top of it trivially.

We first bring two minor changes to the standard deduction rules presented in chapter 2, which cater better to the Barrelfish architecture:

1. Based on the shared key notation, we write that two dispatchers P and Q can engage in one-directional communication through an LMP or UMP channel as $P \xrightarrow{K} Q$, K is a $P \to Q$ channel. The message-meaning rule becomes

$$\frac{P \textbf{ believes } Q \xrightarrow{K} P, \quad P \textbf{ sees } \{X\}_K}{P \textbf{ believes } Q \textbf{ said } X},$$

   reading *if P believes it can receive messages from Q through the $Q \to P$ channel K and P sees message X on channel K, then P believes that Q said X.*

2. Since Barrelfish endpoints and capabilities in general do not have an implicit time-to-live, the formula $fresh(X)$ is tautological in the context of our protocol. Therefore, the nonce-verification rule becomes

$$\frac{P \textbf{ believes } Q \textbf{ said } X}{P \textbf{ believes } Q \textbf{ believes } X},$$

   reading *if P believes that Q said X, then P believes that Q believes X.*

Next, let us analyze the protocol assumptions in BAN logic notation. It has been established that *monitor.0 provides proc_mgmt.0 with an LMP endpoint to itself; both domains know that proc_mgmt.0 can thus send messages to monitor.0.* Writing P for the process manager, M for the monitor and K_{PM} for the spawnd-to-monitor binding, we obtain:

$$P \textbf{ believes } P \xrightarrow{K_{PM}} M \tag{5.1}$$

$$M \textbf{ believes } P \xrightarrow{K_{PM}} M \tag{5.2}$$

The next assumption was that *monitor.0 provides spawnd.0 with an LMP endpoint to itself; both domains know that spawnd.0 can thus send messages to monitor.0.* Writing S for spawnd and K_{SM} for the monitor binding that it holds, it follows that:

$$S \textbf{ believes } S \xrightarrow{K_{SM}} M \tag{5.3}$$

$$M \textbf{ believes } S \xrightarrow{K_{SM}} M \tag{5.4}$$

We have also said that *proc_mgmt.0 trusts that monitor.0 can provide it with the correct iref for spawnd.0.* Writing K_{PS} for the channel which the process manager can open on that iref[1] yields:

$$P \textbf{ believes } M \textbf{ controls } P \xrightarrow{K_{PS}} S \tag{5.5}$$

There are two more assumptions hidden in the protocol flow. Firstly, the monitor actually *expects* the process manager to provide it with a trusted endpoint for establishing a channel to communicate the iref for spawnd. Such a channel can be written as K_{MP}, yielding:

$$M \textbf{ believes } P \textbf{ controls } M \xrightarrow{K_{MP}} P \tag{5.6}$$

Secondly, the monitor also expects spawnd to provide it with its iref for the process manager. Similarly to Equation 5.6, we can write:

$$M \textbf{ believes } S \textbf{ controls } P \xrightarrow{K_{PS}} S \tag{5.7}$$

The first step of the protocol consists in the process manager's allocating an LMP endpoint for the monitor. Let the channel eventually established using that endpoint be K_{MP}. It follows that:

[1] An $A \to B$ LMP channel can be established using either an LMP endpoint or an iref for B. Therefore, in our formal notation both an endpoint and an iref equate to the same K_{AB} channel.

$$P \textbf{ believes } M \overset{K_{MP}}{\rightarrow} P \tag{5.8}$$

The process manager sends the newly allocated endpoint to the monitor through its monitor binding as per Equation 5.1, resulting in:

$$M \textbf{ sees } \{M \overset{K_{MP}}{\rightarrow} P\}_{K_{PM}} \tag{5.9}$$

Applying the message-meaning rule using Equation 5.9 and Equation 5.2 yields:

$$M \textbf{ believes } P \textbf{ said } M \overset{K_{MP}}{\rightarrow} P \tag{5.10}$$

Which, under the nonce-verification rule for $\mathit{fresh}(X) = \mathit{true}$, gives:

$$M \textbf{ believes } P \textbf{ believes } M \overset{K_{MP}}{\rightarrow} P \tag{5.11}$$

Applying the jurisdiction rule using Equation 5.11 and Equation 5.6, we have that:

$$M \textbf{ believes } M \overset{K_{MP}}{\rightarrow} P \tag{5.12}$$

Equation 5.12 states that the process manager is authenticated with the monitor through the channel built on top of the dedicated LMP endpoint. Together with Equation 5.8, we have so far formally proven that the process manager and the monitor believe they are talking to each other.

The next step amounts to spawnd's sending its iref to the monitor via its monitor binding, as per Equation 5.3:

$$M \textbf{ sees } \{P \overset{K_{PS}}{\rightarrow} S\}_{K_{SM}} \tag{5.13}$$

The message-meaning rule for Equation 5.13 and Equation 5.4 yields:

$$M \textbf{ believes } S \textbf{ said } P \overset{K_{PS}}{\rightarrow} S \tag{5.14}$$

Which, under nonce-verification, results in:

$$M \textbf{ believes } S \textbf{ believes } P \overset{K_{PS}}{\rightarrow} S \tag{5.15}$$

Applying the jurisdiction rule for Equation 5.15 and Equation 5.7, we have that:

$$M \textbf{ believes } P \overset{K_{PS}}{\rightarrow} S \tag{5.16}$$

Since the monitor now believes it has an iref which the process manager can use to send messages to spawnd, it can use the $M \overset{K_{MP}}{\rightarrow} P$ channel given by Equation 5.12 to inform the process manager of the iref:

$$P \textbf{ sees } \{P \overset{K_{PS}}{\rightarrow} S\}_{K_{MP}} \tag{5.17}$$

Using the message-meaning rule for Equation 5.17 and Equation 5.8, it follows that:

$$P \textbf{ believes } M \textbf{ said } P \overset{K_{PS}}{\rightarrow} S \tag{5.18}$$

Applying the nonce-verification rule, Equation 5.18 becomes:

$$P \textbf{ believes } M \textbf{ believes } P \overset{K_{PS}}{\rightarrow} S \tag{5.19}$$

Finally, the jurisdiction rule for Equation 5.19 and Equation 5.5 gives:

$$P \textbf{ believes } P \overset{K_{PS}}{\rightarrow} S \tag{5.20}$$

Hence the proof that the process manager has discovered an authentic spawnd server is complete. Appendix B gives an Isabelle[14][15] implementation of a set theory-based version of this proof.

5.4.4 Brute-force vulnerability

The fact that spawnd expects requests from the process manager on the iref which it passes to the monitor during the discovery protocol can be written as:

$$S \textbf{ believes } P \overset{K_{PS}}{\rightarrow} S \tag{5.21}$$

Together, Equation 5.20 and Equation 5.21 formally state that the process manager and spawnd believe they are talking to each other. However, in practice, there is a problem with this conclusion: K_{PS} denotes an iref, which is actually a *number*[2]. The security implication is that a malicious domain could theoretically brute-force attack the iref space to guess where spawnd is and establish a connection. Since spawnd trusts that the monitor will forward its iref to the process manager, when it sees an incoming connection request it will assume it is coming from the process manager. This would lead to the latter's being impersonated. Therefore, the discovery protocol does not guarantee that requests received by spawnd actually originate in the process manager – only that it is likely this is the case.

It is worth mentioning that the aforementioned problem does not apply to the authentication phase between the process manager and the monitor on core 0, because those principals use LMP endpoints to establish a connection. LMP endpoints are capabilities, thus they cannot be forged, however they cannot be used with UMP for multi-core discovery[3].

5.4.5 The ProcessManager capability

The persisting question is *how can spawnd tell when requests are coming from the process manager?* One solution consists in having the latter sign the messages it sends. A Barrelfish-friendly way to accomplish this is to have the process manager include a capability with its requests. Such a capability needs to uniquely identify an entity which equates either to the process manager or to a domain which the process manager delegates.

The capability can be of two types among the existing ones. The first is the ID capability, which encapsulates the index of the core where it was created, alongside a unique core-local index. It is guaranteed that every two ID capabilities in the system are different. However, since any domain can create ID capabilities at will, it would be impossible to identify the one denoting the process manager.

The second existing type that could be used is the endpoint capability. Specifically, spawnd could allocate a special endpoint for the process manager, assume that the process manager eventually gets hold of that endpoint and treat all subsequent requests received on that endpoint with the certainty they originate in the process manager. However, this scenario only holds when the interconnect driver between spawnd and the process manager is LMP. This is not always the case, as the process manager and spawnd need to communicate via UMP when they run on different cores.

All of the above stand as reason for introducing a new capability type, the *ProcessManager capability*. A single instance of this capability is given to the CPU driver on core 0 to init, where it is stored in the *task cnode* in slot `TASKCN_SLOT_PROCMNG`. When init.0 boots monitor.0, it copies the process manager capability into the latter's cspace. The monitor then starts the process manager domain on core 0 and passes on the capability. Thereafter, when spawnd receives a process management request, it inspects the enclosed capability's type: if it is ProcessManager, then the request is considered valid. The flow of the ProcessManager capability from the CPU driver to proc_mgmt.0 is illustrated in Figure 5.3. This design requires that init and the monitor can be trusted to deliver the capability to the process manager and only to the process manager, which is a fair assumption given they are both privileged domains.

[2] 32-bit unsigned, as of when this thesis was written.

[3] Connecting to servers running on different cores is exactly the problem irefs solve.

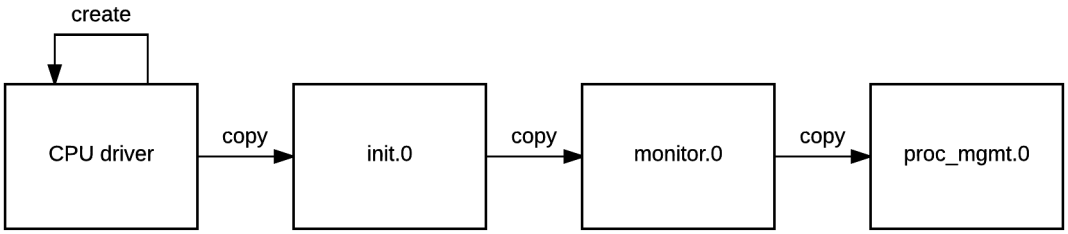


Figure 5.3: A single instance of the ProcessManager capability is created by the CPU driver on the bootstrap core. The capability is delivered to the process manager through init and the monitor. Slot `TASKCN_SLOT_PROCMNG` of the task cnode is used to store it.

Chapter 6

Implementation

Everything presented in the previous chapters helped consolidate the interface for the process management service. Since it was designed to act as middleware between clients and the spawnd backend, the process manager implements an interface for the former and one for the latter, as depicted in Figure 6.1.

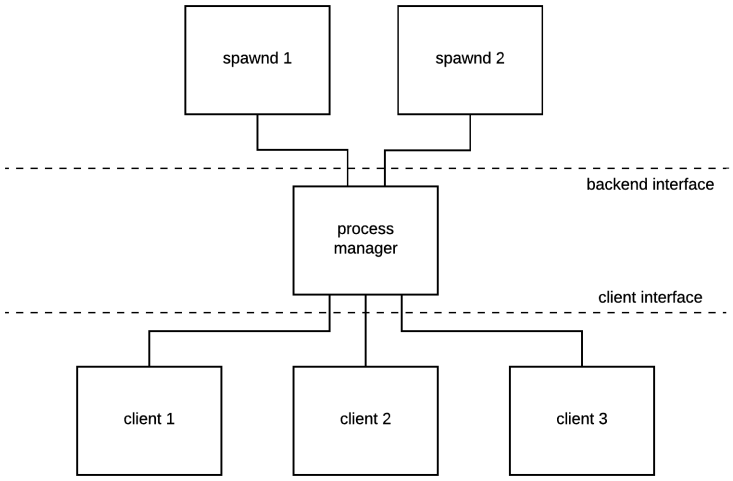


Figure 6.1: The process manager acts as middleware between outside clients and the spawnd backend.

6.1 The process management client interface

The client interface definition can be found in `if/proc_mgmt.if`. Due to the fact that the process manager expects special messages from the monitor for the spawnd discovery protocol, the client calls it exposes are separated in two categories: *general-purpose* and *monitor-only*.

6.1.1 The general-purpose interface

All general-purpose calls can be used by any domain which establishes a client connection to the process manager. All calls cause the client to block. Below, we list the API calls in the Flounder language used by Barrelfish.

The spawn call. The spawn call creates and runs a new domain. It expects similar arguments to the corresponding call offered by the old spawn service interface. It takes the target core ID, the program name, an argument and an environment buffers and flags as input parameters; it

returns an error code and a domain capability as output parameters. The key point is the output capability used to identify the newly spawned domain, which contrasts with the numerical identifier traditionally returned by the spawn interface. The motivation was that it is easier to guarantee a capability's system-wide uniqueness, as well as the fact that a numerical ID can be easily forged in a man-in-the-middle attack. The domain capability is also copied into the newly spawned domain's cspace, in slot `TASKCN_SLOT_DOMAINID` of the task cnode. Domain capabilities will be described in more detail in the following sections.

```
rpc spawn(in coreid core,
          in String path[2048],
          in char argvbuf[argvbytes, 2048],
          in char envbuf[envbytes, 2048],
          in uint8 flags,
          out errval err,
          out cap domain_cap);
```

The spawn with caps call. The spawn with caps call is identical to the plain spawn one with the exception of two input parameters: a capability for a cnode to be inherited by the spawnee and a capability for an argument cnode. These parameters exist for compliance with the `spawn_domain_with_caps` call exposed by the old spawn interface and are meant to be used by the spawner to provide the spawnee with additional capabilities.

```
rpc spawn_with_caps(in coreid core,
                    in String path[2048],
                    in char argvbuf[argvbytes, 2048],
                    in char envbuf[envbytes, 2048],
                    in cap inheritcn_cap,
                    in cap argcn_cap,
                    in uint8 flags,
                    out errval err,
                    out cap domain_cap);
```

The span call. The span call creates and runs a new dispatcher under the requesting domain. It expects a capability identifying the caller, the target core ID, a capability for the root of the caller's virtual address space, memory for the new dispatcher's frame and an error code. The first argument is used to associate the new dispatcher with the caller domain. The third argument is used to share the caller's virtual address space with the new dispatcher.

```
rpc span(in cap domain_cap, in coreid core, in cap vroot,
         in cap dispframe, out errval err);
```

The kill call. The kill call stops and cleans up after all dispatchers belonging to a domain identified by the capability passed as argument, returning an error code.

```
rpc kill(in cap domain_cap, out errval err);
```

The exit message. The exit message allows the domain identified by the capability to exit execution with the provided exit code. It is not implemented as an RPC, as the requester is not expected to run after the call completes.

```
message exit(cap domain_cap, uint8 status);
```

The wait call. The wait call blocks the requesting domain until the domain identified by the capability passed as argument has finished execution, returning an error code and the requested domain's exit status.

```
rpc wait(in cap domain_cap, out errval err, out uint8 status);
```

6.1.2 The monitor-only interface

In addition to the calls listed above, the monitor has access to the `add_spawnd` message:

```
message add_spawnd(coreid core, iref iref);
```

which expects the core where the new spawnd runs and its iref. The message is used to implement the discovery protocol and does not block the caller. In order to guarantee that only the monitor can trigger the discovery of a new spawnd, the process manager only assigns a handler for this type of message to the binding initiated on the monitor-dedicated endpoint which it allocates during the first step of the authentication protocol. If an `add_spawnd` message is received on any binding other than the monitor one, the process manager disregards it, doing nothing.

