Förderkennzeichen: 01IH130003
Vorhabensbezeichnung: MyThOS
Modulares Betriebssystem für Massiv Parallele Anwendungen

MyThOS Base Architecture ver.2

Robert Kuban, Randolf Rotta

16th August 2016

Contents

1 Preface

The only way to avoid these traps is to encourage a software culture that knows that small is beautiful, that actively resists bloat and complexity: an engineering tradition that puts a high value on simple solutions, that looks for ways to break program systems up into small cooperating pieces

Eric S. Raymond, The Art of UNIX Programming

1.1 Issues with Previous MyThOS Designs

(1) The object capabilities were present as a translation table from user-visible logical identifiers to actual kernel-level pointers. However, these translations did not cooperate with the resource management. Hence, it was possible to delete kernel objects without cleaning up all dangling references. Deletion could overlap with concurrent method calls to the deleted object.

(2) The object capabilities translated just to kernel objects and any access control was implemented by pointing to a more restricted interface. Combining all relevant access rights, for example readable, writeable, delegatable, referencable, and so on, leads to a combinatorial explosion of interface variants.

(3) Each execution context (user thread) required a dedicated kernel stack that was used for the entry from system calls and interrupts. These stacks were mostly unused because the kernel's task scheduling had to switch to the hardware thread's main stack anyway. Allocating and mapping these kernel stacks into the kernel space increased the thread creation overhead.

(4) All kernel objects were placed into specific expandable logical address ranges in the kernel space. The implied dynamic kernel address space management introduced dependencies to physical memory allocators for the on-demand creation of page maps. This introduced non-deterministic overheads whenever the kernel space page maps had to be updated.

(5) In order to improve the scalability of the kernel's internal memory allocators, private memory pools were used at each hardware thread. Because of the limited memory capacity, these pools had to allocate from a globally shared reserve anyway. The assignment of resources to these pools and the return to the global pool required complex policies, statistics, and algorithms. Nevertheless these strategies did not match the application's actual needs.

(6) The implicitly allocated kernel-space objects had no user-visible address. Hence, there was no mechanism to clean up and recycle these resources on demand. Smart pointers with reference counting were used at some places but did not cooperate well with the translation of user capabilities to kernel objects. Especially, it was easy to create cyclic dependencies that would never get cleaned up. This could have been solved with a much more restrictive process-thread model, tying the kernel and supervisors to a quite limited policy.

(7) The high-level stubs for system calls tried very hard to resemble a homogeneous symmetric remote method invocation model similar to, for example, JavaRMI. This made the actual implementation of kernel objects and services overly difficult because the generated kernel-side stubs either hid too much context information, for example the calling process and numeric object handles, or pre-processed the arguments more than actually needed by the kernel object.

(8) The generator for the high-level stubs increased dependencies on complex tools like the python-clang binding. Despite all the help from modern compilers, extracting interface definitions from the kernel's C++ code had a hard time to separate interface-relevant information from unrelated definitions. This polluted the user-space client stubs with arbitrary kernel-internal definitions.

(9) Like with the widely used locks for critical code sections, all synchronisation was in the responsibility of the users/callers of kernel objects and services like the address space management. However, often the caller side did not know the exact synchronisation requirements , or the correct core, that was supposed to process the requests. Therefore, some synchronisation was overly pessimistic while some was insufficient and led to race conditions.

(10) The kernel-level asynchronous communication between hardware threads was unbounded based on the on-demand allocation of message buffers. This introduces hidden overheads for the allocation and deallocation of messages. The missing flow control allowed misbehaving code to use up all kernel memory. Likewise, the asynchronous inter-process communication service allowed misbehaving applications to deplete all kernel memory.

(11) The kernel operated without masking the interrupts. All data structures based on atomic operations required careful design in order to avoid deadlocks created by interrupts at intermediate steps—especially when interrupt handlers might access the same data structures. Hence, complex wait-free data structures had to be used instead of simpler lock-free and obstruction-free variants. In consequence, interrupts had to be disabled at quite some places anyway.

1.2 Issues with Existing (Micro-)Kernels

(a) Synchronous IPC and synchronous system calls: Long running operations such as resource cleanup, e.g. a capability revoke, block the calling thread. In order to offload such work to another thread, a complex combination of asynchronous notifications and user-level communication via shared memory is needed.

(b) Big Kernel Lock:

(c) Read-Copy-Update:

(d) Frequent context switching:

(e) Not all microkernels are as small and simple as desirable for many-core processors: Kernel-level on-demand memory allocation and deallocation add considerable complexity. The separation between core kernel components and platform- and application-specific extensions is not always clear, which hampers the extensibility.

(f) Licensing: While some kernels are available under BSD-alike licenses, we repeatedly found crucial code under GPL license, for example in code generators for stubs and data structure access.

1.3 Key Aspects of the Presented Design

- No on-demand memory allocation and deallocation inside the kernel: objects are created only through user requests (system calls) and the user supplies the needed memory pool.

- No logical/virtual memory in the kernel space: after booting, the kernel's logical to physical address translation tables are not extended, which removes the need for on-demand page maps. The kernel operates inside a 4GiB direct-mapped address range, which removes the need to differentiate between physical and logical addresses and enables compact 32bit pointers.
- No direct access to user-space memory: the user tells the kernel which frames will be used for sharing between kernel and user space. The kernel accesses the physical frames directly, which removes all in-kernel page faults and dependency on the user-space address translation.
- No interrupts during usual kernel-mode execution: Just the kernel's task scheduler contains a preemption point to check for pending interrupts. All kernel tasks are very short running or decompose into multiple tasks and can be implemented more efficiently when interrupts do not need to be feared.
- Unified smart pointer and weak reference mechanism, includes precise tracking of resource inheritance. This is used for long-term storage of references and enables to inform all affected reference holders and tear down all contained sub-resources.
- Unified translation from user handles to kernel objects: Separate translation table implementations for the various types of user-visible kernel objects are no longer needed. Instead the user space references any type of kernel objects through capability pointers and these are translated into kernel objects and access rights through per-process capability spaces. The capability spaces are the user's entry point to the kernel's unified resource management and prevent premature deletion of kernel objects.
- Simplified software layers: The synchronisation mechanisms do no longer need to differentiate between purely local synchronisation and remote communication. It is no longer necessary trying to choose a appropriate memory allocators in various places all over the kernel, which often fails because information about future sharing of the allocated memory is not available. Now, only factories for kernel objects allocate memory and are passed appropriate memory allocators by the caller.
- No implicit sharing of kernel objects between processes, user threads, and hardware threads. Through the explicit allocation of kernel objects and user-driven partitioning of the physical memory pool, the applications have full control over all aspects of sharing.

- Non-blocking synchronisation in shared kernel objects: A delegation mechanism based on small task objects unifies various types of synchronisation monitors and provides lightweight non-blocking mutual exclusion. The monitors automate the flow control such that no on-demand allocation of message/task buffers is necessary.
- Unified event-style task execution model: System calls, interrupt epilogues, delegated critical sections, asynchronous method calls, and the return to user-mode execution are expressed through small tasks and are scheduled by a lightweight scheduler on each hardware thread.
- The kernel code does not hide that all addresses are in the direct mapped part of the kernel space. Physical addresses can be easily accessed by adding the direct mapping base address and physical addresses can be obtained from kernel addresses by subtraction.

1.4 Relation to seL4

It is undeniable that the design of MyThOS as presented in this document borrowed many clever ideas from the seL4 design and implementation. We are very thankful for the inspiring papers and presentations of Gernot Heiser et al. and the decades of collective experience on which this work could build upon. Many projects, for instance also the Barrelfish multikernel, learned a lot from seL4, which helped to speed up development and allow to focus on specific key aspects. Despite the many similarities, MyThOS features a few interesting differences, which we hope to be very useful for the targeted HPC application scenarios:

- seL4's prefix serialisation of the capability derivation tree is used in MyThOS for the prefix serialisation of the resource inheritance tree. We analysed the inheritance rules and successfully separated the tree management from any type-specific knowledge about kernel objects. Kernel objects implement a simple C++ interface in order to take part in the kernel's resource management.
- seL4's compact storage of in-kernel pointers was applied to the resource inheritance tree in order to achieve a very compact representation even on 64-bit processors. By coincidence, the 64-bit variant does not consume more memory than seL4's 32-bit variant.

- MyThOS replaces seL4's widespread use of switch-case type dispatches by C++ virtual methods of the kernel object interface. This enables the easy introduction of new kernel objects for specific use cases and hardware-specific device support. Whether the virtual method dispatch is faster than seL4's long chains of conditional branches has to be seen. Of course, such vtable-based dispatches complicate formal verification very much.
- The Untyped Memory objects in seL4 use a simple watermark scheme. This simplifies verification but leaves significant (internal) fragmentation when large kernel objects like page maps require a specific alignment. The MyThOS design allows for the use of in-place free lists and similar heap structures inside the Untyped Memory objects. It still needs to be evaluated which variant balances overhead and compactness well enough.
- Through the monitors and scheduling contexts, the MyThOS kernel has an understanding of the processor's distributed topology. This makes it possible to assign kernel services and application threads to dedicated cores.
- MyThOS has no big kernel lock and allows concurrency inside the kernel. Lightweight non-blocking synchronisation and delegation are used to protect shared kernel objects. However, the decision to share kernel objects between hardware threads is a pure application choice. The kernel can be operated like a multikernel or a clustered multikernel. Of course, respective implementation variants should be selected to skip superfluous synchronisation.
- Based on the scheduling contexts, the kernel's inter-process communication supports deferred synchronous communication between different hardware threads. Most L4-based kernels support just asynchronous notifications based on inter-process interrupts between hardware threads.
- MyThOS separates the sender-side communication portal from the user thread. This allows applications to set up multiple portals per thread in order to enable multiple overlapping deferred synchronous system calls and IPC messages. We hope that this simplifies the implementation of user-level asynchronous programming frameworks.
- The resource management around the inheritance tree, kernel objects, and factory objects is designed to simplify the introduction of new kernel objects with minimal impact on the existing base kernel.

2 Architecture Overview

Unix's syscalls all are synchronous. That makes them a bad target for a microkernel, and the primary reason why Mach and Minix are so bad - they want to emulate Unix on top of a microkernel. Don't do that.
If you want to make a good microkernel, choose a different syscall paradigm. Syscalls of a message based system must be asynchronous (e.g. asynchronous IO), and event-driven (you get events as answers to various questions, the events are in the order of completion, not in the order of requests). You can map Unix calls on top of this on the user side, but it won't necessarily perform well.

Bernd Paysan, alt.os.multics, 2003.

This chapter discusses the global architecture and gives an overview about the basic abstractions and dynamics of the system. Subsequent chapters focus in more detail on individual subsystems.

The following sections follow the 4+1 view model of Kruchten [Kru95]. The first section summarises the requirements as Application View. Section 2.2 describes the logical view, which consist of common services, mechanisms, and design elements that fulfill the functional requirements. The physical view in Section 2.3 discusses the deployment with respect to the placement and replication of components onto processors and hardware threads as well as the basic memory layout. Subsection 2.4 discusses the dynamical view, which focuses on interactions inside the kernel, between applications and kernel, and between applications as well as the life-cycle management of components and the necessary scheduling. Finally, the development view in Section 2.5 describes implementation aspects, for example, the files and folders structure and the configuration management via code modules.

2.1 Application View

The target application domain of MyThOS is dynamic and elastic high-throughput parallel computing applications on many-core processors. For example, complex multi-physics simulation codes exhibit multiple coupled applications operating in turns and in parallel on shared data. *Invasive computing* [THH+11, OSK+11] tackles the difficult load balancing by trading cores dynamically between the individual applications. Similarly, new applications in big data, deep learning, and the internet of things raise the need for *elastic* compute clouds [RKZB11]. In addition to the dynamic assignment of cores to compute tasks, these scenarios require effective isolation and supervised communication between third-party compute codes.

The requirements from the application's viewpoint divide into three general types of user-space activities: actual applications that perform computations, supervisors that globally coordinate between the applications, and basic system services that provide a shared infrastructure. The supervisors require mechanisms to exert control over applications such as assigning and revoking physical memory, storage in general, compute cores, and communication channels. This includes the ability to preempt and migrate application threads as well as to intercept protection violations and similar events. In this way, policies such as the mapping of application threads to physical cores and invasive computing [THH+11] can be implemented. Beyond this, the three types share requirements with respect to the management of threads, address spaces, shared memory, and communication.

Application threads provide logical control flows that are scheduled for execution by the operating system and operate in an application-specific environment. Interfaces are needed to create/delete and suspend/wakeup application threads. The thread's execution environment is composed of a logical address space, thread-specific data (stack, basic communication buffers, thread-local data) inside this address space and access rights to system services and communication channels. Hence, an interface is needed to map actual memory into these address spaces and configure the processor's support for thread-local data. On many-core processors, an additional core-local storage relative to physical hardware threads seems useful.

Basic communication support by the OS kernel is needed for inter-process communication (IPC) between application, supervisors, and system services, e.g., to communicate requests and set up high-level communication. If system services are partially implemented by kernel objects instead of user-space threads, a uniform interface is needed to hide the implementation-specific differences. High-volume data transfers are usually facilitated via shared memory, which requires means to setup shared memory across address spaces. In this context, asynchronous and optionally preemptive notification mechanisms are useful in order to avoid busy polling on shared message buffers. The actual communication between application threads is up to the application's parallel programming runtime.

2.2 Logical View

This section describes the logical view, which consist of common services, mechanisms, and design elements that fulfill the functional requirements of the system.

From the application's point of view, abstractions for communication with the operating system services, for communication with other applications, and for the management of their execution environment are needed. These abstractions define the overall system's style and versality. For example, Unix-based systems expose a large number of kernel functions via the system call interface. Where a reference to a specific instance, for example a file, is needed, a numeric handle, for example a file descriptor, is passed via the call arguments and the kernel function looks up the instance in a type- and process-specific table. Other designs focus directly on these instances and let the application communicate with them via a small set of system calls, see for example [Lie96]. This style is closer to method invocations on kernel objects. A major advantage is, that the management of instances can be unified while giving the application/user more explicit control.

2.2.1 Vertical Layers and Object Types

Different styles of interaction between objects can be found inside the kernel. This is caused by the differentiation between purely local actions that do not leave their hardware thread, communication mechanisms that coordinate the kernel's activities across hardware threads, and the communication between user-mode applications and kernel via system calls and invocation messages.

This section introduces vertical layers of the software architecture in order to simplify the later discussion of the respective interaction styles. The interaction styles also constrain the object model in order to lead to interoperable interfaces. A short summary of the layers and object models in shown in Figure 2.1.

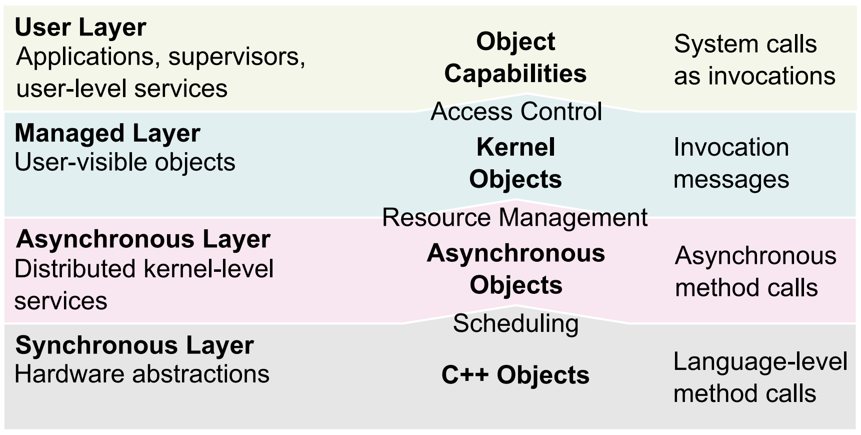


Figure 2.1: Vertical layers and object types.

Synchronous Layer: This is the lowest, most basic layer of the architecture. Ordinary *C++ objects* are used to implement fundamental data structures and basic hardware abstraction components. These objects are used via the ordinary C++ method and function calls and their implementation is executed immediately in the current logical control flow.

This layer is responsible to provide the runtime environment that is needed by the asynchronous layer. This includes the scheduling of asynchronous activities across hardware threads and mechanisms for concurrency and locality control. *Tasklet* objects provide the abstraction for asynchronous activities and *monitor* objects implement asynchronous synchronisation policies such as mutual exclusion.

Asynchronous Layer: On top of the runtime environment from the synchronous layer, this layer hosts *asynchronous objects* that provide communication via asynchronous method calls. In contrast to arbitrary C++ object methods, the asynchronous methods consume a Tasklet as small state buffer and a reference to an asynchronous response handler object. The actual implementation of the called method can be executed at a later time and also on a different hardware thread. The call returns by calling a respective asynchronous response method, passing along the Tasklet.

This layer implements shared kernel infrastructure and is not directly visible to the user. Examples are the *resource inheritance tree* and supporting asynchronous objects for the implementation of kernel objects.

