Global Architecture of MyThOS

Whitepaper

Randolf Rotta, rottaran@b-tu.de
Vladimir Nikolov, vladimir.nikolov@uni-ulm.de
Lutz Schubert, lutz.schubert@uni-ulm.de

19th April 2016

Contents

1 Overview **1**

2 Logical View: Components, Layers, and Subsystems **3**
2.1 Software Layers 3
2.2 Functional Subsystems 5

3 Physical View: Deployment, Placement, Connections **9**

4 Process View: Dynamic Interactions **11**

5 Development View: Implementation Artifacts, Interfaces, Types, Modules **13**
5.1 Software Components 14
5.2 Build-Time Configuration Modules 16
5.2.1 Module Specification 16
5.2.2 Target Configuration 17

1 Overview

This chapter introduces the global architecture and design of MyThOS. Summarized in broad terms it is a monolithic modular multikernel with optional sharing and a fine-grained asynchronous execution model.

Classic microkernel designs combine a modular component-based software architecture with strong isolation between components by executing them in separate protection domains. MyThOS follows the same component-based design style but makes isolation optional by allowing kernel-mode components for performance critical system services. This hopefully simplifies the implementation of high throughput task and thread scheduling approaches. Hence, from a puristic point of view, MyThOS has a monolithic kernel but nevertheless is based on a modular design.

Systems like Barrelfish [BBD+09] propose the multikernel style in which each hardware thread runs an independent kernel and all communication is facilitated via explicit message passing. This shared-nothing approach enables the operating system to coordinate across heterogeneous processors and distributed memory whereas applications can use shared memory where supported. MyThOS follows the same style by deploying a local task and memory management on each hardware thread. However, MyThOS also allows hybrid deployments in which groups of threads can share local resources and tasks. This should improve the load balancing between service threads in order to increase the overall responsiveness and throughput.

In order to actually achieve high throughput on many-core architectures, parallelism and locality is needed. To this end, MyThOS uses a fine-grained asynchronous execution model based on *tasklets* as non-blocking pieces of work. The basic communication across cores and threads is achieved by sending tasklets like active messages and parallelizable tasks can generate multiple tasklets for local load balancing. On top of this, interactions between components are mapped to request tasklets in combination with continuation passing for the responses. The classic interrupt handling epilogues also map to tasklets naturally.

The chapter follows along the 4+1 view model of Kruchten [Kru95] with some minor modifications. The scenarios were already summarized in the previous chapter in the form of objectives and requirements. Subsequent chapters will discuss the design of selected subsystems in more detail.

The first subsection (Sec. 2) describes the logical view, which consist of common services, mechanisms, and design elements that fulfill the functional requirements. This analysis extends the scenarios from the previous section by adding the services and system components that are needed to provide the requested functionality. The physical view in Section 3 discusses the deployment. This includes the placement and replication of specific components onto processors or hardware threads as well as the basic memory layout. Subsection 4 discusses the dynamic view, which focuses on life-cycle management of components, how interactions between components are carried out, and the necessary scheduling of tasks on hardware threads. Finally, the development view in Section 5 describes implementation aspects, for example, how interfaces and components are implemented in principle, as well as how files and folders should be structured.

2 Logical View: Components, Layers, and Subsystems

The logical organization of the operating system is based on reusable software components that provide a specific service by encapsulating the necessary code and state. Variants of the operating system are deployed by composing and configuring these components. Many of the hardware-independent components can also be reused via user-mode libraries.

In order to restrict dependencies and improve reusability, the logical architecture is decomposed vertically into software layers, e.g. local versus global services, and horizontally by functional subsystems, e.g. memory versus thread management. The following subsection (Sec. 2.1) gives an overview about the software layers of MyThOS. Then, Section 2.2 discusses important subsystems.

