Whitepaper

Programming Model

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

Disclaimer

Das im Folgenden vorgestellte Programmiermodell für Anwendungen, die auf das MyThOS Betriebssystem aufsetzen sollen, beschreibt die essentielle Vorgehensweise bei der Anwendungsentwicklung, die Schnittstelle zwischen Applikationen und dem Betriebssystem-Kernel und das erwartete Verhalten des Kernels. Um auch nicht deutsch-sprachige Entwickler zu unterstützen, wird dieses Dokument in englischer Sprache bereitgestellt.

Introduction

This document summarizes the essential parts of the MyThOS operating system from application development perspective. It provides a general development and programming model for applications which shall be executed on top of the MyThOS-kernel. The suggested API and functionality provided by means of a set of system calls and methods is primarily driven by the specific MyThOS kernel architecture, which was established with focus on management and fast work distribution over a magnitude of integrated computing resources, e.g. CPU cores. On the one hand, applications are required to follow that model in order to gain a maximum level of configurability, optimal execution performance and parallelism. On the other hand, specific application requirements for functionality on kernel level were also reflected in this API.

We start our discussion with a general description of the core MyThOS kernel components and their functionality. Then we present the current system interface, i.e. a set of system calls or library functions which can be invoked by the application, some of which have already been fully implemented and tested for the current kernel version at the time of writing this. Some additional functions were identified during recent discussions about applications behavior and an analysis of their respective kernel-level support required – these functions will be implemented and provided in a second iteration.

A special focus of the API rests on sharing and dynamically managing resources. Generally, dynamic resources allocation and usage implies bookkeeping mechanisms on kernel-level, which may result in additional overhead for the system. Therefore, the dynamicity, performance and minimalism (in terms of the system interface) always have to be considered. This document provides a conceptual comparison between static versus dynamic management strategies, with respect to the management of Execution Contexts (i.e. threads), isolation of applications and their components as well as memory organization.

I. General Overview of Components

Based on the description of the initial MyThOS architecture, which was provided within the project deliverable D2.1 [1], we now give a general overview over the most relevant system components from application perspective, i.e. Execution Contexts, Protection Domains, Portals and Storage Objects.

A. Execution Context

The central computational software “unit” in MyThOS is an Execution Context (EC) which is a software construct providing a minimal executional environment for applications tasks and logic. Logically, an EC corresponds to a traditional *software thread* or *lightweight process* (LWP) as known from general purpose operating systems like Linux (among many other). However, MyThOS ECs are not bound statically to the actual code they should execute, but the latter is assigned dynamically on each EC invocation, as will be shown later. So, essentially the EC corresponds to a “wrapper” around traditional software threads. Furthermore, in MyThOS threads (or ECs) are not encapsulated on a higher abstraction level by processes. Also, their execution is not decided and controlled by a central scheduling mechanism, but on application level. Thus, an application has control over when and on which computing resource an EC is initialized and activated. Consequently, an EC can be dynamically or statically bound to a hardware computational unit, i.e. a hardware thread (HWT) residing on a specific core and CPU. Each EC within the system can be uniquely identified independent of its current location. Therewith, ECs can be moved to different computational units if the application decides to do so. This is however currently considered secondary for the use cases to be supported and may be pursued in future iterations.

B. Protection Domain

Although MyThOS’ ECs are completely autarchic and act autonomously in terms of allocation, mapping and sharing of physical memory and computational resources, they can be grouped and hierarchically organized within so called Protection Domains (PD). The PDs provide isolation between different thread groups, e.g. different applications or application components, operating on separate address spaces with different resource allocation and functional capabilities. However, if only one application is running within the system all ECs can belong to the same PD or they can be also dynamically assigned to different PDs according to application-specific isolation requirements or functional necessity.

Figure 1 depicts the basic system model of MyThOS with respect to an execution context: Each EC consists of a *user-stack* for the application logic and a *kernel-stack* for system calls and interrupt handling. The user-stack is provided by the application when an EC is created, as will be shown later. The kernel-stack is automatically maintained for all ECs by the operating system itself.