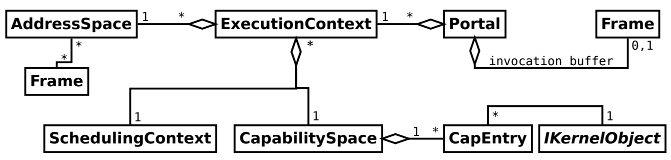


Figure 2.2: Interaction between the basic kernel objects.

Managed Layer: This layer manages the resources and life cycle of *kernel objects* through *capabilities* as an unified smart pointer mechanism. The capabilities track the resource inheritance beginning from memory ranges over kernel objects allocated in this memory to derived weak references that point to the same kernel object. In cooperation with the memory management this inheritance enables the clean deallocation of kernel objects and recycling of the respective memory. Alongside the basic state information that is used by the kernel's resource management, the capabilities include generic and type-specific access rights. These are used by the system call interface to restrict the user's capability invocations. Such invocations are forwarded as asynchronous method call's to the *invoke()* method of the targeted kernel object.

See also 2.5.4

This layer contains all user-visible and call-able system services, which are introduced in Section 2.2.2. A small set of system calls allows to push capability invocations from the application to the kernel objects.

Application Layer: Applications, supervisors, and other system services live in the application layer. Application threads interact with *kernel objects* via *capability pointers*, which are logical indexes into the thread's capability space. Communication on the application layer is possible via shared memory in the logical address spaces and via inter-process communication (IPC) services of the kernel. Libraries can introduce various middleware layers for higher-level parallel abstractions.

2.2.2 Core Abstractions: Kernel Objects

Kernel objects are the smallest components with a managed life cycle. Figure 2.2 shows a class diagram of the interactions between the core kernel objects.

Address Space: The logical address space configuration, that is the translation from logical to physical addresses and the access protection, is represented

See also 2.3.3

through Address Space kernel objects. They allow to manipulate the protection flags, map physical memory frames into a logical address space, and unmap address ranges. The Address Space is responsible to react to the deletion of mapped frames, which happens through the resource revocation mechanism and unmaps the frame. The TLB invalidation on all affected hardware threads has to be implemented by the owner of unmapped or revoked frames because the user-level code has to keep track of the used address spaces anyway.

Capability Space: These manage the application's access to kernel objects by implementing a mapping from numerical capability pointers to capability entries. These entries contain the pointer to the kernel object, meta-data such as access rights, and information about the inheritance history for the resource management.

Execution Context: User-mode threads are represented by Execution Contexts. Each execution context is associated with an address space, a capability space, and a scheduling context. It is responsible to track the execution state (waiting, running, preempted. . .) and the user-mode processor state (register contents, the thread-local storage configuration. . .). The address space, capability space, and scheduling context can be shared between many threads and arbitrary combinations are possible.

See also 2.3.1 **Scheduling Context:** Scheduling contexts manage the processing time of execution contexts on hardware threads. It is their responsibility bring ready execution contexts into actual execution by scheduling them on a selected hardware thread. In the basic variant of MyThOS, each scheduling context represents a single hardware thread and the associated execution contexts are scheduled only on this thread with FIFO order. Thread migration is achieved by rebinding execution contexts to other scheduling contexts.

See also 2.4.4 **Portal:** Portals facilitate the deferred synchronous *communication* between execution contexts as well as between applications and kernel objects. Each execution context needs a portal in order to issue capability invocations, send and receive IPC requests, and send responses. Portals hold a capability to an *invocation buffer* frame, which is used as shared message buffer between application and kernel. Each portal is associated with one execution context, which is resumed whenever a message arrives at the portal.

Frame: In order to map physical memory into an address space, especially to established shared memory across address spaces, a handle to such physical memory regions is needed. These are represented by frames as contiguous well-aligned ranges of physical addresses. The frame objects are responsible for tracking the memory's use in address spaces and enforce effective revocation by unmapping themselves from all affected address ranges. Frame capabilities can be inherited into subranges of the frame.

Untyped Memory: Appropriate (physical) memory is needed in order to create new kernel objects. These memory ranges cannot be frames, because this would allow to map the memory of kernel objects into user-level address spaces. Hence, untyped memory is used as origin of the available physical kernel memory. Like frame objects, untyped memory objects represent contiguous ranges of physical addresses. They are responsible to manage the used versus free parts of the range. Resource inheritance is used to split free untyped memory into smaller portions. This is the key ingredient to the user-controlled distributed concurrent allocation of kernel objects.

See also 2.3.4

Base Factory: Converting untyped memory into actual kernel objects is achieved by invoking factory objects. The base factory is responsible for all the basic kernel objects as described in this section. The creation of custom, application-specific kernel objects is facilitated by additional factories or by replacing the base factory implementation.

Discussion. All kernel object that are purely invoked by the user through capabilities can easily be replaced by portals. This forwards the invocation to a user-space supervisor, which just has to implement the interfaces that the client expected and respond with according result messages. Likewise, a supervisor can establish man-in-the-middle proxy portals between applications and use these to monitor and manipulate all communication and capability transfers. On the opposite extreme, performance-critical services can be moved from the user-space supervisors into dedicated kernel objects without any visible differences for the supervised applications.

Table 2.1 gives an overview of related concepts in other operation systems. Nova's Protection Domains [SK10] are the fixed combination of an Address Space and a Capability Space. This might be caused by Nova's recursive page mappings. Similarly, Unix systems combine both into processes and enforce that each process always contains at least one user thread. In comparison, enforcing this tight combination requires to implement respective policies inside the kernel while the benefits are small. One motivation between the Unix process+thread combination was the easier

Table 2.1: Comparable concepts in seL4, Nova and Unix-based systems.

MyThOS	seL4	Nova	Unix
Address Space	Page Table	Protection Domain	Process
Capability Space	Cnodes		
Execution Context	TCB	Execution Context	Thread/TCB
Scheduling Context	?	Sched. Context	Scheduler
Untyped Memory	Untyped Memory	frame pool	frame pool
Frame	Frame	pages	mapped files
Portal	Endpoint	?	Sockets etc.

memory cleanup when threads exited. This is addressed by MyThOS in a more unified way through the resource inheritance and revocation mechanisms.

In seL4 [HE16] following naming scheme is used: Execution Contexts are Thread Control Block (TCB), Capability Spaces are Cnodes. The seL4 Endpoints are similar to our Portals on the receiver side. However, MyThOS Portals are used at the client and server side to send and receive requests and responses. The client-side portals enable a limited degree of asynchronous operation by using multiple portals from within the same execution context.

The seL4 design exposes the individual page table structures as Page Directory, Page Table kernel objects and so on. These objects just verify that page tables are constructed correctly while leaving other policy aspects such as, for example, sharing or replication of address space segments to the user. The adress spaces in MyThOS follow this example because it provides better control over address space sharing and simplifies the kernel. Note that the current seL4 design does not seem to support the sharing of page tables and address space fragments.

Finally, seL4 combines the management of usable memory and factory-alike allocation inside the untyped memory objects. With a fixed set of kernel object types such as in seL4, this strategy is sufficient. However, a separate allocator is easier to implement in order to support custom kernel objects and replaceable implementation variants of the basic kernel objects.

2.2.3 Concurrency Control: Tasklets, Monitors, Lock-Free Algorithms

In a system with many concurrently running hardware threads, multiple threads can perform system calls in parallel. This introduces a consistency challenge because one thread could manipulate or delete a kernel object while another thread tries to use it. In addition, interrupt signals and traps from the processor and devices can preempt the execution of a hardware thread and jump into kernel mode. Preemption of user-space activities looks very similar to entering the kernel via system calls. But preemption of kernel-mode execution can introduce subtle deadlocks and inconsistencies. Hence, a concurrency control strategy is needed that ensures consistency of all kernel state.

The MyThOS kernel does not allow on-demand allocation. In consequence, no message objects, state copies, or kernel threads can be allocated on the fly. The concurrency control has to operate with statically allocated resources. This excludes unbounded asynchronous messages, read-copy-update mechanisms, and cooperative kernel threads for lock-based critical sections. The kernel should not wait busily in fine grained locks. Event-style execution models can be a useful alternative if all aspects of the system design and implementation respect their limitations and constraints. This has the additional benefit, that single-stack kernels need an event-based model anyway.

- All asynchronous operations have to be deferred synchronous cycles. Requests transfer the logical control flow and responses return it.
- Transitional states and the transitional use of internal resources has to be protected by a flow control mechanism. The next incoming request can be processed just after the all outstanding responses of previous deferred synchronous cycles have been processed.
- There is just a finite number of entry points into the kernel (system calls and interrupts) and each entry point can initiate just a finite number of logical control flows.
- Forking of logical control flows requires the ownership of statically allocated resources.

In MyThOS, *asynchronous objects* are the main synchronisation mechanism throughout the kernel, providing mutual exclusion, operation ordering, flow control, and bounded asynchronous communication. These objects provide *asynchronous methods*, which are called like C++ methods but are actually executed later and possibly on a different hardware thread.

See also 2.5.3 **Tasklets:** The signature of asynchronous methods follows a specific pattern: a reference to a *Tasklet* object is passed as temporary storage for the asynchronous execution state. The Tasklet objects have a fixed structure that enables enqeueing into queues, reflecting about source and destination of the task, and a handler function with a limited space for additional state data or arguments.

See also 2.3.1 **Monitors:** Inside the asynchronous methods, *monitor* objects are used to implement the actual synchronisation and communication. Different variants are available to choose between simple mutual exclusion, multiple reader/single writer, response-before-request synchronisation patterns. The monitors do not need dynamic memory management because they use the forwarded tasklet argument to manage their processing state and communication.

It is the monitors responsibility to bring enqueued tasks into actual execution on a suitable hardware thread. As example implementation, each monitor is bound to a hardware thread and all Tasklets are enqueued into a task queue at this thread. This ensures mutual exclusion. Priorities like preferring the processing of responses over beginning of new requests can be implemented by a respective priority hierarchy of task queues at each hardware thread. A different implementation, similar to delegation queue locks, uses a Tasklet queue inside the monitor. Whenever a new task is enqueued as first task the hardware thread has mutually exclusive access and can process the task directly or enqueue the monitor into the threads task queue.

Discussion. The seL4 kernel is in a very similar situation because it disallows on-demand allocation too. Their solution is to disallow concurrency and preemption inside the kernel. Hence, no critical section are present and a kernel monitor is also not necessary. This strategy exploits that all system calls are very quick and the interrupt latency is small if there are just a few hardware threads. With many-core processors the number of hardware treads that compete for the big kernel lock might be too large and severely limit the systems scalability. Therefore, MyThOS trades latency of individual system calls for higher scalable throughput of the overall system.

The life cycle of a Tasklet can be interpreted as a credit-based flow control: Client objects can issue asynchronous calls only if they own a currently unused Tasklet. The request call passes the Tasklet credit to the server object and, there, it can be used for further internal asynchronous processing. Finally, the response call returns the

Tasklet and the associated credit back to the client, which enables the client to issue a new asynchronous request.

The Tasklet life cycle can also be interpreted as a logical control flow. The request call forks the logical control flow and the Tasklet is used as message from client to server. Inside the server, the Tasklet stores the logical control flow's state. Finally, the response call finishes the logical control flow and the Tasklet acts as message from server to client.

Note that asynchronous methods are not forced to use monitors. If their duty can be achieved via wait-free data structures or similar atomic memory operations, they can be used instead. Asynchronous objects may use read-copy-update mechanisms internally and just return the tasklet by immediately calling the response method. The important point is, that this implementation decision is transparent to all users of such asynchronous objects.

The monitors are implemented on the synchronous layer where just ordinary synchronous C++ method calls but no mutual exclusion or locking mechanisms are available. Here, synchronisation and communication primitives of the hardware have to be used. For example, atomic compare-and-swap, fetch-and-add, and exchange operations on shared memory are sufficient to implement the wait-free FIFO queues needed for the monitors.

Related Work. Many operating system kernels split the handling of preemptive interrupts into a synchronous prologue and an asynchronous epilogue, which is executed before returning to the user mode. In the asynchronous model of MyThOS, the prologue calls an asynchronous method that acts as epilogue. The called object's monitor takes the role of the kernel's monitor on an object granularity. Therefore, the prologue needs a Tasklet and can reuse it for scheduling the next epilogue only after the previous epilogue was executed and has returned the Tasklet. Because overlapping interrupts are aggregated anyway, the interrupt source can be masked until the tasklet is returned. This corresponds to masking the interrupt source until the end of the epilogue.

Some operating systems, like Linux and Mach, used a big kernel lock to prevent concurrent entry into kernel-mode execution. In order to increase the scalability on multi-core processors, the big kernel lock has been replaced by fine grained locks on individual data structures or subsystems, lock- and wait-free data structures [BWCM+10], and read-copy-update transactions [MJK+02] that retain consistency without delaying execution. Kernel-mode interrupts are handled by splitting the reaction into a prologue and epilogue. The prologue is executed immediately by the processors preemption mechanism and appends a respective prologue, which is executed before leaving kernel-mode.

Most L4-based operating system kernels use a big kernel lock to protect all kernel objects against concurrent access and, except for selected preemption points, interrupts are disabled as well. This is sufficient for a small number of hardware threads because the time actually spent inside the kernel is very short [HE16]. However, this does not scale to larger numbers of hardware threads. The multikernel design obliterated any concurrency inside the kernel [BBD+09]. Each hardware thread operates only on local private kernel objects and, thus, does not need to care about concurrency. Instead, all concurrent actions across hardware threads are outsourced to user-mode. However, as the authors of [BBD+09] note, this introduces additional context switches for non-local IPC, a user-mode capability management responsible for supervised capability transfer between cores, and just shifts the concurrency challenges into these supervisors. The Barrelfish OS uses message passing via shared memory for all communication between the monitor application instances, which act as supervisor and local scheduler. In order to reduce the local replication of kernel objects in shared caches, the clustered multikernel design [vT12] groups multiple hardware threads into a shared kernel that is protected with a big kernel lock.

See also 2.4.3

The monitors can be interpreted as actors [HBS73] that send Tasklets as messages between each other. The request-processing-response cycle between asynchronous objects is inspired by the TinyOS communication model and credit-based flow control schemes. Instead of unlimited asynchronous message queues like in actor models, the degree of asynchronous activity is bounded by the limited amount of statically allocated Tasklets. TinyOS components limit the degree of asynchronous activity on the receiver side through statically allocated task objects. Unlike such bounded mailbox, the MyThOS Monitor's can receive an unlimited amount of tasklets without failing or blocking.

2.2.4 Dynamic Resource Management through Capabilities

One of the main problems that a kernel has to solve is the management of dynamic resources—especially the dynamic allocation and deallocation of objects in the kernel's memory. Which and how much memory is an application allowed to use? What happens if the kernel's memory reserve is depleted? Which kernel objects are

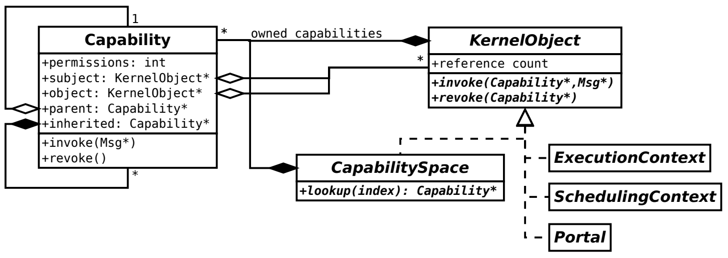


Figure 2.3: Conceptual interactions between capabilities as smart pointers and kernel objects. The implementation stores just a subset of these pointers.

affected when deleting an object? When is it safe to delete objects and which are still used by an application?

MyThOS solves these problems through three principles. First, all kernel objects are designed to operate without any on-demand allocation. For example, the asynchronous communication layer pre-allocates all needed message buffers inside the callers of asynchronous objects. Likewise, the IPC portals communicate with pre-allocated invocation buffers. Hence, the system's normal operation is not impacted by hidden memory management overheads and cannot fail due to depleted resources. Sec. 2.2.3

Second, just the creation and deletion of kernel objects involves dynamic memory management. The respective memory pools are represented via Untyped Memory kernel objects. When an application wants to create a kernel object, it first has to own an Untyped Memory capability with sufficient free space. The boot sequence hands over the initial Untyped Memory. In combination, this steers the applications' use of kernel memory and using up all memory does not affect unrelated applications. Sec. 2.4.9

Finally, based on the concept of object capabilities, the resource inheritance tree implements smart pointers and weak references to kernel objects in combination with tracking the object's resource inheritance. Figure 2.3 shows a conceptual model of the interactions between capabilities and kernel objects. When a kernel object is allocated from an Untyped Memory, a respective Original Capability is created that points to the kernel object and has the Untyped Memory as parent. Therefore, the inheritance tree enumerates all objects that are contained inside the memory region and allows to force the deletion of the contained objects. Likewise, all long term references to kernel objects are stored as children inside the inheritance tree. Sec. 2.5.4

The inheritance acts as a weak reference mechanism with the added benefit of being able to inform the affected subjects about the ongoing deletion. By removing

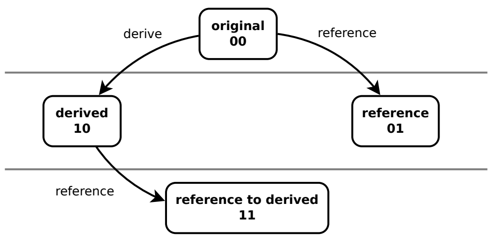


Figure 2.4: Capability Types and their relations. References to derived capabilities can be revoked without revoking other derived capabilities and references to the parent.

all inherited capabilities, the kernel ensures on behalf of the user that no new asynchronous activities can be started on the object. On top of this mechanism, the deferred deletion support of the asynchronous objects layer ensures that objects are not deleted before all deferred activities have completed.