6.2 The Domain capability

The domain identifier returned by `spawn` and `spawn_with_caps` and required by the other general-purpose interface calls is implemented as a capability, motivation being that the said identifier must:

- uniquely reference exactly one domain;
- be non-forgeable (only the process manager can create it).

Following a train of reasoning similar to the one explaining the ProcessManager capability in the previous chapter, the domain capability could not be of any of the existing system types. Firstly, the capability could not be of type ID because any entity in the system can create arbitrary ID capabilities, potentially allowing domains to be impersonated. Secondly, if the capability was an endpoint, then it could uniquely reference exactly one domain by, say, retyping it from the domain's main dispatcher. However, any domain in the system would be able to retype such a capability if it held a reference to the originating dispatcher.

Instead, stemming from the restriction that the process manager should be the only one that can create domain identifiers, a new capability type was introduced: the *Domain* type, which can only be created by retyping from the ProcessManager capability. Similarly to the ID type, a Domain capability encapsulates the index of the core where it was created alongside a unique core-local index. Consequently, whenever the process manager receives a spawn request, it will assign the newly spawned domain a Domain capability, which will be copied into the spawnee's cspace and returned to the caller. That capability is required for any subsequent requests targeting the newly spawned domain.

6.2.1 Preallocating domain capabilities

A potential problem with this approach is that retyping capabilities is an expensive operation in so far as it requires switching to kernel space for checks that only the CPU driver is allowed to perform. Retyping the ProcessManager capability into a new Domain one every time a domain is to be spawned might hence be undesirable from the performance perspective. Instead, similarly to the slab and slot refill mechanisms in Barrelfish, the process manager preallocates a level 2 cnode and fills it with 256 capabilities of type Domain at startup. When subsequent spawn calls use up all the preallocated identifiers, a new level 2 cnode will be allocated and filled with 256 more capabilities. The process manager stores pointers to the domain capabilities in a list residing in its address space. The algorithm for allocating Domain capabilities is presented in algorithm 3 and implemented in `usr/proc_mgmt/domain.c`.

6.2.2 Identification and retrieval

In the Barrelfish user space, capabilities act like black boxes. A domain can hold a reference to a capability which it can show to the CPU driver to prove it is entitled to use the resource it denotes, however the actual capability is only held by the CPU driver. When the process manager is presented with a capability which is claimed to represent a domain, it needs to acquire two

```
1  new_cnode := create_l2();
2  retype(cap_procmng, new_cnode.0, 256);
3  for i from 0 to 256 do
4  |  cap_list_add(new_cnode.i);
5  end
6  while spawn_call do
7  |  if cap_list_freecount() = 0 then
8  |  |  new_cnode := create_l2();
9  |  |  retype(cap_procmng, new_cnode.0, 256);
10 |  |  for i from 0 to 256 do
11 |  |  |  cap_list_add(new_cnode.i);
12 |  |  end
13 |  end
14 |  domain_cap := cap_list_get_first();
   |  /* Do the rest of the spawning work.                                  */
15 end
```

Algorithm 3: Pseudo-code for the domain capability allocation algorithm. Lines 1-4 represent the initial preallocation phase. Lines 5-14 correspond to acting on a new spawn call. The variable cap_procmng holds the ProcessManager capability; new_cnode.i represents the capref in slot i of new_cnode. The resulting identifying capability for the new spawnee is denoted as domain_cap.

pieces of information about it: whether it is actually a capability of type Domain and if so, which exactly is the identified domain. This is done through the capability identifying RPC provided by the monitor:

```
rpc cap_identify(in cap cap, out errval err, out caprep caprep);
```

which takes a capability reference of type `struct capref` and returns an error code and a `struct capability` object. The latter is the internal representation of a capability in the kernel and contains a field marking the object type. For capabilities of type Domain, the encapsulated core number and core-local ID are also present in the returned capability object.

Once the domain's originating core and local ID have been inferred from the capability, the process manager needs to store this information in a manner which facilitates future identification. Specifically, when the process manager is subsequently presented with a reference to the same domain capability, it needs to associate it with the same domain structure. To this end, domain capabilities are hashed using the function:

$$h : \{0, 1, ..., l_{max}\} \times \{0, 1, ..., c_{max}\} \to \{0, 1, ..., 2^{64} - 1\}, \quad h(l, c) = (1 + l_{max}) \cdot l_d + c_d,$$

where l_{max} is the maximum value of a core-local id, c_{max} is the maximum core index, l_d is the domain's core-local id and c_d is the originating core. The core-local id field of a capability is of type `uint32_t`, therefore $l_{max} = 2^{32} - 1$; the index of a core is of type `uint8_t`, therefore $c_{max} = 2^8 - 1 = 255$.

In order to correctly map capabilities to the domains they represent, the function h as defined above needs to be injective, which we prove below:

Claim. $\forall (l_1, c_1) \neq (l_2, c_2) \in \{0, 1, ..., l_{max}\} \times \{0, 1, ..., c_{max}\} \;\; h(l_1, c_1) \neq h(l_2, c_2)$

Proof.

$$\begin{aligned}
&\text{Assume } \exists (l_1, c_1) \neq (l_2, c_2) \in \{0, 1, ..., l_{max}\} \times \{0, 1, ..., c_{max}\} \;\; s.t. \;\; h(l_1, c_1) = h(l_2, c_2) \\
&\Rightarrow (1 + l_{max}) \cdot l_1 + c_1 = (1 + l_{max}) \cdot l_2 + c_2 \\
&\Rightarrow (1 + l_{max}) \cdot l_1 + c_1 - c_2 = (1 + l_{max}) \cdot l_2
\end{aligned}$$

Case $c_1 \neq c_2$ $\Rightarrow c_1 - c_2 = k > 0$, assuming w.l.o.g. $c_1 > c_2$
$\Rightarrow (1 + l_{max}) \cdot l_1 + k = (1 + l_{max}) \cdot l_2$
$\Rightarrow (1 + l_{max}) \cdot l_1 + k \equiv (1 + l_{max}) \cdot l_2 \pmod{1 + l_{max}}$
$\Rightarrow k \equiv 0 \pmod{1 + l_{max}}$ **contradiction** since $k > 0$ and $k < 1 + l_{max}$

Case $c_1 = c_2$ $\Rightarrow (1 + l_{max}) \cdot l_1 = (1 + l_{max}) \cdot l_2$
$\Rightarrow l_1 = l_2$ **contradiction** since $(l_1, c_1) \neq (l_2, c_2)$ and $c_1 \neq c_2$

■

The key computed by the hash function h maps to a `struct domain_entry` object in a hashtable residing in the process manager's address space. The table uses chaining for solving conflicts and is instantiated with 6151 buckets. The `domain_entry` structure collects domain metadata such as the identifying capability, list of cores which the domain has dispatchers on, list of waiters and exit status. The complete definition can be found in `usr/proc_mgmt/domain.h` and is illustrated in Listing 6.1.

```
struct domain_entry {
  struct domain_cap_node *cap_node;
  enum domain_status status;

  struct spawnd_state *spawnds[MAX_COREID];
  coreid_t num_spawnds_running;
  coreid_t num_spawnds_resources;

  struct domain_waiter *waiters;

  uint8_t exit_status;
};
```

Listing 6.1: The domain entry structure, containing, in order: corresponding node in list of domain caps; current domain status (state as per the formal model); set of cores that run a dispatcher; number of cores that run a dispatcher; number of cores where resources are still in use; list of waiters; exit status.

6.3 The spawn service backend

After it has assigned a domain capability (in the case of spawn calls) or inspected the capability presented with to verify that the request is valid (for other API calls), the process manager sends a request to the spawn service backend implemented by a spawnd server on the target core or cores. The target core equates to the one the client passes if the request is a spawn or span one, or all cores which the domain has a dispatcher on if the request is of type kill or exit.

6.3.1 Interfacing spawnd with the process manager

The spawn service interface implemented by spawnd was extended for the purpose of communicating with the process manager. Several process manager-only requests were added, implemented asynchronously as messages, instead of RPCs. The need for asynchronicity derived from the fact that a single process manager server needs to overview as many spawnd servers as there are cores in the system. Consequently, if the process manager blocked waiting for a given spawnd to complete the backend call, then all cores would effectively be serialized with respect to domain-related operations. Although the process manager does not block during backend calls, spawnd does.

The extended spawn interface can be found in `if/spawn.if` and contains the messages described below. All process-manager only requests need to by signed with a capability of type

ProcessManager to enforce authenticity, as described in the previous chapter. In order for spawnd to assess whether the passed capability truly identifies the process manager, a `cap_identify` RPC is used to tell the capability type, similarly with the case of the process manager identifying requesting domains.

The spawn request. The spawn request expects the process manager capability, an identifying capability for the new domain, the program path, argument and environment buffers and flags. The first two arguments constitute the process manager's guarantee that the request is authentic and valid. The last four arguments are simply forwarded to spawnd as they have been received from the process management client. The request creates, sets up and runs a *main dispatcher* for the given domain. This is the backend correspondent of `rpc spawn` exposed by the client-side process management interface.

```
message spawn_request(cap procmng_cap,
                      cap domain_cap,
                      String path[2048],
                      char argvbuf[argvbytes, 2048],
                      char envbuf[envbytes, 2048],
                      uint8 flags);
```

The spawn with caps request. The spawn with caps request is similar to the simple spawn one, with the exception of the two additional capabilities for cnodes to be passed to the spawnee. It is the backend correspondent of `rpc spawn_with_caps` in the client-side interface.

```
message spawn_with_caps_request(cap procmng_cap,
                                cap domain_cap,
                                String path[2048],
                                char argvbuf[argvbytes, 2048],
                                char envbuf[envbytes, 2048],
                                cap inheritcn_cap,
                                cap argcn_cap,
                                uint8 flags);
```

The span request. The span request expects the process manager capability, an identifying capability for the domain to span, as well as the vspace root and dispatcher frame capabilities as passed to the process manager. The last two capabilities must be already allocated and set up. The request causes a new dispatcher to be run for the given core. It is the backend correspondent of `rpc span` in the client-side interface.

```
message span_request(cap procmng_cap, cap domain_cap, cap vroot,
                     cap dispframe);
```

The kill request. The kill request expects the process manager capability and an identifying capability for the victim domain. It removes the victim domain's local dispatcher from the core's run queue by revoking the DCB capref and then destroying it. It is the backend correspondent of both `rpc kill` and `message exit` in the client-side interface. Both client-side calls generate the same backend since the exit status is only stored in the process manager.

```
message kill_request(cap procmng_cap, cap domain_cap);
```

The cleanup request. The cleanup request expects the process manager capability and an identifying capability for the domain to clean up. It releases the given domain's capabilities by revoking its root cnode capref and then destroying it. It is meant to succeed the kill request.

```
message cleanup_request(cap procmng_cap, cap domain_cap);
```

The spawn reply. The spawn reply is a general-purpose message which spawnd sends back to the process manager to communicate the error status of one of the requests listed above.

```
message spawn_reply(errval err);
```

The *spawn, spawn with caps* and *span* requests implement the *span* message part of the model presented in chapter 4. In addition, the *kill* and *cleanup* requests implement the *stop* and *free resources* messages respectively. There is no equivalent to *allstop*, as this message does not trigger any physical action. The purpose of *allstop* in the model was only to highlight that all the victim domain's DCBs have been removed from the run queues, hence implementing it in practice would mean an extra round-trip with no effect. Instead, the process manager updates the aggregated domain state to *stop* when all spawnd instances have replied to its *kill* requests.

In order to correctly identify the domain referred by the requests received from the process manager, spawnd performs the same steps described in subsection 6.2.2. Firstly, an RPC is made for retrieving the contents encapsulated in the domain capability. Secondly, a hashtable is used for storing domain structures. The hashing function is the same h defined in subsection 6.2.2. The object mapped to the resulting hash code is a representation of the domain on the local core, containing the domain capability, the DCB for the local dispatcher, as well as a reference to its root cnode. The mapping is created when either a span or span call is received. Subsequently, when a kill or cleanup call arrives, the DCB or, respectively, the rootcn is retrieved from the mapping, revoked and destroyed.

6.3.2 Asynchronous message queues

It has been mentioned that the process manager does not block when communicating with the spawnd backend, by sending and receiving asynchronous messages. However, spawnd does block, *i.e.* it only acts on one request from the process manager at the time. Two questions stemmed from this implementation decision:

1. What happens if the process manager receives two or more simultaneous requests intended for the same spawnd instance?

2. How does the process manager resolve the client to respond to when it receives an asynchronous message from spawnd?

The first question is partially answered by the Barrelfish message-passing infrastructure. Assume two clients running on two different cores send simultaneous requests to the process manager. Flounder, the RPC mechanism, will serialize the requests and deliver them sequentially to the process manager, which will thus forward them to spawnd. Similarly, if the two requests arrive at the spawnd server in quick succession, then the message-passing infrastructure will enqueue the request which arrived second until spawnd has finished serving the first one.

