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

MyThOS D2.3 Architekturplan

Stefan Bonfert, Vladimir Nikolov, Robert Kuban, Randolf Rotta

28. März 2017

Zusammenfassung

Dieses Dokument beschreibt notwendige sowie potentiell interessante Richtungen zur Weiterentwicklung der MyThOS Softwarearchitektur im Sinne einer langfristigen Weiterentwicklung und Nutzung. Abschnitt 1 diskuttiert mögliche Verbesserungen der Performance. Der nachfolgende Abschnitt 2 stellt Verbesserungsmöglichkeiten für die Benutzbarkeit vor. Der letzte Abschnitt 3 beschreibt mögliche Vereinfachungen der Architektur.

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 Potential Performance Improvements

1.1 Page Maps: ASID versus Embedded Capabilities

When mapping a frame or a page map into a page map, some bookkeeping is necessary in order to be able to clean up the mapping when the mapped frame or mapped page map gets deleted. This is also necessary when replacing a previous mapping. Shared memory between address spaces is usually implemented by mapping the same frame to multiple pages. Hence, a one-to-many relation with quick queries in both directions has to be maintained.

MyThOS page maps combine the processor-specific page table with an embedded capability map. Each page table entry has a respective capability entry. When mapping a frame, a reference capability to the mapped frame is stored in this entry. This capability is a child of the original frame in the resource inheritance tree. When replacing the mapping, the reference capability is removed from the tree. When

deleting the frame, all children in the tree are revoked, which informs the page map to undo the mapping. The same applies for mapping lower-level page maps.

This approach enables shared frames as well as shared page maps for partially shared address spaces. However, each page map takes 3x the size of the actual page table, which is a waste of space for sparsely filled address spaces.

The seL4 kernel uses a more compact strategy: When mapping a frame, the used capability is replaced with a mapped-frame capability directly in its current capability entry. The new capability contains an address space identifier (ASID) and the logical address of the page. The root supervisor manages the system-wide ASID pool, which maps from the identifiers to the root page map of the address space. When deleting the frame, this structure allows to clean up the affected address spaces. Unfortunately, cleaning up this capability entry when removing page maps or deleting whole address spaces is not possible. The mapped-frame capabilities are not deleted, which is no problem as long as the ASID is not reused.

Advantages are the low overhead (no insert/removal in the resource tree needed), the easy integration of process context identifiers (PCIDs) for the TLB, and easier global TLB invalidation (ASID can track the hardware threads). On the downside, partially shared address spaces by sharing page tables is not possible and larger capability entries are required for 64bit address spaces.

1.2 Low-Overhead TLB invalidation

On multi-threaded architectures, the kernel has to protect itself against exploits of the Translation Lookaside Buffer (TLB) caches. For example, a frame of physical memory can be used as level-2 page map. On processors with TLB cache, the address of this page map would be stored in the TLB cache in order to speed up future table walks in the same 2MiB range. Now, a malicious application could delete the page map and map the same frame as normal memory into its address space. Because the frame's address is still in the TLB cache it will be used as page map for table walks. But now, the application can write arbitrary address translation into the fake page map and, hence, access any memory. In consequence, either the kernel or a trusted supervisor has to implement the necessary global TLB invalidation before reusing the memory of page maps.

As a minimal precaution, the kernel broadcasts an address space reset whenever a level-4 map is deleted. This changes the address space (CR3 register) back to

the kernel's internal address space on all the hardware threads that were using the deleted page map. The broadcast is implemented similar to the deletion broadcast for kernel objects. It interrupts all places that currently use the affected address space by sending them a Tasklet. In order to skip unaffected places, each place stores the currently used CR3 value in a variable that is readable from the other threads.

The overhead of this approach comes from reading 240 possibly remote cache lines for the current CR3 value plus one asynchronous request for each affected hardware thread. Because of the ring-structured broadcast, the effort is distributed over the affected threads. This ensures good throughput in widely-used address spaces. Instead of storing the currently used CR3 value in each place, each level-4 map could use a bitmap to mark the hardware threads that currently use the address space. Reading these 30 bytes is much faster, but adds larger overhead to loading and unloading of an address space. The best approach likely depends on the application scenario.