There are four different Capability types with a different purposes. Figure 2.4 illustrates the four types and their relations. Consult Section 2.5.4 for the role of the Capability types in implementation the Resource Inheritance Tree.

Original Capabilities: *manage the life-time* of Kernel Object. When a Kernel Object is created, its Original Capability is created. Whenever the Original Capability is deleted, the Kernel Object will be destroyed. As the *single owner* of an Kernel Object, the Original Capability can only be referenced or derived, which creates a *child* in the Resource Inheritance Tree.

Reference Capabilities: always are a child of an Original Capability and a weak reference to the object its parent manages. Reference Capabilities allow to *share access to an original capability*, such as the right to create Derived Capabilities, without transferring the ownership of the object. Reference Capabilities can be copied, which creates another Reference Capability as a *sibling* in the Resource Inheritance Tree.

Derived Capabilities: always are, like the Reference Capabilities, a child of an Original Capability and a weak reference to the object its parent manages. Unlike Reference Capabilities, Derived Capabilities cannot be copied, but only be referenced, which creates a Reference Derived Capability as a *child* in the Resource Inheritance Tree. Derived Capabilities facilitate *right management* by grouping references.

Reference Derived Capabilities always are a child of an Derived Capability and a weak reference to the object its Original Capability manages. Reference Derived Capabilities can only be copied, which creates another Reference Derived Capability as a *sibling* in the Resource Inheritance Tree.

Discussion. This design is heavily inspired by the seL4 capability model [HE16, GW16], which uses capabilities as resource and access control mechanism. We solved implementation details with respect to the handling of concurrent inheritance tree manipulation and the representation of weak references to mapped frames and mapped page maps differently. seL4 has Original and Derived Capabilities, but introduces a additional layer through Batched Originals for Endpoints.

The capability paradigm of Miller et al. [MTS05] differentiates between subjects, objects, permissions, and invocations. *Subjects* are the smallest units of computation that can hold access rights and *objects* are the smallest units to which access rights can be provided. A *direct access right* gives the subject the permission to invoke the behavior of the object. In this sense, the kernel objects take the role of capability subjects and objects. The user cannot access capabilities directly, that is their instance data structures as C++ object. Instead, a capability space maps from numerical indexes to capability entries. Some capability entries are stored inside other kernel objects and can be used only indirectly.

2.2.5 Physical Memory and Address Space Management

Virtual memory with access control is the most important hardware-level protection mechanisms. Thus, any manipulation of the hardware's address translation tables has to be supervised by the operating system. Otherwise, applications might be able to map foreign physical memory into their own address space and compromise the consistency of the overall system.

The current x86-64 processor architectures support up to three different frame sizes (4KiB, 2MiB, 1GiB) and the address translation employs a four-level page table structure. The following kernel object types are used to represent them:

Frame: A frame represents a contiguous range in the physical address space that is backed by some kind of memory or a memory-mapped device. The range is restricted to certain sizes and respective alignment in order to reduce implementation overhead. Currently these are 4KiB, 32KiB, 256KiB, 2MiB, 16MiB, 128MiB, 1GiB, and 8GiB.

It is not useful to dynamically allocate additional kernel objects whenever smaller frames are derived from a large frame. These objects would inherit resources from *both* the original Frame and an Untyped Memory, which violates the resource inheritance tree. Instead, the *flyweight* pattern [GHJV94] is used, that is Frame capabilities point to the same statically allocated Frame kernel object.

MemoryRegion: For frames that lie outside of the kernel's initial Untyped Memory, a respective base for the flyweight Frame object is needed. These are allocated as MemoryRegion objects and provide the shared Frame object for all frames that are derived from this region.

PageMap: Each page map represents an individual table in an address space structure. Page maps are bound to a specific level during their creation. When mapping a frame or a page map into a page map's entry, the kernel checks the frame size and page map levels. One can map either a page map of the next lower level or a frame of the same level.

The top level table is called *page map level 4* (PML4) and each of the 512 entries represents 512GiB of the logical address space. The PML4 entries point to *page map level 3* tables (PML3 aka page directory pointer table PDP) where each entry can represent a 1GiB page or point to the next level. The entries of the *page map level 2* tables (PML2 aka page directory PD) can represent a 1MiB page or point to a *page map level 1* (PML1 aka page table PT), which contains 512x 4KiB pages.

MappedFrame, MappedPageMap: The owner of a frame retains the right to revoke the frame, which equates to unmap the frame from all affected page maps. The same is true for mapped page maps. In consequence, references from the frames and page maps to their mapped entries are needed. At the same time, page maps reference up to 512 mapped frame or page maps. This many-to-many relation has to be updated when existing mappings are overwritten, page maps are deleted, or frames are revoked. Therefore, MyThOS page maps include the storage for one weak reference per entry and these are implemented through respective capability entries. Again, the flyweight pattern is used to share the kernel object between all mapped frames and mapped page maps.

Discussion The seL4 implementation uses similar abstractions for Frame, MemoryRegion, and PageMap objects. Their 32-bit variant uses some clever tricks in order to fit the physical offset relative to the memory region, the virtual address, access rights, and an address space identifier into the capability's meta-data. Unfortunately, this strategy result in too large capability entries on 64-bit systems. MyThOS shifts some of the information about mapped frames into the page map.

In addition, an indirection through a 2-level address space lookup table is used in seL4. This indirection is necessary because mapped frame capabilities are stored in capability spaces instead of the page map. When the address space is deleted, seL4 has thus no direct mechanism to update the respective mapped frame capabilities. Instead, they later hit on a zombie entry in the address space table. MyThOS stores the mapped frame capabilities inside the PageMap kernel objects, which allows for straight forward updates. This design allows to map page maps and frames into multiple address spaces, which enables shared memory and page map sharing. The downside is, however, that storage for the weak references is wasted when just a few entries are actually mapped.

The deletion of a Mapped-type Capability does not force a *TLB invalidation* on all the hardware threads that use the mapping. In implementing such an behavior would imply traversing large portions of Inheritance Tree in order to find every EC that had access to the mapping, a costly operation that would be carried out redundantly for larger address space reconfigurations. For *HPC applications*, this does not matter, as the application manages itself and is able to issue TLB invalidations only when necessary. In other scenarios, the *supervisor* that manages the address space must track all user-threads that share that the mapping and invalidate the TLB when in doubt.

Exokernels [EKO95] allow the user to directly read tables and only control write access. In later iterations, MyThOS might allow mapping the hardware PML tables read-only in order to provides low-cost access for bookkeeping purposes.

2.2.6 User Access through Capability Spaces

Applications need some access path to kernel objects and not all applications should be able to access any kernel object. Hence, a translation from a per-process or per-thread name space to the process's kernel objects is needed. It should be possible to share these translation fully or partially between threads and between processes.

For the compute cloud scenario, it is useful to hand access to kernel objects from the supervisor to unprivileged applications while keeping the final control of the kernel object in the supervisor. Finally, a mechanism is needed to safely delete kernel objects and revoke access rights.

MyThOS uses an object capability model that brings together the kernel-internal resource management and the various user-to-kernel translation tables. On the resource management level, capabilities are smart pointers to kernel objects combined with some meta-data about access rights and resource inheritance information for clean deallocation. The translation from user-level capability pointers to actual capabilities and kernel objects is facilitated through capability spaces. Each execution context (user thread) is assigned to one capability space.

Capability Map: In order to balance the storage consumption and the actually used name space, the capability spaces are implemented as radix trees.[1] The root and intermediate tree nodes are implemented by *capability map* kernel objects. Each capability map has a fixed power-of-2 size in order to work with static memory allocation. In addition, each map can have a guard prefix bit pattern in order to skip intermediate maps in sparse capability spaces.

Lookup Reference: The pointer translation is ambiguous when looking up capability maps instead of using them as intermediate nodes in the radix tree. For example, the capability pointer `0` could address the first entry of the root capability map or, if this entry points to a capability map, the first entry of this lower map and so on.

This ambiguity is resolved by separating the leave entries and intermediate capability maps through *lookup reference capabilities*. The translation descends recursively only via lookup references to capability maps and any other entry behaves as a leave of the tree even if it points to a capability map. In order to insert or remove nodes to the radix tree, non-lookup references to the respective capability maps have to be used.

Capability Pointer: The user-space applications and supervisors use *capability pointers* as a numerical index into their own capability space. These pointers are a 32bit integer and the root capability map interprets the most significant bits of the pointer. First, the guard bits are checked and, if they match, they are shifted out to the left. Then depending on the map's size, the first most

[1]aka compact prefix trees, https://en.wikipedia.org/wiki/Radix_tree

significant bits are used as index into the own capability entry table and are shifted out to the left as well. The capability entry points to a kernel object and the object's `lookup()` method is used to implement the recursive descend. Leave objects check that the remaining capability pointer is zero. Only the lookup method of lookup references repeats the procedure on the next lower tree node.

Discussion. The node versus leave ambiguity is solved in seL4 by using a separate depth argument in system calls that manipulate the tree structure [GW16].

2.3 Physical View

The physical view discusses the deployment with respect to the placement and replication of components onto processors and hardware threads as well as the basic memory layout.

2.3.1 Locality Control via Monitors and Scheduling Contexts

With a large number of hardware threads and non-uniform hardware topology, it becomes increasingly important to control the placement of both data and tasks. For example, access to remote memory in a NUMA systems can degrade performance of a system considerately by delaying serial program phases and congesting memory interconnects. Even worse, there are low-level operating system tasks that can only be executed on certain hardware threads: configuration registers such as the address space (CR3), devices such as the LAPIC, and configuration operations such as cache flushing and TLB invalidation can only be accessed by the affected hardware thread.

On the asynchronous layer, locality control is exerted through the *monitor* objects. Because monitors provide the interface for the asynchronous execution of method calls, they can forward these calls to kernel-level task schedulers on other hardware threads. Various policies can be implemented and configured via the monitors. This ranges from unconditional delegation to a fixed hardware thread, over load balancing across groups of threads, to opportunistic delegation [FK12] in order to improve the cache locality. The delegation to fixed threads can be used to bind operating system services to dedicated processor cores. On the other end, monitors for purely thread-private asynchronous objects do not not need additional concurrency control mechanisms because the single physical control flow serializes the execution implicitly.

On the managed and application layers, the locality of user-level threads is controlled through the *scheduling context* of an execution context. These are responsible for bringing runnable execution contexts into actual execution on an appropriate hardware thread. The scheduling context implementations use the monitor concept of the lower layers to implement these policies. The simplest variant is bound to a fixed hardware thread. All execution contexts that use such a scheduling context are run on this hardware thread. More complex scheduling contexts can implement load balancing across local groups of hardware threads.

See also 2.3.6

Discussion. In both multikernels [BBD+09] and clustered multikernels [vT12], sharing of kernel structures is restricted to small local subsystems such as individual threads or cores. This greatly reduces the need to further control the locality of kernel-level objects. Because operating system services and device driver are implemented in user mode, their placement can controlled on the level of execution contexts. Likewise, data locality for applications can be achieved by mapping local memory.

Corey [BWCC+08] allows applications to control the locality of kernel subsystems or devices by binding them to physical cores (*pcores*). Moreover, control over sharing of kernel structures, such as file descriptor tables, can be use in order to exploit cache locality.

Shared memory microkernels and pure multikernels have no concept of locality: kernel objects and user threads simply exist and all inter-process communication is performed synchronously within the same hardware thread, that is within the same location. All cross-core scheduling is achieved solely at the user level via shared memory user-level communication. For example, the supervisor application on a hardware thread receives the message to schedule a specific application and, then, does a local IPC to this application. Also, supervisor applications may implement preemptive scheduling by using the local interrupt controllers in order to interrupt other hardware threads. The respective interrupts handlers switch to the local supervisor, which, then, can switch to the actual implementation. In order to provide cross-core IPC at the kernel level, the kernel requires an own concept of locality such as the Scheduling Contexts in MyThOS.

2.3.2 Kernel- versus User-Level OS Services

For a specific operating system service, it is often not obvious whether its more efficient if implemented inside the kernel as kernel object, inside the application as library, or as a separate user-space deamon. Liedtke's principles for L4-style microkernels [Lie96], urge to move as much as possible outside of the kernel except for basic isolation and communication mechanisms. However, frequent context switching between applications, service deamons, and kernel impairs the efficiency because of the simple cores of many-core architectures. In addition, the main challenge shifts from fast local IPC to fast coordination across cores and hardware-threads because of their large number.

One key feature of MyThOS is the support for kernel-level services and drivers in the form of custom kernel objects. This allows to move performance-critical components that cannot be implemented as application-level library into the kernel. Due to the microkernel inheritance, the basic kernel objects are sufficient to run a fully featured software environment. Hence, custom kernel objects are optional and should be designed based on actual performance bottlenecks.

Apart from less frequent context switches, custom kernel object might benefit from the kernel's lightweight synchronisation and communication framework. The flexibility is gained by two strategies: Applications interact with service deamons and with kernel objects via *capability invocation* without seeing the difference. Such invocations can be processed by a custom kernel object or can be forwarded to a user-mode deamon by a portal. On the other hand, custom kernel objects can act as proxies in order to forward kernel-internal asynchronous requests to a portal into user mode by translating them into invocation messages.

2.3.3 Logical Address Spaces: Kernel versus User Space

It is necessary to differentiate two principal types of logical address spaces. The *kernel space* is used by the operating system kernel with full access permissions whereas the *user space* is used by applications with user-mode execution. The construction of user spaces is supervised and restricted by the kernel in order to enforce isolation and protection between independent applications.

Some hardware architectures switch between completely independent kernel and user address spaces on each system call. However, x86-based processors use a shared approach where each address space contains both a kernel and a user space in order to reduce the overhead of system calls. The difference is achieved by a access permission flag that protects kernel-space pages from access during user-mode execution. On x86-64 architectures, the actually 48-bit logical address space has a natural separation into a lower 128TiB half from `0x0` to `0x7FFF FFFF FFFF` and an upper 128TiB half from `0xFF80 0000 0000 0000` to `0xFFFF FFFF FFFF FFFF`.

MyThOS, like most operating systems, uses the upper half as kernel space and the lower half as user space. This affects the Page-Map Level 4 table (`PML4`), where just the 256 lower entries are writeable for the user-space management while the upper 256 entries have fixed values that point to the kernel-space's Page-Map Level 3 tables (`PML3` aka PDPT). These are shared across all hardware threads and are rarely modified by the kernel as discussed in Section 2.3.4. When creating PML4 objects, the upper 256 entries are directly filled with the kernel space default entries. Therefore, later manipulation of the kernel space is transparent to all address spaces and does not require to touch every PML4 object.

Discussion. Some operating system designs propose a single-address space approach for the applications. There, all applications share the same address space layout, that is the same translation from logical to physical addresses. Just the access permisions vary between the applications. One advantage is, that pointers are globally unique because all applications see the same memory at that address. One example is Multics, which used segments to describe access permissions and the segment selector was part of the pointers instead of separate registers like on x86.

On x86 architectures in 32bit mode, segments can be used to put multiple applications into the same logical address translation while hiding this fact from the application's code [SGI14]. However, x86-64 in 64bit mode simply ignores the base addresses of code, data, and stack segments because almost all operating systems were not using segments anyway.

2.3.4 Kernel-Space Memory Management: Physical Memory

In order to be independent of the user address space layout, the kernel has to map the memory of all kernel objects and everything that is accessed from within kernel-mode execution into the kernel space. Access to user space addresses is possible in principle but requires to track which address space the address belongs to and handle page access exceptions. All objects and data structures need to placed appropriately into the kernel space, whether they are created during the boot sequence or dynamically allocated and deallocated during the system' lifecycle. Sec. 2.4.9

All of the kernel's memory allocation is facilitated through Untyped Memory kernel objects. Based on the firmware's memory map, the initial untyped memory is created and, then, used to allocate all initial data structures—not just initial kernel objects. The remaining untyped memory is passed via a capability to the root application.

The kernel-mode code operates on physical addresses. This removes the need to establish dynamically kernel-space page tables, which would require a kernel-space memory management for the on-demand allocation of page table structures. Of course, the processor architecture does not allow to access the physical addresses directly. Therefore, a contiguous *direct mapped range*, similar to an identity mapping, is initialised in the kernel-space during the boot sequence. Physical addresses are accessed by using them as offset into this range.

