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

MyThOS D4.3 Developer Cookbook

Stefan Bonfert, Vladimir Nikolov, Robert Kuban, Randolf Rotta

28. März 2017

Zusammenfassung

Dieses Deliverable richtet sich an externe Anwender und dokumentiert die Tools als auch die Grundzuege des Programmiermodells anhand von Beispielen (Kochrezepten).

Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministerium für Bildung und Forschung unter dem Förderkennzeichen 01IH13003 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Inhaltsverzeichnis

1 Introduction

MyThOS, as a reduced operating system, gives application developers the possibility to control its behavior and configuration at a fine-grained level. For this purpose, it can be configured to suit the needs of the application to a much higher degree, than traditional operating systems. However, in order to utilize this configurability, application developers need additional knowledge about MyThOS in order to write performant applications. This document is intended to supply required knowledge to them and serve as both a starting point for development and a reference document.

Therefore, this document contains both descriptions of MyThOS' interfaces and examples how to use them.

In Section 2, an example application is presented, including hints for the development of different applications and information about the compilation process required for them to run on MyThOS. Section 3 describes the system call interface of MyThOS, that may be used by applications. Since this interface is very reduced and generic, various user space objects are provided to conveniently interact with the kernel and the application's kernel objects. These are further described in section 4. Section 5 serves as a guideline for the development of own kernel objects, that can be used to move functionality from the user space into the kernel. Finally, Section 6 illustrates these development steps with a concrete example.

2 Creating and Compiling a User Application

The MyThOS kernel automatically loads a default initial application after the operating system is booted and the available hardware is appropriately configured. The code of the init application lies in the `kernel/app/init-app/app` directory of the kernel source tree and the `int main()` function in `init.cc` is the entry point for the application execution. In the following just some code excerpts are depicted explaining the general application behavior. For better understanding please introspect in parallel the original source code in `init.cc`.

The init application first creates a predefined kernel object (KO) of type `mythos::Example`[1] using a preconfigured untyped memory space (`kmem`), portal (`portal`) and object factory (`mythos::init::EXAMPLE_FACTORY`):

```
1  mythos::Portal portal(mythos::init::PORTAL, msg_ptr);
2  mythos::UntypedMemory kmem(mythos::init::UM);
3  mythos::CapMap myCS(mythos::init::CSPACE); // Capability Space
4  ...
5
6  int main() {
7    ...
8    mythos::Example example(mythos::init::APP_CAP_START);
9    auto res1 = example.create(portal, kmem, mythos::init::EXAMPLE_FACTORY);
10   res1.wait();
11   ASSERT(res1.state() == mythos::Error::SUCCESS);
12   ...
13 }
```

[1] implemented in kernel/runtime/kobject/runtime/Example.hh

Each kernel object inherits from a generic type `KObject`[2]. It encapsulates (and thus is referenced by) a certain kernel capability (e.g. `mythos::init::APP_CAP_START`) which is later used for object registration and query within application's capability space. While `example` is only a user-space wrapper of the KO, the actual object creation within the kernel is triggered by its `create(...)` function. The `create(...)` function is object-specific and has to be provided by the object developer. It triggers the required creation steps, which are performed on behalf of a special predefined object protocol within the kernel. That protocol guides the creation procedure and genral usage of the object (see later Section 5 for more information). The procedure automatically enacts the referenced object factory and registers the newly created object within application's capability space. Thereby, the `portal` object is required for communication with the kernel and it performs the actual system call and protocol invocation. Since system invocations are asynchronous, waiting for the result of each kernel invocation via `res1.wait()` blocks until completion.

As next, the init application invokes a basic `printMessage(...)` function of the `Example` object (on line 1) and finally deletes that object from its capability space (on line 4):

```
1 auto res2 = example.printMessage(res1.reuse(), obj, sizeof(obj)-1);
2 res2.wait();
3 ASSERT(res2.state() == mythos::Error::SUCCESS);
4 auto res3 = myCS.deleteCap(res2.reuse(), example);
5 res3.wait();
6 ASSERT(res3.state() == mythos::Error::SUCCESS);
```

For each new invocation the same protal is reused. The result of each invocation is a reference to the engaged portal and its invocation buffer. Beyond retrieving system call results, this mechanism also allows for checking the execution status of a respective invocation, for example for polling or waiting for a result (see `kernel/runtime/async/runtime/PortalBase.hh`). MyThOS also supports a variety of error codes which are defined in `kernel/mythos/error/mythos/Error.hh`, like for example `SUCCESS`, `INVALID_ARGUMENT`, `NOT_IMPLEMENTED`, `INVALID_CAPABILITY`, `REQUEST_DENIED` etc.

For more information on the peculiarities and the procedure of object creation and invocation from a user application please refer to Sections 5 and 6.