Many multi-threaded operating system kernels use interrupt broadcasts in order to implement the TLB invalidation. This makes the latency of individual invalidations more predictable but requires interrupts from ground up. In contrast, MyThOS streams a large number of invalidation tasks through the system and uses interrupts just once for the first preemption of affected hardware threads. While increasing the latency of invalidation broadcasts in widely-used address spaces, this approach reduces the overhead per invalidation and increases the throughput over a large number of simultaneous invalidations.

An obvious step would be to disallow shared page maps, that is, each page map is mapped at most once. This allows to store the pointer to the higher level page map inside each page map object and quickly walk up to the address space through this chain. Then, all kinds of TLB invalidation tasks can be handed to the single affected address space and from there broadcasted to all affected hardware threads.

However, this restriction can be circumvented: Each page map stores its original capability in an own capability entry and all mapped page map capabilities are a child of this entry. Hence, the higher-level page maps can be enumerated by inspection of the resource inheritance tree. This can be used for all page maps that are mapped in more than one maps.

1.3 Interactions between Portals and Execution Contexts

From the application's perspective, communication with other threads and with the kernel objects is achieved through capability invocations through portals. Each portal combines the necessary resources to issue an asynchronous request message and wait for the response message. In order to be able to receive messages from other threads and exception messages from the kernel, portals can also be used to receive a request and answer with a response.

The most important ingredient for the capability invocations is the invocation buffer that holds the request and response message data. It is the only regularly shared data structure between user-space and kernel. In order to avoid accessing user-space memory, the invocation buffers reside inside frames of physical memory such that the kernel can access the contents directly in its own address space. Each invocation buffer can either hold the request or the response message from a single capability invocation and cannot be used for multiple invocations at the same time. Hence, the portal kernel object manages the state of a single invocation buffer.

The current portal design in MyThOS combines the functionality for sending and for receiving requests in one kernel objects. However, portals would be used just for one of these two purposes and many receiver portals could easily share a single invocation buffer. In order to achieve a high throughput, multiple portals are needed and a mechanism to route incoming requests to free portals. Thus, it is a good idea to decrease the management overhead.

A significant architectural improvement would be the differentiation between *Portals* for outgoing requests and *Endpoints* for incoming requests. Then, a large number of endpoints could share a smaller set of invocation buffers. The implementation of portals would be simpler, too. The current MyThOS designs aims at multiple portals per thread. In contrast, Barrelfish uses multiple endpoints with separate buffers whereas, in seL4, multiple endpoints share the thread's single buffer.

In order to achieve higher throughput, for example for central supervisor threads, incoming requests should be automatically routed to a free buffer. In order to facilitate parallel processing, multiple threads should be able to receive messages via the same set of endpoints (and buffers). That is, the requests could be distributed over a group of (ideally nearby) supervisor threads without the need to keep a dedicated thread or portal available for every client or hardware thread.

On the sending side, routing incoming system calls to specific or unused hardware threads would be a useful architectural extension. This is useful, for example, to

offload long-running actions such as the capability revocation. Also, the initialisation of application threads could be easily offloaded to the target hardware thread. In current designs (Barrelfish, MyThOS), the supervisor or application would have to keep respective supervisor threads available and notify them about their asynchronous task. This separation of system activities from application threads is similar to other *core separation* approaches in the Linux world [RLM13, SFL+11].

1.4 Lightweight Execution Contexts

Execution Contexts represent user-level threads. They combine an address space, a capability space and communication portals/endpoints. In addition, they are bound to a scheduling context, which represents the hardware thread that will execute the user-level thread, and an exception handler endpoint, which is supplied by the supervisor. The communication portals and endpoints again are bound to the execution context for message notifications and frames for the invocation buffers.

Managing all these references and bindings to other kernel objects adds overhead to the startup and teardown of execution contexts. Many application scenarios feature a large number of relatively short-lived, very simple execution contexts. For example, they share address spaces, capability spaces and a fixed set of communication channels. Hence, this exact combination could be combined into a specialised execution context variant. One example are the *codelets* used by OctoPOS to simplify and speed up X10 activities [MBZ+15]. The *Basslets* in Barrelfish follow similar ideas [GZAR16].

1.5 Latency-Aware Placement of Synchronisation Variables

Many-core architectures such as the Intel XeonPhi Knights Corner employ address interleaving over cache coherence directories in addition to interleaving over memory controllers. This helps to increase the throughput of the cache coherence protocol but introduces quite large latency variations for inter-core synchronisation.