This direct mapping segment covers the whole physical memory or at least a large enough portion to provide space for the later kernel objects. The covered range is handed as untyped memory to the user. Any additional memory is handed as frame objects to the user. This ensures, that the kernel can directly access all kernel objects without having to fear page access faults.

The kernel can implement an optional memory protection scheme in order to detect bugs earlier. Most parts of the direct mapped range are read&write protected until they are explicitly used for allocation via an untyped memory or via a temporary access to specific physical addresses. This requires separate tables with reference counters in order to prevent revoking access concurrently. These reference counters are held per 2MiB page. Access to smaller objects is simply extended to the enclosing 2MiB aligned range.

The only user-space memory that is accessed by the kernel are the portal's invocation buffers. Here, seL4's strategy is applied in order to avoid all the hassles with kernel-mode page faults: the kernel accesses the buffer directly via its physical address, which is known through the frame object that was registered as buffer at the portal. The user maps the same frame into its user-space. Additional care is necessary to differentiate frames inherited from untyped memory against frames that are not accessible via the kernel's direct mapped range. When registering a buffer frame at a portal, the portal has to check whether the physical address range lies within the direct mapped range. Alternatively, the frame capabilities can contain a kernel access flag that shows which frames are accessible to the kernel.

Custom kernel objects might need to access additional regions of the physical address space, for example to implement device drivers. These ranges are not part of the direct mapped range and, hence, mapping them into the kernel space requires dynamic allocation of page table structures. Note that the mapping is created on user request during the creating of these custom kernel objects. Hence, the user can provide the needed untyped memory.

Discussion. Linux and Unix-based systems have to deal with kernel-mode access to user-space addresses because of the design of many system calls from the POSIX interface. First, each access to user space was protected by a software-implemented page table walk to check whether the address is accessible. This proved to be very slow and mostly unnecessary. In addition, it is not secure against concurrent manipulation of the page tables. Thus, the Linux kernel replaced these checks with a lazy page fault strategy. Just a selected set of functions implements user-kernel data transfers. The page fault interrupt handler checks whether the faulted instruction is part of one of these functions and, in this case, jumps to a respective handler code, similar to the C++ exception mechanism. While efficient and effective, this approach does not answer what the kernel should do when such a page fault happens. It increases the number of possible error conditions that have to be handled by the kernel.

The differences between synchronous and asynchronous IPC are quite relevant for non-local communication. Support for unbounded asynchronous IPC messages, like in Mach, would require dynamic memory management for the unbounded cross-core message queues. Instead, MyThOS provides just bounded asynchronous communication in the form of deferred synchronous IPC via pre-allocated portals. The number of messages in flight is bounded by the existing portals and all necessary data structures were allocated together with these portals.

2.3.5 Core-Local Memory: FS/GS Segment Base

Multiple hardware threads share the same kernel-space address layout. However, each hardware thread requires a few thread-specific data structures such as local task queues and state information about the currently active user thread or execution context. Global variables, that is outside of function bodies and class definitions, and static member functions cannot be used for these structures because, then, all hardware threads would access the same instance instead of their own. Hence, a fast lookup mechanism is needed that translates logical identifiers into the respective hardware-thread-local instances.

This situation is very similar to thread-local storage in application threads and uses the same hardware support. The main difference is, that the kernel's core-local memory is relative to the physical position, that is the hardware thread, whereas the application's thread-local storage is relative to the logical control flow, that is the execution context.

MyThOS uses the base address of the x86 architecture FS and GS segments. This enables fast access to thread-local storage via segment-relative load and store instructions. For historical reasons, the segment descriptors can only hold 32-bit base addresses. These can be used as thread-specific 32-bit offset with the logical identifier as 64-bit base. However, special model specific configuration registers in x86-64 processors allow to override the segment descriptor's base address with a 64-bit address.

The base kernel requires only a limited set of core-local variables that are known at compile time. Because the maximal number of hardware threads is also known when linking the kernel image, a sufficiently large core-local memory segment is reserved in the kernel image. The static addresses to these core-local variables point to the instances of thread 0. The GS segment base contains a 32bit displacement for the hardware thread's instances. Any GS-relative memory access to core-local variables thus uses the sum of the first thread's address to the variable and the actual thread's GS displacement.

This strategy does not require additional address mappings in the kernel-space. In principle, dynamic allocation of additional core-local variables is possible: Unused parts of the linked core-local segment can be used and additional segments can be allocated from untyped memory at run-time. This works because the GS base address always is a displacement offset relative to the beginning of such segments.

Discussion. Several options are available to implement such local storage lookup. Basically, the global logical identifiers are an offset into a thread-specific address range. On some architectures, the compiler reserves a register and this contains all the time the fixed thread-specific base address. Unfortunately, this wastes an expensive register that is more useful for actual computations. The thread-specific base address can also be placed at the bottom of each stack and by limiting the stack size and alignment, the bottom's address can be calculated from the current top of the stack. This costs additional computations and memory lookups while constraining the stack size.

Core-local memory is used by all shared memory operating system kernels. The multikernel design avoids this issue by using a separate kernel address space per hardware thread, which makes all global and static variables core-local. Nova seems to use a dedicated address range in the kernel space. This range is mapped to different physical memory for each hardware thread and, thus, static variables in this range are core-local. However, this prohibits the sharing of PML4 tables across hardware threads. Either, the user would have to create separate page tables on each hardware thread like in a multikernel or the kernel would have to allocate and synchronise core-local PML4 tables implicitly for each logical address space.

2.3.6 Multikernel Support

What is needed in order to implement a multikernel or a clustered multikernel based on MyThOS? In a multikernel, each hardware thread runs its own kernel instance and does not share kernel objects with other hardware threads. Hence, no concurrency control is needed except for dealing with interrupts. A clustered multikernel shares a kernel instance in a small group of hardware threads. This balances the utilisation of locally shared hardware resources such as caches and TLB versus the contention caused by concurrent kernel activities.

The kernel instance is defined completely through its kernel address space layout. Multiple instances can be created during the boot sequence by creating multiple kernel space page tables. The kernel's code segment can be shared because it is read-only. The kernel's segment for global variables and static object instances has to be mapped to separate physical memory and initialised by copying from the first hardware thread. Similarly, the segment for the core-local memory needs to be replicated and mapped to separate physical memory. Finally, the direct mapped range is replicated but maps to the same physical addresses. Mutually exclusive use of this memory is ensured via separating the untyped memory per kernel instance. The kernel trusts the user-mode supervisor to not share the untyped memory across kernel instances.

Sec. 2.4.9

The supervisor on top of each kernel instance is responsible for establishing shared user-space memory across kernel instances for communication and to implement remote capability transfers. When receiving a frame capability, the respective untyped memory range has to be converted into a local frame kernel object. This is simplified by handing the complete untyped memory range to all supervisor instances and let them coordinate who is using which part.

2.4 Dynamical View

This section focuses on interactions inside the kernel, between applications and kernel, and between applications as well as the life-cycle management of components and the necessary scheduling.

2.4.1 Error handling: IPC Messages to Supervisor

The error handling differentiates between asynchronous exceptions and synchronous errors. Asynchronous exceptions preempt the execution of user- or kernel-mode execution and jump to the kernel's interrupt handling. Synchronous errors are detected by assertions during normal operation, for example when a system call fails due to insufficient access rights. Both situations require a recovery strategy that forwards the exceptions and errors to appropriate handlers.

Synchronous errors appear during system calls or are related to (deferred) synchronous calls. They are passed to the application via a normal system call return with the error encoded in the return values.

Asynchronous exceptions arise from interrupts, processor traps, and all faults that cannot be tracked back to a synchronous system call. Exceptions during user-mode execution are transformed into an invocation message and sent to a *exception handler kernel object*. This object is selected based on the affected execution context and is configurable. The basic variant uses a portal that activates an execution context, which can be used to implement supervisor hierarchies. The user-mode handler receives the exception message, can inspect and handle the causes and optionally restart the affected execution context.

Because of concurrent state changes, exceptional situations can occur during kernel-mode execution. These are detected through assertions like synchronous errors but cannot be associated with a specific, currently active system call. For example, a response message can hit a revoked or mis-configured client portal. This exception is not interesting for the server side and, thus, has to be handled by the client's exception handler in the same style as asynchronous exceptions.

Discussion. Unix systems use a signal mechanism to jump into a handler function in the context of the affected thread. Signals are used for both asynchronous exceptions, for example the kill signal, and synchronous exceptions, for example a memory protection fault. This forced jumps inside a single user thread pose some serious problems with respect to stack space and interrupt-alike preemption. Windows NT uses separate handler threads for asynchronous exceptions. Synchronous exceptions are translated into localised signal handlers very similar to the C++ exception mechanism but with the option to retry the affected instruction. MyThOS, like many microkernels, hands control over the exception handling to supervisors instead of implementing a fixed policy. This exception handler could then, for example, implement the affected execution context's jump to an exception code like in Windows or C++.

2.4.2 Concurrent Object Deallocation

See also 2.2.4

In a concurrent system, races between object usage, such as method calls and access to member variables, and object deallocation can easily occur. The capability-based weak references mechanism cleans up long-term references before any object deallocation. However, temporary references in messages and concurrent control flows, for example between capability lookup and using the object, could still lead to race conditions and premature deallocation.

Our solution delays the destruction of objects until all temporary references and in-flight messages are cleared. First, kernel objects are deallocated only when the Untyped Memory they were allocated from is recycled. Before this happens, all long-term references are cleared through revoking all inherited capabilities in the resource inheritance tree.

See also 2.2.3

In order to deal with outstanding messages, pending responses, and references contained inside in-flight messages, the kernel object's Monitor tracks both the number of temporary references and pending messages of the object. Object destruction is facilitated through an asynchronous method and the Monitor executes this call after all temporary references and pending messages are finished. Because all references on the capability layer are already revoked, no new temporary references or pending messages can be created.

This leaves short-term references in concurrent kernel control flows. For example, another hardware thread could have successfully looked up the object pointer before the capability was revoked but still have not issued its request and, hence, is invisible to the object's Monitor. To resolve this race between a thread increasing the counter after reading a soon deleted capability and the deallocation of the object, the memory is only recycled after all hardware threads have either left the kernel or returned to the task scheduler once.

A hardware thread that is not inside the kernel can hold no short-term references. Likewise, returning to the hardware thread's task scheduler ensures that all activities that could have held a short-term reference have completed and have updated the Monitor's reference count. This situation is detected lazily by scheduling a tasklet at every hardware thread that is in the kernel at the time of the Untyped Memory's recycling and waiting for the execution of these tasks. To this end, a broadcast ring of asynchronous objects is used. Hardware threads that are not currently inside the kernel are skipped in order to avoid waking up sleeping threads or needlessly interrupting applications.

Discussion: Reference counting is a well-known concept for lifetime management of objects. One major problem of reference counting in a non-distributed context are cyclic references that are usually avoided by using weak references to break the circle.

In MyThOS, the general lifetime of an object is managed by the original capability, and all other capabilities are basically weak references. However, as capabilities are not counted, creating an temporary reference or entering the Monitor races with the destruction of the object. To resolve this race, the broadcast-based mechanism has been introduced. This mechanism is very similar to *basic RCU deletion schemes* [McK04, MW07] which delay the deletion of copies by rescheduling on every hardware thread. If frequent recycling of Untyped Memory is needed, enhanced RCU schemes may provide further ideas on how to increase the throughput.

Multikernels like Barrelfish prohibit shared kernel objects and, hence, no invisible temporary references should exist. Similarly, seL4 and clustered multikernels use a big kernel lock to prohibit concurrency inside the kernel, which eliminates data races.

2.4.3 Credit-based Flow Control

The amount of memory in many-core architectures is quite small in relation to the number of hardware threads. With asynchronous programming models, an even larger number of logical control flows exists. Therefore, assigning each control flow a practically infinite, hence “large enough”, message/context buffer is not feasible. Conversely, dynamic memory allocation for buffers injects additional runtime dependencies and, thus, latencies into the communication. Last but not least, the *system throughput is limited* by the maximal throughput of the interconnect and the maximal throughput of the execution units. Once one of them is saturated, being able to create further messages or control flows does not increase the overall throughput any further.

See also 2.2.3

In MyThOS, these difficulties are avoided by incorporating the statically allocated bounded message buffer into the design of the asynchronous objects model: *Tasklets* serve both, as a *buffer* to hold a message or a control flow context, and as a *token* to implement *credit-based flow control*. Because each call to an asynchronous method consumes a Tasklet, the number of messages and control flows originating from a single asynchronous object is bounded by the number of Tasklet that it owns. New asynchronous activities can be started only after the Tasklet was returned through a call to a response method. Passing the Tasklet is *explicit and visible to the programmer*. This forces the designers and developers to reason about the implications of asynchronous calls.

Inside asynchronous objects, the passed tasklet is used to store all information about the issued method call. The tasklet is scheduled through the object’s Monitor for mutually exclusive execution and the actual implementation of the called method calls an asynchronous response method on the caller’s continuation object in order to pass along or return the Tasklet and flow control credit.

A more complex Monitor is needed if incoming asynchronous requests lock object-internal resources such as additional Tasklets for parallel communication with multiple other objects. In this case, new requests can be executed only after the previous request is completed, that is after the request’s processing received all internal asynchronous responses. In such situations, a Monitor implementation is used that queues incoming new request separate from incoming response messages. Response processing is prioritised and the oldest of the pending requests is executed only after the previous request was declared as finished.

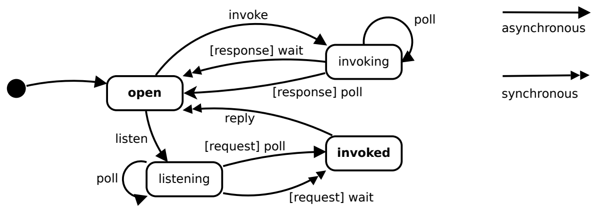


Figure 2.5: User-visible states of the Portal object.

Discussion. In TinyOS, the server objects have to supply the task objects for bounded asynchronism, which simplifies the flow analysis. However, in multi-threaded environment, the concurrent client requests would, first, have to allocate one of these task objects via a wait-free dispatching data structure and, then, have to enqueue the task to a shared wait-free scheduler queue. MyThOS uses static allocation of Tasklets at the client object in order to reduce the need for dynamic dispatching. Many simple asynchronous object own no or just a single Tasklet.

2.4.4 Portal: States and Operations

Capability invocation is the main mechanism for user-mode applications to interact with the kernel objects and to initially communicate between applications/processes, for example between worker process and supervisor. Portals are the key component for capability invocation and inter-process communication. This section describes the system calls that enable sending and receiving invocation and discusses the Portal's respective life cycle.

See also 2.2.2

Figure 2.6 shows the Portal's states and transitions. Initially, the Portal is in the *open* state and can be configured to either listen for incoming requests or to send a request in the form of an invocation message. The Portal returns to the *open* state by sending a reply in the *invoked* state or by receiving a reply from the *invoked* state. The *invoke*, *listen*, *wait*, and *reply* transitions are triggered through system calls of the Portal's owner. The synchronous transitions return from the system call after the operation completed whereas the asynchronous variants return immediately.

The message data is stored in an *invocation buffer*, which is shared memory between the application and the Portal on the kernel side. Applications should write to the invocation buffer only when the Portal is in the *open* or *invoked* state. However, this is not enforced by the operating system because non-conforming applications would just interfere with their own operation. The kernel objects that read from the invocation buffer first copy the needed values before checking their validity. Hence, concurrent manipulation of the invocation buffer is ignored.

The following system calls directly interact with the owner's Portal. For convenience and in order to reduce the number of system calls, combined system calls such as *invoke+wait* are available.

invoke(portal, uctx, dest): the content of the invocation buffer is sent asynchronously as an invocation to the specific destination `dest`, which is a capability pointer into the caller's capability space. In general, any kernel object that implement the invoke operation is a valid destination. When used for communication between applications, the destination will be a Portal. The *user context* `uctx` is an opaque pointer (64bit integer) that is returned by `wait()` and `poll()` on arrival of the reply. It is used by the user-space runtime environment to associate received messages with the respective communication channel.

listen(portal,uctx): this asynchronous operation switches the portal to receive mode and allows the Portal to write into the invocation buffer. Otherwise, only the application is allowed to write. Again, the user context `uctx` is returned by `wait()` and `poll()` when a request was received.

reply(portal): is a synchronous operation that returns after the invocation buffer is copied into the other side's invocation buffer. The operation will succeed quickly because the other side has to wait for the reply already. If the other side's Portal is not ready to receive the response or does no longer exist, the reply operation fails immediately. This ensures that misbehaving clients cannot delay the supervisor's control flows.

bind(portal,ec): rebinds the Portal to a different Execution Context, which will receive future invocation and reply messages of the Portal. This operation can be issued in any state and just selects the Execution Context that will be resumed by incoming messages. There is a race between rebinding and receiving a message. Whichever is processed first decides which Execution Context handles the message. However, applications that perform rebind in such a receiving state should be prepared to handle the message in both Execution Contexts anyway.

wait()→uctx: is a synchronous operation that blocks until a message on any Portal bound the Execution Context has arrived. The message can be either a invocation or a reply, and may has been arrived on any portal that is in the *listening* or *invoking* state. The user context `uctx` of the respective `listen()` and `invoke()` calls is returned. The `wait()` call can be interrupted without receiving any message. In that case the value 0 is returned as user context and the error number is set.

poll()→uctx: works like `wait()` but returns immediately if no received messages were pending.