The interconnect driver hence handles the case of multiple requests arriving simultaneously or in quick succession. However, before sending requests through a binding, Flounder first enqueues the requests in the sender's transmit (TX) buffer, to be popped and sent when the binding is ready. If the process manager attempts to send too many requests to spawnd too quickly, its TX buffer might therefore become full and unable to enqueue more messages, returning an error. The buffer could be polled until there is room to add another request, but this would block the executing thread.

One possible approach to solving this problem is having the process manager delegate a new thread for attempting to enqueue in the TX buffer. However, Barrelfish traditionally prefers event-based design to threaded concurrency. A cleaner solution hence consists in storing to-be-sent requests in event queues on top of every spawnd binding that the process manager holds. The process manager has one send queue per spawnd it is bound with, to which it adds requests instead of sending them right away. For every queue, an event is scheduled that will pop the next request and send it. On encountering a TX buffer full error, the request is re-added to the front of the queue[1]. This strategy is similar to the one used by monitors to perform capability operations through the intermon channels. Pseudo-code is given in algorithm 4.

Registering send events from request queues is an elegant solution to the TX buffer problem, as it is single-threaded and does not block the process manager. However, it reiterates the second

[1] If the process manager only sends requests to spawnd by registering for send events, then it should never receive a `TX_BUSY` error. Testing for this error and adding requests to the front of the queue is only done as a safeguard, in case the process manager will ever need to call a tx send function directly, bypassing the queue.

```
1  Function handle_request(request)
2      enqueue(request);
3      if queue_size = 1 then
4          register_send(send_from_queue);
5      end
6  Function send_from_queue()
7      request := dequeue();
8      err := send(request);
9      if err = FLOUNDER_ERR_TX_BUSY then
10         enqueue_front(request);
11     end
12     if queue_size > 0 then
13         register_send(send_from_queue);
14     end
```

Algorithm 4: Pseudo-code for managing the send queue for a given spawnd instance. Function *handle_request* (lines 1-5) is called for every new request that the process manager needs to send to spawnd. The request is added to the send queue; if the queue was empty before, a send event is registered on the *send_from_queue* handler (lines 6-14). The handler is called by the standard Barrelfish dispatch mechanism. It pops the next request from the queue and tries to send it; if the TX buffer is full, the request is added back to the front of the queue. The handler re-registers itself if the send queue is not empty.

question stated before, that of telling what requests an asynchronous reply from spawnd corresponds to. One of the investigated options involved the process manager's sending a token to spawnd identifying the client which the request originated in. On replying, spawnd would include the same token, thereby letting the process manager know what client the response is for. Such a token could be the domain capability of the client.

A lighter approach relies on the fact that every spawnd instance serves requests from the process manager sequentially. Since Flounder guarantees FIFO delivery of messages, it is safe to expect that the replies from spawnd will arrive at the process manager in the same order as the requests were sent. For example, if the process manager sends requests x and y to a spawnd server in this order, then that server must reply to x before it replies to y. This conclusion is proven formally below, using the following notation:

- x is a request from the process manager to spawnd and x' the corresponding reply from spawnd;
- $T_{departure}(m) \in (0, \infty)$ is the timestamp when message m (either request or reply) leaves the sender; requests depart from the process manager, whereas replies depart from spawnd;
- $T_{arrival}(m) \in (0, \infty)$ is the timestamp when message m (either request or reply) arrives at the recipient; requests arrive at spawnd, whereas replies arrive at the process manager;
- $D_{service}(x) \in (0, \infty)$ is the time required for spawnd to serve the request x;
- $B(x) = \{y : T_{arrival}(y) < T_{arrival}(x)\}$ is the set of requests which arrived at spawnd before x;
- $T_{queue}(x) = \sum_{y \in B(x)} max\{D_{service}(y) - T_{arrival}(x) + T_{arrival}(y), 0\}$ is the time spent by x in queue before spawnd starts serving it;

The time when spawnd sends a reply can be expressed as:

$$T_{departure}(x') = T_{arrival}(x) + T_{queue}(x) + D_{service}(x) \tag{6.1}$$

Furthermore, the guarantee that messages arrive in-order can be expressed as:

$$T_{departure}(x) < T_{departure}(y) \Leftrightarrow T_{arrival}(x) < T_{arrival}(y), \forall x, y \tag{6.2}$$

The proof then proceeds as follows:

Claim. Let x, y requests s.t. $T_{departure}(x) < T_{departure}(y)$. Then $T_{arrival}(x') < T_{arrival}(y')$.

Proof.

$$
\begin{aligned}
& T_{departure}(x) < T_{departure}(y) \Rightarrow T_{arrival}(x) < T_{arrival}(y) \text{ (using Equation 6.2)} \\
& \Rightarrow B(x) = \emptyset \Rightarrow T_{queue}(x) = 0 \\
& \qquad \Rightarrow T_{departure}(x') = T_{arrival}(x) + D_{service}(x) \text{ (using Equation 6.1)} \\
& B(y) = \{x\} \Rightarrow T_{queue}(y) = max\{D_{service}(x) - T_{arrival}(y) + T_{arrival}(x), 0\} \\
& \text{Let } T_{arrival}(y) - T_{arrival}(x) = t > 0 \Rightarrow T_{arrival}(y) = T_{arrival}(x) + t \\
& \Rightarrow T_{queue}(y) = max\{D_{service}(x) - t, 0\}
\end{aligned}
$$

Case $D_{service}(x) - t \leq 0 \Rightarrow T_{queue}(y) = 0$

$$
\begin{aligned}
& \Rightarrow T_{departure}(y') = T_{arrival}(y) + D_{service}(y) \text{ (using Equation 6.1)} \\
& \qquad = T_{arrival}(x) + t + D_{service}(y) \\
& \Rightarrow T_{departure}(y') - T_{departure}(x') = T_{arrival}(x) + t + D_{service}(y) \\
& \qquad - T_{arrival}(x) - D_{service}(x) \\
& \qquad = t - D_{service}(x) + D_{service}(y) \\
& \Rightarrow T_{departure}(y) - T_{departure}(x') > 0 \text{ (using } t \geq D_{service}(x) \text{ and } D_{service}(y) > 0) \\
& \Rightarrow T_{departure}(x') < T_{departure}(y') \Rightarrow T_{arrival}(x') < T_{arrival}(y') \text{ (using Equation 6.2)}
\end{aligned}
$$

Case $D_{service}(x) - t > 0 \Rightarrow T_{queue}(y) = D_{service}(x) - t$

$$
\begin{aligned}
& \Rightarrow T_{departure}(y') = T_{arrival}(y) + D_{service}(y) + D_{service}(x) - t \text{ (using Equation 6.1)} \\
& \qquad = T_{arrival}(x) + t + D_{service}(y) + D_{service}(x) - t \\
& \qquad = T_{arrival}(x) + D_{service}(x) + D_{service}(y) \\
& \Rightarrow T_{departure}(y') - T_{departure}(x') = T_{arrival}(x) + D_{service}(x) + D_{service}(y) \\
& \qquad - T_{arrival}(x) - D_{service}(x) \\
& \qquad = D_{service}(y) > 0 \\
& \Rightarrow T_{departure}(x') < T_{departure}(y') \Rightarrow T_{arrival}(x') < T_{arrival}(y') \text{ (using Equation 6.2)}
\end{aligned}
$$

■

This result inspired the implementation of *receive queues* in the process manager. Similarly with send queues, the process manager has one receive queue per spawnd instance it is bound with. Every receive queue stores bindings for the clients that have sent requests for the corresponding spawnd, in the order the requests were forwarded by the process manager. When a spawnd server sends a reply, the process manager dequeues the next client binding from the receive queue associated with that spawnd, performs any necessary post-processing and finally responds back to said client. The flow of a request through one of the process manager's send queues and its corresponding receive queue is illustrated in Figure 6.2.

6.4 Example scenario

Suppose there are two cores in the system, 0 and 1. Each of them runs an instance of spawnd and core 0 also runs the process manager. Suppose a is an arbitrary client domain running on core 0, which wants to spawn the program "b" on core 0. Figure 6.3 illustrates the flow of messages between a and the process manager, as well as those between the process manager and spawnd.0.

The result of the previous operation is a new domain, b, running a single dispatcher on core 0. The identifying domain capability for b has been added to its cspace, as well as returned to a.0. Figure 6.4 shows what happens if b.0 wants to run a dispatcher on core 1. Finally, Figure 6.5 illustrates what happens if a requests that the process manager kill b.

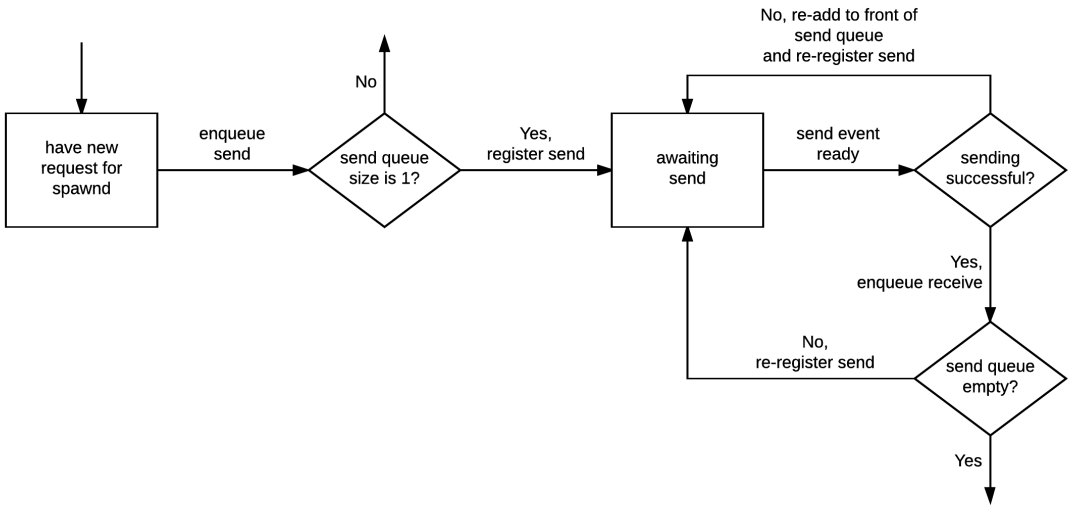


Figure 6.2: Flow of a request through one of the process manager's send queues. The initial state corresponds to receiving a new request from a client. The state machine proceeds as per algorithm 4. On successfully sending the request to spawnd, the originating client is added to the receive queue, where it will be popped from and responded to when spawnd sends the matching reply. States with outwards-pointing arrows are final.

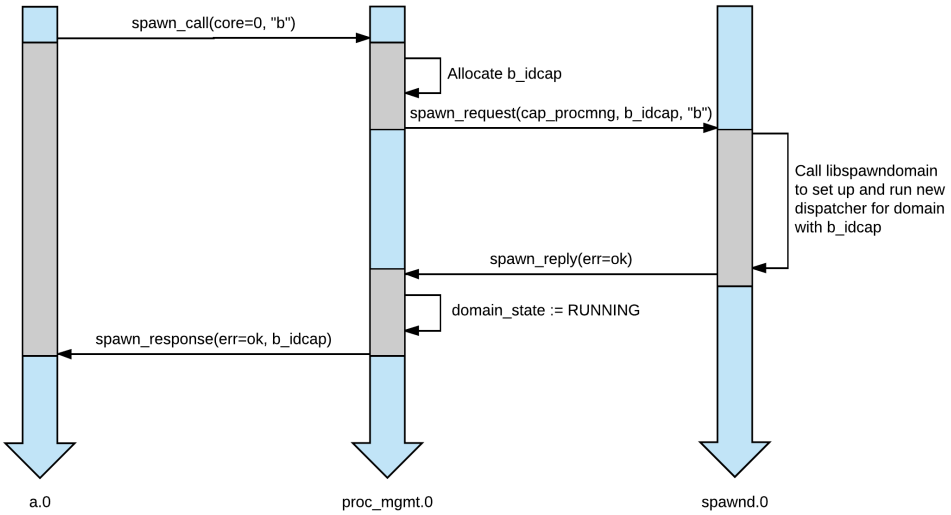


Figure 6.3: A spawn request. Time flows from top to bottom. Client *a* running on core 0 asks to spawn program "b" on the same core. The process manager (proc_mgmt.0) allocates an identifying capability for the new domain b.0 and it instructs spawnd.0 to run program "b" in that domain. When spawnd.0 replies with a success message, proc_mgmt.0 sets the domain state of b.0 to RUNNING, before responding back to a.0. Time slices during which the dispatchers are blocked are marked in gray. Owing to asynchronous message-passing, the process manager does not need to wait while spawnd finishes running the new domain.

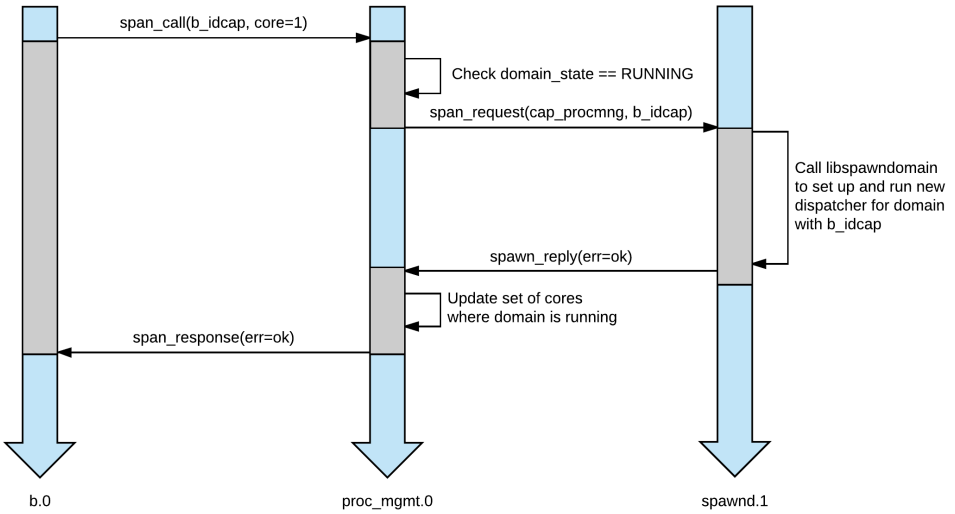


Figure 6.4: A span request. Time flows from top to bottom. Client b.0 asks to run a dispatcher on core 1. The process manager checks the client's domain state. If the state were not RUNNING, proc_mgmt.0 would stop immediately and respond with an error. Since b's state is RUNNING, its request is forwarded to spawnd.1, which runs the new dispatcher and replies back to the process manager. The latter then updates the set of cores which b is running on and informs b.0 the spanning was successful. Time slices during which the dispatchers are blocked are marked in gray.

