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

MyThOS D3.3
Softwareentwicklungsplan

Stefan Bonfert, Vladimir Nikolov, Robert Kuban, Randolf Rotta

18. Oktober 2016

Zusammenfassung

Dieses Dokument umfasst den Strategieplan für die Weiterentwicklung des Prototypen und Anleitungen für Systementwickler. Abschnitt 1 gibt Beispiele und Hinweise für die Entwicklung eigener Kernel-Objekte, die Behandlung von Systemaufrufen, die Synchronisation asynchroner Aufgaben, sowie die Verwaltung von Beziehungen zwischen Kernel-Objekten. Abschnitt 2 fasst nützliche Erweiterungen des bestehenden Kernels zusammen. Dazu gehören zum Beispiel Verbesserungen bei der Fehler- und Performanceanalyse, Schnittstellen zur Konfiguration der Interruptbehandlung und Capability Transfers zwischen Kommunikationsportalen.

Inhaltsverzeichnis

1 Kernel Development

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.

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.

```
1  class IKernelObject : public ICastable {
2  public:
3    virtual Range<uintptr_t> addressRange(Cap self);
4    virtual optional<Cap> mint(Cap self, CapRequest request);
5    virtual optional<void> deleteCap(Cap self, IDeleter& del) = 0;
6    virtual void deleteObject(Tasklet* t, IResult<void>* r) = 0;
7    virtual Range<uintptr_t> objectRange() const = 0;
8    virtual optional<CapEntryRef> lookup(Cap self, CapPtr needle, CapPtrDepth
         maxDepth);
9    virtual void invoke(Tasklet* t, Cap self, IInvocation* msg);
10 };
11
12 class ICastable {
13 public:
14   virtual optional<void const*> vcast(TypeId id) const;
15 };
```

Listing 1: The `IKernelObject` interface.

```
1 optional<void> ExampleObj::deleteCap(Cap self, IDeleter& del) {
2   if (self.isOriginal()) del.deleteObject(del_handle);
3   return Error::SUCCESS;
4 }
```

Listing 2: Example handler for capability revocation.

1.1 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 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.

```
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.

`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.

1.2 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 **??** shows an example of the invocation handling. The `invoke()` method dispatches the message based on the message label and asynchronously calls the appropriate 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

```
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.

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.

1.3 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 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.

```
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.

The converted pointer is then retrieved via the `cast()` helper method as shown in Listing 6.

1.4 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, 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

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

Listing 7: The `IFactory` interface.