The combination `invoke+wait` is synchronous and returns when any message was received on one of the execution context's Portals. In contrast, `invoke+poll` is asynchronous and returns immediately. If a received message was pending in one of the Portals, the respective user context value is returned.

Note that *poll* races with enqueuing a message on the Portal. If a application requires not to miss an incoming message, it must use *wait* instead.

explain send-wait-reply-wait sequence

how to remove in-flight messages when the sender's capability to the destination is revoked before delivery?

2.4.5 Execution Context: States and Operations

See also 2.2.2

Execution Contexts represent user-mode application threads by combining a reference to a logical address space, a reference to a capability space, a reference to a scheduling context, and an interface to the user-mode processor state such as register contents. The execution state is modified by three factors: synchronous system calls, synchronous exceptions/traps, asynchronous preemptive interrupts. This section describes the life cycle of Execution Contexts and their interaction with communication and notification mechanisms.

Figure 2.6 summarises the states and transitions of Execution Contexts. Initially, the Execution Context is *blocked*. When it transitions to *ready*, the associated Scheduling Context is informed, which is responsible to schedule the Execution context on a hardware thread. Once this hardware thread returns from the kernel mode into the Execution Contexts user-mode execution, the state changes to *running*. An Execution Context can be in this state only on a single hardware thread at any time. When the hardware thread enters the kernel again, the Execution Context's state transitions into the *blocked* or the *ready* state.

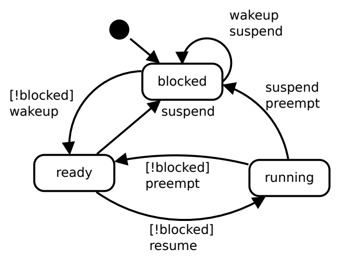


Figure 2.6: State machine for an Execution Context.

Synchronous system calls enter the kernel-mode execution from the Execution Context's user-mode thread and the Execution Context is blocked until the call has completed. The main set of system calls is presented in Section 2.4.4. Synchronous exceptions and traps are caused by the user-mode execution, enter the kernel, and block the Execution Context until the exception is handled. The exception handling is discussed in Section 2.5.9. Asynchronous preemption is caused by interrupt signals received from devices and by inter-processor interrupts (IPIs) that were sent from other hardware threads. This enters kernel mode as well but does not automatically block the Execution Context. Finally, an Execution Context can be marked as blocked via an invocation message, for example, from a supervisor. This does not interrupt the running execution but prevents any future return to user mode from interrupts, exceptions, or system calls. Therefore, a supervisor should first block the execution context and then raise an IPI in order to suspend a Execution Context preemptively.

In conclusion, a number of distinct reasons can block the Execution Context and multiple reasons may overlap. A field of *block flags* is used to represent these reasons. Only after all block flags are cleared, the Execution Context transits into the *ready* state. Therefore, after each update of one of the block flags, the overall state is checked. The following block flags are used:

No Scheduling Context: This flag is set when the Execution Context was not yet bound to an Scheduling Context or the Scheduling Context was revoked. Without a Scheduling Context, the kernel does not know how and where to run this Execution Context.

No Address Space: This flag is set when the Execution Context was not yet bound to an Address Space or the Address Space was revoked. Without an Address Space, the kernel cannot switch into the correct logical user address space.

Faulty This flag is initially set to show that its processing state is not fully configured. It is also set by synchronous exceptions and cleared by the exception handler. Any supervisor that holds a reference to the Execution Context can set this flag in order to prevent its execution.

Waiting This flag is set by the `wait()` system call and is cleared when an associated Portal receives a message.

Execution Context are not blocked when no Capability Space was assigned or it was revoked. Without a Capability Space, almost all system calls will fail. However, the system call entry code has to check for the presence of a Capability Space anyway. It might happen that some application or supervisor revokes a Capability Space while threads are still using it. In this case, the next system call has to detect the missing capability space and has to fail gracefully.

The implementation of Execution contexts uses a more general internal interface to the block flags based on bit fields and masks. The following methods are available on Execution Contexts. Not all of them are accessible through invocation messages.

suspend(bitfield)→bitfield: or-combines the old block flags with the given bit field and returns the old flags. When this blocks the Execution context from the *ready* state, no additional action is needed. The Scheduling Context checks the block flags before switching to the *running* state. While in `running` state no preemption is done automatically. This can be issued by the supervisor separately.

wakeup(bitmask)→bitfield: resets the block flags according to the given mask. If the block flags are all cleared, the Execution Context is scheduled via the *Scheduling Context*.

isRunnable()→bool: returns true if none of the block flags is set.

preempt(): if the Execution Context is in the *running* state, a kernel-internal doorbell interrupt is sent to the hardware thread that currently runs the Execution Context. The preemption request is handed to the respective Scheduling Context, which uses the kernel's asynchronous infrastructure to issue the interrupt signal.

dequeue(): if the Execution Context is enqueued for execution in a Scheduling Context, this operation removes it from the queue. The implementation reuses capabilities as queue and weak reference mechanism in cooperation with the Scheduling Contexts.

In order to forcefully stop an Execution Context, its supervisor has to first set one of the block flags and then send a preemption. Normally, this would return from the user-mode execution into the kernel and, then, see the block flag in order to not reschedule the Execution Context. However, the targeted Execution Context could have been scheduled for execution without running yet. Hence, the Scheduling Context has to check the block flags before switching to the user mode.

There is potential for a race condition between running, blocking, and preempting the Execution Context. Either the preemption could be dropped because the thread is not running yet or the blocking reason could be missed because the thread already runs. This is resolved as follows: The target hardware thread first sets the state to *running* and then checks the block flags. The other side updates the block flags first and then calls `preempt()`, which looks for the *running* state.

The `dequeue()` is not necessary during normal operation, because Execution Contexts that became blocked after enqueuing are ignored lazily. However, rebinding to a different Scheduling Context and deleting Execution Contexts require an immediate mechanism.

2.4.6 Interrupt Handling and the Scheduler Monitor

2.4.7 Monitor Example

This section describes the interface and dynamic behavior of a homed monitor with separate *request* and *response queues*. All asynchronous methods protected by the monitor are executed on a specific hardware thread, the *home place* of the monitor. The monitor prioritizes response messages in order to prevent blocking to many in-object resources with requests.

request(tasklet, function) increases the request counter atomically by one. If the counter has been zero, this request uses the supplied tasklet to schedule the function at the home place. Otherwise, it enqueues the function on its internal *request queue*

response(tasklet, function) increases the request counter by one and uses the supplied tasklet to schedule the response method on the home place. If the counter has been zero, this response uses the supplied tasklet to enqueue the function at the home place. Otherwise, it enqueues the function on its internal *response queue*.

requestDone()/responseDone() atomically decreases the request counter by one. If the response counter reaches zero, there are no pending requests and responses. Otherwise, it first checks the *response queue* and schedules it the response if any is found. If there is no pending response, a pending request from the *request queue* is scheduled on the home place.

acquireRef() increases the reference counter by one.

releaseRef() decreases the reference counter by one. If the reference counter reaches zero, there are no further references.

Discussion. The monitor presented here provides concurrency and locality control by allowing execution only on its home place. More sophisticated monitors may support delegation locking schemes in order to relax the locality constraints, or reader-writer synchronization. Additional waiting queue can be added in order to support condition variables.

2.4.8 Kernel Entry and Exit

There are various reasons why a user-thread may enter the kernel: synchronous and asynchronous system calls, hardware exceptions (traps) or interrupts. Fortunately, on implementation level, there are only two main mechanism to enter the kernel: interrupt and system calls. In the rest of the section, both are called an *entry event*.

In MyThOS, the only the information required to continue the execution of a user-thread is the processor state. As described in Section 2.4.5, a Execution Context can even be blocked for multiple reasons at once. However, the entering reason determines what processor state is expected by the user-level code when it continues running. For a *system call*, it is expected that the processor state is *modified* to reflect its return value. This further implicates that the value to be stored depends on the system call and reflects what may be returned. *Portal operation system calls* can either return no value, an error code, or a user context pointer. For example, *wait* and *poll* operations on a Portal are expected to return a *user context*, whereas other portal operations will only return error codes. For an *interrupt*, it is important that the saved processor state is *preserved*.

See also 2.4.4

MyThOS separates blocking from the reason the kernel has been entered. As illustrated in Table 2.2, the method of entering the kernel and blocking are to two separate concerns and can occur in various combinations. Moreover, a Execution Context might enter the kernel for a non-blocking reason, but is concurrently suspended by its supervisor.

	blocking	non-blocking
interrupt	page fault	hardware timer
system call	invoke (async.)	wait (sync.)

Table 2.2: Combinations of different entry methods and blocking semantic.

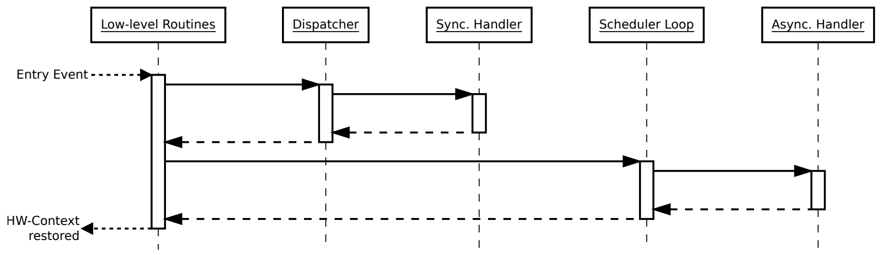


Figure 2.7: Kernel entry and exit sequence.

Entry Event Handling Both event types look very similar from the operating systems perspective. They both stop the execution of the user-thread currently running, start as synchronous events, and may schedule asynchronous event handlers. Figure 2.7 illustrate the complete entry event handling from entering the kernel till exiting the kernel.

Kernel Entry Routine (sync): User-level interrupts automatically switch to a kernel stack and then store parts of the hardware context on the stack. System calls are issued by a user thread and must to manually switch to the kernel stack and store the hardware context. Therefor, in contrast to interrupts and by convention, some parts (registers) of the hardware context are used for passing system call parameter, known to be volatile, or used to return a result value.

Both kernel entry routines save the hardware context and the entry reason into a context object pointed to by a hardware-thread-local variable. Typically, this object will be part of the current EC. The kernel entry routines are *synchronous*, implemented in *assembler* and call a global dispatcher.

Global Dispatcher (sync): For both event types, there is a global dispatcher that looks up a synchronous handler based on the *interrupt number*, respectively *system call number*. The dispatcher then reads a reference to the current

Execution Context from a hardware-thread-local variable and calls the handler, passing the Execution Context reference.

The global dispatchers is *synchronous*, implemented in *C++*, called by the kernel entry routine and calls a synchronous handler.

Synchronous Handler (sync): The synchronous handler is the first thing that must know about the semantics of the event. It may decode parts of the hardware context in order to extract parameter or error codes, and selects what can be returned by a system call. The synchronous handler may call arbitrary synchronous methods or manipulate the stored hardware context in order to handle the event, but must not block.

If the handler requires to issue asynchronous calls in order schedule a an asynchronous handler, it must first acquire a Tasklet. There are two possibilities: either the event is related to the hardware thread (hardware exception or system call) or to a specific interrupt handler (async. interrupt). In the first case, the handler can synchronously obtain a Tasklet from the current EC, which will be marked as blocked until the Tasklet is returned. In the second case, the handler must own at least one Tasklet, and either masks its interrupt source or must accumulate additional interrupts until the Tasklet is return until a Tasklet is returned.

Scheduling Loop (sync) / Asynchronous Handler (async): After the asynchronous handler returns, the dispatcher enters the task loop, that possible execute the asynchronous handler or other asynchronous tasks. The task executed may alter the state of the current execution context and its hardware context. After all local tasks have been completed, the kernel returns into the kernel exit routine with the next runnable EC.

Kernel Exit Routine (sync): The kernel exit routine does not depend on the entry event type of the last EC or next runnable EC. It simply restores the user-threads hardware context therefore exits the kernel.

The kernel exit routine is *synchronous* and implemented in *assembler*.

Discussion. [DBRD91] introduces the Continuation model after discussing two different models: The *process model* have a separate kernel stack for each process. The *interrupt model* uses a single kernel stack for every entry into the kernel. When a thread would block in-kernel, it saves its whole contest, hence the stack. In contrast, the *continuation model* only stores a smaller context when resuming a system call.

As the event-based in-kernel programming model in seL4, OKL4 and Nova [HE16], MyThOS programming model only requires one stack per hardware thread. Logical control flow is represented by asynchronous object state and Tasklets, which enable MyThOS to only store the processor state and the method of entering in order to resume a Execution Context later on.

2.4.9 Boot Sequence

The user-visible part of the system begins at the initial user thread, which usually is the first supervisor. In order to reach this point, it is necessary to initialise address spaces, start up the other hardware threads, create first kernel objects, and load the initial user thread.

The boot sequence is performed in a sequence of stages that initialise subsequent layers of the kernel architecture. *Stage 0* is loaded and executed by the platform's boot loader or firmware in the raw physical address space. It creates an initial page table that contains the lower and higher half kernel codes at the correct logical addresses. Then, it switches to the x86-64 long mode and jumps to the higher half kernel code.

The *Stage 1* sets up the final kernel address space without the lower half kernel, because the lower half will be used as user space. This stage also sets up additional mappings as needed and configures sensible access rights. Its advantage over the previous stage is, that it can already use all of the kernel's code.

After switching to the final kernel address space, the *Stage 2* is executed on the first hardware thread (aka Bootstrap Processor, BSP). It configures the root Untyped Memory object by setting the managed physical address range and inserting all free ranges that are backed by usable memory. For this purpose it can be necessary to parse the platform's memory table. The GDT, Core-Local Memory, Local APIC, and initial objects for all hardware threads are allocated from the root Untyped Memory. Finally, this stage uses the LAPIC to start up all other hardware threads and switches to the hardware threads actual kernel stack.

The *Stage 3* is responsible for loading the initial user thread. This is part of the kernel as binary blob with a `text`, `bss`, and `data` segment with fixed logical addresses and limited size. A respective address space is created by allocating Page Maps, creating Frames out of the current physical position of the binary blob, and mapping these Frames at the predetermined logical addresses. The initial Capability Space is created by allocating Capability Maps and filling in all Scheduling Contexts (one per hardware thread), an initial Portal for system calls, the remaining Untyped Memory, and references to the own Execution Context, Address Space, and Capability Space. Then, the Execution Context's register set is initialised with the code entry point as instruction pointer and additional registers for the number of Scheduling Contexts and Untyped Memory objects. A user stack is not needed yet, because the supervisor can set the stack frames up on its own.

Stage 4 is initiated by scheduling the initial user thread on the bootstrap hardware thread. All other hardware threads are still in their scheduling loop and wait. The initial user thread performs system calls through its Portal in order to configure its Address and Capability Spaces, create more Execution Contexts and so on. An initial communication channel to the external management service on the host computer is needed. This has to be provided by the Stage 3 setup. The implementation of this channel depends on the platform. Based on commands and data from this channel, the initial user thread loads applications and schedules them on the desired Scheduling Context.

2.5 Development View

This section describes implementation aspects, for example, the files and folders structure and the configuration management via code modules.

2.5.1 Compile- and Run-Time Dependency Injection

Whenever objects or software components are created by instantiating classes, the question arises how the new objects will interact with the outside world. They require connections to other used services and usually also some context-dependent configuration. These dependencies can be fulfilled by queries to omnipresent directory services, by conventions, or via configuration methods and constructor arguments.

In order to simplify all aspects of instantiation, MyThOS applies the principle of *dependency injection* throughout all levels of the architecture and source code. This implies that no object or software component shall search on its own for its dependencies through global variables or omnipresent directory services. Instead, all dependencies and configuration is injected by the creator through constructor arguments and dedicated configuration methods.

This strategy separates the code responsible for instantiation and configuration from the component's implementation. This helps to reduce the component's assumptions about its usage environment and protects against premature policy decisions.

Run-time value injection: During object creation, the creator passes pointers to used objects and configuration data by value via constructor arguments and additional configuration methods. This is the most commonly used form of dependency injection.

Compile-time template injection: In C++, objects are created by instantiating classes and generic class definitions expose template arguments that need to be filled with actual type names and constants. This provides a mechanism for a template type based dependency injection. For example, in order to remove the overhead of virtual methods, the types of used objects can be injected alongside with the pointers to these used objects. This strategy is applied wherever type polymorphism is needed only for configuration but not during the actual execution.

See also 2.5.2

Compile-time code injection: Some definitions and constants cannot be injected via template arguments and, sometimes, overuse of template arguments impedes the readability just a bit too much. In these situations, static code injection is applied by including generic header files that will provide the missing definitions. The build configuration management has to supply actual contents for these header files.