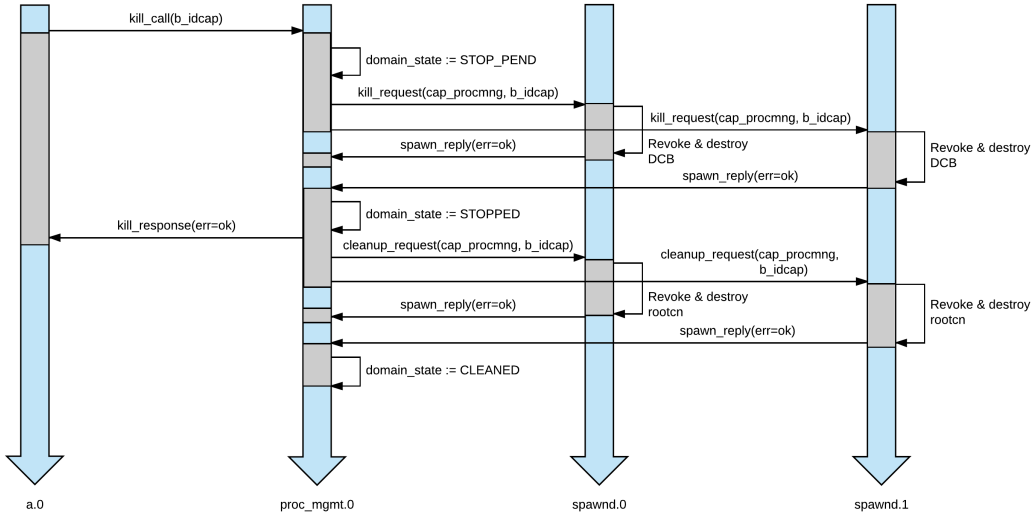


Figure 6.5: A kill request. Time flows from top to bottom. Client a.0 shows proc_mgmt.0 the identifying capability for b, asking that it be stopped. The process manager marks the pending stop for b, so that it will not be able to span any more dispatchers. An asynchronous kill request is then sent to each of the two cores running dispatchers for b. When both spawnd servers confirm that they have removed their dispatcher, proc_mgmt.0 marks b as STOPPED, informs the client and all waiters of the successful kill and starts cleaning up. An asynchronous cleanup request is hence sent to each spawnd. When both requests have been replied to, b is marked as CLEANED.

Chapter 7

Experiments

Performance of the process management subsystem was tested by comparing empirical run times with queuing theory models. This chapter describes *what* was measured and *why* measuring it was important. It then presents the strategy behind building theoretical models used to predict what the measurements *should be*. The objective is to analyze how the system handles multiple client connections, as well as how much overhead the process manager service adds to spawning operations.

7.1 Experimental setup

Experiments were run on the rack-mounted x86 machines in the Systems Group at ETH Zürich. The machine used for the most part was Babybel1, which has 20 hyperthreading-enabled Intel® Xeon® CPU E5-2670 v2 cores at 2.50 GHz and 256 GB of RAM. It is connected to Infiniband and 10 Gigabit Ethernet.

Throughout all tests, the process management server was run on core 0. The spawnd backend was run on all cores, however only cores 0 and 7 were targeted with requests. The client domains ran on core 3.

Deploying Barrelfish to the rack machines and benchmarking were achieved using Harness[16].

7.2 Measuring response time

The first question we sought to provide an answer to by means of benchmarking was *how long does it take to spawn a domain?* To this end, the round-trip response time of spawn requests was measured on the client side. A Harness test launched from 1 to 5 client domains on core 3 and connected them to the process manager on core 0. The clients sent a total of 150 requests of type *spawn on core X*, where X was either 0 or 7 for different runs.

The domains spawned following client requests exited execution as soon as they entered the main function, performing no operations. Instrumentation was achieved through the *bench* library implemented in `lib/bench/bench.c`. The test can be found under `usr/tests/proc_mgmt_test` and the Harness script is `tools/harness/tests/proc_mgmt_test.py`.

Among all process management API calls, `rpc spawn` was chosen for instrumenting due to its complexity in terms of steps performed. Killing and cleaning up could potentially be more complex in sophisticated setups due to revoking the DCB and rootcn capabilities, operations which can expand system-wide. However, in order to kill and clean up domains it is necessary to spawn them in the first place, hence the experiment design.

7.2.1 Spawning on core 0

For the first experiment, the clients running on core 3 spawned domains on core 0. This meant that the process manager, the spawnd server doing all the spawning work and the newly created domains all ran on the same core. Most system services are also offered on core 0, *e.g.* the memory

and ramfsd servers. Therefore, the expectation was that spawning on core 0 would be hindered by all system services' competing for scheduling. Average response time, standard deviation and coefficient of variation measured for this experiment are given in Table 7.1. The values are also illustrated in Figure 7.1.

The response time varied from 231 *ms* when there was only 1 client in the system to 1300 *ms* when there were 5. However, the coefficient of variation calculated as $\frac{stdev}{mean}$ decreased with the number of clients. In terms of response time, the difference between two successive cases is greater than the base 1 client case, which can be explained from the perspective of two factors. Firstly, the more clients core 3 runs, the smaller the fraction of time every one of them is allocated becomes. Secondly, the clients communicated with the process manager via UMP, which relies on cache coherence to send messages. It is hence possible that delivering a message to one client could impact another's optimal cache layout.

Load	1 client	2 clients	3 clients	4 clients	5 clients
Average response time (μs)	231,266.500	471,773.125	740,489.188	1,012,757.875	1,300,052.250
Standard deviation	36,040.464	56,870.260	81,281.411	106,100.757	113,415.016
Coefficient of variation	0.156	0.121	0.110	0.105	0.087

Table 7.1: Average response time and standard deviation in microseconds, as well as coefficient of variation for up to 6 clients spawning domains on core 0. All values are rounded to three decimal places.

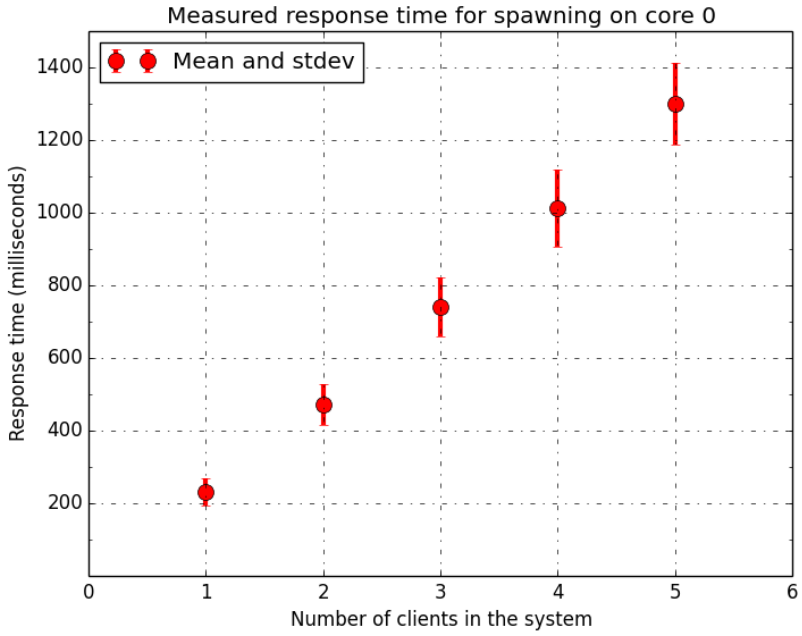


Figure 7.1: Average response time and standard deviation (in milliseconds) for 1 to 5 clients spawning domains on core 0.

7.2.2 Spawning on core 7

The second experiment had clients spawn domains on core 7. The aim was to compare the performance of spawning on the bsp core to that achieved when spawning on an app core. On one

hand, spawning on an app core means that communication between the targeted spawnd and other servers, *e.g.* the process manager and the memory server, is done through UMP rather than LMP, which might hint at worse performance. On the other hand, targeting a spawnd on an app core might lead to lower response times due to its not competing for scheduling with all the system services provided on core 0.

Results for this experiment are given in Table 7.2 and illustrated in Figure 7.2. The conclusion is that both hypotheses were true to some degree. Mean response times were approximately 1.2 to 1.4 times lower than when spawning on core 0, however the system was more unstable as the standard deviation to mean ratios were considerably higher. The speedup can be attributed to the activity of spawnd.7 being more seamless due to the domain's not competing for CPU time with the process manager and the memory server. The drop in stability can be blamed on switching over to UMP as the interconnect driver used by spawnd.7 to exchange messages and ask for resources.

Load	1 client	2 clients	3 clients	4 clients	5 clients
Average response time (μs)	170,941.453	382,508.219	586,136.563	780,862.250	936,249.625
Standard deviation	71,615.853	162,426.578	237,445.443	138,870.097	178,352.671
Coefficient of variation	0.419	0.425	0.405	0.178	0.190
Relative average speedup	1.353	1.233	1.263	1.297	1.389

Table 7.2: Average response time and standard deviation in microseconds, as well as coefficient of variation for up to 6 clients spawning domains on core 7. Relative speedup is given compared to the average response time recorded in Table 7.1. All values are rounded to three decimal places.

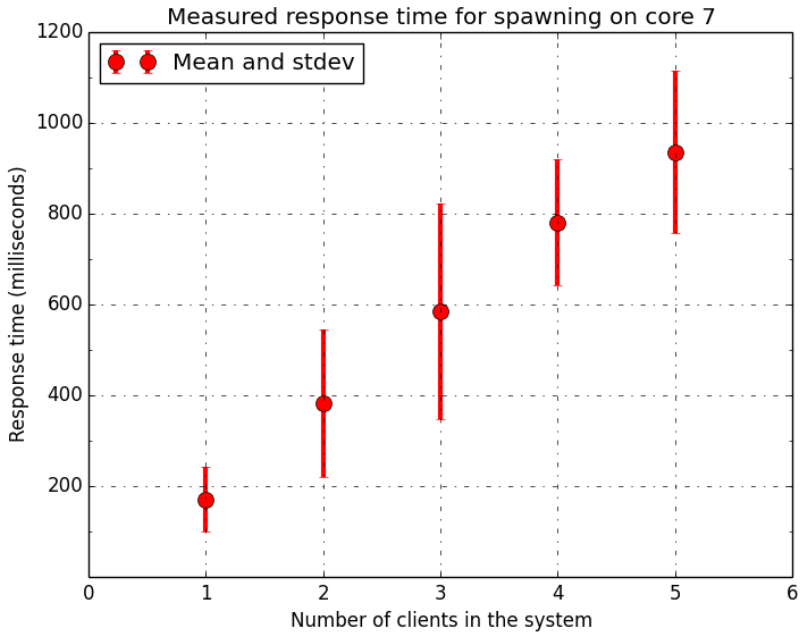


Figure 7.2: Average response time and standard deviation (in milliseconds) for 1 to 5 clients spawning domains on core 7.

Lastly, Figure 7.3 below depicts the comparison between the response times measured when

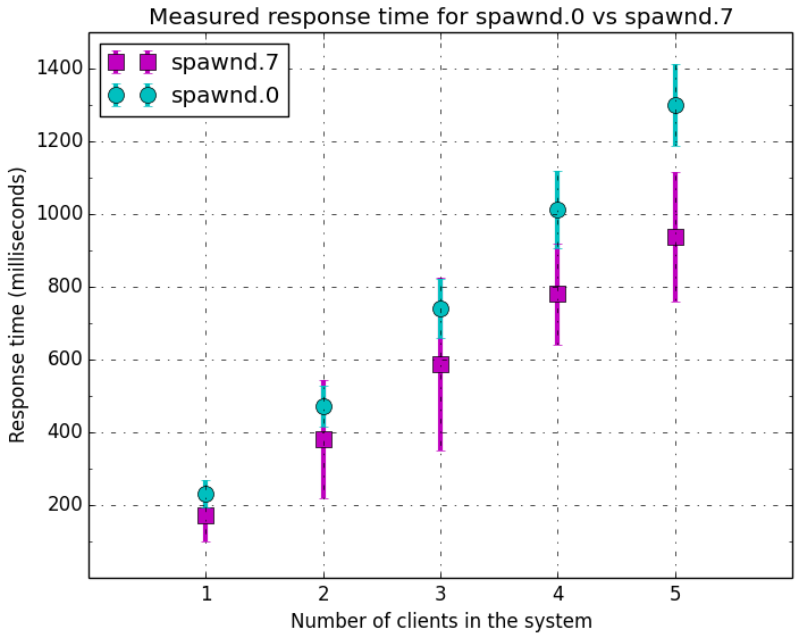


Figure 7.3: Comparison between response times measured when spawning on core 0 and core 7.

spawning on core 0 and core 7.

7.3 System as a network of queues

Although we have discussed the relative performance of spawning on two different cores, there has so far been no indication of how good or poor the response times were in absolute terms. We also do not know how strongly the process management middleware impacted these values. In order to tackle these problems, we present an approach at modeling the system by means of queuing theory. Queuing theory is useful for analyzing how systems behave under stress and can be conveniently applied to the event-oriented message-passing infrastructure in Barrelfish. The process management as-a-service implementation and the request queues described in the previous chapter further render the system suitable to be modeled using queuing theory. This section therefore aims at accomplishing two objectives:

1. Assessing how much of the response time of a spawn RPC is spent in the process management server;

2. Providing theoretical estimates for the performance of the whole process management subsystem.

7.3.1 The model

All valid domain-related requests arrive at spawnd through the process manager and all responses travel back to clients through the process manager. Consequently, the process management subsystem can be thought of as a linear queuing network of three nodes. The first node maps to the validation and preprocessing done by the process manager, such as checking the domain capability and state. From there, clients queue up to be served by the second node, the spawnd backend. When the latter's job is done, clients move forward to the third node, which corresponds to the process manager's postprocessing and resolving of where to send the reply. The network is illustrated in Figure 7.4.