Finally the init app creates and activates two execution contexts `ec1` and `ec2` which (when later executed) simply print a short debug message (line 6) and then suspend

[2] see `kernel/runtime/async/runtime/PortalBase.hh`

(line 7). The following code excerpt shows the creation and activation procedure of one `ExecutionContext`:

```
1  mythos::PageMap myAS(mythos::init::PML4);  // Address Space
2
3  void* thread_main(void* ctx)
4  {
5    char const str[] = "hello thread!";
6    mythos::syscall_debug(str, sizeof(str)-1);
7    return 0;
8  }
9
10 int main() {
11 ...
12 mythos::ExecutionContext ec1(mythos::init::APP_CAP_START);
13 auto res1 = ec1.create(portal, kmem, mythos::init::EXECUTION_CONTEXT_FACTORY,
14                         myAS, myCS, mythos::init::SCHEDULERS_START,
15                         thread1stack_top, &thread_main, nullptr);
16 res1.wait();
17 ASSERT(res1.state() == mythos::Error::SUCCESS);
18 ...
19 }
```

An execution context is also a predefined MyThOS kernel object. First a user space wrapper `ec1` is instantiated (line 12) and linked with a certain capability[3]. The actual creation and activation of the thread is triggered again by its `create(...)` function, which accepts a variety of parameters, as for example the thread's address and capability space, stack, a thread factory and scheduler references, as well as a reference to thread's entry point `thread_main(...)` (line 3).

The init application is exported as a module (see `kernel/app/init-app/mcconf.module`), which can be selectively included in kernel's build process via the `mcconf` tool. The module definition is the place, where also the build configuration and procedure of the application can be customized in terms of build commands, compiler/linker flags and further options. These are automatically included to the respective build targets for the application in the final `Makefile` of the kernel. However, the application is linked to a separate ELF-binary, which is automatically embedded into the kernel image. A basic application loader in the kernel automatically queries, loads and executes the application during runtime. A dynamic application loader is intended to be included in future versions of MyThOS. The latter will allow for on-demand application loading and start from a Xeon Phi host system.

[3] In this example the capability previously used for the `Example` object is reused.

The init application provides a good starting point and a general guideline for the development of new MyThOS applications. An implementation should follow that example and it must be integrated in a similar way into the directory structure of the kernel (e.g. `kernel/app/{my-app}/app/init.cc`). It can be exported and composed within the kernel as a different module (e.g. `kernel/app/{my-app}/mcconf.module`). The provided module can be then simply added to a general kernel configuration like for example `kernel-amd64.config`. However, an application should at least implement an `int main()` function. The kick-off procedure of the init application is defined within the assembly file `kernel/app/my-app/app/start.S`, which should be reused by custom applications.

Limitations:

Due to the reduced nature of MyThOS, applications have to be compiled carefully to be executable on MyThOS. This will also be true once the dynamic application loader is in place. Currently, MyThOS does not offer the full interface of commonly used libraries like the LibC. Therefore, the compiler has to be instructed not to include exception handling into the code (`-fno-exceptions`) and omit including stack protectors (`-fno-stack-protector`). The linker has to be configured to not link against the standard libraries (`-nostdlib`) and link against all other libraries statically (`-static`), since dynamically loaded libraries are not yet supported by MyThOS. However, GCC-specific functions are included in the code of MyThOS. Therefore, these libraries have to be included into the code (`-lgcc`).

Since MyThOS is designed to run on manycore architectures, which often have limited support for advanced istruction set extensions, these should be disabled in the compiler. When compiling an application for the Intel© Xeon Phi™, all of MMX, SSE, SSE2, SSE3 and 3DNow should be disabled (`-mno-mmx -mno-sse -mno-sse2 -mno-sse3 -mno-3dnow`).

3 Basic System Call Interface of MyThOS

Compared to general purpose operating systems, MyThOS offers a relatively small set of system calls for the interaction of applications with the operating system kernel. In the following all system calls are listed, including a description of their semantics. The user space wrappers of those system calls are defined in `kernel/mythos/invocation/mythos/syscall.hh`.

Note: This is the lowest level interface to the system. More general abstractions based on that definitions exist. For example consider object and system related invocations through a portal defined in `kernel/runtime/async/runtime/PortalBase.hh`. These functions finally decompose into basic system calls as defined in `kernel/mythos/invocation/mythos/syscall.hh`.

void syscall_debug(char const* start, uint64_t length): This system call is used to print debug messages to the text console. In the current implementation, the output is redirected to a static memory buffer, from where it is then output to the user's terminal by an helper application running on the MyThOS host. The parameter `start` points to the buffer containing the text to be output, `length` determines the length of the string. After the output is written into the buffer, the syscall returns to the application. The actual output to the user's terminal is then executed asynchronously on the host computer.

void syscall_exit(uint64_t rescode): This system call is called to exit the currently running application. When issuing this system call, the currently running `ExecutionContext` exits itself and yields control to the kernel. The parameter `rescode` follows the usual semantics of a result code, where the value 0 indicates successful execution of the application or thread, while all other result codes indicate unsuccessful execution, with a semantic, that is up to the application.

KEvent syscall_invoke(CapPtr portal, CapPtr object, void* userctx): All functionality in the kernel of MyThOS is encapsulated into kernel objects. Using this system call, applications can interact with these kernel objects in a well-defined and controlled manner. This system call can be used to request a specific service, from the kernel object `object`. For the communication a portal is used, that is passed to the system call in the parameter `portal`. The called kernel object does not necessarily have to be located locally. The translation of the application's system calls to either local or remote system calls is transparent to the user. An additional reference `userctx` is passed, which is later returned to the application upon reply of the kernel object. This system call is executed asynchronously. Additional system calls have to be issued to check for the result of the invocation. *Note:* as already mentioned, for particular cases there are already wrappers providing a higher level interface for handling results and invocation status, as for example `PortalBase.h`.