```
1  optional<void> ExampleFactory::factory(Cap umcap, IAllocator* mem,
2                                         IInvocation* msg, CapEntry* tgtentry) {
3    if (!tgtentry->acquire()) return Error::LOST_RACE;
4    auto obj = mem->create<ExampleObj,64>(mem, ...); // create with 64b alignment
5    if (!obj) {
6      tgtentry->reset();
7      return Error::INSUFFICIENT_RESOURCES;
8    }
9    Cap tgtcap(obj);
10   auto res = cap::inherit(msg->getCapEntry(), obj->ownRoot, umcap, tgtcap);
11   if (!res) {
12     mem->release(obj);
13     tgtentry->reset();
14     return res.state();
15   }
16   return cap::inherit(obj->ownRoot, tgtentry, tgtcap, tgtcap.asReference());
17 }
```

Listing 8: An example factory implementation.

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.

1.5 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.

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 that do not acquire exclusive access will be processed at this place, too. The last

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

Listing 9: The weak reference capability interface.

`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*.

1.6 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.

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

Listing 10: An example use of weak references.

2 Strategy for further Development

2.1 Capability Transfer

Capability transfer and capability unwrapping in inter-process calls are not necessary for the applications targeted by MyThOS. However, they promise to simplify the interaction between unrelated application without a common supervisor.

In order to support the future implementation of such transfers, the invocation buffer format mirrors the seL4 message format and reserves the necessary space. In order to support a capability transfer, a portal implementation would read the desired target capability address from the receiver's invocation buffer, look up the respective entry in the receiver's capability space, and insert a reference capability there.

2.2 Image-based Debugging Facilities

Because the system memory in MyThOS is just 4GiB large, it is feasible to create an image of the whole system memory by reading it from the host over PCIe. Host access to the coprocessor memory is 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 and interesting details about the state of each hardware thread are hidden in their registers.

Two approaches can be implemented in order to obtain a consistent kernel image: *Snapshot Images* capture a consistence state by issuing a non-maskable interrupt (NMI), which dumps the processor state into each thread's NMI stack and, and

then, busy waits 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 NMI handler, all threads wait before the next Tasklet is executed. This allows to create a kernel image that is easier to analyse because all transactions and critical sections were completed.

2.2.1 Doorbell Interrupts and User-Level Interrupt Handling

The Intel Xeon Phi Processor has a doorbell interrupt, which can be triggered from the host software in order to wakeup the sleeping processor or interrupt its current activity in order to handle urgent requests. This mechanism is based on configuration registers in the processor's PCIe MMIO space and the XeonPhy's IOMMU. The respective configuration can be added through a KNC-specific source code module. One imminent application would be the interrupt broadcast for debugging purposes.

The present interrupt handling in MyThOS is sufficient to handle traps and interrupts inside the kernel but is not extensible. A useful improvement would be the implementation of user-level interrupt handling. The basic idea is to represent interrupt gates as kernel objects and triggered interrupts send a message or notification to a portal or execution context.

2.3 Exploiting Hardware Transactional Memory

Hardware Transactional Memory such as Intel's TSX extension can speed up the execution by enabling lock elision and much simpler non-blocking algorithms. Previous related work about transactional memory in microkernels [SLEV15, Fuc14] focused on the latency of the IPC path. The results were disappointing because one or two atomic exchange or CAS operations are difficult to beat.

MyThOS contains quite a lot non-blocking algorithms, especially for double-linked queues of pending messages and the resource inheritance tree. These can be replaced by hardware transactional memory in order to reduce complexity and busy waiting. Very likely, TSX would not speed up the serialisation monitors and the tasklet queue.

2.4 Conservative Sleeping

When there is no work to do, a hardware thread should switch into a sleep state in order to minimize energy consumption and to lend its thermal budget to other cores. Currently, a hardware thread in MyThOS starts sleeping as soon as there are no more executable Tasklets and Execution Contexts (application threads). However, this is not optimal because waking up from a sleep state requires an interrupt to be send by another thread, which imposes a latency penalty. Entering and exiting deeper sleep modes also adds a considerable overhead.

A possible solution to this problem is introducing an active waiting phase, which resembles the *pause* phase for mutexes first introduced by Ousterhood. Heuristics might be used to find a tradeoff between active waiting phase length, hence energy consumption, and the latency penalty of waking up from a sleep state. MyThOS can support the development of such heuristics by providing diagnostic data like active waiting time or sleeping phase length.

2.5 Sleeping through MWAIT

Many x86 Prozessors, except the Intel XeonPhi Knight Corner, support a more efficient sleep&wakeup mechanism for the hardware threads. It is based on monitoring a cache line for access by other hardware threads with the `MONITOR` instruction. Then the thread can sleep with `MWAIT` and is woken up whenever the monitored line is modified or evicted from the thread's cache.

The benefit of this approach is, that client threads that send a Tasklet do not need to check whether they have to send an IPI interrupt. The sleeping thread does not need to go through the interrupt entry routine when waking up. In MyThOS this would allow to leave out the release of the thread's Tasklet queue prior to sleeping.

2.6 Adapt Tracing for Tasklet Queues and Synchronisation Monitors

The previous implementation of MyThOS used fixed-size Ringbuffers for the exchange of Tasklets between hardware threads. In addition, Tasklets were allowed to enter multiple queues at the same time and were allocated dynamically. This introduced interesting challenges for the global reconstruction of thread-local traces.

In contrast, the present design is much simpler because all Tasklets have to follow a request-response cycle between kernel objects. This allows to simplify the trace recording and reconstruction. The port of the trace subsystem to the new design is still pending.

References

[Fuc14] Raphael Fuchs. Hardware transactional memory and message passing. 2014.

[SLEV15] Till Smejkal, Adam Lackorzynski, Benjamin Engel, and Marcus Völp. Transactional IPC in Fiasco.OC. *OSPERT 2015*, page 19, 2015.