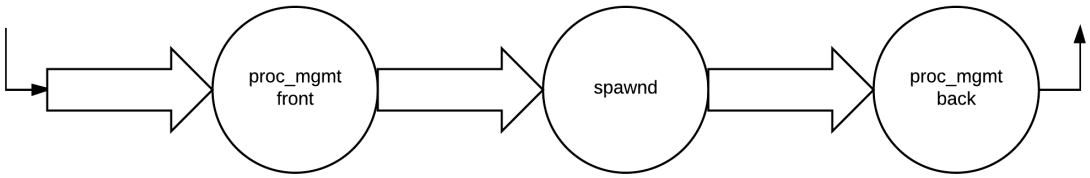


Figure 7.4: The basic process management queuing network. Clients enter the system through the queue for the first process manager node and must proceed though all queues in succession. Leaving the system means being served by the second process manager node.

7.3.2 Microbenchmarking

Analyzing the network of queues required knowledge of base service rates for every node. To this extent, the process manager and spawnd servers were instrumented to also collect statistics from the experiments described in section 7.2. The initial case of a single client spawning domains was used to determine base service time for the network's nodes, which are presented in Table 7.3 alongside the resulting service rates (calculated as $\mu = \frac{1}{t_s}$, where μ is the service rate and t_s is the service time). The first insight is that the process management nodes serve clients tremendously faster – the front and back nodes put together are about 3000 times faster than the spawnd node. This result is intuitive, as spawning a domain is considerably more complex than the routing and bookkeeping jobs of the process manager. The spawnd backend scored a throughput of only approximately 15.3 queries per second, which strongly suggests spawnd is the bottleneck of the process management subsystem performance-wise.

Node	Average service time	Service rate
proc_mgmt front	1.014039 μs	986,155.364833 q/s
spawnd.0	65,323.699219 μs	15.308379837 q/s
proc_mgmt back	20.545126 μs	48,673.3447145 q/s

Table 7.3: Average service time and resulting service rates for the network nodes, using requests of type *spawn on core 0*. Time is measured in microseconds. Service rates are measured in queries per second.

The same microbenchmarking technique was applied to the experiment targeting the app core. Data collected by the process manager and spawnd servers showed the first process management node and the spawnd node to be 2.72 and 1.12 times slower than their first experiment counterparts. In the case of spawnd, the slight increase in service time could be due to the fact that core 7 exchanges messages through UMP instead of LMP. All capability transfers are slower through UMP, since the capabilities travel through the monitors on both cores. As for the process manager front node, while the relative increase in service time seems high, the absolute difference is only about 1.75 μs, which can be attributed to arbitrary noise in the system. A similar argument can be made for the process manager back node, which measured a 242% speedup, but was only absolutely faster by 12 μs. Service rates are presented in Table 7.4.

Node	Average service time	Service rate
proc_mgmt front	2.762133 μs	362,039.047359 q/s
spawnd.7	73,448.007812 μs	13.6150731625 q/s
proc_mgmt back	8.476534 μs	117,972.74688 q/s

Table 7.4: Average service time and resulting service rates for the network nodes, using requests of type *spawn on core 7*. Time is measured in microseconds. Service rates are measured in queries per second.

7.3.3 Mean value analysis

The results obtained from microbenchmarking were fed into an implementation of the mean value algorithm (MVA)[17][18][19]. The algorithm uses base service rates of the network's nodes to estimate performance in terms of queue lengths and waiting times when there are multiple clients in the system.

Table 7.5 presents the waiting time at every node and resulting queue lengths when there is only 1 client in the system spawning domains on core 0. For this base case, every node's waiting time is equal to its service time. For every node k, the length of its queue is given by $v_k \cdot W_k \cdot \lambda$, where v_k is the node's visit ratio ($v_k = 1$ since the network is linear), W_k is its waiting time and λ is the system throughput. For a given number of clients m, the throughput is calculated using Little's law[20] as $\lambda = \frac{m}{\sum_k v_k \cdot W_k}$.

Node	Waiting time	Queue length
proc_mgmt front	1.014039 μs	$1.551817 \cdot 10^{-5}$
spawnd.0	65,323.699219 μs	0.999670
proc_mgmt back	20.545126 μs	0.000314

Table 7.5: Waiting time and queue lengths for every node, as calculated by the MVA algorithm. Values are based on microbenchmarking with 1 client spawning domains on core 0. The waiting time is the same as the service time in Table 7.3. Queue lengths mark how saturated the nodes are, *i.e.* clients need to queue up for a node when its queue length hits 1. Additionally, for a single-client case, queue lengths add up to 1, suggesting how large of a fraction of the overall response time every node is responsible for.

Running further iterations of the MVA algorithm produced estimates of the system response time if multiple clients connected simultaneously. Table 7.6 lists values for 1 up to 6 clients. The response time in the base $n = 1$ client case is equal to the sum of all nodes' service times. The difference between $n+1$ and n clients amounts to the service time of the spawnd node, as it is the slowest in the network.

Load	1 client	2 clients	3 clients	4 clients	5 clients	6 clients
Response time (μs)	65,345.258	130,647.412	195,971.098	261,294.797	326,618.496	391,942.195

Table 7.6: System response time predictions for 1 to 6 clients, according to MVA based on the data in Table 7.5. All values are in microseconds and are rounded to three decimal places.

Glancing at the numbers in Table 7.6 reveals that they are considerably lower than the actual measured values given in Table 7.1. One explanation is that model devised so far only takes into

account the work done by spawnd and the process manager in function calls which implement the spawn API call. Consequently, the model fails to encompass a few items, such as:

- other domains being scheduled into execution on the bsp core, most notably the monitor and memory server;
- the time needed by the interconnect driver to deliver messages between dispatchers (UMP to and from clients, LMP between the process manager and spawnd);
- the fact that domains spawned as part of the experiment exit execution as soon as they enter main, meaning they trigger kill and cleanup calls back and forth between the process manager and spawnd.

To correct for these shortcomings, a second version of the queuing model was designed to include an extra *black box node* representing all the work undetectable by the instrumented code. The base service time of the additional node was set to the difference between the response time measured by the client and the total service time of the other three nodes, equaling to approximately 165921.242 μs. The first iteration of the MVA algorithm on the revised network produced the values listed in Table 7.7.

Node	Waiting time	Queue length
proc_mgmt front	1.014039 μs	4.384721 $\cdot 10^{-6}$
spawnd.0	65,323.699219 μs	0.282461
proc_mgmt back	20.545126 μs	8.883745 $\cdot 10^{-5}$
black box	165,921.241616 μs	0.717446

Table 7.7: Waiting time and queue lengths for the extended network. Values are based on microbenchmarking with 1 client spawning domains on core 0, with the service time for the black box node calculated as the difference between the response time measured by the client and the sum of the other nodes' service time.

Adding the extra node reduced the queue length of spawnd from 0.999670 to 0.282461. Consequently, while spawnd still accounted for roughly 28% of the total response time, it is clear that more time is spent in the intrinsic corners of the system. Proceeding with the additional iterations of the MVA algorithm resulted in the system response time estimates presented in Table 7.8. It is apparent that the extended model predicts values closer to the ones measured in section 7.2.

Load	1 client	2 clients	3 clients	4 clients	5 clients	6 clients
Response time (μs)	231,266.500	368,757.423	517,381.666	673,592.673	834,410.358	997,784.072

Table 7.8: System response time predictions for 1 to 6 clients, according to MVA based on the extended queuing network. All values are in microseconds and are rounded to three decimal places.

Together, the MVA values shown in Table 7.6 and Table 7.8 give two upper bounds on the performance of the system when spawning domains on core 0. The first bound is looser and can be interpreted as giving the performance of spawning a domain according only to spawnd and the process manager's service implementation code. The second one is tighter, since it also tries to predict the work done by the operating system under the hood. Figure 7.5 illustrates the two bounds together with the empirical values measured in section 7.2.

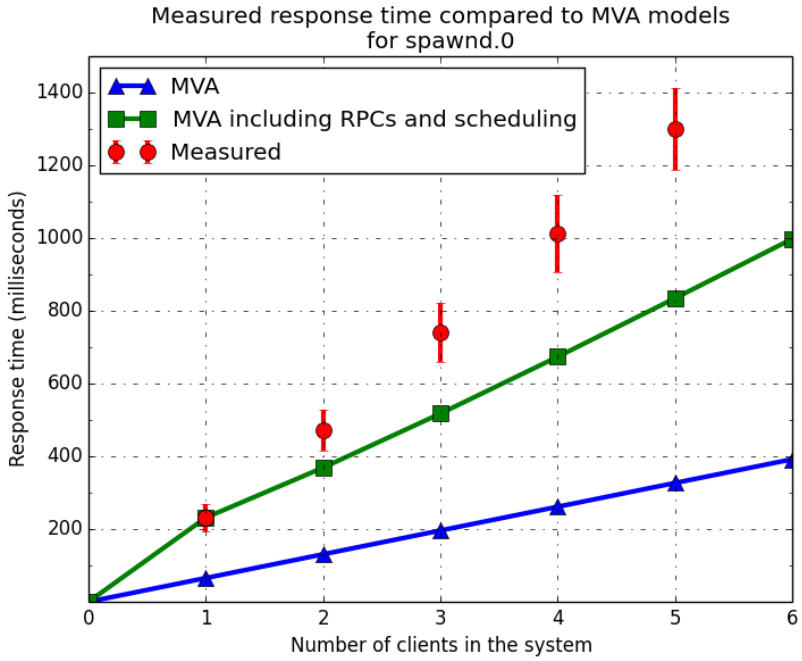


Figure 7.5: Comparison between measured response time and MVA-predicted values for spawning on core 0. The blue and green dots are connected by lines to highlight the bounds.

The same modeling steps were applied to the experiment involving spawning on core 7. Table 7.9 shows the waiting time and queue lengths for the three nodes of the original queuing network, based on the microbenchmarking results presented in Table 7.4. Furthermore, Table 7.10 lists response times estimated by further MVA iterations.

Node	Waiting time	Queue length
proc_mgmt front	2.762133 μs	3.760088 $\cdot 10^{-5}$
spawnd.7	73,448.007812 μs	0.999847
proc_mgmt back	8.476534 μs	0.000115

Table 7.9: Waiting time and queue lengths for every node, as calculated by the MVA algorithm. Values are based on microbenchmarking with 1 client spawning domains on core 7. The waiting time is the same as service time in Table 7.4

Load	1 client	2 clients	3 clients	4 clients	5 clients	6 clients
Response time (μs)	73,459.246	146,896.018	220,344.023	293,792.031	367,240.039	440,688.040

Table 7.10: System response time predictions for 1 to 6 clients, according to MVA based on the data in Table 7.9. All values are in microseconds and are rounded to three decimal places.

Since once again the 3-node queuing model seemed to overestimate the previously calculated system response times, the additional black box node accounting for the difference was added. Updated queue lengths for the 4-node model are given in Table 7.11. This time, spawnd accounts for slightly less than 42% of the system response time, which is more than when spawning on core 0. The black box node is responsible for 53% of the response time, down from 71.7% in the core

0 experiment. These observations suggest a few conclusions paramount to the implementation of the process management subsystem:

- the actual spawning work done by spawnd takes longer on app cores – potentially because spawnd needs to transfer capabilities via UMP instead of LMP;
- the overall spawn domain RPC is faster on app cores, implying that messages might be delivered faster on average owing to scheduling: spawnd does not compete with the process manager, the memory and ramfsd servers etc;
- although faster on average, spawning on app cores is also more unstable compared to the bsp core; this is likely also due to communicating through UMP, which relies on memory to send messages.

Using this data, response time predictions for 1 to 6 clients were again computed using MVA and are shown in Table 7.12. Finally, a graphical comparison of the MVA-generated bounds and measured response time is illustrated in Figure 7.6.

Node	Waiting time	Queue length
proc_mgmt front	1.014039 μs	1.615836 $\cdot 10^{-5}$
spawnd.0	65,323.699219 μs	0.429668
proc_mgmt back	20.545126 μs	4.958735 $\cdot 10^{-5}$
black box	97,482.206646 μs	0.570267

Table 7.11: Waiting time and queue lengths for the extended queuing network based on the app core experiment.

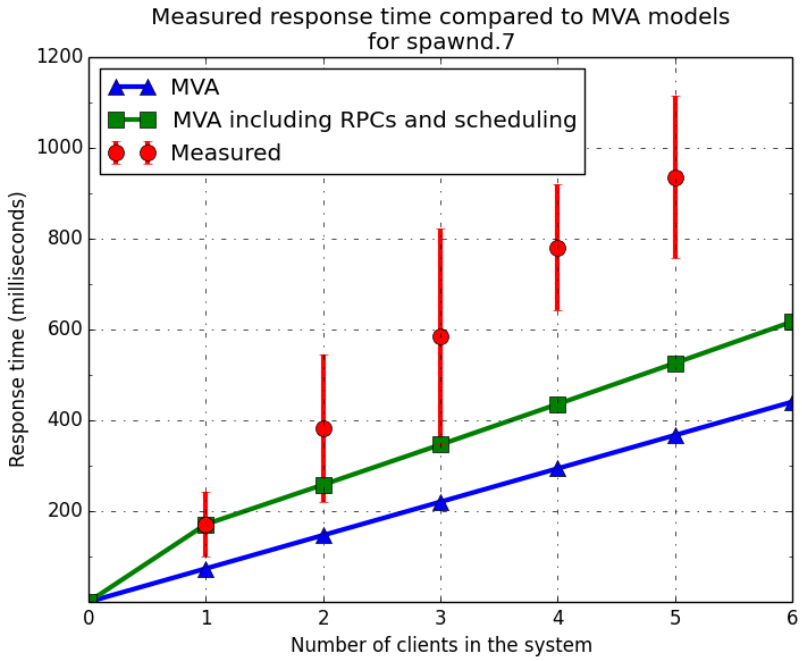


Figure 7.6: Comparison between measured response time and MVA-predicted values for spawning on core 7. The blue and green dots are connected by lines to highlight the bounds.