KEvent syscall_poll(): This system call checks, whether the application received a response to a previously issued system call or an invocation by a remote portal. If a notification was received, this is indicated in `state`-member of the returned `KEvent`-object. The reference, that was passed to the corresponding invoke system call (either locally or remotely via a portal) is made available to the application in the `user`-member of the returned object. This system call returns immediately, even if there was no notification pending.

KEvent syscall_wait(): This system call follows the same semantics as the `syscall_poll` system call, but blocks until a notification was received. It's therefore follows a synchronous processing model.

KEvent syscall_invoke_poll(CapPtr portal, CapPtr object, void* userctx)

KEvent syscall_invoke_wait(CapPtr portal, CapPtr object, void* userctx): These two system calls combine an invocation of a kernel object with the check for pending notifications. They follow the semantics of the individual system calls as described above and are available in a blocking (wait) and a non-blocking (poll) implementation. The combination of two system calls into a single one resembles the common send-receive pattern and potentially can reduce the number of required system calls and therefore the number of switches from user space to kernel space and back.

Note: for portal based communication with the kernel, future objects of type `kernel/runtime/async/runtime/FutureBase.hh` are used in order to poll and synchronize on results. These are abstractions of the basic `syscall_{poll,wait}()` functions and the `PortalBase` type inherits the implementation provided in `FutureBase.hh`.

4 Interface of the MyThOS Runtime Environment

Due to the focus of the MyThOS project on developing an operating system, the support for applications in terms of runtime environment are limited, compared to general purpose operating systems like Linux. The main point of interaction with the operating system is the use of portals, which represent the communication endpoint between the application and MyThOS. The interaction between the application and the portals is detailed in the previous section of this document.

For the interaction of applications with other kernel objects, some user space representations are provided to application developers, that encapsulate the

communication with the respective kernel object and therefore provide the developer with a more specialised interface than generic portals. In the following, these user space representations and their interfaces are presented. For further reference, the user space representations are located in the folder `kernel/runtime/kobject/runtime`, the message formats used for communication with the respective kernel objects are specified in protocols, which are stored in `kernel/mythos/invocation/mythos/protocol` and the kernel space implementations of the kernel objects can be found in the folder `objects`. The filenames and folder names are identical to the headings and class names used in this chapter.

4.1 CapMap

A capability map object is used to manage the application's capability space by creating, moving, deleting or revoking capability pointers. These are references to kernel objects that can be accessed and utilised by the application.

A new capability map object can be instantiated by calling:

```
PortalFutureRef<void> create(PortalRef pr, UntypedMemory kmem, CapPtr factory,
    CapPtrDepth indexbits, CapPtrDepth guardbits, CapPtr guard)
```

This method creates a new capability map via the portal `pr` in the memory area `kmem` using the factory referenced by `factory`. Additionally, indexbits and guardbits can be set, which are further detailed in Deliverable D2.2.

An existing capability map can be utilised to manipulate the capability entries, which it contains. New capabilities can be created using the methods

```
PortalFutureRef<void> derive(PortalRef pr, CapPtr src, CapPtrDepth srcDepth,
    CapPtr dstCs, CapPtr dst, CapPtrDepth dstDepth, CapRequest req)
```

and `reference(...)`, which implements the same method signature. These methods create new capabilities, that are children of or references to an existing capability `src` in the called capability space. The newly created capabilities are then inserted into the capability space `dstCs` at the location `dst`. Additionally, desired capability flags can be specified in the parameter `req`.

The method

```
PortalFutureRef<void> move(PortalRef pr, CapPtr src, CapPtrDepth srcDepth, CapPtr
    dstCs, CapPtr dst, CapPtrDepth dstDepth)
```

can be used to move a capability `src` from the called capability space to another capability space `dstCs` at location `dst`. Thereby, capabilities can be separated into multiple capability spaces to pass them to individual applications or application parts.

To delete a capability from the capability map the method

```
PortalFutureRef<void> deleteCap(PortalRef pr, CapPtr src, CapPtrDepth srcDepth)
```

can be called, which deletes the capability `src` from the map. Furthermore, the method

```
PortalFutureRef<void> deleteCap(PortalRef pr, KObject src)
```

can be used to delete a given kernel object from the map, regardless of its assigned corresponding capability. For these two methods, respective `revokeCap(...)` methods exist with the same signatures, that can be used to revoke individual capabilities.

4.2 DebugSink

This class represents a small wrapper for the debug system call and can be used to output debug information (or any other text) to the text console and thereby make it available to users or developers. Its functionality is contained in the function

```
void write(char const* str, size_t len)
```

which takes a string and its length as input parameters and outputs this string to the console.