2.5.2 Source Code Composition: Code Modules

Early work with MyThOS showed that a single runtime-only configuration strategy is not sufficient. Some components depend on specific hardware and compiler support, some performance critical options require a static compile-time configuration. At the same time, subtle differences between various platforms (xeonphi, quemu, bochs,

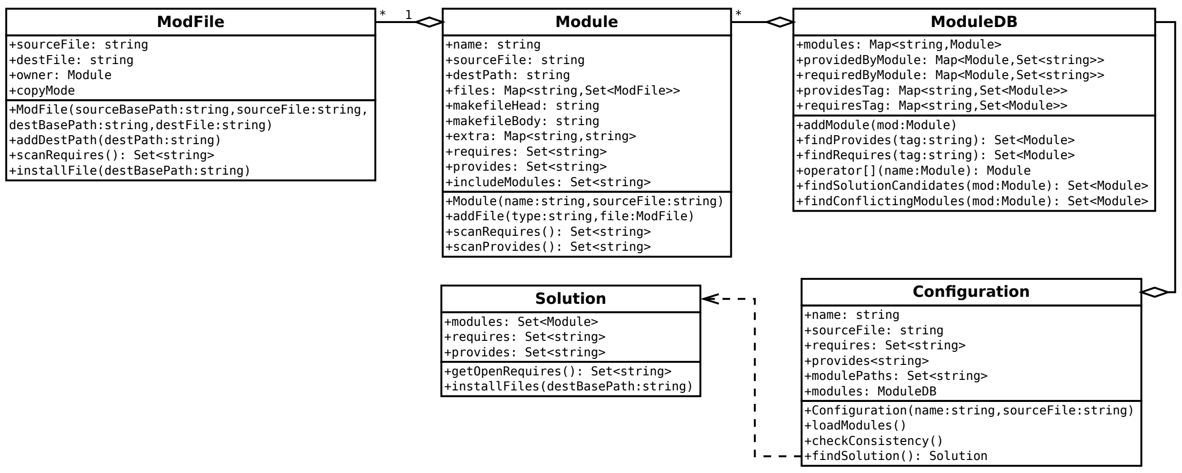

ModFile
+sourceFile: string
+destFile: string
+owner: Module
+copyMode
+ModFile(sourceBasePath:string,sourceFile:string, destBasePath:string,destFile:string)
+addDestPath(destPath:string)
+scanRequires(): Set<string>
+installFile(destBasePath:string)
*
1
Module
+name: string
+sourceFile: string
+destPath: string
+files: Map<string,Set<ModFile>>
+makefileHead: string
+makefileBody: string
+extra: Map<string,string>
+requires: Set<string>
+provides: Set<string>
+includeModules: Set<string>
+Module(name:string,sourceFile:string)
+addFile(type:string,file:ModFile)
+scanRequires(): Set<string>
+scanProvides(): Set<string>
*
ModuleDB
+modules: Map<string,Module>
+providedByModule: Map<Module,Set<string>>
+requiredByModule: Map<Module,Set<string>>
+providesTag: Map<string,Set<Module>>
+requiresTag: Map<string,Set<Module>>
+addModule(mod:Module)
+findProvides(tag:string): Set<Module>
+findRequires(tag:string): Set<Module>
+operator[](name:Module): Module
+findSolutionCandidates(mod:Module): Set<Module>
+findConflictingModules(mod:Module): Set<Module>
Solution
+modules: Set<Module>
+requires: Set<string>
+provides: Set<string>
+getOpenRequires(): Set<string>
+installFiles(destBasePath:string)
Configuration
+name: string
+sourceFile: string
+requires: Set<string>
+provides<string>
+modulePaths: Set<string>
+modules: ModuleDB
+Configuration(name:string,sourceFile:string)
+loadModules()
+checkConsistency()
+findSolution(): Solution

Figure 2.8: Data model of the module and build target description.

gem5...) and the desire to evaluate implementation variants required the ability to replace and recombine code fragments.

MyThOS avoids the conditional C preprocessor spaghetti by applying the principle of dependency injection to the source code organisation. Similar to grouping tightly related objects into components, related source files are grouped into *modules*. In this setting, the kernel objects of MyThOS are the user-visible components and modules provide the class definitions that are needed to create the objects of these components.

The build process is configured by the combination of *module specifications* and a *target specification*. Figure 2.8 gives an overview of the underlying data model. Basically, each module provides a set of source files and requires a set of files. For example, such dependencies can be header files with type definitions or source files with a specific implementation variant of a global function. Most dependencies are extracted automatically from the `#include` directives in the provided source files.

The target specification lists a set of requested modules or files. A simple resolution scheme is used to resolve dependencies: a list of missing files is collected from the requested modules and then additional modules are selected one by one if they provide one of the missing files. Two modules conflict when they provide files with the same destination name. In that case the dependency resolution reports an error and the developer has to manually select one of the possible modules by editing the target specification. In order to reduce the tedious work of selecting large groups of

```
[module.boot-memory-multiboot]
  requires = ["platform:multiboot"]
  incfiles = ["boot/memory/Stage3Setup.h"]
  kernelfiles = ["boot/memory/Stage3Setup.cc", "boot/memory/Stage3Setup.cc",
      "boot/memory/Stage3Setup-multiboot.cc"]

[module.boot-memory-gem5]
  requires = ["platform:gem5"]
  incfiles = ["boot/memory/Stage3Setup.h"]
  kernelfiles = ["boot/memory/Stage3Setup.cc", "boot/memory/Stage3Setup.cc",
      "boot/memory/Stage3Setup-e820.cc"]

[module.boot-memory-knc]
  requires = ["platform:knc"]
  incfiles = ["boot/memory/Stage3Setup.h"]
  kernelfiles = ["boot/memory/Stage3Setup.cc", "boot/memory/Stage3Setup.cc",
      "boot/memory/Stage3Setup-sfi.cc", "boot/memory/Stage3Setup-knc.cc"]
```

Listing 2.1: An example module specification.

related module variants individually, the respective variants depend on a artificial *tag* as resolution filter and the target specification provides this flag. For example, all modules that target the XeonPhi depend on `platform:knc` and all modules specific for the x86-64 architecture depend on `cpu:x86-64`.

Listing 2.1 shows three example modules that provide the same header file `Stage3Setup.h` together with with different implementation source files. Instead of selecting one of the variants explicitly, build target specifications declare that they provide one of the platform tags, for example `platform:knc`. Then, just a single variant has all dependencies fulfilled and will be selected automatically to fulfil the boot codes dependency on `Stage3Setup.h`.

With respect to the layout of files and folders, modules are grouped into folders based on their respective subsystem with one sub-folder per module. These module folders contain a `modules.mcconf` file or several `.conf` files containing module specifications. The pathes to source files are relative to the position of the module specification. The target specifications contains a relative path to the root of the module folders.

The build process works as follows: given a target specification, the `mcconf` tool reads all module specifications, selects additional modules in order to fulfil open requirements, and then copies the references source files into the target folder. The configuration tool also generates a `Makefile` that contains rules for the compilation of the kernel source files and the linking of the final kernel image. Then, `make` can be used to compile and link the configured kernel. For this purpose, module specifications can contain Makefile fragments.

The top level source folder structure is roughly based on the horizontal layers. The folder name `tag` is reserved for *tags*, a symbolic requirements do not refer to a file and is used to specify a configuration without the need to manually pick each platform-dependant module.

util contains language level helpers and meta programming as well as logging facilities and generic data types.

cpu contains synchronous hardware abstractions.

async implements the runtime for asyncronous objects.

objects contains the implementation of concrete kernel objects and their runtime environment.

boot contains platform-dependent boot implementation, initializations and glue code.

mythos contains the interface and definitions to be shared between kernel and user space.

runtime contains the user-space runtimen environment.

lib contains shared application code that is not part of the operating system.

app contains individual applications.

The modules folder structure mirrors the structure of the source folder. Each module lives in its own subfolder, shared code is extracted into modules with the suffix "-common". There is an additional *build* folder that contains modules facilitating the build process, for example by adding appropriate compiler flags to the makefile.

Discussion. A *software component* is a C++ object or a group of C++ objects that work tightly together to provide a specific service and that share the same life cycle. Although less relevant for MyThOS, components should be designed with substitute-ability, reusability, and composition in mind. The difference between arbitrary language-level C++ objects and components are the more restricted rules governing the component's interfaces, life cycle, and interactions. Language-level objects are used to implement components but also to implement data structures, containers, communication channels, function objects, messages and so on.

The seL4 kernel uses some clever Makefile constructs in order to copy the platform- and process-specific files into the right place. Dependency injection is applied by including header files at generic locations, for example `arch/types.h`, that are not actually present in the source code. The actual files are created during the build process by generators or by copying specific implementations, for example `arch/x86/types.h`. The Fiasco.OC kernel uses a mix of a custom C++ preprocessor and the Linux kernel's preprocessor-based configuration tool.

2.5.3 Interacting Asynchronous Objects via Tasklets

The asynchronous objects' responsibility is to implement delayed and delegated execution as base mechanism for the kernel's concurrency and locality control. Main challenges for any asynchronous execution are the encoding and decoding of the desired action into messages, the storage and transfer of these messages, and the storage of the dynamic processing state.

Listing 2.2 shows an example implementation of an asynchronous object. The `Adder` provides the asynchronous method `Adder::add()` that allows two add two integer numbers. Given a Tasklet `msg`, a pointer `a` to an Adder instance, and a pointer `r` to a matching response handler object, the asynchronous method is called with `a->add(&msg, r, 1000, 1)`. In line 4, the call and its arguments are captured into a C++11 anonymous function (aka closure, lambda expression) and passed to the Adder's monitor (line 4). The monitor uses the passed Tasklet to schedule the delayed and possibly remote execution of the anonymous function. The actual implementation `Adder::addImpl()` in line 10 performs its work and, then in line 12, calls one of the result handler's methods, passing along the Tasklet for the asynchronous execution of the handler method. In line 13, the monitor is informed, that the request processing has finished. This is necessary for the asynchronous object deletion.

See also 2.4.2

```
class Adder {
public:
  void add(Tasklet∗ msg, AdderResponses∗ res, int a, int b) {
    monitor.exclusive(msg, [=](Tasklet∗ msg){this->addImpl(msg,res,a,b);} );
  }

private:
  MutexMonitor monitor;

  void addImpl(Tasklet∗ msg, AdderResponses∗ res, int a, int b) {
    // do something ... then respond
    res->addResult(msg, a+b);
    monitor.release();
  }
};
```

Listing 2.2: An example asynchronous object.

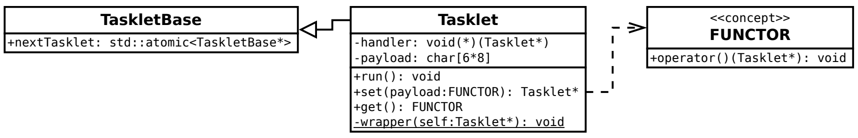

TaskletBase
+nextTasklet: std::atomic<TaskletBase*>
Tasklet
-handler: void(*)(Tasklet*)
-payload: char[6*8]
+run(): void
+set(payload:FUNCTOR): Tasklet*
+get(): FUNCTOR
-wrapper(self:Tasklet*): void
<<concept>>
FUNCTOR
+operator()(Tasklet*): void

Figure 2.9: Class diagramm of the Tasklet implementation.

All execution is non-blocking and, hence, waiting synchronously for results of calls to asynchronous methods is not possible—and not necessary. Instead, *request* methods also retrieve a reference to a response object that acts as continuation and sink for the result data. Such *response* methods consume the tasklet and results as arguments but have no argument pointing to another response handler.

Figure 2.9 shows the Tasklet's class diagram. The base class contains an C++11 atomic variable that is used by monitor implementations to manage task queues without dynamic memory management. The pointer is in a separate base class in order to simplify the usage of small dummy objects for the queue management.

The Tasklet class itself contains a C function pointer very similar to *active messages*. This handler function is responsible for executing the actual function object that was stored through the `set()` method in the payload. Instead of the function pointer, a virtual method could have been used. This would consume the same space but adds a second memory lookup via the vtable pointer. The Tasklet implementation contains a generic static implementation for these handler functions.

The size of the payload is chosen such that Tasklets are exactly one x86-64 cacheline large (64 byte). This leaves four 64-bit words for direct call arguments. Any additional arguments have to be passed via a pointer to an argument structure, which is provided by the caller.

Discussion. Often, user threads or coroutines are used to store the dynamic processing state. These are based on a separate stack for each of the currently active executions. The separate stacks allow to suspend and resume the current execution at arbitrary points by switching between stacks. While very convenient, a major drawback is the need to reserve a sufficient amount of large enough stack in advance. Suspending an execution usually requires to switch to a new coroutine or thread in order to be able to begin the next pending execution. This again requires dynamic memory management.

Message passing as a middleware layer tends to hide the allocation and deallocation of message buffers in order to increase the conceived convenience. This introduces a hidden dynamic memory management and adds considerable send and receive overhead. The MyThOS design avoid both by using Tasklets as caller-provided message buffers and, hence, can be used with no or very simple caller-internal allocation mechanisms.

Decoding retrieved request and response messages involves figuring out what should be done with the message and, then, to unpack the attached payload into usable, correctly typed arguments. A method call dispatcher would interpret a logical identifier in order to choose which code to execute. This usually introduces multiple conditional branches into the execution path. Instead, Tasklets as active messages use a single unconditional jump.

Encoding and decoding the arguments requires a client and a server stub code. The publicly visible asynchronous method fulfills the role of the client stub and the anonymous function acts as server stub. The use of C++11 lambda expressions shifts the tedious work of argument encoding and decoding into the C++ compiler. This exploits that the caller and callee code is compiled by the same compiler and resides in the same kernel image.

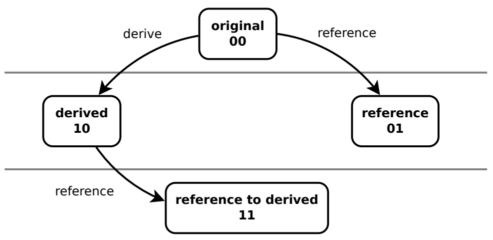


Figure 2.10: Derivation and reference capability flags. References to derived capabilities can be revoked without revoking other derived capabilities and references to the parent.

2.5.4 Managing the Resource Inheritance Tree

Storing the complete resource inheritance tree can be very space consuming and most of the information is never used. This reduces the system's efficiency through increased cache and TLB trashing and complicates the concurrent update operations on the inheritance tree. In addition, not all possible inheritance operations have a well-defined meaning. For example, what should happen with the children when a capability that has derived children is copied? Who is responsible for the object deletion when the root capability of an object is copied?

MyThOS uses a set of rules to restrict the types of capabilities that may be derived to create children and which may be copied to create siblings in the inheritance tree. The basic idea is to maintain a strict *contained-in* relation between the kernel objects and their children. For example, a Portal that was allocated from an Untyped Memory is contained within the address range of the Untyped Memory.

Now, just *references* to kernel objects require separate handling because they represent the same address range as the capability they were derived from. The basic idea is to use a flag in the capability in order to mark it as weak reference. This allows to revoke all references and inform the affected capability subject or object.

However, supervisors may want to transfer access capabilities, which are weak references too, to other applications and retain the option to revoke these individually. For example, a different clients retrieve Portal references to the same portal. In order to keep the different clients apart, these references do not inherit from the original portal but from an intermediate *derived* capability. Thus, two flags in the capability are used to represent *original*, *reference*, *derived*, and *reference to derived*.

This derivation and reference scheme is illustrated in Figure 2.10. For example, the original capability can point to a Portal. A reference to the original enables the holder to operate on the Portal as if owns the original. The actual access rights can be restricted through the reference's meta-data. The derived capability of the Portal can be used to hand out derived references to a client application. These can be revoked by revoking the derived capability.

The same rules are required to differentiate Frames, mapped Frames, derived Frames, and mapped derived Frames. The mapped frames are weak references and are mainly used by the Page Maps to track the usage dependencies between address spaces and frames. The derived frames are used to hand out the right to map frames without loosing the ability to revoke just these mappings.

The above rules have following implications: inheritance from a capability creates a child if the new capability is (a) a reference to an original capability, (b) a reference to a derived capability, (c) a derivation from an original capability, or (d) an original capability that is a real subset of the parent original capability. Inheritance creates a sibling in the tree if the new capability is (e) a reference created from a reference, (f) a derivation created from a derivation, or (g) an original capability that has the same address range as its parent capability. All other cases are prohibited and the rights management may further restrict the inheritance.

In combination, this allows to store just the *prefix serialisation* of the inheritance tree in a double linked circular list. Figure 2.12 gives an example. The root of the tree is the initial Untyped Memory object that represents all of the kernel's usable memory and is statically owned by the kernel. The circular list avoids to deal with the special case of the list's end. Children are inserted directly behind the parent into this chain. In order to insert a sibling behind a capability, it would be necessary to skip all existing children. Instead, a sibling is inserted before the capability that it inherited from, which equally ensures that both stay within their parent's subtree.