2.1 Software Layers

The main purpose of the software layers in MyThOS is restricting dependencies in order to improve the reusability. The lower layers are based on dependencies to basic mechanisms such as local memory management, asynchronous task scheduling, or inter-core and inter-thread communication. The upper layers are responsible for hardware-independent abstractions and application-specific support. Figure 2.1

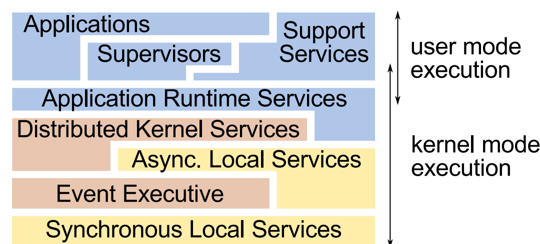

Figure 2.1: MyThOS architecture layers

shows the layers' vertical organisation and the following paragraphs describe each layer.

Synchronous Local Services (SLS): This layer contains components for the local hardware configuration and management on each hardware thread and processor core. The requirements of these components are as simple as possible, for example no asynchronous processing, no communication, and just statically allocated memory.

The components of this layer operate synchronously and immediately, based on request from higher layers and from hardware events. Policies and decision making should be left to higher layers in order to improve the reusability. Hence, hardware events are passed to the layers above via registered callback functions or specific hook functions. All processing should be non-blocking and wait-free in order to simplify the scheduling on higher layers.

Event Executive (EVE): This layer provides the mechanisms for all asynchronous processing ranging from scheduling local activities within hardware threads to pushing tasks to other threads, cores, and processors. All events generated from incoming messages and the hardware that cannot be processed locally and immediately are encoded into tasklets. The EVE layer schedules the execution of these tasklets.

Besides processing local and received tasklets, distributed and parallel processing also requires some concurrency control mechanisms. The EVE layer provides ordering mechanisms such as future variables (promises) as well as basic mutual exclusion mechanisms. The mutual exclusion mechanisms should be non-blocking, that is, enqueue waiting tasklets.

This layer uses just the data structures and components from the lower Synchronous Local Services layers. Some convenience functions for faster prototyping may rely on dynamic memory allocation, which may fail if the local reserves are depleted.

Asynchronous Local Services (ALS): This layer builds upon the low-level components of the Synchronous Local Services in order to add asynchronous processing interfaces that make use of the Event Executive layer. This is mostly a glue layer to connect the local services with the upper Distributed Kernel Services. In contrast to the distributed services, this layer does operate only locally without relying on the existence of other hardware threads or cores.

Distributed Kernel Services (DKS): The components of this layer, finally, bring together all the hardware threads by coordinating their activities globally. These services can be provided via centralized components that are assigned to a specific thread group or distributed/replicated components that coordinate in a peer-to-peer style. Examples are the management of shared memory pools and shared address spaces, configuring thread groups, implementing consistency protocols for replicated objects, and global garbage collection if necessary.

Application Runtime Services (ARtS): This layer implements runtime support for applications ranging from high-level abstractions for protection domains (system call interfaces and access control), execution contexts (software threads), and scheduling control up to application-specific programming environments like C/C++, Fortran and mechanisms like MPI/OpenMP. Parts of this can be integrated as components into the kernel or reside as library inside the applications. The components of this use the Local and Distributed Kernel Services and, hence, should be reasonably hardware-independent.

Support Services, Supervisor, Applications: On top of the hierarchy, the user-mode threads reside. The application threads use system calls to communicate with components from the ARtS layer. Depending on the deployment and usage scenario, the calls are forwarded to a separate supervisor application and external support services. The supervisor is helpful to manage the high-level scheduling and migration of data and activities in scenarios like the media cloud.

2.2 Functional Subsystems

The functional subsystems introduce a horizontal decomposition that focuses on specific responsibilities. Subsystems may span multiple layers by composing components from different layers. In order to gain wider flexibility in the application layers, the subsystems are little bit more fine grained than might be necessary with a fixed process/thread model.

Address Space Management: These components manage logical address spaces for kernel and application threads. For this purpose, the processor's address translation and access protection is configured by creating respective mapping tables from logical to physical addresses. The final logical address space is

composed of various partial spaces, for example by combining the application-independent kernel-space with the current application's user-space layout.

Components for optional reverse mappings from physical addresses to all uses in logical address spaces are provided as well. These are necessary, for example, to properly release physical memory ranges when removing or overwriting existing mappings. Likewise, a mapping from partial address spaces to all hardware threads that currently use these is provided in order to update the processor's translation caches when needed.