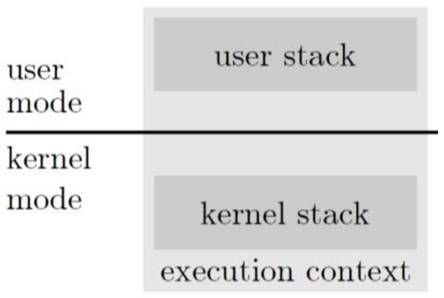

FIGURE 1: BASIC SYSTEM MODEL OF AN EXECUTION CONTEXT

Generally, when an EC is created it is bound to a specific HWT and a Protection Domain (PD), which manages the isolation between the application logic and the operating system kernel. A Protection Domain consists of three subcomponents: (a) an *address space*, (b) an *object space* and (c) a *system* *call interface*. Several different implementations of a Protection Domain can coexist within a system implementation for a specific use case. They can be dynamically assigned to newly created ECs. Changing the PD for an initialized EC at runtime requires some mechanism for capabilities transfer, which is currently considered secondary in the development plan.

The *address space* of a Protection Domain represents a logical address space, i.e. the default mapping from logical addresses to a region of physical memory, which is shared between all ECs that belong to that PD. The default address mapping provided by a PD is inherited by the ECs when they are created and bound to that PD. In addition to this, each EC is able to freely allocate/de-allocate further physical memory and to map/un-map it into that address space. Thus, the address space and its physical mapping can be extended and modifications are instantly visible to all ECs sharing the same PD. On kernel-level each Protection Domain is equipped with its own memory management interface. Invocations of memory-related functions and system calls from user-space are forwarded to the respective kernel-instance over the Protection Domain of the calling thread and its private memory interface. This allows on the one hand for different PD-dependent allocation strategies, but also for application- and EC-specific memory organization.

Kernel objects encapsulate standard system-functionality and services and they are accessible from kernel space through predefined interfaces and lookup facilities. The *object space* of a PD defines and controls an application's mapping and access to kernel functionality by means of a set of referenced kernel objects and services. However, that functionality is provided on the application level by connecting objects with system calls. This connection is realized and controlled by the respective Protection Domain of an Execution Context. On this way it is possible that different PDs provide a different implementation of the same system call. With this model even requestforwarding can be implemented, where system calls are delegated to a different PD providing their actual implementation. On this way, even a hierarchy of PDs can be implemented, if required by the particular use-case, and system invocations can be delegated to different levels of that hierarchy.

The system call interface is the system entry point, i.e. a hardware-dependent mechanism for handling system calls. It is responsible to dispatch a system call invocation to an appropriate kernellevel service or function. This way, a system call can be served by any EC (within any PD).

Besides Protection Domains, MyThOS defines two further components for the applications, namely *portals* and *storage objects*:

C. Portals

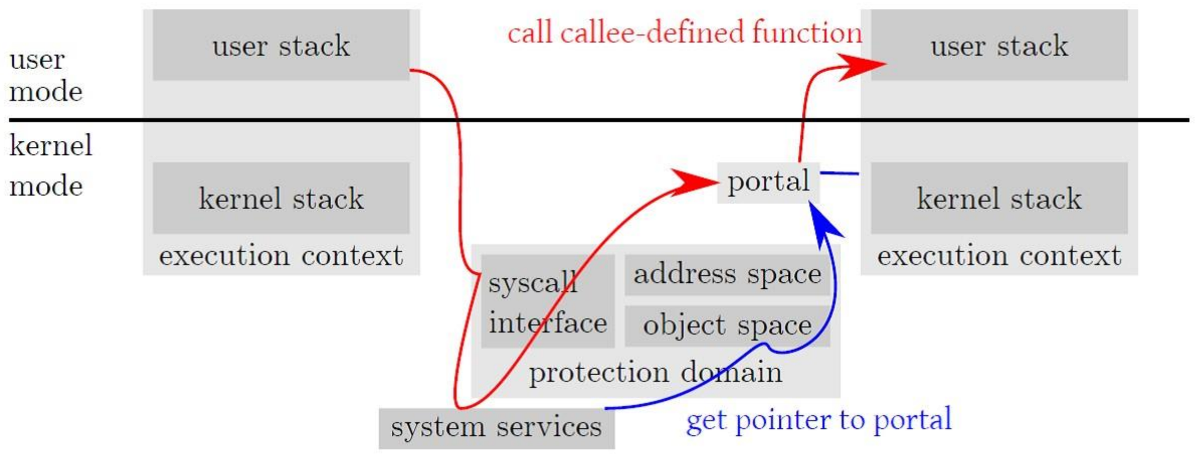

FIGURE 2: USAGE OF PORTALS

Portals are used to allow for interactions between ECs residing in different Protection Domains and address spaces. A portal is a dynamically configurable endpoint for message-based communication. It is associated with an EC and a fixed function reference. Figure 2 depicts a usage scenario of a portal which is invoked as a consequence of a system call. A system service handling the system call may not implement the functionality itself, but fulfill it by invoking a remote instance via a portal bound to a specific EC. Whenever a message arrives at the portal its contents are parsed and the referenced function is scheduled for execution by the respectively called EC. Portals allow easy communication between applications, or application parts in different address spaces and PDs, because no foreign function pointers need to be known and both calling and called EC operate just in their own address space.

D. Storage Objects

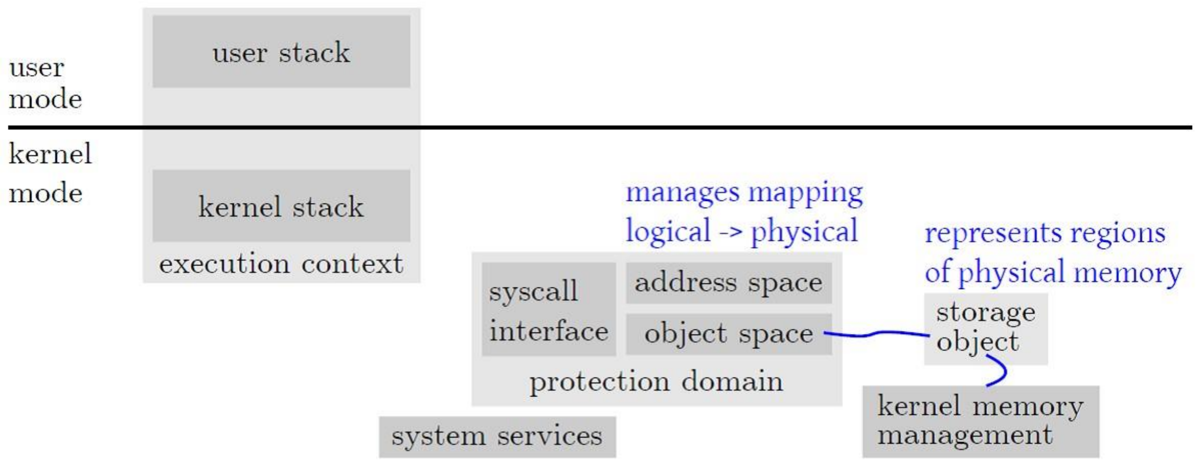

FIGURE 3: MEMORY AND STORAGE OBJECTS

Storage objects represent a logically contiguous piece of physical RAM. Depending on the implementation, they can grow and shrink or be sparse. Figure 3 exemplifies the connection between an EC, its PD-specific memory management interface and storage objects. Storage objects give memory an identity, which allows it to be dynamically used and even shared between different address spaces. In MyThOS a storage object is represented by a physical memory area (PMA) object. ECs can arbitrarily allocate PMAs and then map them into their logical address space. A garbage collection mechanism is also necessary for cleaning up unused and released resources. It is automatically provided by the kernel without the need for a special intervention from user-space.

Some memory management functions (e.g., see `mmap`, `alloc` and `free` in chapter II.D) automatically allocate and release storage objects in behind. In such cases, the respective PMA handles for that physical memory are not exposed to the application. For that reason, that physical memory remains invisible or “anonymous” from application point of view.

II. API Reference

This chapter gives an overview of the current user-level API methods with respect to manage Protection Domains, Execution Contexts, Portals, memory and asynchronous system calls. Each method provided can be implemented as a wrapper of a particular system call or within a user-level library which may invoke several system calls in order to accomplish the specific task.

A. Application Management and Protection Domains

The amount of Protection Domain types in a system will be defined by the actual system requirements of a specific use-case – however, the amount of actual PD-instances is dynamic, since privileged ECs can dynamically create and destroy instances of certain PD-types and bind other ECs to them. Such privileges are regulated by the according PD to which an EC belongs.