The mapping of cache lines to coherence directories and the distance between cores and directories is quite well known. Thereby, synchronisation variables such as Tasklets, queues, locks, and reference counters could be placed into nearby directories by choosing respective cache lines.

This would be easy for statically allocated kernel objects because just a small known number of cache lines is needed. For the dynamic allocation of kernel objects,

a pool of appropriately placed cache lines is needed inside each Untyped Memory. Very often, this pool would be sparsely used and, therefore, a recursive scheme that uses the parent's pool first could be integrated. One challenge is the design of an appropriate allocation/deallocation interface for this pool.

1.6 Coarse-Grained Scheduling Contexts

Many-core processors provide a large number of cores and an even larger number of hardware threads. At the same time it is necessary to split the work into smaller tasks and distribute them over multiple threads in order attain the processor's throughput. Distributing the application's and system's tasks leads to an increased overhead. One aspect is the effort to choose a suitable hardware thread for the execution of the task.

Currently, MyThOS leaves the scheduling of application threads (Execution Contexts) to hardware threads (Scheduling Contexts) to the application or supervisor. Each application thread has to be bound to an hardware thread and a simple cooperative round-robin scheduler is used on each.

Managing the placement on the level of individual hardware threads is a tad too fine-grained because the differences between the threads of a core are insignificant. Instead, whole cores should be assignable as scheduling context. Then, the core's scheduler (somehow) distributes the ready application threads on the available hardware threads. The design can exploit the very fast synchronisation inside each core because of the shared caches.

2 Usability Improvements

2.1 User-Space Runtime Environment

The present implementation misses a usable runtime environment on the user-space side. Applications and supervisors have to use individual system calls (capability invocations to kernel objects) in order to set up capability and address spaces and create worker threads. A lot of the intermediate steps can be hidden in a system library similar to the seL4re.

2.2 Futex Support or Notifications

User-level critical sections and object monitors require fast mutex and semaphore implementations. These can either employ busy waiting, which does not hand over the compute time to other threads, or use a wait-signalling mechanism provided by the operating system, which adds the system call overhead. In practise, many critical sections are entered and left without competing threads. In these cases, the help of the kernel would not be necessary.

The *Futex* mechanism in Linux enables the implementation of fast user-level mutexes and semaphores that only use system calls when necessary, that is to actually put a thread to sleep or to wake up a sleeping thread [Dre09, FRK02]. A similar mechanism would be very helpful for parallel MyThOS applications or their runtime environments. The seL4 kernel seems to circumvent the complexity, by providing a low-overhead notification mechanism based on bitmask wait-sets. The futex implementation of the magenta-kernel might be interesting because it is restricted to synchronisation within a single address space[1].

2.3 (Clustered) Multikernel

The idea of *multi-kernels* like Barrelfish is, that the kernel on each hardware thread operates in isolation. The benefits are that no synchronisation is needed within each kernel and that heterogeneous kernels can be combined. Sharing is implemented and coordinated by a user-level supervisor, which uses message passing between the threads.

The seL4 project proposed a *clustered multi-kernel* approach that groups a small number of nearby hardware threads together. Inside each cluster, a big kernel lock is used and coordination between clusters works like in multi-kernels. The benefit is, that the locally shared caches are more effective while the number of messages for global coordination is reduced.

Booting MyThOS as a clustered multi-kernel is relatively easy. The basic idea is to create disjoint untyped memory objects and separate supervisor processes per cluster. This task can be left to the initial supervisor [ZGKR14]. It has to create initial exclusive kernel objects for the other clusters.

[1] https://github.com/fuchsia-mirror/magenta/blob/master/docs/futex.md

The actual challenge of multi-kernels is the distributed rights management. In order to support multi-threaded applications, access rights need to be transferred and shared between clusters. To a large degree, replication and update broadcasts solve this requirement. Because multiple threads can try to manipulate the same access right simultaneously, a transaction or consensus model is needed. Barrelfish allows the sharing of some kernel objects without replication, for example to share whole address spaces efficiently. This is achieved by replicating just the access right capabilities. Each capability does not just point to a kernel object, they also contain a 32bit context identifier as home of the kernel object and a few additional flags for the replication state. When access rights are transferred between clusters, the receiver has to find the right position in its resource inheritance tree. This takes linear time because of the prefix-order encoding of the tree.