Managing free versus used ranges in logical address spaces is left to the Memory Management subsystem. The needed data structures are very similar to the management of free memory. The Memory Management uses the Address Space Management for on-demand growing and shrinking of memory heaps. The Address Space Management just needs a simple physical page allocator to dynamically create and reuse mapping tables.

Memory Management: This subsystem contains reusable components for managing free and used address ranges, generic memory allocators, and specific allocator instances for common purposes.

Core-Private Memory (like thread-local memory in the user-space) is provided to manage objects with global identity but a separate instance per hardware thread. This is needed, for example, to access the local scheduler on each hardware thread.

Each hardware thread has a local synchronous memory allocator that works independent from the other threads and enables quick dynamic allocation of local kernel objects. However, the limited amount of physical memory combined with the large number of hardware threads implies that the local memory reserve is quite small. The physical memory of large objects has to be allocated asynchronously from a shared pool. For this purpose, a global memory allocator is provided that leaves the mapping into thread-specific address spaces to the user.

Protection Management: Protection domains are a software-level abstraction that control the access to kernel objects, communication channels, storage (physical memory) and logical address spaces. For this purpose, the Protection Management subsystems implements an object-capability model [DVH66] with communication semantics based on the actor model [HBS73].

The Protection Management subsystems provides components that handle system calls from the local application thread, system calls received from other threads, and requests from external management tools received via communication channels. This is a two step procedure. First, a shared component implements the capability-based dispatching of requests to service components. These service components are responsible for the service-specific decoding of the requests into method calls or appropriate actions and the encoding of responses. Also, this subsystem provides portals for basic communication between applications.

Scheduling: The scheduling subsystem provides quite different components mechanisms on the various layers. On the Event Executive layer (EVE), tasklets are implemented as abstraction for lightweight kernel-internal asynchronous tasks. A tasklet scheduler for each hardware thread is provided, which processes local and received tasklets. Finally, the subsystem provides mechanisms to create and manage optional hardware-thread groups.

On the Kernel Services layers (ALS, DKS), execution contexts are implemented as abstractions for application threads. These include mechanisms to suspend and activate execution contexts on a hardware thread and migration mechanisms between hardware threads.

On the Application Runtime Services layer (ARtS) and the Supervisor layer, more specific application process and thread models are implemented based on the execution contexts and tasklet-based coordination of the lower layers. This concerns the creation, activation and migration of application threads. The Hardware Thread Manager (HWTM) components keep track of the utilization state of the hardware threads and help to direct thread creation requests to free hardware threads.

Communication: This subsystem implements the various communication channels required for the tasklet exchange between hardware threads, request/response exchange to external tools, remote system call channels, and user-mode communication. The channel variants cover, for instance, message queues for small fixed-size messages over cache-coherent shared memory as well as non-coherent PCIe2 shared memory. For the use in simulators/emulators, simple channels for communication over virtual serial ports are provided. Finally, the communication subsystem is responsible for the high-level implementa-

tion of preemptive notifications that interrupt threads and enforce handling of important messages.

Processor Management: The Processor Management subsystem contains mostly support components on the Synchronous Local Services layer. These provide access to hardware-specific configuration registers, the thread-local interrupt controllers, and the frequency/voltage scaling. They set up the processor-specific interrupt handling as well as the system call entry and return. On the Distributed Kernel Services layer, the processor wide thermal and power management is implemented.

Debugging Support: The debugging subsystem provides easy to use mechanisms for generating textual log messages and detailed trace events. The log/event recording is designed from ground up for highly concurrent usage with high throughput and low overhead.

This subsystem would also be the home of remote execution control services like debugger stub. Such components provide means to suspend and resume software and hardware threads and to query memory contents and component states.

3 Physical View: Deployment, Placement, Connections

The physical view discusses deployment aspects such as the placement of software components onto processors and their hardware threads as well as the placement of crucial data structures in the physical and logical address spaces. This also depends on the layout of private, local and shared memory regions and the sharing of address spaces accross hardware threads.