Load	1 client	2 clients	3 clients	4 clients	5 clients	6 clients
Response time (μs)	231,266.500	368,757.423	517,381.666	673,592.673	834,410.358	997,784.072

Table 7.12: System response time predictions for 1 to 6 clients, according to MVA based on the extended queuing network for the app core experiment. All values are in microseconds and are rounded to three decimal places.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

To conclude, we reiterate through the main research objectives, stating brief answers to the questions presented in chapter 1:

- **What is a process?** The closest equivalent to a UNIX process is a dispatcher; however, Barrelfish features an additional layer of abstraction on top of dispatchers: *domains*. A domain is set of dispatchers uniquely identified by a domain capability, which share the same vspace and have the purpose of running a single program using one or more cores. Every domain has at least one dispatcher, the main one; conversely, every dispatcher belongs to exactly one domain.

- **What can a process do?** A domain exists to run the program which the main dispatcher was created for. Among a domain's dispatchers, the main one executes the program code, while subsequent ones exist to grant it access to other cores' resources by leveraging on their shared virtual address space. Another way to answer this question is: a domain can perform any action for which it is authorized by a capability held by one of its dispatchers. From the process management perspective, such actions include adding spanning to a new core, spawning new domains, killing or waiting for domains for which a domain capability is referable and requesting to exit execution.

- **What actions can be performed on a process and who can perform them?** Firstly, a domain can be spawned by another domain with a client connection to the process manager – which is how domains come into existence. Once running, a domain can be killed or waited on by any other domain that holds its identifying capability. Once killed, all of the domain's dispatchers are removed from the run queues of all cores and the capabilities held by the dispatchers are released. If a domain is waited on however, its course will not be impacted.

- **What is the life cycle of a process?** The life cycle is presented in chapter 4 and features five distinct states: *nil, run, stop pending, stop* and *cleanup*. The first three states exist both globally across the system and locally, projected on every core. Globally, state *nil* means the domain has not been created yet, *run* means at least one core is running a dispatcher for the domain and *stop pending* means that an authoritative entity has requested the domain be killed. Locally for a fixed core, nil means the core has no knowledge of the domain, *run* means the core is running a dispatcher for the domain, while *stop pending* means the core has removed its local dispatcher from the run queue. Finally, state *stop* means all cores have removed their dispatchers for the domain from their run queues and state *cleanup* means that the domain's capabilities have also been released from all cores.

- **Can a process subsystem benefit from the TTY subsystem?** In its current iteration, the process management system is not tied to the TTY subsystem in Barrelfish. One question that was investigated while the system was being designed is whether Angler-created sessions should be used to control for which domains are allowed to kill other domains. In the end, the

path taken was that of using identifying domain capabilities to validate all domain operations, as it offered more granular control. However, a potential direction for merging the process management and TTY subsystems consists in assigning one process manager instance per session. If this were the case, the process manager could share the identifying capabilities of all the domains it controlled with all its clients, as they all belonged to the same user session. The result would be functionality similar to the session-based process control in UNIX.

The questions stated above were first approached by devising a domain model. The model aggregated the state of each of the domain's dispatchers into one domain state machine. Pre- and post-conditions were formally validated for every state using TLA+ and the TLC model checker.

Based on the formal model, we presented a process management subsystem for Barrelfish. The system was designed *as a service*. The main advantages of such a solution include providing a trusted authority for keeping track of domain state, validating domain operations and ensuring killing and cleanup. Furthermore, the process manager was designed as a gateway between clients and spawnd, allowing the latter to be privately registered with it instead of publicly available through the nameserver.

Although concealing spawnd from clients offered more control over domain-related operations, it also generated a need for authenticating spawnd and the process manager. To this end, an authentication protocol was designed to formally guarantee that the two parties believe they are talking to each other, with the help of the monitor. The protocol was reinforced against brute-force discovery of spawnd irefs by having the process manager enclose a special capability with its requests to facilitate validation.

Domains created through the process manager were marked with a special domain capability, which served to account for their state and validate subsequent message exchanges they partake in. To ensure the domain capability's uniqueness and authenticity, creating it was only allowed by retyping from the process manager capability itself. Instead of performing a retype operation every time a new domain is created, the process manager preallocated an L2 cnode and filled it with domain capabilities to amortize complexity, similarly to the slab and slot reallocation techniques. Moreover, to ease identification and lookup, spawnd and the process manager stored domain capabilities in a hashtable under keys computed using an injective function.

Saturating the process manager's TX buffer for spawnd bindings was avoided by adding outgoing requests to higher-level queues instead of attempting to send them straight away. Requests were dequeued and sent to spawnd in an event-based manner. For every send queue, the process manager held a receive queue where it stored clients to respond to on receiving replies from spawnd. The correlation between the send and receive queues was based on the guarantee provided by Barrelfish that messages are transmitted in a FIFO manner and proven correct mathematically.

Finally, the system's performance was measured in terms of response time for spawn requests. Values obtained when spawning on the bsp core were compared to those measured when spawning on an app core. Results determined experimentally were compared against theoretical models computed using queuing theory and mean value analysis, generating two upper bounds on the system's performance. The insight drawn consisted of the following:

- the process management middleware is about three orders of magnitude faster than the spawnd backend;
- the spawnd backend has higher service time on an app core than on the bsp core, resulting in approximately 43% of the total response time compared to 28%;
- the overall spawn RPCs completed faster on average when spawning on the app core, however their completion times varied consistently more.

8.2 Future work

This section lists the shortcomings of the process management subsystem in its current form, as well as techniques for addressing them in the future.

Firstly, a general vector for improvement amounts to distributing the process management system to multiple cores, matter on which a discussion was made in chapter 5. The distribution strategy could be linked to the TTY subsystem in Barrelfish, as mentioned in the previous section.

A second design detail to consider is that the process manager does not currently recycle identifying capabilities once the designated domains have been stopped and cleaned up after. Domain entries persist in the process manager's address space holding the list of cores the domain was run on. In the future, it might be desirable that domain capabilities include a time-to-live field so that they can be reused and the domain entries freed.

8.2.1 Flaws in the process management interface

Spawn requests do not require domain capabilities. The motivation was to keep the spawn API call compatible with domains that were not created by the process manager (*e.g.* started by the monitor before the process manager). However, this implies that a malicious domain can keep spawning other domains even after an authorized kill request has been received for it. This problem could be circumvented by having the monitor create identifying capabilities for itself and the domains it spawns before the process manager.

Another problem in the process management interface is that the exit message expects the domain capability of the sender so that the process manager can know which domain to stop. However, a requesting domain could pass in the identifying capability for another domain instead. The effect would be that of a kill call, but with a custom exit status provided by the requester. In order to avoid this, instead of expecting a domain capability, the process manager should resolve the identity of domains issuing exit messages. The resolution could be performed using the process management client binding, which could be mapped to the domain capability when the client establishes a connection.

8.2.2 Rights for revoking and invoking capabilities

Since domains hold references to their own identifying capability, a malicious domain could prevent others from killing it by calling `cap_revoke` on their own capref. What would happen is that all other copies of the rogue domain's identifying capability would be deleted from the system, thereby the process manager would have no means of requesting that spawnd dequeue the domain's DCB. A potential solution is to introduce a new *is_revokable* capability right which would be set to 0 for all domain capabilities created by the process manager.

Lastly, dispatchers are added to a core's run queue by invoking the DCB capability, which is currently done by spawnd. The process management system assumes that spanning happens strictly under the supervision of the process manager and spawnd, however there currently is no explicit way to enforce that only spawnd can invoke DCBs. One consequence is that a malicious domain a running on core X could bypass the process manager and spawnd to add a dispatcher to core Y, if it had an accomplice b on core Y. Together, a and b could set up a new DCB for a and have b invoke it, thus adding it to the run queue on core Y. This scenario could be prevented by implementing extra checks for `cap_invoke` on DCBs, such as requiring an additional capability that only spawnd possesses.

Appendices

Appendix A

TLA+ Specification for the Formal Process Model

The specification starts with the PlusCal code implementing the algorithm 1 and algorithm 2 given in chapter 4.

The first part of the code declares the variables:

MODULE *SpanStopCleanup*

EXTENDS *Integers*, *Sequences*, *TLC*
CONSTANT N
ASSUME $(N \in Nat) \land (N > 0)$

--algorithm *ProcSpec*{
ProcSpec uses one manager process and N worker processes to simulate N cores
running concurrently in the system, responding to external messages.
For a simplified single-core scenario, set N ← 1 in the TLC model checker.

variables st is the state array for each of the worker processes.
It has N elements, one for worker/worker. Initially set
to "nil" for the starting state, the values any state element can take are
"nil", "run", "stop_pend", "stop", "cleanup".
$st = [i \in 1 \,..\, N \mapsto$ "nil"$]$,
dcb_rq is a boolean mapping from every worker instance, where
dcb_rq[inst] = 1 if that instance's DCB is in the run queue and 0 otherwise.
$dcb\_rq = [i \in 1 \,..\, N \mapsto 0]$,
res is a boolean mapping from every worker instance, where res[inst] = 1
if that instance still uses resources (e.g. memory) and 0 otherwise.
$res = [i \in 1 \,..\, N \mapsto 0]$,
turn denotes who acts next, initially set to 0 for manager; subsequently,
worker[i] will act when turn = i, for i ¿ 0.
$turn = 0$,
old_st denotes the state the machine transitions FROM in every iteration
of the simulation.
$old\_st = [i \in 1 \,..\, N \mapsto$ "nil"$]$,
msg_in is the incoming message, i.e. what caused this state transition.
It is initially set to "nil", corresponding to the initial "nil" state of
the machine.
$msg\_in = [i \in 1 \,..\, N \mapsto$ "nil"$]$,
msg_out is the outgoing message, i.e. what the worker instance replies
to the manager (the entity sending it messages).
$msg\_out = [i \in 1 \,..\, N \mapsto$ "nil"$]$,
last_proc identifies the last process which actioned on a message.

last_proc = 0,
domain_state identifies the global state of the single domain,
aggregated from the individual state of all the dispatchers.
domain_state = "nil" ;

The *manager* process is then defined. This is what the process management server implementation is based on.

```
Simulates the manager, sending messages to all worker instances and tracking the
global domain state by aggregating individual spanwd instance (dispatcher) states.
process ( Manager = 0 ) {
  mon: while ( TRUE ) {
        await turn = 0;

        with ( proc ∈ 1 .. N ) {
          if ( msg_out[proc] = "nil" ∨ msg_out[proc] = "run" ) {
              Process is either in state "nil", or "run" =¿ send "span" and "stop".
              "span" should cause it to transition to "run", while "stop" should
              cause it to transition to "stop_pend".
            with ( action ∈ {"span", "stop"} ) {
              msg_in[proc] := action;
            }
          } else if ( msg_out[proc] = "stop_pend" ) {
              Process is in state "stop_pend".
            if ( ∀ i ∈ 1 .. N : msg_out[i] = "stop_pend" ∨ msg_out[i] = "stop" ) {
                All processes have received the "stop" message =¿ proceed with
                global transition to "stop".
              msg_in[proc] := "allstop";
            }
          } else if ( msg_out[proc] = "stop" ) {
            if ( ∀ i ∈ 1 .. N : msg_out[i] = "stop" ∨ msg_out[i] = "cleanup" ) {
                Send "cleanup" message".
              msg_in[proc] := "free_res";
            }
          }
        } ;

        if ( ∀ proc ∈ 1 .. N : msg_out[proc] = "nil" ∨ msg_out[proc] = "run" ) {
            All dispatchers are either in state "nil" or "run". Note that the
            initial domain_state value is "nil".
          if ( ∃ proc ∈ 1 .. N : msg_out[proc] = "run" ) {
              Some dispatcher is running, hence the domain is running.
            domain_state := "run";
          }
        } else {
          if ( ∀ proc ∈ 1 .. N : msg_out[proc] = "cleanup" ) {
              All dispatchers have performed cleanup.
            domain_state := "cleanup";
          } else if ( ∃ proc ∈ 1 .. N : msg_out[proc] = "stop" ) {
              Cleanup has not been performed yet, but all workers have stopped
              running their local dispatcher.
            domain_state := "stop";
          } else {
              Stopping the domain has been initiated, but not all workers have
              stopped their local dispatcher.
            domain_state := "stop_pend";
          }
```

```
        } ;

        last_proc := 0 ;
        turn := 1 ;
      }
  } ;
```

The *worker* process then follows, modeling the spawnd backend:

```
  Simulates a worker instance, acting on messages from the manager.
process ( Worker ∈ 1 .. N ) {
  spn: while ( TRUE ) {
        await turn = self ;

        old_st[self] := st[self] ;

        if ( msg_in[self] = "span" ) {
            Received "span" message.
          if ( st[self] = "nil" ∨ st[self] = "run" ) {
              Only action on it if current state is "nil" or "run".
            st[self] := "run" ;
            dcb_rq[self] := 1 ;
            res[self] := 1 ;
            msg_out[self] := "run" ;
          }
        } else if ( msg_in[self] = "stop" ) {
            Received "stop" message.
          if ( st[self] = "nil" ∨ st[self] = "run" ) {
              Only action on it if current state is "nil" or "run".
            st[self] := "stop_pend" ;
            dcb_rq[self] := 0 ;
            msg_out[self] := "stop_pend" ;
          }
        } else if ( msg_in[self] = "allstop" ) {
            Received "allstop" message.
          if ( st[self] = "stop_pend" ) {
              Transition to "stop" if current state is still "stop_pend".
            st[self] := "stop" ;
            msg_out[self] := "stop" ;
          }
        } else if ( msg_in[self] = "free_res" ) {
            Received "cleanup" message.
          if ( st[self] = "stop" ) {
              Free resources if current state is "stop".
            st[self] := "cleanup" ;
            res[self] := 0 ;
            msg_out[self] := "cleanup" ;
          }
        } ;

        last_proc := self ;
        turn := (self + 1)%(N + 1) ;
      }
  }
}
```

The PlusCal algorithm is translated into a TLA+ spec modeling the behavior of the manager

and worker processes:

BEGIN TRANSLATION

VARIABLES st, dcb_rq, res, $turn$, old_st, msg_in, msg_out, $last\_proc$, $domain\_state$

$vars \triangleq \langle st, dcb\_rq, res, turn, old\_st, msg\_in, msg\_out, last\_proc, domain\_state \rangle$

$ProcSet \triangleq \{0\} \cup (1 \,..\, N)$

$Init \triangleq$ Global variables
$\land\ st = [i \in 1 \,..\, N \mapsto \text{“nil”}]$
$\land\ dcb\_rq = [i \in 1 \,..\, N \mapsto 0]$
$\land\ res = [i \in 1 \,..\, N \mapsto 0]$
$\land\ turn = 0$
$\land\ old\_st = [i \in 1 \,..\, N \mapsto \text{“nil”}]$
$\land\ msg\_in = [i \in 1 \,..\, N \mapsto \text{“nil”}]$
$\land\ msg\_out = [i \in 1 \,..\, N \mapsto \text{“nil”}]$
$\land\ last\_proc = 0$
$\land\ domain\_state = \text{“nil”}$

$Manager \triangleq \land\ turn = 0$
$\land\ \exists\, proc \in 1 \,..\, N :$
IF $msg\_out[proc] = \text{“nil”} \lor msg\_out[proc] = \text{“run”}$
THEN $\land\ \exists\, action \in \{\text{“span”}, \text{“stop”}\} :$
$msg\_in' = [msg\_in \text{ EXCEPT } ![proc] = action]$
ELSE $\land$ IF $msg\_out[proc] = \text{“stop\_pend”}$
THEN $\land$ IF $\forall\, i \in 1 \,..\, N : msg\_out[i] = \text{“stop\_pend”} \lor msg\_out[i] = \text{“stop”}$
THEN $\land\ msg\_in' = [msg\_in \text{ EXCEPT } ![proc] = \text{“allstop”}]$
ELSE $\land$ TRUE
$\land$ UNCHANGED msg_in
ELSE $\land$ IF $msg\_out[proc] = \text{“stop”}$
THEN $\land$ IF $\forall\, i \in 1 \,..\, N : msg\_out[i] = \text{“stop”} \lor msg\_out[i] = \text{“cleanup”}$
THEN $\land\ msg\_in' = [msg\_in \text{ EXCEPT } ![proc] = \text{“free\_res”}]$
ELSE $\land$ TRUE
$\land$ UNCHANGED msg_in
ELSE $\land$ TRUE
$\land$ UNCHANGED msg_in
$\land$ IF $\forall\, proc \in 1 \,..\, N : msg\_out[proc] = \text{“nil”} \lor msg\_out[proc] = \text{“run”}$
THEN $\land$ IF $\exists\, proc \in 1 \,..\, N : msg\_out[proc] = \text{“run”}$
THEN $\land\ domain\_state' = \text{“run”}$
ELSE $\land$ TRUE
$\land$ UNCHANGED $domain\_state$
ELSE $\land$ IF $\forall\, proc \in 1 \,..\, N : msg\_out[proc] = \text{“cleanup”}$
THEN $\land\ domain\_state' = \text{“cleanup”}$
ELSE $\land$ IF $\exists\, proc \in 1 \,..\, N : msg\_out[proc] = \text{“stop”}$
THEN $\land\ domain\_state' = \text{“stop”}$
ELSE $\land\ domain\_state' = \text{“stop\_pend”}$
$\land\ last\_proc' = 0$
$\land\ turn' = 1$
$\land$ UNCHANGED $\langle st, dcb\_rq, res, old\_st, msg\_out \rangle$

$Worker(self) \triangleq \land\ turn = self$
$\land\ old\_st' = [old\_st \text{ EXCEPT } ![self] = st[self]]$
$\land$ IF $msg\_in[self] = \text{“span”}$
THEN $\land$ IF $st[self] = \text{“nil”} \lor st[self] = \text{“run”}$
THEN $\land\ st' = [st \text{ EXCEPT } ![self] = \text{“run”}]$

$\land\ dcb\_rq' = [dcb\_rq \text{ EXCEPT } ![self] = 1]$
$\land\ res' = [res \text{ EXCEPT } ![self] = 1]$
$\land\ msg\_out' = [msg\_out \text{ EXCEPT } ![self] = \text{“run”}]$
ELSE $\land$ TRUE
$\land$ UNCHANGED $\langle st, dcb\_rq, res, msg\_out\rangle$
ELSE $\land$ IF $msg\_in[self] = \text{“stop”}$
THEN $\land$ IF $st[self] = \text{“nil”} \lor st[self] = \text{“run”}$
THEN $\land\ st' = [st \text{ EXCEPT } ![self] = \text{“stop\_pend”}]$
$\land\ dcb\_rq' = [dcb\_rq \text{ EXCEPT } ![self] = 0]$
$\land\ msg\_out' = [msg\_out \text{ EXCEPT } ![self] = \text{“stop\_pend”}]$
ELSE $\land$ TRUE
$\land$ UNCHANGED $\langle st, dcb\_rq, msg\_out\rangle$
$\land\ res' = res$
ELSE $\land$ IF $msg\_in[self] = \text{“allstop”}$
THEN $\land$ IF $st[self] = \text{“stop\_pend”}$
THEN $\land\ st' = [st \text{ EXCEPT } ![self] = \text{“stop”}]$
$\land\ msg\_out' = [msg\_out \text{ EXCEPT } ![self] = \text{“stop”}]$
ELSE $\land$ TRUE
$\land$ UNCHANGED $\langle st, msg\_out\rangle$
$\land\ res' = res$
ELSE $\land$ IF $msg\_in[self] = \text{“free\_res”}$
THEN $\land$ IF $st[self] = \text{“stop”}$
THEN $\land\ st' = [st \text{ EXCEPT } ![self] = \text{“cleanup”}]$
$\land\ res' = [res \text{ EXCEPT } ![self] = 0]$
$\land\ msg\_out' = [msg\_out \text{ EXCEPT } ![self] = \text{“cleanup”}]$
ELSE $\land$ TRUE
$\land$ UNCHANGED $\langle st, res, msg\_out\rangle$
ELSE $\land$ TRUE
$\land$ UNCHANGED $\langle st, res, msg\_out\rangle$
$\land$ UNCHANGED dcb_rq
$\land\ last\_proc' = self$
$\land\ turn' = (self + 1)\%(N + 1)$
$\land$ UNCHANGED $\langle msg\_in, domain\_state\rangle$

$Next \triangleq Manager$
$\lor (\exists\, self \in 1 \,..\, N : Worker(self))$

$Spec \triangleq Init \land \Box[Next]_{vars}$

END TRANSLATION

Invariants check the pre-and post-conditions of every state of the original domain state machine:

Invariants
1. Transitioning to state ”run” REQUIRES that the msg received be ”span”
and the previous state be ”nil” or ”run”.
$RunReqsSpanMsgFromNilOrRun \triangleq last\_proc > 0 \land st[last\_proc] = \text{“run”} \Rightarrow msg\_in[last\_proc] = \text{“span”} \land (old\_st[last\_proc] = \text{“nil”} \lor old\_st[last\_proc] = \text{“run”})$
$RunReqsHappens \triangleq last\_proc > 0 \land st[last\_proc] = \text{“run”} \Rightarrow \text{FALSE}$

2. If a process is in state "run", then its DCB is in the run queue.
In other words, state "run" ENSURES that the local DCB is in the run queue.

$$RunEnsDcbEnq \triangleq last\_proc > 0 \land st[last\_proc] = \text{“run”} \Rightarrow dcb\_rq[last\_proc] = 1$$

3. Transitioning to state "stop_pend" REQUIRES that the msg received be "stop".

$$StopPendReqsStopMsg \triangleq last\_proc > 0 \land st[last\_proc] = \text{“stop\_pend”} \Rightarrow msg\_in[last\_proc] = \text{“stop”}$$

$$StopPendingReqsHappens \triangleq last\_proc > 0 \land st[last\_proc] = \text{“stop\_pend”} \Rightarrow \text{FALSE}$$

4. If a process is in state "stop_pend", then its DCB is not on the run queue.
In other words, state "stop_pend" ENSURES that the local DCB is NOT on the run queue.

$$StopPendEnsDcbDeq \triangleq last\_proc > 0 \land st[last\_proc] = \text{“stop\_pend”} \Rightarrow dcb\_rq[last\_proc] = 0$$

5. State "stop" REQUIRES that no process be in "nil" or "run"..

$$StopReqsAllInStopPendOrStop \triangleq (\exists\, p \in 1 \,..\, N : st[p] = \text{“stop”}) \Rightarrow (\forall\, p \in 1 \,..\, N : \neg(st[p] = \text{“nil”} \lor st[p] = \text{“run”}))$$

$$StopReqsHappens \triangleq (\exists\, p \in 1 \,..\, N : st[p] = \text{“stop”}) \Rightarrow \text{FALSE}$$

6. State "stop" ENSURES that all DCBs have been removed from their processes' run queues.

$$StopEnsAllDcbsDeqd \triangleq (\exists\, p \in 1 \,..\, N : st[p] = \text{“stop”}) \Rightarrow (\forall\, p \in 1 \,..\, N : dcb\_rq[p] = 0)$$

7. State "nil" REQUIRES that no domain-related messages have been received.

$$NilReqsNoMsg \triangleq last\_proc > 0 \land st[last\_proc] = \text{“nil”} \Rightarrow msg\_in[last\_proc] = \text{“nil”}$$

$$NilReqsNoMsgHappens \triangleq last\_proc > 0 \land st[last\_proc] = \text{“nil”} \Rightarrow \text{FALSE}$$

8. State "nil" ENSURES that there is no DCB in the run queue.

$$NilEnsNoDcb \triangleq last\_proc > 0 \land st[last\_proc] = \text{“nil”} \Rightarrow dcb\_rq[last\_proc] = 0$$

9. State "cleanup" REQUIRES all worker instances transitioning from state "stop".

$$CleanupReqsTransFromStop \triangleq (\exists\, p \in 1 \,..\, N : st[p] = \text{“cleanup”}) \Rightarrow (\forall\, p \in 1 \,..\, N : st[p] = \text{“cleanup”} \lor st[p] = \text{“stop”})$$

$$CleanupReqsHappens \triangleq (\exists\, p \in 1 \,..\, N : st[p] = \text{“cleanup”}) \Rightarrow \text{FALSE}$$

10. State "cleanup" ENSURES all domain resources have been freed.

$$CleanupEnsNoRes \triangleq (\forall\, p \in 1 \,..\, N : st[p] = \text{“cleanup”}) \Rightarrow (\forall\, p \in 1 \,..\, N : res[p] = 0)$$

11. Domain state "nil" is equivalent to all dispatcher states being "nil".

$$DomainStateNil \triangleq last\_proc = 0 \Rightarrow (domain\_state = \text{“nil”} \Rightarrow (\forall\, p \in 1 \,..\, N : st[p] = \text{“nil”})) \land ((\forall\, p \in 1 \,..\, N : st[p] = \text{“nil”}) \Rightarrow domain\_state = \text{“nil”})$$

12. Domain state "run" is equivalent to at least one dispatcher state being "nil"
and all other dispatcher states being either "nil" or "run".

$$DomainStateRun \triangleq last\_proc = 0 \Rightarrow (domain\_state = \text{“run”} \Rightarrow ((\exists\, p \in 1 \,..\, N : st[p] = \text{“run”}) \land (\forall\, p \in 1 \,..\, N : st[p] = \text{“nil”} \lor st[p] = \text{“run”}))) \land (((\exists\, p \in 1 \,..\, N : st[p] = \text{“run”}) \land (\forall\, p \in 1 \,..\, N : st[p] = \text{“nil”} \lor st[p] = \text{“run”})) \Rightarrow domain\_state = \text{“run”})$$

13. Domain state "stop_pend" is equivalent to at least one dispatcher being in state
"stop_pend", but no dispatcher being in state "stop".

$$DomainStateStopPend \triangleq last\_proc = 0 \Rightarrow (domain\_state = \text{“stop\_pend”} \Rightarrow (\exists\, p \in 1 \,..\, N : st[p] = \text{“stop\_pend”}) \land \neg(\exists\, p \in 1 \,..\, N : st[p] = \text{“stop”})) \land ((\exists\, p \in 1 \,..\, N : st[p] = \text{“stop\_pend”}) \land \neg(\exists\, p \in 1 \,..\, N : st[p] = \text{“stop”}) \Rightarrow domain\_state = \text{“stop\_pend”})$$

14. Domain state "stop" is equivalent to at least one dispatcher being in "stop".

$$DomainStateStop \triangleq last\_proc = 0 \Rightarrow (domain\_state = \text{“stop”} \Rightarrow (\exists\, p \in 1 \,..\, N : st[p] = \text{“stop”}))$$

$$\wedge\,((\exists\, p \in 1\,..\,N : st[p] = \text{“stop”}) \Rightarrow domain\_state = \text{“stop”})$$

15. Domain state "cleanup" is equivalent to all dispatchers being in state "cleanup".

$$DomainStateCleanup \triangleq last\_proc = 0 \Rightarrow (domain\_state = \text{“cleanup”} \Rightarrow (\forall\, p \in 1\,..\,N : st[p] = \text{“cleanup”})) \wedge ((\forall\, p \in 1\,..\,N : st[p] = \text{“cleanup”}) \Rightarrow domain\_state = \text{“cleanup”})$$

Lastly, temporal properties model the necessity of the transition between the states *nil* and *stop pending* in order to ensure that malicious domains cannot outrace the cleanup process.

Temporal properties

1. Given:
i) a domain D which is to be completely stopped and removed from the system,
ii) a worker instance W on some core where D has not run yet, which is informed via a "stop" message that D is to NEVER span on its worker,
the transition "nil" -¿ "stop_pend" (through the "stop" message) is mandatory to ensuring that W will never allow D to span to its worker.