The parent-child relation is not stored explicitly because the objects and properties of the capabilities are sufficient to decide this relation. This is just needed for the revocation of access rights and deletion of kernel objects. By construction, all children are directly behind the parent in the chain. The subtree ends when reaching a capability that cannot be a child of the subtree's root. This *is-child-of* test is successful if (a) the address range owned by the child is a real subset of the parent's range, or their ranges are equal and (b) only the child has the reference flag, or (c) only the child has the derived flag.

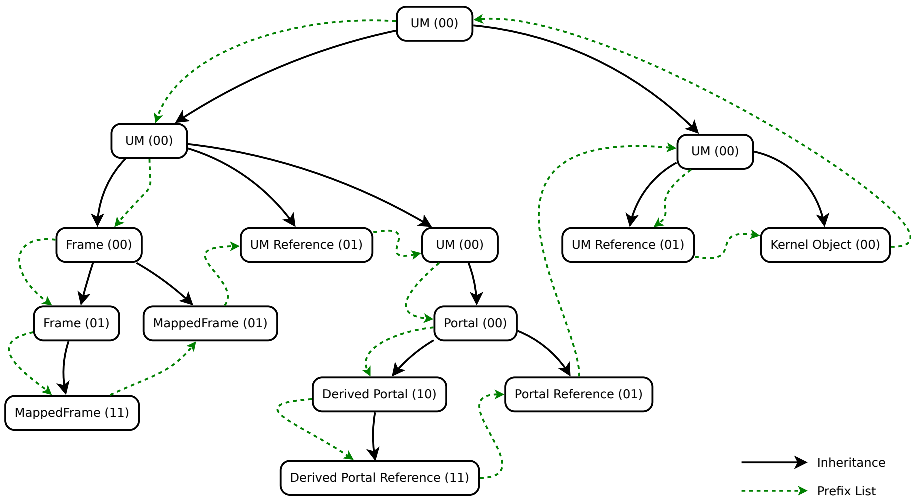


Figure 2.11: An example resource inheritance tree. UM is Untyped Memory. The two digits mark original, derivation and reference capabilities. The dotted arrow shows the actually stored prefix serialisation.

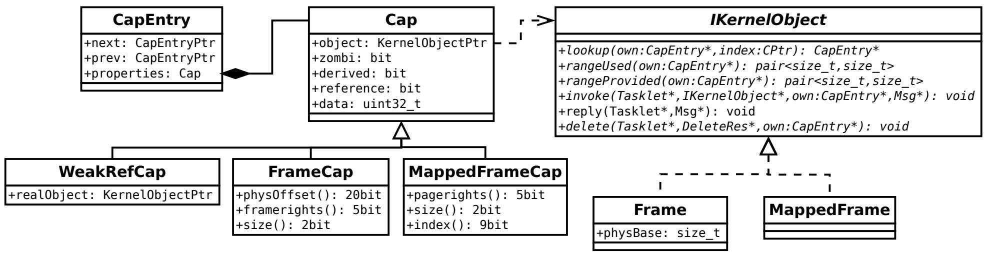


Figure 2.12: Class diagram of the prefix-serialised resource inheritance tree.

Figure 2.12 shows the class diagram of the inheritance tree. The capability entries `CapEntry` store the double-linked chain and the meta data `Cap`. The meta data is designed to fit into a 64-bit register. The first 32 bit contain a compressed pointer to the kernel object that is the capability's object. The capability's subject, that is, the holder of the capability entry is not stored by default. The kernel object has to provide the actual information for the inheritance tree management. Most important are `rangeUsed` and `rangeProvided` for the *is-child-of* test. The second 32 bit contain arbitrary type-specific meta-data that is processed only by the kernel object. For this purpose, a pointer to the capability entry is passed to all respective methods.

Because of the limited size of the kernel space and the 8-byte alignment of all objects, pointers can be represented with just 29 bits. The remaining three bits are used for status flags such as zombi, derived, and reference. Frames store a 4KiB aligned offset into a 4GiB address window, which requires just 20 bits. The base address of the window is stored in the Frame object.

Weak reference capabilities, for example mapped frames in a page map, require additional consideration. On revocation, the `delete()` method of the capability's object pointer is called in order to inform the affected kernel object. If this points to the original referenced object, a weak-reference flag and a pointer to the actual subject (the reference holder) is needed in the meta-data. In many cases, it is easier to replace the capability object by a helper kernel object that is part of the subject and knows how to clean up the revoked reference. For example, the object of `MappedFrameCap` points to a respective kernel object that is part of a `PageMap`. This leaves more space for other meta-data.

Discussion. The idea of using the prefix serialisation of the inheritance tree was inherited from seL4's implementation. They differentiate between capabilities on untyped memory, *original* capabilities that can only be derived from, and *derived capabilities* that can only be copied [GW16]. Unlike our presented solution, their *is-child-of* test inspects the types of the capabilities and applies type-specific rules. Our solution requires just a call to a virtual method in order to query the type-specific address range calculation and everything else is independent of the kernel objects.

Ideally, it would suffice to compare the managed address range of the parent against the used address range of the child only if both are original capabilities. In most cases, the reference and derivation capabilities that follow in the chain have to be children. Unfortunately, the Untyped Memory nodes in the inheritance tree can contain original capabilities as well as own references and derivatives. Hence, the range comparison is mandatory. We do not optimise away this comparison because it is used just during revocation, which is not expected to be fast anyway.

2.5.5 Operation Implementation in the Inheritance Tree

Even given the structure of the Inheritance Tree, described in Section 2.5.4, and the approach to safely delete individual objects, described in Section 2.4.2, it is non-trivial to implement high-level operations on Capability considering these operation might be executed parallelly.

There are three major categories of operations to consider: The *read operations* read a capability in order to create a reference or call a asynchronous method. The *local operations* locally update the Inheritance Tree, like *derive*, *reference*, but also *moving* a capability into another entry. The *subtree operations* consists of operations that may delete whole subtrees of the Inheritance Tree, such as *delete* and *revoke*.

There are several race conditions between operations from different or the same category. The value of a Capability is only updated atomically, and can be marked as a zombie to prevent reader from using the value. A simple solution to resolve the races between writers is protecting the whole Inheritance Tree with a global mutex. However, this serializes all operations which manipulate capabilities.

...

2.5.6 Invocation Buffer

The invocation buffer is 512 byte buffer in a Frame created from Untyped Memory. Therefor it is read- and writeable by the kernel using direct mapped memory. The overall invocation process is not concerned with the semantics of the Invocation Buffer. However, Capability Transfer demands to distinguish Capability Pointers from plain binary data. On the other hand, Invocations on Kernel Objects require a calling, respectively, marshaling convention. Table 2.3 shows how the Portal operations interact with the Invocation Buffer.

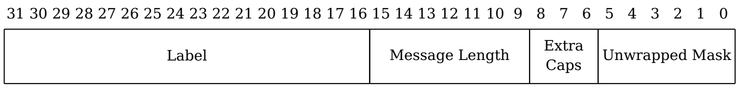
See also 2.5.7
See also 2.4.4

field	size	invoke (in)	invoke+wait (out)	listen (in)	listen+wait (out)	reply (out)
msg. tag	4B					
label		request	response	ignored	request	response
msg. length		req. len.	res. len.	480	req. len.	res. len.
extra cap.		request	response	0	request	response
unw. mask.		0	0	0	0	0
msg. buf.	480B	request	response	-	request	response
cap. array [0..6]	7*4B					
dest. cap [0]		dest.	-	-	dest. badge	-
extra cap [1..6]		request	response	-	request	response

Table 2.3: Invocation Buffer Format. op+wait denotes either for a wait or a successful pull operation after a operation. Numbers denote constants that must be written into reserved fields.

Message Tag The message tag (32bit) consists of a Label, the message length, the number of referenced capabilities and a mask.

this might be to specific?



The *Label* (16bit) is typically not interpreted by the invocation process. However, kernel objects may interpret this value and generic error codes might be returned in the Label. The *Message Length* field holds values from 0 to 480 and denotes the length of the message. The *Extra Capability* (3bit) field holds values from 0 to 6 and describes how many capability are referenced. The *Unwrapped Mask* (6bit) determines which Capabilities should be tried to unwrap (in) and which capabilities have been unwrapped (out). As Capability Transfer will not be implemented, this field is reserved for later iterations of MyThOS.

Message The message is a oblique byte array with a size of 480 bytes. Capabilities are referenced by an index into the Capability Pointer Array.

Capability Pointer Array The Capability Pointer Array consists of 7 entries (32bit, 4bytes), with the first entry being reserved for the invocation target address.

Discussion The format of the Message Buffer and Message Tag format is heavily inspired by seL4 IPC IPC buffer data structures [GW16]. However, the number of extra capabilities has been increased in order to reflect the needs of factory method calls.

seL4 implements Capability Transfers between user-threads. This is especially useful for capability exchange between applications that do not have supervisor-supervised relationship. However, the current application requirements for MyThOS do not demand for such an mechanism. Therefore, it will not be implemented in the current iteration. Supervisors have control over the supervised Capability Space and can therefor use numeric identifiers to manually lookup and create Capabilities on the appropriate positions.

seL4 furthermore unwraps capability badges of capabilities that reference the invocation target. This enables supervisors to store internal identifiers directly in the supervised Capability Space. The reduce the need to track the content of the supervised Capability Space in the supervisor.

MyThOS might implement Capability Transfer and Capability Unwrapping in later iterations, as it promises to simplify the implementation of supervisors for complex programs. Therefore, the Unwrapped Mask in the Message Tag is reserved. For the supervisors required by the compute cloud scenario, Capability Spaces are expected to by simple enough to be managed without further support.

2.5.7 Invocation Handling

In Section 2.5.6, the low-level format of an invocation message and a simple marshaling convention has been presented. In contrast, this section illustrates by examples how Kernel Obects handle invocations and how concurrent application runtimes may expose the invocation to the programmer.

Invocation Handling in Kernel Objects Listing 2.3 illustrates invocation handling in the kernel: *invoke* is a ordinary asynchronous function and part of every the Kernel Objects interface. It multiplexes between *different protocols* and invokable *methods* using the *protocol identifier* and *method identifier* introduced in Section 2.5.6. As most Kernel Objects, the example object only support a single protocol. After the concrete method has been identified, the monitor is requested to

```
void Adder::invoke(Tasklet* t, InvocationCtx* ctx, CapEntry* cap) {
  switch (ctx->protocol()) {
  case ADDERPROT:
    switch (ctx->method()) {
      case ADD:
        monitor.request(t, [=](Tasklet*t){this->add(t,ctx,cap)} );
        return;
      }
    break;
  }
  ctx->response(t, CORE, UNKNOWN_METHOD);
}

void Adder::add(Tasklet* t, InvocationCtx* ctx, CapEntry* cap) {
  int a, b;
  bool success = ctx->readMsg(a,b);
  if (!success || !AdderCap(cap).canAdd()) {
    ctx->response(t, CORE, success ? DENIED : INVALID_MSG);
    monitor.requestDone();
    return;
  }
  // work with ctx->thread(), cap, a, b ...
  ctx->response(t, ADDERPROT, SUM, a+b);
  monitor.requestDone();
}
```

Listing 2.3: Kernel-side dispatch and implementation of a method invocation.

execute the implementation of the method. When a method or protocol is requested that is not implemented by the Kernel Object, it immediately returns calls the response handler with a error code.

In the implementation of the invoked method, the invocation message is interpreted further to extract the arguments. Moreover, the capability is inspected to check if the caller is allowed to invoke the function. If anything fails, the appropriate response is issued. Otherwise, the actual work is done and the context of the invocation is used to return a response according to the protocol. As required for every asynchronous method, the implementation must signal the completion of the request to the monitor.

See also 2.4.4

Invocations in Application Runtime Environments Application runtime environments are expected to provide a concurrent programming model, such as

```
void Adder::add(Portal∗ p, Future f, int , int b) {
  p->invoke(this->obj, ADDERPROT, ADD, f, a, b);
}

// ... in app’s idle loop:
Portal∗ res = wait();
if (res) res->handleResponse(); // pushes results to f
```

Listing 2.4: Example for issuing invocation user-side.

coroutines or events. Listing 2.4 illustrates how asynchronous invocations can be integrated into an environment which facilitates deferred synchronous execution with *futures*: The invocation is wrapped into a deferred synchronous function that stores its result into a future. Depending on the application-side programming model, the future may trigger an event or reschedule a coroutine when the invocation is completed. Therefore, the application-level idle loop waits for *any* invocation to complete and uses the *user context* in order to dispatch the appropriate response handler. User-level server or services can be implemented analogously: Listening on a Portal also is a deferred synchronous method that returns when a request arrives.

See also 2.4.4

Marshaling Conventions Per convention, all invocation messages where the most significant bit in the first byte of the message is set are in the kernel invocation format.

In the message, Capabilities should referenced by there offset in the Capability Index Field.

2.5.8 System Calls

Invocations are implemented on top of implementation level system calls, which modify the state of *Portals*. The semantic of these calls is described in Section 2.4.4.

2.5.9 Error Detection and Handling

Errors in a operation system can have multiple causes: incorrect usage of system call, hardware faults and programming errors. Almost every operation systems must be resilient against misuse of system calls and programming errors, which

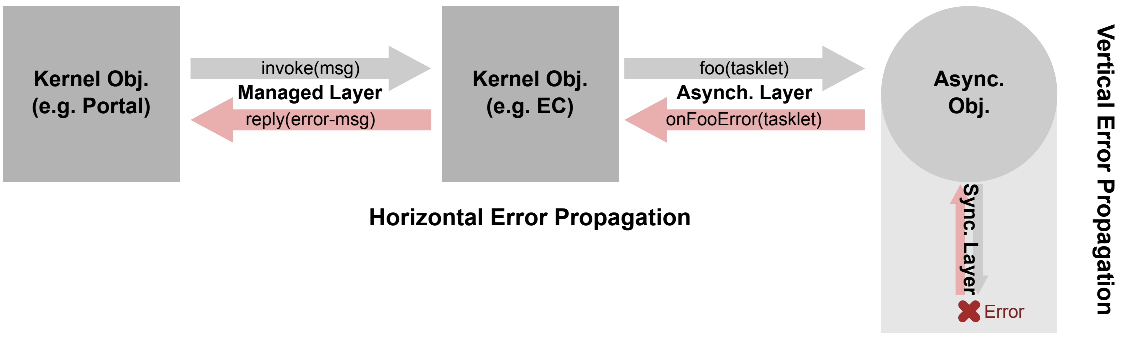


Figure 2.13: Vertical and horizontal error propagation across different layers.

are converted into *logical exceptions* when detected, and may lead to *hardware exceptions* if not detected or not handled properly. *Hardware faults* can either lead to *hardware exceptions*, e.g. ECC fault, or to *interrupts*.

Hardware exceptions occur synchronously when processing a certain instruction. Typical examples are Page and Protection Faults, and Invalid Instruction Exceptions. As such, hardware exceptions can be attributed to the asynchronous object currently being run and incorporated into a error handling scheme as described for logical errors. However, currently this is not the case. *Interrupts* occur asynchronously and can not be attributed to a concrete object or control flow. As such they are best handled by the normal interrupt handling procedure. Finally, *logical exceptions* are handle error discovered by the normal control flow. Triggering logical exceptions often is an appropriate mean to prevent hardware exceptions: for example, checking for a null pointer might prevent a page fault exception. The rest of the section focuses on the propagation and handling of logical errors.

After a exception is raised, it must be propagated to an appropriate handler. In MyThOS, we differentiate between two different dimensions: *horizontal* error propagation between *asynchronous and kernel objects*, and *vertical* error propagation in the *synchronous control flow*. The next paragraphs describe the error propagation in on the different *vertical layers*, as illustrated in Figure 2.13.

See also 2.2.1

Synchronous Error Propagation: In a *synchronous* control flow, errors are propagated *vertically* up in the stack. In the current MyThOS implementation, *vertical error propagation* is implemented via return values with the `optional` type. A function returning a optional value can either return *a concrete value* or an *error* *code and no value*. This is different from, e.g., returning a null pointer, which is in itself a possible value and does not allow returning a error code. When required, this mechanism might be switched out for another vertical error propagation mechanism, for example *C++ language level exceptions*.

Error Propagation between Async. Objects: If the error can not be handled in the current synchronous control flow, it must be propagated *horizontally* between *asynchronous objects*. *Horizontal error propagation* is implemented by exploiting the *client/server pattern* between asynchronous objects. Instead of calling the *response handler* of the client, the server responds by calling an separate *error handler*. This separates all handling of non-synchronous errors into the error handler, freeing the response handler from verbose error checking. Hence, all errors are propagated to the client. The error handler can then decide whether it can handle the error, for example by retrying, or propagates the error even further. An asynchronous object might enter an *error state* where it replies to all pending and incoming requests through a designated error handler, unrelated to the semantic of the request. This mechanism might be used in order to isolate objects in which a hardware exception has occurred.