In MyThOS the execution of software threads is not dedicated and controlled by a central scheduling mechanism. An application has control about when and on which computing resource an EC is initialized and activated. For this purpose an EC can be dynamically or statically bound to a hardware computational unit, i.e. a hardware thread residing on a specific core and CPU. Applications are also provided with the possibility to move ECs to different computational units on its own discretion. When using the thread management facility provided by MyThOS, the responsibility of selecting a computational unit for a newly created EC is transferred to the kernel. However, there are no guarantees about which computational unit is selected for a specific EC.

Upon creation, each EC is assigned to a Protection Domain (PD), which manages the logical address space and therefore the physical memory of this EC. The address space of a PD represents a logical address space, i.e. the mapping from logical memory addresses to a region of physical memory, which is then shared between all ECs that belong to that PD. In addition, each EC is able to freely allocate or deallocate further physical memory and to map or unmap it into that address space.

Physical memory is structured into several parts, that serve different purposes. The init segment is placed into the area of physical memory with the lowest addresses. It contains the boot code for both the BSP and all APs. Since these code parts are executed in real mode, they cannot be located at high memory addresses. These code parts are called as the first instructions of a newly booted processor and are responsible to set up basic page tables to switch to protected mode. These page tables and all page tables created in later stages are located in the following part of the physical memory. Subsequently, all kernel code is located, which during runtime

is mapped into the upper half of the logical address space. Directly behind the Core Local Memory is located, where configurations for each hardware thread is stored. The remainder of main memory is left for runtime mappings by the operating system, which are managed by the memory management facility within the kernel. From this memory area mappings into both user- and kernel-space are possible, which are then located in either the lower or upper half of the logical address space.

4 Process View: Dynamic Interactions

In MyThOS dynamic interactions between processes residing on different computational units are realized with an asynchronous communication strategy. More formally, different software components residing on different processor cores can interact over well defined interfaces and communication channels by transmitting *asynchronous tasks* between each other. Thereby, tasks are serialized within so called “active messages”, containing parameters for their remote invocation and a reference to an object and a function which is to be invoked. The asynchronicity is then achieved by decoupling a request from its actual execution time via differently prioritized task-queues for each processing unit. When a task is picked-up from a queue it is executed by the respective processing unit to completion. This allows for an efficient implementation on task-level, where all tasks of one priority level can share the same stack, memory segment and protection level.

Our current strategy involves three different priority levels for the tasks, i.e. *kernel*, *user* and *idle* tasks.

Kernel Tasks are high-priority jobs which involve kernel-specific functions. They have the highest eligibility for execution and are therefore processed in advance to user or idle tasks. Typical representatives for such high-priority tasks are for example remote system call invocations, or epilogues of locally received interrupts. In general, kernel-tasks require almost immediate kernel intervention and should not be preempted by user-activities on the receiving core’s side. Moreover, although the kernel address space layout may vary between hardware threads, the layout of the kernel code and core-local storage is the same. Protection mechanisms between kernel components are not needed and they may fully trust each other. Therefore, communication channels can pass function and object pointers without the need of pointer conversions.

User Tasks are communication requests on application-level, i.e. between user-processes and components. This kind of communication and asynchronous task processing has a lower urgency than the one on kernel-level. From a more abstract point of view, tasks placed in the user-level queue of a core are used to call user-space code on the receiving core. Since there is no global

scheduling mechanism for processes, the process activation is controlled by the asynchronous user-level communication requests.

Idle Tasks have the lowest priority within our architecture. They are processed locally by a core only in presence of spare time, i.e. when no kernel- or user-activities are pending and waiting for their execution. Idle-Tasks are destined for local cleanup operations, e.g. data deallocation or management of freed address spaces within free memory pools.

Tasks are represented by either a function pointer, a functor or a lambda expression, regardless of their priority level. For transmission those are wrapped into a tasklet and placed in the queue of the given priority level of the receiving core. When issuing a call to an execution context, an additional function can be passed. This *continuation* is a function pointer, functor or lambda expression, which is valid in the sending core's context. After the completion of the task by the execution context of the receiving core, the continuation is placed into the task queue of the sending core. Thereby the continuation notifies the sending core of the finished execution and enables the core to process the results of the original message, without actively waiting for the task to finish. Continuations can also be used to be notified of the completion of other types of messages.