The property reads: if there is a "stop" message eventually and there is never a local dispatcher, then eventually the transition "nil" -¿ "stop_pend" (via msg "stop") MUST happen.

$$MustHaveNilStopPend \triangleq \forall\, p \in 1\,..\,N : \Diamond\neg(st[p] = \text{“nil”}) \wedge \Box(dcb\_rq[p] = 0) \Rightarrow \Diamond(old\_st[p] = \text{“nil”} \wedge st[p] = \text{“stop\_pend”} \wedge msg\_in[p] = \text{“stop”})$$

$$MustHaveNilStopPendHappens \triangleq \forall\, p \in 1\,..\,N : \Diamond\neg(st[p] = \text{“nil”}) \wedge \Box(dcb\_rq[p] = 0) \Rightarrow \text{FALSE}$$

$$NilStopPendIsSuff \triangleq \forall\, p \in 1\,..\,N : \Diamond(old\_st[p] = \text{“nil”} \wedge st[p] = \text{“stop\_pend”} \wedge msg\_in[p] = \text{“stop”}) \Rightarrow \Diamond\neg(st[p] = \text{“nil”}) \wedge \Box(dcb\_rq[p] = 0)$$

$$NilStopPendIsSuffHappens \triangleq \forall\, p \in 1\,..\,N : \Diamond(old\_st[p] = \text{“nil”} \wedge st[p] = \text{“stop\_pend”} \wedge msg\_in[p] = \text{“stop”}) \Rightarrow \text{FALSE}$$

Modification History
Last modified Wed Aug 09 14:50:02 CEST 2017 by razvan
Last modified Mon May 22 13:22:06 CEST 2017 by damachir
Created Thu May 11 15:35:45 CEST 2017 by razvan

Appendix B

Isabelle Implementation of the BAN Logic Authentication Proof

Implementation of the BAN logic proof of the single-core process manager - spawnd authentication protocol. For every principal, sets are used to implement:

- what messages it sees;
- what channels it believes it can use to communicate;
- what entities it believes control channels.

```
datatype princ =
    ProcMngr
  | Monitor
  | Spawnd
```

An LMP or UMP channel.

```
type_synonym chan = "char list"
```

Connection: a principal sending messages through a channel to another principal. In classic BAN logic terms, this says that two principals can communicate through a public or private key. In Barrelfish, keys are replaced by LMP or UMP channels.

Connections are the main *belief* held by principals in our BAN logic implementation.

```
type_synonym conn = "princ * chan * princ"
```

A message (k, c) where c is the channel the message is received on and k is the enclosed connection. For example, a message can read as: "I received (P, c1, Q) on channel c2", that is "I received that P can use channel c1 to send messages to Q, on channel c2."

```
type_synonym msg = "conn * chan"
```

A "said" statement, e.g.: "P said that (Q, c, P)", that is "P said that Q can send messages to P through channel c".

This is the second type of *belief* held by principals in our implementation.

```
type_synonym said = "princ * conn"
```

A set of seen messages.

```
type_synonym seen = "msg set"
```

A set of connections held as beliefs.

```
type_synonym bel_conn = "conn set"
```

A set of "said" statements held as beliefs.

type_synonym *bel_said = "said set"*

The message-meaning rule, in set formulation for a fixed principal. Given:

- a set of believed connections to this principal,
- a set of seen messages,

this function matches believed channels to those through which messages where received, to deduce what the principals at the other end of the channels said.

definition *msg_mean :: "bel_conn ⇒ seen ⇒ bel_said"* **where**
"msg_mean Bc S = {(q2, k2). ∃ (q1, c1, p) ∈ Bc. ∃ (k1, c2) ∈ S.
q1 = q2 ∧ k1 = k2 ∧ c1 = c2}"

A set of connection beliefs held by other principals, *e.g. I believe principal X holds beliefs k1 and k2.*

type_synonym *bel_other_bel = "(princ * conn) set"*

The simplified nonce-verification rule, in set formulation. It reads: "if we believe another principal said X, then we believe that it believes X".

definition *nonce_ver :: "bel_said ⇒ bel_other_bel"* **where**
"nonce_ver s_bel = s_bel"

The "control" belief, e.g. (Q, (P, c, Q)) means that principal Q controls the connection(s) which P can use to communicate with Q .

type_synonym *controls = "princ * conn"*

A set of control beliefs.

type_synonym *bel_ctrl = "controls set"*

The jurisdiction deduction rule, in set formulation for a fixed principal. Given:

- the control beliefs of the principal, *e.g. I believe that Q controls k*,
- what the principal believes other principals believe, *e.g. I believe that Q believes k*,

this function returns what this principal should believe based on other principals' authority and beliefs, *e.g. I believe k.*

definition *jurisd :: "bel_ctrl ⇒ bel_other_bel ⇒ bel_conn"* **where**
"jurisd Ctrl Bob = {k3. ∃ (q1, k1) ∈ Ctrl. ∃ (q2, k2) ∈ Bob.
q1 = q2 ∧ k1 = k2 ∧ k1 = k3}"

The authentication lemma, in set formulation.

lemma *auth:*
fixes *p :: princ*
and *p_bel :: bel_conn*
and *p_bel_ctrl :: bel_ctrl*
and *p_seen :: seen*

and *s :: princ*

and *m :: princ*
and *m_bel :: bel_conn*
and *m_bel_ctrl :: bel_ctrl*
and *m_seen :: seen*

```
    and c_pm :: chan
    and c_mp :: chan
    and c_sm :: chan
    and c_ps :: chan

    and m_nonce_ver :: bel_other_bel
    and m_bel' :: bel_conn

    and p_nonce_ver :: bel_other_bel
    and p_bel' :: bel_conn

  assumes i1: "(m, c_mp, p) ∈ p_bel"
      and i2: "(p, c_pm, m) ∈ m_bel"
      and i3: "(s, c_sm, m) ∈ m_bel"
      and i4: "(m, (p, c_ps, s)) ∈ p_bel_ctrl"
      and i5: "(s, (p, c_ps, s)) ∈ m_bel_ctrl"
      and i6: "(p, (m, c_mp, p)) ∈ m_bel_ctrl"

      and m1: "((m, c_mp, p), c_pm) ∈ m_seen"
      and m2: "((p, c_ps, s), c_sm) ∈ m_seen"
      and m3: "((p, c_ps, s), c_mp) ∈ p_seen"

      and d1: "m_nonce_ver = nonce_ver(msg_mean m_bel m_seen)"
      and d2: "m_bel' = m_bel ∪ jurisd m_bel_ctrl m_nonce_ver"
      and d3: "p_nonce_ver = nonce_ver(msg_mean p_bel p_seen)"
      and d4: "p_bel' = p_bel ∪ jurisd p_bel_ctrl p_nonce_ver"

  shows "(p, c_ps, s) ∈ (m_bel' ∩ p_bel')"

proof -
```

1.1) M believes P said $M \overset{K_{PM}}{\rightarrow} P$

```
  have 1: "(p, (m, c_mp, p)) ∈ {(q2, k2). ∃ (q1, c1, p) ∈ m_bel.
                                        ∃ (k1, c2) ∈ m_seen.
                                        q1 = q2 ∧ k1 = k2 ∧ c1 = c2}"
    using i2 m1 by blast
  have 2: "msg_mean m_bel m_seen = {(q2, k2). ∃ (q1, c1, p) ∈ m_bel.
                                           ∃ (k1, c2) ∈ m_seen.
                                           q1 = q2 ∧ k1 = k2 ∧ c1 = c2}"
    by(auto simp: msg_mean_def)
  from 1 2 have 3: "(p, (m, c_mp, p)) ∈ msg_mean m_bel m_seen" by simp
```

1.2) M believes $M \overset{K_{PM}}{\rightarrow} P$

```
  hence 4: "(p, (m, c_mp, p)) ∈ m_nonce_ver" using d1 by(auto simp: nonce_ver_def)
  hence 5: "(m, c_mp, p) ∈ {k3. ∃ (q1, k1) ∈ m_bel_ctrl.
                              ∃ (q2, k2) ∈ m_nonce_ver.
                              q1 = q2 ∧ k1 = k2 ∧ k1 = k3}"
    using i6 by auto
  have 6: "jurisd m_bel_ctrl m_nonce_ver = {k3. ∃ (q1, k1) ∈ m_bel_ctrl.
                                           ∃ (q2, k2) ∈ m_nonce_ver.
                                           q1 = q2 ∧ k1 = k2 ∧ k1 = k3}"
    by(auto simp: jurisd_def)
  from 5 6  have 7: "(m, c_mp, p) ∈ jurisd m_bel_ctrl m_nonce_ver" by simp

  hence 8: "(m, c_mp, p) ∈ m_bel'" using d2 by simp
```

2.1) M believes S said $P \overset{K_{PS}}{\rightarrow} S$

```
  have 9: "(s, (p, c_ps, s)) ∈ {(q2, k2). ∃ (q1, c1, p) ∈ m_bel.
                                        ∃ (k1, c2) ∈ m_seen.
```

```
                                        q1 = q2 ∧ k1 = k2 ∧ c1 = c2}"
  using i3 m2 by blast
from 9 2 have 10: "(s, (p, c_ps, s)) ∈ msg_mean m_bel m_seen" by simp
```

2.2) M believes $P \overset{K_{PS}}{\rightarrow} S$

```
hence 11: "(s, (p, c_ps, s)) ∈ m_nonce_ver" using d1 by(auto simp: nonce_ver_def)
hence 12: "(p, c_ps, s) ∈ {k3. ∃ (q1, k1) ∈ m_bel_ctrl.
                               ∃ (q2, k2) ∈ m_nonce_ver.
                               q1 = q2 ∧ k1 = k2 ∧ k1 = k3}"
  using i5 by auto
from 12 6 have 13: "(p, c_ps, s) ∈ jurisd m_bel_ctrl m_nonce_ver" by simp

hence 14: "(p, c_ps, s) ∈ m_bel'" using d2 by simp
```

3.1) P believes M said $P \overset{K_{PS}}{\rightarrow} S$

```
have 15: "(m, (p, c_ps, s)) ∈ {(q2, k2). ∃ (q1, c1, p) ∈ p_bel.
                                         ∃ (k1, c2) ∈ p_seen.
                                         q1 = q2 ∧ k1 = k2 ∧ c1 = c2}"
  using i1 m3 by blast
have 16: "msg_mean p_bel p_seen = {(q2, k2). ∃ (q1, c1, p) ∈ p_bel.
                                             ∃ (k1, c2) ∈ p_seen.
                                             q1 = q2 ∧ k1 = k2 ∧ c1 = c2}"
  by(auto simp: msg_mean_def)
from 15 16 have 17: "(m, (p, c_ps, s)) ∈ msg_mean p_bel p_seen" by simp
```

3.2) P believes $P \overset{K_{PS}}{\rightarrow} S$

```
hence 18: "(m, (p, c_ps, s)) ∈ p_nonce_ver" using d3 by(auto simp: nonce_ver_def)
hence 19: "(p, c_ps, s) ∈ {k3. ∃ (q1, k1) ∈ p_bel_ctrl.
                               ∃ (q2, k2) ∈ p_nonce_ver.
                               q1 = q2 ∧ k1 = k2 ∧ k1 = k3}"
  using i4 by auto
have 20: "jurisd p_bel_ctrl p_nonce_ver = {k3. ∃ (q1, k1) ∈ p_bel_ctrl.
                                               ∃ (q2, k2) ∈ p_nonce_ver.
                                               q1 = q2 ∧ k1 = k2 ∧ k1 = k3}"
  by(auto simp: jurisd_def)
from 19 20  have 21: "(p, c_ps, s) ∈ jurisd p_bel_ctrl p_nonce_ver" by simp

hence 22: "(p, c_ps, s) ∈ p_bel'" using d4 by simp

from 14 22 show "(p, c_ps, s) ∈ (m_bel' ∩ p_bel')" by simp
qed
```

Bibliography

[1] Andrew Baumann et al, "The Multikernel: A new OS Architecture for Scalable multicore systems," 2009.

[2] The IEEE and The Open Group, "The Open Group Base Specifications Issue 7, 2016 Edition," 2016.

[3] M.K. McKusick, G.V. Neville-Neil, R.N.M. Watson, *The Design and Implementation of the FreeBSD Operating System, Second Edition.* Addison-Wesley, 2015.

[4] The Microsoft Developer Network. `_spawn, _wspawn` Functions. [Online]. Available: https://msdn.microsoft.com/en-us/library/20y988d2.aspx

[5] Team Barrelfish, "Barrelfish Architecture Overview, Technical note 000," 2013.

[6] Leslie Lamport, *Specifying Systems.* Pearson Education, 2002.

[7] ——. (2015) The PlusCal Algorithm Language. [Online]. Available: http://lamport.azurewebsites.net/tla/pluscal.html

[8] ——. (2011) TLC – The TLA+ Model Checker. [Online]. Available: http://lamport.azurewebsites.net/tla/tlc.html

[9] M. Burrows, M. Abadi, R. Needham, "A Logic of Authentication," 1990.

[10] A.C. Bomberger, N. Hardy, A.P. Frantz, C.R. Landau, W.S. Frantz, J.S. Shapiro, A.C. Hardy, "The KeyKOS Nanokernel Architecture," 1992.

[11] N. Hardy, "KeyKOS Architecture," 1985.

[12] United States Patent 4 584 639.

[13] Trustworthy Systems Team, Data61. (2017) seL4 Reference Manual Version 6.0.0. [Online]. Available: http://sel4.systems/Info/Docs/seL4-manual-latest.pdf

[14] The University of Cambridge and Technische Universität München. (2016) Isabelle. [Online]. Available: https://www.cl.cam.ac.uk/research/hvg/Isabelle/

[15] Tobias Nipkow. (2016) Programming and Proving in Isabelle/HOL. [Online]. Available: https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2016-1/doc/prog-prove.pdf

[16] Team Barrelfish. (2016) Harness. [Online]. Available: http://wiki.barrelfish.org/Harness

[17] P.J. Schweitzer, G. Serazzi, M. Broglia, "A survey of bottleneck analysis in closed networks of queues," 1993.

[18] Y. Bard, "Some Extensions to Multiclass Queueing Network Analysis," 1979.

[19] M. Reiser, S.S. Lavenberg, "Mean-Value Analysis of Closed Multichain Queuing Networks," 1980.

[20] J.D.C. Little, "A Proof for the Queuing Formula: L = λW," 1961.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PROCESS MANAGEMENT IN A CAPABILITY-BASED OPERATING SYSTEM

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
DAMACHI	RĂZVAN - GABRIEL

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, 29.08.2017	Damachi R.

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.