4.3 ExecutionContext

As detailed in deliverable D2.2, Execution Context objects are used to represent the execution state of a single software thread. Therefore, they are a very important interaction point for application developers to control the degree of parallelism in their applications and to achieve actual concurrency within MyThOS. Their interface looks as follows:

```
PortalFutureRef<void> create(PortalRef pr, UntypedMemory kmem, CapPtr factory,
    PageMap as, CapMap cs, CapPtr sched, void* stack, StartFun start, void*
    userctx)
```

This function is used to create a new `ExecutionContext` kernel object. The parameter `pr` specifies the pointer to the portal that should be used for this operation. `kmem` specifies the kernel memory area, in which the new EC should be allocated. `factory` is a pointer to the kernel object responsible for the actual creation of the object and has to be provided by the operating system beforehand. In the parameters `as` and `cs` the address space and the capability space of the application are passed. The parameter `sched` specifies the scheduling context, which is designated to execute the newly created EC, while its user space stack is placed at the virtual address specified by the parameter `stack`. `start` is a pointer to the initial function executed by the EC in user space and `userctx` can be used by the application developer to later identify individual ECs and is passed to the function specified by `start`.

```
PortalFutureRef<void> configure(PortalRef pr, PageMap as, CapMap cs, CapPtr sched)
```

This method is used to configure the EC, which is accessed via the local portal `pr` to use the page map, capability map and scheduling context, specified by `as`, `cs` or `sched` respectively, in the future.

4.4 Frame

A frame object represents a memory frame, that can be accessed by the application and is allocated from an `UntypedMemory` object via the method

```
PortalFutureRef<void> create(PortalRef pr, UntypedMemory kmem, CapPtr factory,
    size_t size, size_t alignment)
```

Additionally to the local portal `pr`, the allocation pool `kmem` and the factory for `Frame` objects `factory`, the size and the alignment requirements for the newly allocated frame are passed as arguments in the parameters `size` and `alignment`.

The method

```
PortalFutureRef<protocol::Frame::Info> info(PortalRef pr)
```

returns various information about a given memory frame, like its size and alignment accompanied by permissions, that are set to this memory frame, like its executable and writable flags.

4.5 PageMap

Objects of the class `PageMap` represent a single table in the hierarchical page table structure of the application. On creation, each page map and their contents are bound to a specific level in the page table hierarchy. The entries of a page map point to either a memory frame of the same level or a page map of the next lower level. `PageMap` objects can be created using the method

```
PortalFutureRef<void> create(PortalRef pr, UntypedMemory kmem, CapPtr factory,
    size_t level)
```

Besides the local portal `pr`, the memory area used for allocation `kmem` and the page map factory `factory`, the level of the page map has to be specified.

After creation (and potentially configuration) a page map can be put into action by calling the method

```
PortalFutureRef<protocol::PageMap::Result> installMap(PortalRef pr, PageMap
    pagemap, uintptr_t vaddr, size_t level, MapFlags flags)
```

Thereby, the page map is mapped into the higher level table `pagemap`, which then contains mappings for the virtual addresses in the range of `vaddr` at the given level of page tables. Additionally, flags can be set on the page map, e.g. write permissions or caching behaviour. A page map can be removed from the page table hierarchy using the method

```
PortalFutureRef<protocol::PageMap::Result> removeMap(PortalRef pr, uintptr_t
    vaddr, size_t level)
```

With the method

```
PortalFutureRef<protocol::PageMap::Result> mmap(PortalRef pr, Frame frame,
    uintptr_t vaddr, size_t size, MapFlags flags)
```

a given memory frame can be mapped to the virtual user space address `vaddr`. Again, flags can be specified to customise the mapping process. The reverse operation is available via the method

```
PortalFutureRef<protocol::PageMap::Result> munmap(PortalRef pr, uintptr_t vaddr,
    size_t size)
```

which removes the frame from the page map. Existing mappings can be remapped from `sourceAddr` to `destAddr` by calling the method

```
PortalFutureRef<protocol::PageMap::Result> remap(PortalRef pr, uintptr_t
    sourceAddr, uintptr_t destAddr, size_t size)
```

Flags, that were previously set for a page map or a frame can be altered to a new set of flags by using the method

```
PortalFutureRef<protocol::PageMap::Result> mprotect(PortalRef pr, uintptr_t
    vaddr, size_t size, MapFlags flags)}
```

The semantics of these method calls are comparable to the linux system calls `mmap`, `munmap`, `mremap` and `mprotect`.

4.6 Tasklet

Tasklets are used for two purposes in MyThOS. They represent the smallest unit of executed code and at the same time serve as a token in a token-based flow control system. A tasklet encapsulates a single functor, that can be set and retrieved. For this purpose the `Tasklet` class exposes two self-explanatory methods:

```
template<class FUNCTOR>
void set(FUNCTOR const&& fun);

template<class MSG>
MSG get() const;
```

4.7 UntypedMemory

This object represents a memory pool, from which memory can be allocated to be either used by the application (Frames) or to be used to host kernel objects. The `UntypedMemory` objects only expose one function to the user space, which is defined as

```
PortalFutureRef<void> create(PortalRef pr, UntypedMemory kmem, CapPtr factory,
    size_t size, size_t alignment)
```

This method is used to derive an `UntypedMemory` object from an `UntypedMemory` object, splitting one of these objects into two. Thereby, a hierarchical memory structure is built up. For this purpose, it is passed a pointer to the local portal `pr`, the original `UntypedMemory` object `kmem`, a pointer to an object capable of creating new `UntypedMemory` objects `factory` and the new objects desired size and memory alignment.