For the management of protection domains the following system calls were defined:

`int createPD()`

This method can be used on application-level in order to create and register a new Protection Domain instance of a certain type within the kernel. The return value of that method is the unique ID of the new Protection Domain instance. Thus, the operating system kernel is responsible for the organization of PDs and their unique IDs. A calling thread has to keep track of its created child PDs and it also has to release them when they are not needed anymore.

`void releasePD(int id)`

Unregisters and releases the Protection Domain with the provided id.

`int getCurrentPD()`

Returns the ID of the Protection Domain to which the calling Execution Context is actually bound.

`void changePD(int ec, int pd)`

This method changes the Protection Domain of an Execution Context `ec` registered in the PD specified by pd. This method implies a capability transfer for the target Execution Context `ec`. It is up to the kernel to delay changing the Protection Domain for a thread, as long as there are pending system calls executed in the scope of the target-threads old PD. The appliance of that system call depends on the calling threads capabilities as mandated by its current Protection Domain.

Usage

After a new domain instance is created with `createPD()` new Execution Contexts can be created and connected to that PD. This may be accomplished by the thread that created the new domain or any other thread that belongs to this particular PD. The capabilities of the new ECs apply automatically with respect to their assigned Protection Domain. The respective PD id of an Execution Context is accessible via `getCurrentPD()`. The respective `releasePD(id)` system call implies resource management issues, i.e. the Protection Domain cannot be deleted as long as it is used by any Execution Context or referenced via another Protection Domain.

A default Protection Domain is provided by the system, which is automatically set up for a newly created Execution Context, unless a specific PD instance is specified during EC creation.

The implementation details of a Protection Domain are out of the scope of this deliverables, please refer to the code documentation on GIT.

B. Thread Management

This chapter elaborates the system calls used for the creation, invocation and release of Execution Contexts. The presented methods in this chapter imply synchronous calls, while asynchronous calls are discussed later in chapter II.E.

`int createEC(int pd, int hwtid, addr_t stackptr)`

This method creates a new Execution Context which is initialized on a hardware thread with ID `hwtid`. The new EC belongs to the Protection Domain with the ID given by pd or to the default Protection Domain if a null value is specified. The value given by `stackptr` is the start address of the ECs user-space stack. The returned value is the ID of the newly created Execution Context.

`addr_t callEC(int id, addr_t funcptr, addr_t args)`

This method invokes a referenced Execution Context identified by `id`. The invoked EC then executes the function given at address `funptr`. `args` passes a pointer to an arguments structure in some shared memory between caller and callee. If the Execution Context is already active with another call, this call is <u>enqueued</u> for later execution. `callEC` is synchronous and returns only after the execution of the referenced function `funcptr` completes (asynchronous calls are discussed later in chapter II.E). In the current implementation `args` has to point to shared memory and the caller supplies the function pointer. Therefore, `callEC`() is useful only inside the same Protection Domain and address space (for cross-PD and address space invocations see portals in chapter II.C). The return value is a pointer to a structure with the result value(s) for that call.

`void exit(int result)`

Forces the calling EC to immediately stop and leave user-mode. The thread is not destroyed. The `result` value is pushed back to the caller of that thread. The Execution Context is not destroyed automatically. If there are pending calls to that EC, the next one is executed.

`void releaseEC(int id)`

This method triggers the release and destruction of an Execution Context identified by `id`.

With the distributed (asynchronous) architecture of MyThOS, all release operations require special care. An Execution Context can request the release of another EC (or portal) while calls and messages for it are still pending or in transit. These objects can be released only after all tasks have finished. This leads to the need for a sophisticated garbage collection strategy on kernel-level.

Alternatively, the kernel can trust the application and get along without kernel-internal garbage collection. Though this would not be acceptable for general purpose operating systems, MyThOS is focusing on HPC and HPC-cloud applications with a tendency to isolated and controlled application execution, where little conflicts arise from different application spaces. For example, the management services of the cloud environment has to keep track of active and releasable resources anyway. In that case, garbage collection can be merged with the management implementation.

Most HPC applications work well without any dynamic thread management. The application simply starts with one Execution Context per hardware thread and generally does not create new ECs as this would imply other application spaces.