Communication among different hardware threads or different nodes may sometimes be difficult, because of differences in data representation like e.g. endianness or word lengths. In order to provide applications with an easy way of communication, stubs are employed. On the sender side the stub encodes the message into a standardized format, which is then used for transmission. On the receiver side a corresponding stub function unwrappes the received message and delivers it to the appropriate endpoint. Thereby, stubs act as a transparent way of communication, without the need of the application to know technical details about the receiver side. This method can for example be used to transparently delegate system calls to a distant hardware thread, rather than executing them locally. However, the interface for applications does not change, regardless of the actual point of execution.

5 Development View: Implementation Artifacts, Interfaces, Types, Modules

The development view describes implementation aspects on the architectural level of abstraction. This covers, for example, interaction rules between architectural layers, implementation strategies for interfaces and components, the compilation process, as well as the basic file and folder structure.

In order to be able to actually reuse software components, a strategy for their configuration and deployment is needed. This is an old problem with many good solutions. Unfortunately, the initial development work in MyThOS showed that a single runtime-only configuration strategy is not sufficient because some components depend on specific hardware and compilers whereas some performance critical options require a static compile-time configuration. Hence, the following three configuration strategies have to work together in MyThOS and the implementation of components and interfaces needs to be aligned to these strategies.

Run-time instantiation. MyThOS, as many-core operating system, bears similarity to distributed system because many of the system's services cannot simply exist globally but, instead, require a separate instance per core or hardware thread. This is achieved through the use of software components. The run-time deployment creates, configures and connects these components. The reusability of the components is improved further by applying the *dependency injection* principle, preferring *composition* over specialisation by inheritance, and using generic type parameters instead of interfaces where suitable.

Link-time composition. Not all parts of the operating system code can be compiled for every target architecture and even within a single architecture not everything is needed for each configuration. Therefore, the build configuration has to select which files are compiled and linked into the final kernel image, application libraries and support tools.

Compile-time configuration. Unfortunately, some configuration options cannot be deferred to the run-time instantiation because they interact with hardware

requirements or would introduce indirections with significant impact on the performance. For example, the global descriptor table's layout is clearly configurable but has to be known to low-level code at compile-time. In order to achieve this configurability, MyThOS uses file-based code *modules* that represent implementation and configuration variants. These modules are also used to configure the link-time composition of the kernel and tools.

The following subsections discuss the applied implementation strategies for software components and the code modules in more detail.

5.1 Software Components

A software component is an object or a group of objects that work together to provide a specific service and is designed with substitute-ability, reusability, and composition in mind. The distinction to arbitrary language-level objects are the more restricted rules governing the component's life cycle and interactions. Language-level objects are used to implement components, but also to implement data structures, containers, communication channels, functors, messages and so on.

Most of the kernel's components follow a quite simple life cycle. The high-level boot sequence of the operating system is implemented in `boot/PLATFORM/kernel.cc`. Here, all *static components* of the kernel, which will be present until power-off, are instantiated and configured. A couple of typical deployment structures can be reused across platforms and various parts of the kernel. In order to facilitate their reuse, the deployment code is factored out into helper components, which names begin with `Deploy` for easy recognition.

Later on, additional components representing Execution Contexts, Protection Domains, and similar have to be created on demand. These *dynamic components* will be torn down based on garbage collection strategies or based on management decisions from higher layers such a the cloud management or hypervisor. In order to factor out the details of their creation, static factory components are used. For example, the external application loader is connected to a factory for Execution Contexts and Protection Domains, which enables the loader to start actual applications.

The reusability of components is improved further by applying the *dependency injection* principle. This strategy separates all code responsible for instantiation and configuration from the component's implementation. Instead of searching for other needed components from within the component, all such dependencies and configuration options are injected from the outside via constructor arguments and

setter functions. This helps to reduce the component's dependencies and assumptions about its runtime environment. For example, such components can be instantiated and connected very differently depending on the chosen hardware platform without the need to change the component implementation.