4.8 Library Functions

In addition to the functions detailed above, that directly interact with the operating system via system calls, a small set of convenience functions was is implemented, that conform to the default interfaces known for these functions from the C/C++ string library. Currently, the set of functions comprises of `memcpy` to copy memory ranges, `memset` to initialise memory regions, `strlen` to determine the length of a string and `strcmp` to check two strings for the lexical order and equality.

5 Development of Kernel Objects

This section describes the steps necessary to create a new kernel object. First, the `IKernelObject` interface (Listing 1) is discussed. It has to be implemented by all kernel objects in order to participate in the resource management and processing of system calls. Then, the allocation and initialisation of kernel objects through factories is described. Finally, an overview over the synchronisation monitors and the weak references mechanism is given.

As already explained in Section 2, the kernel source includes an example object, which serves as a starting point for the creation of custom Kernel Objects and as a mock for testing object allocation and deletion. For further information, consider the documentation embedded into the source code. Most of the code embedded here is part of the Example object.

5.1 Ressource management

The `addressRange()` method is used by the resource inheritance tree for parent-ship tests. The method has to return the physical address range that is used by the object. The range has to be a superset of the address ranges of all derived child objects in the capability inheritance tree. Kernel objects that will have no children can simply use the default implementation.

`deleteCap()` and `deleteObject()` are both part of the multistep deletion process. `deleteCap()` notifies the object that a capability is going to be deleted. For most objects, this will have no effect for reference and derived capabilities. For original capabilities, the objects recursively deletes its capability entry and schedules itself

```
class IKernelObject : public ICastable {
public:
  virtual Range<uintptr_t> addressRange(Cap self);
  virtual optional<Cap> mint(Cap self, CapRequest request);
  virtual optional<void> deleteCap(Cap self, IDeleter& del) = 0;
  virtual void deleteObject(Tasklet* t, IResult<void>* r) = 0;
  virtual Range<uintptr_t> objectRange() const = 0;
  virtual optional<CapEntryRef> lookup(Cap self, CapPtr needle, CapPtrDepth
      maxDepth);
  virtual void invoke(Tasklet* t, Cap self, IInvocation* msg);
};

class ICastable {
public:
  virtual optional<void const*> vcast(TypeId id) const;
};
```

Listing 1: The `IKernelObject` interface.

```
optional<void> ExampleObj::deleteCap(Cap self, IDeleter& del) {
  if (self.isOriginal()) del.deleteObject(del_handle);
  return Error::SUCCESS;
}
```

Listing 2: Example handler for capability revocation.

```
1 void ExampleObj::deleteObject(Tasklet* t, IResult<void>* r) {
2   monitor.doDelete(t, [=](Tasklet* t){ this->_mem->release(t, r, this); });
3 }
```

Listing 3: Example handler for asynchronous object deletion.

for asynchronous deletion using the `IDeleter` object given as a parameter. Listing 2 shows a simple example for an object without any embedded capability entries.

`deleteObject()` carries out the final step of the asynchronous deletion. The object will use its deletion monitor to wait for all outstanding asynchronous request to complete and then asynchronously requests the deletion from the `UntypedMemory` object it was allocated from. An example is shown in Listing 3. The memory's `release()` method will call the object's `objectRange()` to query the address range that is used by the object itself. First, the objects virtual destructor `~IKernelObject()` is called in order to let the object free all contained additional memory blocks. Then, the objects memory is freed. Finally, the deleter is notified about the completion by replying to the `IResult` pointer.

The `mint()` method is used when creating a reference or derived capability. This mechanism allows to restrict access rights, add communication badges and similar by altering the data portion of the capability according to the request argument. The meaning of the request arguments is defined by the actual object because the capability data portion is interpreted only by the object itself. If a kernel object does not accept the minting request, it should return an error code.

5.2 System Calls as Capability Invocation

The `lookup()` method is used by the system call entry code in order to find the target object that matches the given capability pointer. The search in the caller's capability space recursively walks through its capability maps. The default implementation of this method fails and returns an error code. A custom implementation is only necessary if the object acts as a capability map.

The `invoke()` method implements the receiving side of the capability invocation to a kernel object. Its default implementation simply returns with an error. Listing 4 shows an example of the invocation handling. The `invoke()` method dispatches the message based on the message label and asynchronously calls the appropriate

```
1  void ExampleObj::invoke(Tasklet* t, Cap self, IInvocation* msg) {
2    switch (msg->getLabel()) {
3    case 0:
4      return monitor.request(t, [=](Tasklet* t) { printMessage(t, self, msg); });
5    default:
6      msg->replyResponse(Error::INVALID_REQUEST);
7    }
8  }
9
10 void ExampleObj::printMessage(Tasklet* t, Cap self, IInvocation* msg) {
11   // may use data portion of self for access control and badges...
12   auto msgdata = msg->getMessage();
13   // do something useful with the message data...
14   message.replyResponse(Error::SUCCESS);
15   monitor.requestDone();
16 }
```

Listing 4: Example invocation handler.

implementation. The implementation checks the caller's access rights by inspecting the capability `self` that was used to access the object. Then, the arguments are copied from the message into local buffers and checked for validity. The copy is crucial for security because otherwise concurrently running application threads could manipulate the arguments after the sanity check have been passed. After carrying out the request, the object replies and releases the its monitor.