C. Portal Management

As already explained, portals provide a mechanism for fast interoperation between different ECs running in the same or even in different address spaces and protection domains – this could also allow communication between different applications. The reason is that the kernel provides an abstraction based on portal identifiers, with which no foreign function pointers have to be known in order to make a remote function call and to get a response. The following methods are provided for portal management on application-level:

`int createPortal(int ec, addr_t funcptr)`

This method creates and registers a new portal object for an Execution Context identified by `ec`. It binds the function `funcptr` to this portal, so that this function is invoked upon calling `callPortal`. Consequently, in the current implementation, the portal creator and the executor ECs have to reside within the same Protection Domain and address space. It returns the `id` of the portal.

`addr_t callPortal(int id, addr_t message)`

This method invokes a predefined portal identified by `id` and invokes the registered function `funcptr` (see `createPortal`). The message provided at address `message` is copied into the address space of the EC bound to that portal and the configured handler function is run by that EC at some point in time. If the EC is currently busy with some other work the current strategy of MyThOS is to enqueue the portal call.

`void releasePortal(int id)`

Releases the portal specified by `id`.

D. Memory Management

This section describes the system calls for memory management:

`PMA newStorage(size_t size)`

Creates a new storage object of `size` bytes in terms of a physical memory area (PMA). The returned value of type PMA is a handler for the newly created storage object. This handler can then be used to dynamically map the physical memory into one or more logical address spaces.

`void mmap(addr_t logptr, size_t size, int flags)`

Maps the requested logical address to an allocated anonymous physical memory. This method allocates an anonymous PMA of `size` bytes, which is then automatically mapped with permissions defined by `flags` at the logical address given by `logptr`.

`void mmap(PMA so, size_t offset, addr_t logptr, size_t size, in flags)`

Maps the requested logical address to the referenced part of the storage object `so`. This method behaves like the `mmap` method based on anonymous physical storage, except that it does not create a PMA itself.

`munmap(addr_t logptr)`

Removes an existing mapping and will also release the physical memory if it is not shared (inside the same address space or between spaces). `logptr` must be identical to the one used in `mmap`

`mremap(addr_t logptr, size_t size, addr_t newptr, int flags)`

Copy the mapping from one logical address to another. The old mapping is not released.

`read(PMA so, addr_t dest, size_t offset, size_t size)`

Read access to a storage object `so`. `size` bytes are read from the relative position `offset` and copied to the memory denoted by the logical address `dest`.

`write(PMA so, addr_t src, size_t offset, size_t size)`

Write access to a storage object `so`.

Basic Memory Allocation Functions

These functions are implemented as library functions in user-space. They use the `mmap` and `unmap` system calls to acquire and release anonymous physical memory. This means that no bookkeeping inside the kernel is required. When the user-space memory management decides to unmap some memory, it can be released immediately at the kernel side too. The current state of implementation does not automatically manage virtually shared memory from the OS side, i.e. without according user-space management.

`addr_t alloc(size_t size, alignment)`

Allocates a piece of memory of *size* bytes. `alignment` allows for aligning the data element of application with memory. The returned value is the logical address of the allocated memory backed by some anonymous physical memory area.

`free(addr_t ptr)`

Releases a previously allocated memory region. The specified address at `ptr` is a logical address. Implicitly, the according (anonymous) physical memory is released as well.

E. Asynchronous System Calls

By default, all functions so far were realized as synchronous system calls. In order to support asynchronous system calls, the following design choices must be considered:

- Which instance will receive the completion event of an asynchronous system call? The caller may pass a destination event queue with the system call invocation.
- How does the receiver match the completion event to the original system call? The asynchronous call can return an identification number and the completion event contains the same identification. But this forces the application to map dynamically from these numbers to the actual event handler.
- Does the completion handler interrupt the original caller or is it enqueued as a message for later handling?
- What are the contents of the event message? This depends on the information that has to be returned by the system call.

That discussion motivates the following further system calls, which will be implemented in a second iteration:

`waitForEvent()`

Returns the user-handle of the next queued event, respectively suspends the Execution Context until an event is available.

`pollForEvent()`

Returns the user-handle of the next pending event, respectively returns null if the event queue is empty.

`callECasync(ec, funptr, arg, result-handle)`

Calls an Execution Context asynchronously, see `callEC()`. A pointer to a completion handler can be provided.