The components can interact with each other through multiple mechanisms: Local Synchronous, Local Asynchronous , Remote Method Call, and System Call Interfaces. The mechanism of choice depends on the architecture layer and the differences between local and remote interactions.

The most well know mechanism are *Local Synchronous Interfaces*, which are conventional C++ method calls. The "client" component has a pointer to a "server" component and the pointer is either typed to the server's implementation type or the a interface class that contains virtual method declarations. In MyThOS, the interface class names use the `I` prefix. At the moment of the method call, the control flow moves into the server component and immediately processes the request. Results can be passed simply via the return value.

In contrast, *Local Asynchronous Interfaces* guarantee that the control flow returns immediately and the request is processed asynchronously. Results cannot be passed via the return value in consequence. No results are needed in many occasions, for example, when emitting event notifications to higher architecture layers. In the cases where a result or completion acknowledgement is needed, the last method argument has to be a continuation object (also known as callback function). In MyThOS, the Local Asynchronous Interfaces are named with the `AI` prefix.

Both interface styles use virtual methods in order to hide the server component's implementation details. Routing all accesses through virtual methods enables the easy substitution of component variants during deployment. However, it incurs an implementation dependency to the imported interfaces and a runtime overhead because the static compiler cannot optimise across calls to virtual methods. In C++, template programming can be used to replace the imported interface with a template type argument. This can be combined with the virtual method trampoline technique in order to enable compiler optimisation whenever the implementation type is known to the client.

The *Remote Method Call Interfaces* and *System Call Interfaces* are generated from interface definitions and look like local synchronous or asynchronous interfaces to the client component. Their major difference is the communication mechanism, which is used internally to transmit the call to the server side. On the server side, the message handling processes these requests similar to active messages. For system calls, argument passing is implemented via the caller's stack and the processor's

system call instructions are used to transition synchronously into the operating system kernel. Within the MyThOS kernel, a mapping to asynchronous interfaces is used in order to be compatible with the non-blocking task-based execution model.

The synchronous interfaces are easy to use and usually the most efficient. However, the methods behind these interfaces are not allowed to wait, for example, for answers from remote or asynchronous interactions. Whenever an asynchronous interaction is needed inside the method implementation, an asynchronous interface has to be exported. Using just asynchronous interfaces for everything is not an option, because they require a basic infrastructure for their execution model and require careful resource management.

Therefore, synchronous interfaces shall be used for purely local interactions from higher to lower architecture layers. In order to avoid deadlock situations, asynchronous interfaces are used for interactions from lower to higher layers. Just the distributed kernel services and asynchronous local services export asynchronous interfaces to higher layers.

5.2 Build-Time Configuration Modules

The process of building MyThOS involves compiling a large amount of source files and linking them appropriately. A convenient way to describe dependencies among source files and build configurations for different target architectures and use cases is provided. Using this mechanism, each functional module can be described independently of the others. This makes the development and integration of additional modules and the adaption to new target architectures more convenient. This configuration facility consists of two parts, which are module specification and target configuration.

5.2.1 Module Specification

Each functional unit, e.g. units for debug output, are described separately from each other. Each source file of the functional unit can be assigned to one of the following groups

Header Files can be included by other modules and should specify the interface of the unit.

Kernel Files are compiled into the kernel image and provide the functionality of the unit.

This corresponds to the traditional separation of interface and implementation in C++. Additionally a module can specify compiler flags, that have to be used to compile the unit.

5.2.2 Target Configuration

The configuration of a build target contains a list of the modules, that should be included into the kernel image for the given target. It also specifies the source directories, containing all required modules and their specifications, and the target directory for the configuration process.

After both module specification and target configuration are present, the provided configuration utility is used to create a valid build directory. For this purpose, it traverses all the modules, which are specified in the target configuration, resolves their dependencies from their source code and gathers all required files into the build directory. Additionally it creates a makefile, which can be used to build MyThOS for the given architecture. After the configuration process, the build directory contains all source files, which are required to build MyThOS, in an appropriate folder structure. Using this utility, the MyThOS kernel can be conveniently adapted to different architectures and extended by additional modules.

Bibliography

[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.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[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.

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