The `IInvocation` interface provides several more methods that help, for example, to look up additional kernel objects in the callers capability space, access the callers `ExecutionContext`, and the caller's logical address space configuration.

5.3 Interactions between Kernel Objects

When a capability lookup is used to retrieve a pointer to another kernel object, this pointer is still typed as `IKernelObject`. This is sufficient to pass invocation messages to the object. However, the invocation mechanism is complex and possibly costly. Therefore it would be more efficient and convenient to call the kernel object's implementation-specific methods directly.

Normal C++ programs use the compiler-generated runtime type information (RTTI) and the `dynamic_cast` conversion. This information is not available in the kernel

```
virtual optional<void const*> vcast(TypeId id) const override {
  if (id == TypeId::id<UntypedMemory>()) return this;
  if (id == TypeId::id<IAllocator>()) return static_cast<IAllocator const*>(this);
  return Error::TYPE_MISMATCH;
}
```

Listing 5: Example run-time type conversion in the `UntypedMemory` object.

```
auto alloc = ptr->cast<IAllocator>();
if (alloc) { // checks whether the conversion was successful
  alloc->allocate(...);
}
```

Listing 6: Casting a kernel object into a more specific type.

in order to reduce its size. Instead, the conversion is accomplished through the `ICastable` mechanism. The method `vcast()` is implemented by each kernel object in order to hand out pointers to more specific interfaces. Listing 5 shows an example. The converted pointer is then retrieved via the `cast()` helper method as shown in Listing 6.

5.4 Object Creation

Kernel objects are created with memory from a single `UntypedMemory` via the help of a factory object. The only exception are a few initial objects that exist statically. The factory has to implement the `IFactory` interface, which is shown in Listing 7.

The `umcap` capability belongs to the `UntypedMemory` that called the factory. This value is later needed to complete the insertion into the resource tree. The `mem` argument points to the `IAllocator` interface of the `UntypedMemory`. Because the memory calls the factory the memory's synchronisation monitor has been entered,

```
class IFactory {
public:
  virtual optional<void> factory(Cap umcap, IAllocator* mem, IInvocation* msg,
                                 CapEntry* tgtentry) = 0;
};
```

Listing 7: The `IFactory` interface.

synchronous calls to `mem` can be used safely. The created object has to store `mem` in order to be able to release its memory later. The `tgtentry` points to the capability entry that shall receive the first capability that points to the newly created kernel object. Finally, the `msg` arguments contains the invocation buffer that was passed from userland with additional initialisation parameters for the new object.

An example factory implementation is shown in Listing 8. First, the target capability entry is acquired and locked in order to prevent data races (line 3). Then, memory for the object is allocated and the object is constructed in this memory. The factory and the object constructor can allocate additional memory from `mem` (line 2). If the allocation fails, the memory has to be released and the target capability entry has to be reset (lines 5–7).

Finally, the original capability of the created object is inserted as child of the `UntypedMemory` that it was allocated from (line 10). The parent entry is retrieved from the invocation message and should match `umcap`. The operation fails when the parent object was deleted concurrently. In this case, the object and the target entry have to be released (lines 11–14). The original capability can be either stored as member variable in the created object, as shown in the example, or can be stored directly in the target entry. In the example, a reference capability is written to the target entry.

5.5 Serialisation Monitors

The monitors are used by kernel objects to serialise asynchronous method calls and select the place (hardware thread) that processes the calls. Currently, four monitor variants are available. The choice depends on the synchronisation needs. In most cases, the `NestedMonitorDelegating` is a good choice.

DeletionMonitor. This monitor implements a reference counter that is used to delay the deletion request until all counted references were released. It is used as base class for the other monitors. The methods `acquireRef()` and `releaseRef()` increase or decrease the reference counter, respectively. The `doDelete()` method schedules a Tasklet to be executed the next time the reference counter reaches zero. The Tasklet is processed at the place of the `releaseRef()` caller. If the reference counter is zero already, the Tasklet processed immediately.

```
optional<void> ExampleFactory::factory(Cap umcap, IAllocator* mem,
                                       IInvocation* msg, CapEntry* tgtentry) {
  if (!tgtentry->acquire()) return Error::LOST_RACE;
  auto obj = mem->create<ExampleObj,64>(mem, ...); // create with 64b alignment
  if (!obj) {
    tgtentry->reset();
    return Error::INSUFFICIENT_RESOURCES;
  }
  Cap tgtcap(obj);
  auto res = cap::inherit(msg->getCapEntry(), obj->ownRoot, umcap, tgtcap);
  if (!res) {
    mem->release(obj);
    tgtentry->reset();
    return res.state();
  }
  return cap::inherit(obj->ownRoot, tgtentry, tgtcap, tgtcap.asReference());
}
```

Listing 8: An example factory implementation.

SimpleMonitorHome. This variant comes close to the classic monitor concept. The `request()` method schedules the given Tasklet at a predefined place called `home`. At the end of each request handler implementation, `requestDone()` has to be called in order to release the reference counter. Because all Tasklet scheduled by the monitor are processed on the same hardware thread, they all are mutually exclusive.