See also 2.2.3

Error Propagation between Kernel Objects: On a higher layer, hence the *Managed Layer*, error still must be propagated horizontally between different objects. However, in the managed layer, only communication through messages is possible. Here, the error is propagated via an error message in a well-defined format.

this paragraph depends much on the invocation process

See also **??**

As *invocation* also is the main communication mechanism between the *application layer* and kernel objects, this is the way how errors are propagated back to the user. Typically, kernel objects perform a first sanity check on the *invocation request message* before starting to carry out the request and potentially delegating sub-tasks as asynchronous calls.

Error Handling in the Application Layer: An application must assume that, potentially, every invocation will return a error message. Application level services might implement sophisticated strategies such as *supervisor trees* [Arm03] in order to systematically handle errors. On the other hand, *hardware exceptions* and *lookup faults* in the Capability Space are translated into invocations to a *exception handler kernel object* which is determined by the *Execution Context*.

See also 2.4.1

Discussion. Choices OS implements a wide variety of error detection and correction methods [DC07]: Code Reloading recovers corrupted code sections, whereas Component Micro-Reboot can recover from corrupted data structures. In the mindset of microkernels, *restartable system components* and their corresponding *supervisors* are canonically implemented outside of the kernel. In-kernel code is expected to be correct, whether proven by verification or assumed after intensive review of the small kernel code base. On the other hand, this does not prevent kernel code and data structures from being corrupted from hardware fault. Choices OS implements restartable components in user-mode.On the error detection side, Choices OS implements *in-kernel C++ exceptions* as a general error handling mechanism, and transforms processor exceptions into language level exceptions.

2.5.10 Image-based Debugging Facilities

Because the system memory in MyThOS is only 4GiB large, it feasible to create an image of the whole system memory by using DMA from the host. Host accesses to the coprocessor memory are cache-coherent if the host-side caching is disabled.

Although it is possible create a kernel image without further kernel support, concurrent changes to the kernel memory can easily lead to an inconsistent image. MyThOS supports two different methods of obtaining a consistent kernel image: *Snapshot Images* capture a consistence state by issuing a non-maskable interrupt (NMI), which dumps the processor state into the stack, and waiting *in the interrupt handler* until the image is captured. In contrast, a *Checkpoint Image* is created by using a interrupt to force all threads into the kernel, but instead of waiting in the interrupt handler, all threads wait *before the next Monitor or Execution Context is dispatched*. This allows to create an kernel image that is easier to analyse, since there are no “non-commited transactions” of asynchronous object monitors.

3 Extensions for x86-64 and Intel XeonPhi KNC

Control Channel: At least the first user-mode supervisor requires a communication channel to external tools on the host processor. MyThOS does not aim to be a self-sufficient kernel and the control channel is the only initial interface from the user into the running system. Depending on the target platform, different implementations are needed, for example via PCIe shared memory on the XeonPhi versus a virtual serial IO port on emulators. Control Channels behave similar to Portals as communication endpoints and badged reference capabilities can be used to multiplex virtual channels.

Debug Channel: this kernel object provides a simple interface to the kernel's internal debug and trace messages subsystem. It is needed to see what the first supervisor does. Additional channels can be set up and configured by the supervisor.

The channel kernel objects could be replaced by platform-specific code in the supervisor. However, the initial channel via shared memory has to be at a fixed physical address in order to simplify the connection initiation. Channels via IO ports requires additional kernel support to enable io port access for the user-mode supervisor. Both would require additional kernel code while increasing the supervisor's complexity. Therefore, we choose to use custom platform-specific kernel objects for this purpose.

3.1 Short Messages over PCIe2

The communication between host services, such as the application launcher and the MyThOS services on the many-core accelerators, is carried out via asynchronous message passing over PCIe2. As already explained, a similar mechanism can be established between different MyThOS instances or components hosted on different processors, which are backed by shared memory. In this particular case, the majority of messages relates to control and configuration akin to remote method invocations or remote system calls. These require just small messages with a size limit in the order of a few cache lines.

Data transfers may need larger messages. One way to implement them, is splitting the transfer into multiple small messages, which does not need additional services. In the long term, dedicated mechanisms for large data transfers, for example via DMA engines, can be added. These would be coordinated through short messages.

Separate message channels are needed for each direction, that is from host to accelerator and from accelerator to host. On the producer site, support for multiple concurrent producers is desirable. This ensures, that all threads can send messages without the need for locking based mutual access or forwarding to a proxy thread. On the consumer side, such support for concurrent access is not strictly necessary. A main thread would be responsible to pull new messages and convert them into local tasks. A single consumer queue can be combined with a non-blocking mutual exclusion mechanism to hand over the work to an idle thread.

The PCIe 2.0 network allows to map the accelerator's physical memory into the logical address space of applications running on the host processor. The host service can directly write and read the accelerator's memory almost like normal shared memory. However, careful protocol design is necessary because the PCIe 2.0 network does not keep the caches coherent, atomic operations like fetch-and-add are not supported, and the involved processors may have a different view on the logical and physical address spaces.

The interaction across multiple address spaces implies the need for memory pinning. The host operating system and MyThOS must be instructed to never migrate the frames that contain the communication channel data. Otherwise, processors will read from and write to wrong positions, which are not observed by their counterparts. The easiest way to achieve this is by placing all shared communication data on the MyThOS side. There we, know the physical address of the channel and can ensure that the data will not be moved. For shared data on the host side, which runs a Linux system for example, two steps are needed: First, the allocated pages have to be pinned in order to prevent migration and, second, their physical addresses have to be provided to the MyThOS side. These need to be translated then into accelerator-physical addresses in order to become accessible from MyThOS.

3.1.1 Logical View: Slotted Circular Buffer

Using a fixed size memory area is the easiest approach for host to accelerator communication because just this area needs to be mapped into the logical address spaces once. Hence, a slotted circular buffer design was chosen: The messages are stored in a fixed size array with slots of fixed size (a few cache lines). The producer and consumer sides use independent counters for their logical read and write positions. The actual positions are modulo the number of buffer slots. Each slot contains a state header for flow control between producers and consumers. The producer writes sequentially into different slots of the circular buffer while the consumer tries to catch-up the producer counter. The flow control information is provided for each slot and it is used to establish an access protocol between the producer and the consumer, which will be explained later in this chapter.

The unidirectional channel is split into three data structures: ProducerSide, ConsumerSide, and ChannelData. This allows to place the respective parts into the appropriate memory, which enables the local use of atomic operations. Just the ChannelData is in the shared memory, which restricts it to ordinary read/write access and C++11 atomic load and store operations.

The ProducerSide and ConsumerSide objects contain the respective logical read/write positions and a pointer to the the ChannelData according to the local logical address space. No access to the other end's state is needed.

3.1.2 Physical View: Consumer vs. Producer Side

For host to accelerator direction, the ProducerSide is placed in host memory while ConsumerSide and ChannelData are placed in the accelerator's memory. For the reverse direction, the ProducerSide and ChannelData are placed in the accelerator's memory and the ConsumerSide is placed in the host memory.

The host has the ChannelData mapped with CacheDisabled and WriteCombining flags. The write combining makes sending more efficient. Ideally, the incoming channel's data should be mapped with caching enabled. Unfortunately, the currently used driver interface does not support this mode.

3.1.3 Dynamic View: Cache Control

Initially, the producer and consumer side logical positions are zero. The slot headers are initialized accordingly to be recognized as “free”.

A producer acquires a logical write position by atomically incrementing a local write counter. This supports concurrent producers. A producer then polls the header of a respective slot until the slot is marked “free” (by being equal to the acquired logical position). Then, the message data is written into the slot and, finally, the slot is marked as “occupied” by setting a respective bit.

The consumer polls the slot of its current logical read position. If the slot is marked as “occupied”, the data is copied or processed directly. Then, the consumer writes the slot’s next logical position into the header in order to mark it as free and selects the next responsible producer at the same time. Finally, the consumer has to invalidate the slot from its L1 and L2 caches and increment the logical read position.

Access to shared memory over PCIe is not cache coherent. Reads to remote memory can be cached in the local cache, but writes to this memory will not invalidate these replica. Hence, the memory has to be mapped in `Cache Disabled` mode or the communication protocol has to implement proper manual cache line evictions. Usually, writes over PCIe should invalidate the destination’s caches automatically. Unfortunately, currently this does not seem to work for writes from host to XeonPhi. Hence, the `CLEVICT0` and `CLEVICT1` instructions were used for manual cache line evictions.

The `CLFLUSH` and `MFENCE` instructions are not available on the XeonPhi. The new `CLEVICTx` instructions evict the requested cache line from the local L1 or respectively L2 cache but do not broadcast the evict request to other caches. Hence, all threads that read from `ChannelData` have to evict their data before reading the next message.

Future: Concurrent consumer support is possible by using a second bit in the slot header to mark the slot as locked by a consumer. This is done with an atomic compare-and-swap such that just a single consumer can succeed. All other consumers proceed to the next slots. Unfortunately, compare-and-swap is available only on the side that has the `ChannelData` in its local memory.

3.1.4 Development View

Because the cache invalidation works differently on host and XeonPhi, two different implementations of the of the producer and consumer sides of the circular buffer are needed. These versions can be also be adapted more easily to specific needs of the respective the side. In our current implementation `ProducerSide` is specialized in the classes `PCIeRingProducerX86` (for the host) and `PCIeRingProducerPhi` (for the MyThOS side). Both extend a generic interface `ISendChannel`. Likewise, a `ConsumerSide` is implemented in the classes `PCIeRingConsumerX86` and `PCIeRingConsumerPhi`. Both extend the interface `IRecvChannel`.

The basic interface provided by `ISendChannel` has the methods `acquireSend()` and `trySend()`. The acquire method returns the logical write position, which is then handed to the send method. `trySend()` has to be repeated for the provided position until it succeeds. This non-blocking interface allows higher software levels to intermix polling for incoming messages.

The basic interface provided by `IRecvChannel` contains the methods `hasMessages()`, `tryRead()`, and `finishRead()`. The first method can be used to check whether there is a pending message without trying to read or write anything. `tryRead()` returns a pointer to message data and might lock an appropriate slot for concurrent consumer support. After a message has been processed, its address is provided to `finishRead()`, which flushes the respective caches and marks the slot as free.

Implementation Note: The `ChannelData` structure is generic and has a slot type attribute as well as a slot count attribute. The slot type is combined with the channels slot header to define the actual slots and slot size. In consequence, the ProducerSide and ConsumerSide classes get the actual ChannelData type as template type argument.

3.2 Textual Debug and Binary Trace Output

MyThOS components produce a configurable amount of textual debug and log messages via the `mlog` subsystem. The `mlog` subsystem is integrated within all mythos modules and provides user interfaces for creating, filtering and sending log messages to customizable log sinks. One such sink is for example the debug output channel to the host. Currently the system is configured to send all levels of debug information of all subsystems over the PCIe2 channel to the host, where it can be further processed, e.g. stored into a file or printed on a terminal console. A per-module configuration of a debug-level is realizable as well.

Textual messages have a variable unbounded length and should be able to contain multi-line messages. Besides this, all hardware threads can produce messages concurrently and their output should be sequentialized for better readability (a multiple producers channel). Writing messages to a single output channel can be a blocking operation, because the communication is just one-way and, thus, no deadlocks are possible. Performance is not important because production systems would disable almost all debugging outputs.

In order to reduce the implementation effort and potential for errors, the short message channel from Section 3.1 is reused. Just a more specific message format had to be defined.

this might belong into the more general dev view

Several different `mlog` channels for different subsystems, e.g. memory management and userspace, are provided. Additional channels can easily be added. Each channel can output messages of different priority levels, as indicated by the programmer, which can be filtered before output. Thereby, developers can specify to see only important messages from the system, while getting detailed information from the subsystem, they are currently working on, without altering existing code. Textual debug messages are stored into an output buffer together with a timestamp, the originating hardware thread ID, the number of remaining messages and the length of the message. This output buffer can then be accessed externally to output the messages. Trace messages use the same `mlog` infrastructure, but are distinguished from debug messages by a type identifier. A different output buffer is used for trace messages to improve performance.

An appropriate terminal application on the host implements the receiver side of the debug communication channel. The process needs to run with superuser access rights, in order to be capable of reading and writing to the channel regions within the mapped physical address space of the extension card. Every received message is processed according to its type and printed on screen. The output can be of course also forwarded to a log file.

3.3 Local and Global TLB Invalidation

see “MOV—Move to/from Control Registers” and Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A

3.4 Global Descriptor Table

3.5 LAPIC for Timer Interrupts and IPIs

3.6 Interrupt Handler Maps

3.7 Architecture-Specific Debugging Support

3.8 XeonPhi DMA Engines

The XeonPhi processor has a couple of DMA engines for bulk data transfers over the PCIe network. In order to use this, the physical memory address of the source and destination have to be known. The host side runs Linux, which can move pages in the physical address space. Hence, the host's memory areas used for DMA transfers have to be locked (aka pinned). Then, the second challenge is to find the physical address of these areas. From this perspective it seems easier, to implement a small Linux kernel module for XeonPhi-to-Host communication. Intel's solution is called "SCIF" and does a lot more than we need.

3.9 Sizes

Warning This whole section is not up-to-date.

Update

Base Sizes			
name	symbol	relation	bits
machine word	MWORD	-	64
machine word aligned	MW_ALIGN	-	3
cache line	CL	64*8	512
cache aligned	CL_ALIGN	-	6
physical address	PHY_ADDR	-	64
logical address	LOG_ADDR	-	48
system space address	SYS_ADDR	-	32
in-page offset	PAGE_OFF	-	12
page table index	PT_IDX	-	9
Derived Sizes			
name	symbol	relation	bits
capability word	CAP_WORD	2*MWORD	128
frame pointer	FRAME_PTR	PHY_ADDR-PAGE_OFF	52
object pointer	OBJ_PTR	SYS_ADDR-CL_ALIGN	26

3.9.1 Capabilities

name	type	bits
CAP_SLOT	4*MWORD	256
prev	SYS_ADDR	32
next	SYS_ADDR	32
cap	CAP_WORD	128
overall	=	182

name	type	bits
FRAME_CAP	CAP_WORD	128
obj	OBJ_PTR	26
prop	>= 5 bits	5
frame	FRAME_PTR	52
frame_rights	>= 3 bits	3
size	2 bits	2
overall	>=	88

name	type	bits
MAPPED_FRAME_CAP	CAP_WORD	128
obj	OBJ_PTR	26
prop	>= 5 bits	5
frame_rights	>= 3 bits	3
index	PT_IDX	9
table	OBJ_PTR	26
overall	>=	95

name	type	bits
MAPPED_TABLE_CAP	CAP_WORD	128
obj	OBJ_PTR	26
prop	>= 5 bits	5
index	PT_IDX	9
table	26	26
overall	>=	95

Bibliography

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, dec 2003.

[BBD$^+$09] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *22nd Symposium on Operating Systems Principles*, pages 29–44. Association for Computing Machinery, Inc., 10 2009. ISBN: 978-1-60558-752-3.

[BWCC$^+$08] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[BWCM$^+$10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. *SIGOPS Oper. Syst. Rev.*, 25(5):122–136, September 1991.

[DC07] F. M. David and R. H. Campbell. Building a self-healing operating system. In *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*, pages 3–10, Sept 2007.

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[FK12] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. *SIGPLAN Not.*, 47(8):257–266, February 2012.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[GW16] Matthew Grosvenor and Adam Walker. *seL4 Reference Manual Version 3.0.0*. Thrustworthy Systems Team, NICTA, March 2016.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[HE16] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Trans. Comput. Syst.*, 34(1):1:1–1:29, April 2016.

[Kru95] Philippe B Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

[Lie96] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9), 1996.

[McK04] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf [Viewed October 15, 2004].

[MJK$^+$02] Paul E. McKenney, Appavoo Jonathan, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read Copy Update. In *OLS '02: Proceedings of the Linux Symposium*, pages 338–367, 2002.

[MTS05] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. *Multiparadigm Programming in Mozart/Oz: Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, chapter The Structure of Authority: Why Security Is Not a Separable Concern, pages 2–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[MW07] Paul E. McKenney and Jonathan Walpole. What is RCU, fundamentally? Available: http://lwn.net/Articles/262464/ [Viewed December 27, 2007], December 2007.

[OSK+11] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat. OctoPOS: A Parallel Operating System for Invasive Computing. In *Proc. of the International Workshop on Systems for Future Multi-Core Architectures (SFMA'11)*, 2011.

[RKZB11] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. Improving per-node efficiency in the datacenter with new os abstractions. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 25:1–25:8, New York, NY, USA, 2011. ACM.

[SGI14] Yuki Soma, Balazs Gerofi, and Yutaka Ishikawa. Revisiting virtual memory for high performance computing on manycore architectures: A hybrid segmentation kernel approach. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, pages 3:1–3:8, New York, NY, USA, 2014. ACM.

[SK10] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[THH+11] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive Computing – An Overview. In *Multiprocessor System-on-Chip*. Springer New York, 2011.

[vT12] Michael von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *2nd Workshop on Systems for Future Multi-core Architectures*, pages 1–6, Bern, Switzerland, apr 2012.