1. Waiting/Sleeping and Wake-Up

Another aspect of the thread management, which is closely related to asynchronicity and event handling, is the capability for waiting/sleeping and wake-on-call. Without operating system support, the applications can merely implement busy waiting by iterative polling. However, the kernel can provide suspend and wakeup mechanisms that utilize the processor's energy saving modes and lets the kernel perform its idle-time tasks. Again, the event channels from the asynchronous system calls can be reused for the waiting mechanism – these system calls are in principle internal to the Operating System and its thread management, but can – due to the architectural setup of MyThOS – also be employed by kernel space developers.

`void wakeup(int ec)`

Send a wakeup signal to the Execution Context identified by `ec` in case it is currently suspended. Multiple wakeup signals can be combined into a single event.

`void setWakeupHandler(addr_t userHandle)`

This handler is invoked upon reception of a wakeup signal.

III. Application Management and Loading

In a typical scenario an application is developed and compiled against MyThOS's interfaces and API for a target platform, which results in a statically linked binary ELF-file. When an application is started, its binary ELF-contents are copied from a persistent storage to the address space of some process initialized by the respective operating system. By default, an ELF-file contains an entry point (e.g. a *main* function), which is then queried and invoked by the main OS-thread. MyThOS follows a similar procedure, with the difference that the app-binary is loaded remotely from a host system onto the target platform, i.e. currently the XeonPhi extension card. The current implementation also does not allow for dynamically loaded shared libraries.

A loader application residing on the host reads the contents of an application's binary ELF-image, allocates physical storage objects and copies the binary data into these objects. It then initializes a Protection Domain and a logical address space for the application, creates a new Execution Context and starts its execution at the application's entry point.

Remote method calls from the host to the kernel are handled similarly to local system calls and can access the same interface, i.e. the system call interface and service implementations are reused where possible. Consequently, the application's entry point is also invoked remotely by means of the `callEC`() system call.

Two strategies are possible to initialize the ECs. The first option consists in the application loader initializing only one EC and PD for the application's entry point and all other necessary ECs, PDs, portals etc. are initialized and executed by the application logic itself. Alternatively, the loader preinitializes all available HWTs with ECs to make them readily available. In this case several application threads can be initialized and started in parallel. For the current implementation, the application loader starts only one EC with the applications entry point which then leaves the creation and execution of further application threads up to the application.

IV. Further API Discussion

This section summarises decisions that were made during recent investigations of the targeted application scenarios.

A. Meeting Application Requirements

1. Simple HPC applications

To simplify resource management and avoid costly bookkeeping, we assume that in HPC cases a single application will run on the OS environment, i.e. that the reserved system space will be allocated completely for the purposes of that single application and the according OS configuration. In such cases, the OS configuration can be statically loaded at boot-up time.

The system starts with a single pre-configured Protection Domain and one Execution Context per hardware thread. The user-level handlers to these ECs are fixed by convention in order to simplify the application's initialization. Once the application is loaded, the "first" Execution Context will be activated to execute the application's `main()` function. Communication memory is mapped and initialized before activating the other Execution Contexts via `callECasync(ec, funptr, arg, result-handle)`. The argument `arg` is passed to the function `funptr` and can be used to transfer suitable thread-specific data.

The application's communication is currently implemented using shared memory (single address space) at user-level. In addition the `wakeup(ec)` and `setWakeupHandler(user-handle)` system calls are provided to implement waiting without polling.

The high-level user-space memory management provides a multi-threaded allocator implementation with `ptr = alloc(size, alignment)` and `free(ptr)` functions. The C++ new and `delete` operators are implemented accordingly. The allocator uses anonymous physical memory that is acquired via `mmap(log, size)` system calls.

To support the HLRS and USIEGEN application cases, explicit support for fast resizing of large arrays is required. The typical memory allocators tend to allocate and free large data structures by using `mmap` and `munmap` directly. This leads to high overheads, especially when resizing arrays. We will provide a `ptr = allocLarge(size, maxsize, alignment)` function together with `freeLarge(ptr)`. This will place the array into a large enough range of the logical address space and map physical memory based on the current size. The implementation should reduce/defer the `munmap` system calls of unused memory.

Some of the target applications need barriers or similar synchronization methods in addition to an easy mechanism to assign tasks to threads. BTU will provide a user-space variant of its tasklet framework. `runAt(queue, taskptr)` schedules the task on a specified queue. Initially, we will have one queue per thread. Later implementations may be able to add some load balancing via shared queues. Part of this framework will be barrier primitives.