NestedMonitorHome. This extension of the previous monitor differentiates between asynchronous *requests* and *responses* in order to implement a form of nested locking. The next requests is processed only after the previous has finished by calling `requestDone()`. Requests can issue sub-requests to other kernel objects and the last response has to call `responseAndRequestDone()` instead of just `responseDone()`. Developers should be careful with cyclic dependencies between kernel objects because the mutually exclusive request are prone to deadlocks – like in any scenario with nested locks that can be entered just once.

NestedMonitorDelegating. This monitor has the same *request*/*response* interface as `NestedMonitorHome` but the monitor is not bound to a specific home place. Instead, the first request that acquires exclusive access sets the home to its caller's place. All responses to sub-requests will processed at this place. Further requests

```
1 template<class Subject, class Object, class Revoker=RevokeByUnbind>
2 class CapRef : public CapRefBase {
3 public:
4   optional<void> set(Subject* subject, CapEntry& src, Cap srcCap);
5   optional<Object*> get() const;
6   void reset();
7 protected:
8   virtual void revoked(Cap self, Cap orig);
9 };
```

Listing 9: The weak reference capability interface.

that do not acquire exclusive access will be processed at this place, too. The last `requestDone()` or `responseAndRequestDone()` releases the exclusive access, such that later requests select a new home. This behaviour is based on the concept of *delegation locks*.

5.6 Connecting Objects through Weak Reference Capabilities

Pointers between kernel objects require special care because the target object can be deleted concurrently. This would leave dangling pointers, which result in fancy bugs and headaches. In many cases, the associations between kernel objects are established by the transfer of access rights. In MyThOS this is implemented by passing a capability to the kernel object and the capability revocation can be used to clean up such pointers. This idea is implemented by the `CapRef` class.

Listing 9 summarises the principal interface of `CapRef`. Basically, the object contains a capability entry that holds a reference capability and a pointer to the `subject` capability holder. The `set()` method inherits the reference from the `src` and `srcCap` pair. It fails if a concurrent call to `set()` was faster or when the source entry was deleted concurrently. The `get()` method returns a pointer to the capability's object casted to the desired object interface. It fails if the capability was never set or was revoked meanwhile. The `reset()` method clears the reference and called automatically from `set()`.

During capability revocation, the `revoked()` method will be called synchronously. It can be overridden by derived types. The default implementation calls `Revoker::apply(subject, object, self, orig)`. The default `RevokeByUnbind` calls `subject->unbind(object)` synchronously in order to notify the reference holder. An example usage is given in Listing 10.

```
1 class Foo : public IKernelObject {
2 public:
3   optional<void> bindPortal(CapEntry& pcap); // calls portal.set(...)
4 protected:
5   CapRef<Foo, IPortal> portal;
6   friend class RevokeByUnbind;
7   void unbind(optional<IPortal*> o); // reacts to the revocation
8 };
```

Listing 10: An example use of weak references.

6 Example: Extending a Kernel Object

When developing an application, the developer at some point may decide to move parts of its functionality to the kernel space to improve the application's performance. In this section we give an example, how to extend an existing kernel object with an additional function, that then can be called from the user space application. The process of initially developing a kernel object including all required methods is laid out in the previous Section 5.

For convenience, each kernel object that is invocable from user space should feature a runtime representation, which serves as a wrapper. For our Example object suppose that it is extended with a `printMessage()` method. The according runtime representation is listed in Listing 11. Runtime representations of kernel objects are typically located in the folder `kernel/runtime/kobject/runtime` and encapsulate the calls to the portal of the current Execution Context (e.g. see Listing 11, Line 13). Here, a runtime method has to be defined, that calls the `tryInvoke` method on a portal and may pass additional arguments to that method.

```
1 #include "runtime/PortalBase.hh"
2 #include "mythos/protocol/Example.hh"
3 #include "runtime/UntypedMemory.hh"
4
5 namespace mythos {
6
7   class Example : public KObject
8   {
9   public:
10    Example(CapPtr cap) : KObject(cap) {}
11
12    PortalFutureRef<void> create(PortalRef pr, UntypedMemory kmem, CapPtr
          factory) {
```

```
13        return pr.tryInvoke<protocol::Example::Create>(kmem.cap(), _cap, factory);
14      }
15
16      PortalFutureRef<void> printMessage(PortalRef pr, char const* str, size_t
            bytes) {
17        return pr.tryInvoke<protocol::Example::PrintMessage>(_cap, str, bytes);
18      }
19    };
20
21  } // namespace mythos
```

Listing 11: Object's Runtime Representation

For data exchange between user space and kernel space, invocation buffers are used, that can be arbitrarily used by applications. Their contents are described by protocols, that are stored in `kernel/mythos/invocation/mythos/protocol`. Listing 12 shows the overall definition of the protocol for our example object. The corresponding class contains a list of available methods (`enum`), that has to be extended with the new one. This list is important for the method dispatching process in the object, i.e. `dispatchRequest()`, which is explained later.

```
1  #include "mythos/protocol/common.hh"
2  #include "mythos/protocol/UntypedMemory.hh"
3  #include <cstring>
4
5  namespace mythos {
6  namespace protocol {
7
8  struct Example {
9    constexpr static uint8_t proto = EXAMPLE;
10
11   enum Methods : uint8_t {
12     PRINT_MESSAGE,
13   };
14
15   struct PrintMessage : public InvocationBase {
16     typedef InvocationBase response_type;
17     constexpr static uint16_t label = (proto<<8) + PRINT_MESSAGE;
18     PrintMessage(char const* str, size_t bytes)
19     : InvocationBase(label,getLength(this))
20     {
21       if (bytes>InvocationBase::maxBytes) bytes = InvocationBase::maxBytes;
22       this->bytes = uint16_t(bytes);
23       this->tag.length = uint8_t((bytes+3)/4);
24       memcpy(message, str, bytes);
```

```
25      }
26      uint16_t bytes;
27      char message[InvocationBase::maxBytes-2];
28    };
29
30    struct Create : public UntypedMemory::CreateBase {
31      Create(CapPtr dst, CapPtr factory) : CreateBase(dst, factory) {}
32    };
33
34    template<class IMPL, class... ARGS>
35    static Error dispatchRequest(IMPL* obj, uint8_t m, ARGS const&...args) {
36      switch(Methods(m)) {
37      case PRINT_MESSAGE: return obj->printMessage(args...);
38      default: return Error::NOT_IMPLEMENTED;
39      }
40    }
41
42  };
43
44  } // namespace protocol
45  } // namespace mythos
```

Listing 12: Object's Protocol Definition

Additionally, the format of the invocation buffer has to be specified here, inside a method-specific struct which includes a constructor that receives all arguments, that were previously passed to the runtime object. Furthermore, the method `dispatchRequest` has to be adapted to call the correct method `printMessage()` (line 37) of the kernel object when receiving a message. This method of the kernel object then carries out the actual work that the developer wanted to move into the kernel.

The actual kernel object's implementation is stored in `kernel/objects/<objectName>/objects` as a respective class `ExampleObj` defined in `Example.hh` and `Example.cc`. Here, the method called by the protocol's dispatcher has to be implemented. This method is responsible for extracting the required information from the invocation buffer and processing it. Data can be returned to the application through the invocation buffer. Listing 13 shows the declaration of the object class and, besides the typical object methods as previously explained in Section 5, the extended method `printMessage()` in line 16. The respective implementation of the method is shown in Listing 14.

```
class ExampleObj : public IKernelObject
{
public:
  ExampleObj(IAsyncFree* mem) : _mem(mem) {}
  ExampleObj(const ExampleObj&) = delete;

  // default IKernelObject methods
  optional<void const*> vcast(TypeId id) const override;
  optional<void> deleteCap(Cap self, IDeleter& del) override;
  void deleteObject(Tasklet* t, IResult<void>* r) override;
  void invoke(Tasklet* t, Cap self, IInvocation* msg) override;

protected:
  ...
  friend struct protocol::Example;
  Error printMessage(Tasklet* t, Cap self, IInvocation* msg);

protected:
  IAsyncFree* _mem;
  ...
};

class ExampleFactory : public FactoryBase
{
public:
  static optional<ExampleObj*>
  factory(CapEntry* dstEntry, CapEntry* memEntry, Cap memCap, IAllocator* mem);

  Error factory(CapEntry* dstEntry, CapEntry* memEntry, Cap memCap,
                IAllocator* mem, IInvocation*) const override {
    return factory(dstEntry, memEntry, memCap, mem).state();
  }
};
```

Listing 13: Kernel Object's Declaration

```
Error ExampleObj::printMessage(Tasklet*, Cap self, IInvocation* msg)
{
  mlogex.info("invoke printMessage", DVAR(this), DVAR(self), DVAR(msg));
  auto data = msg->getMessage()->cast<protocol::Example::PrintMessage>();
  mlogex.error(mlog::DebugString(data->message, data->bytes));
  return Error::SUCCESS;
}
```

Listing 14: Kernel Object's Method Implementation

As a last point, note the `ExampleFactory` class in `kernel/objects/example/objects/Example.hh`, which provides a standard object factory for the creation process of that KO. That factory inherits a generic method `factory(...)` (line 29), which is automatically invoked during the object creation process and redirects to a custom static factory method implementation (in line 27). There the developer has to implement the specific object creation procedure by the factory. A standard implementation can be found in `kernel/objects/example/objects/Example.cc`:

```
1  optional<ExampleObj*>
2  ExampleFactory::factory(CapEntry* dstEntry, CapEntry* memEntry, Cap memCap,
3                          IAllocator* mem)
4  {
5    auto obj = mem->create<ExampleObj>();
6    if (!obj) return obj.state();
7    Cap cap(*obj);
8    auto res = cap::inherit(*memEntry, *dstEntry, memCap, cap);
9    if (!res) {
10     mem->free(*obj); // mem->release(obj) goes throug IKernelObject deletion
           mechanism
11     return res.state();
12   }
13   return *obj;
14 }
```

Listing 15: Kernel Object's Method Implementation

The standard factory implementation basically creates the object with the help of a special memory allocator (line 5) and registers the new object application's capability space (line 8). For more information please refer to the inline documentation in the source code.