Many system calls could be applied to either the own Execution Context or a specific other Execution Context. The same applies to referencing Protection Domains. In order to reduce the amount of system call variants, we only provide the variant with explicit addressing. Special values defined through convention are used to refer to the own Execution Context and Protection Domain. This is similar to the Unix standard file descriptors 0, 1, 2 for `stdin`, `stdout`, and `stderr`.

2. MediaCloud and similar Cloud Environments

The cloud applications are executed similar to virtual machines in separate Execution Contexts inside special Protection Domains. All of their system calls and traps are forwarded as asynchronous events to the VMM. Hence, their system call interface is defined by the VMM implementation. This is equivalent to hypervisor-calls in para-virtualized systems.

The cloud runtime management should be implemented in user-space as a Virtual Machine Monitor (VMM). Its Protection Domain provides a richer set of system calls and the kernel may trust the VMM's resource management. Hence, no reference counting and similar is needed inside the kernel.

B. Thread Manager

To allow for more dynamic thread handling, it is necessary to keep track of busy and idle hardware threads that can serve as resources for new threads. The thread manager will allow the application to request and reserve free hardware threads for working threads, respectively queue a query to reserve a HWT once available. The thread manager shall be implemented as a user-space library which internally invokes appropriate system calls to initialize, call and release ECs. At the moment, UULM provided a detailed cost model and a simulator to analyze the runtime performance of different configurations of the thread manager.

The main method of the hardware thread manager is:

```
void request_hwts(WorkerRequest* request)
```

This method requests N workers and retrieves a list of Execution Context IDs with max. N entries. Then the application can send its tasks there by invoking `callEC()`. After completion, the threads could be automatically or explicitly reported back to the manager.

The `WorkerRequest` is defined as:

```
WorkerRequest {
size_t origin;                 // id of the requesting EC
size_t requested_amount;       // the amount of requested ECs
size_t result[];               // ids of the actually allocated ECs
size_t num_results;            // the number of actually returned ECs }
```

The contents of the given `WorkerRequest` structure are filled partly by the caller and by the thread manager library. If `requested_amount` cannot be fulfilled, i.e. there are not enough free hardware threads available, `num_results` gives the number of actually allocated ECs by the `request_hwts()` call.

`request_hwts()` is the basic function providing a primary interface to the functionality of the hardware thread manager. When using only this function an application is responsible itself for distributing the work over the provided ECs. However, reporting back to the manager could be implemented by an explicit call to a function `release_hwts(size_t ids[])` and as part of the `exit()` system call.

To queue future tasks, rather than reserving HWTs right away, to allow for resource scheduling, we can realise:

```
void request_task_exec(TaskRequest* request)
```

This method sends a task description to the manager. The manager forwards the task to a free thread and also registers when the task is finfished executing.

The `TaskRequest` structure is defined as follow:

```
TaskRequest {
  size_t origin;        // id of the requesting EC
  addr_t funcptr;       // pointer to the function impl
  addr_t args;          // pointer to an args structure parsed by the func
  addr_t return;        // address of a return struct parsed by the caller
}
```

The method `request_task_exec()` serves as a wrapper to `callEC()` which first allocates an appropriate EC for the task execution. Alternatively, a Tasklet-based implementation can be provided which has the benefit that initialization of ECs and reporting of released resources to the manager can be automatically provided instead of being explicitly managed by the application.

C. Event Handling

Each Execution Context creates its own Event Channel to handle messages and events. Registering event handlers is closely linked to creating portals. Portals activate an inactive Execution Context whereas events are returned from `waitForEvent()` system calls from within an active Execution Context. This difference is resolved by registering a default handler function `setEventHandlerFunction(ec, funptr, arg)`. Then, pending events at an inactive Execution Context result in activating the Execution Context and executing `funptr(arg, eventhandle)`. The old `callEC(ec, funptr, arg)` can be replaced by sending a signal that enqueues a respective event at the destination and activates the destination Execution Context if necessary.

V. References

[1] R. Rotta, V. Nikolov und L. Schubert, Projekt-Deliverable D2.1 "Overview of the MyThOS System Architecture", 2015.