In retrospective, a lot of overhead is just shifted from the kernel into the supervisor without simplifying the management. As long as low-overhead is more important than support for complex heterogeneous hardware nodes, the multi-kernel approach might not be worth the effort.

2.4 Easier Memory Sharing between Applications

Shared memory allows applications to exchange large volumes of data without the overhead of kernel-based communication channels. The present architecture supports two ways to establish shared memory. Either a supervisor maps a physical memory frame into both address spaces, or one application transfers a frame capability to the other, which maps the frame to an appropriate logical page. In both cases additional coordination is necessary in order to negotiate which memory is mapped to which address.

The nested address space model of Fiasco, Nova and similar L4-style kernels provides a much more straightforward interface. One application provides the shared memory as a source window in its logical address space and the other application provides a destination window in its own logical address space. The kernel than maps the frames from the source into the destination window. The benefit is, that large windows can be mapped with a single system call and the windows do not have to be aligned on large pages. In addition, both applications do not need to be bothered with the management of individual frames.

3 Architecture Simplification

3.1 Asynchronous Revokation

The deletion of kernel objects is implemented in multiple phases in order to ensure the correct cleanup of references and prevent any concurrent access to the object during its final deletion. The first phase is the revocation of all access rights related to the object. This is achieved by removing children capability entries from the resource inheritance tree.

In the present design, the affected kernel object (usually a capability holder) is informed about the revocation through a call to the synchronous `deleteCap()` method. In consequence, this synchronous call is executed concurrently to the regular asynchronous calls. That is, although normal operations are protected by the object's monitor, references to other kernel objects can disappear at any time. Thus, all code that uses references to other kernel objects has to take precautions when reading these references. Most of the necessary precautions are implemented in the `CapRef` helper.

Instead of the synchronous `deleteCap()` method, this call could be made asynchronous. This likely slows down the revocation but simplifies to kernel objects and enables asynchronous cleanup activities inside of `deleteCap()`. Probably it is necessary to redesign the interaction of `deleteCap()` with the object's monitor.

3.2 Unordered Deletion Lists

The final phase of the deletion mechanism is based on a list of objects that are ready for final deletion. In the present design, this deletion list is processed in FIFO order to delete child objects before the objects that provide their memory resources. The reason is, that the task sent to the object's deletion monitor is delayed infinitely until the reference counter reached zero. This design does not easily allow parallel revocation and the offloading of object deletion because the required deletion order is difficult to maintain.

Two observations can be exploited to simplify the final deletion: First, objects that were allocated from different Untyped Memory objects can be deleted independently. Thus, a separate deletion list per untyped memory can be used. Each object has to know its origin memory anyway. Waiting for the actual deletion can be deferred until

the memory is actually reused, which leaves more offload-able tasks for background processing.

Second, the asynchronous deletion task does not need to be sent into the blue. Instead, the object's reference counter can be queried in order to ensure immediate deletion. When the next object on the deletion list still has pending references, it can be skipped and revisited later. Likewise, any object on a deletion list knows about it state already. When the reference counter reaches zero, the deletion list, that is the untyped memory object, can be informed in order to proceed with the final deletion.

3.3 Big Kernel Lock

Micro-kernels like seL4 and Barrelfish use a big kernel lock to protect the kernel's data structures against inconsistencies and data races from concurrent write access. The reasoning for single-threaded and small multi-threaded processors is, that most kernel activities quickly finish anyway. Thus, finer grained locking would just add overhead without improving the responsiveness.

Obviously the big kernel lock does not scale to many-core processors with a large number of threads. However, getting rid of all the potential races and concurrency inside the kernel allows for tremendous simplifications. A kernel variant based on a big kernel lock would be very interesting for the comparison of scalability aspects and for more efficient support of small processors.

Another interesting variation would be to just protect all resource tree operations, like for example capability inheritance and revocation, with a big kernel lock. The usual operations of kernel objects are protected by their object monitors anyway. Instead of a lock based on busy waiting, the resource tree operations could be offloaded as asynchronous task to a dedicated hardware thread.

References

[Dre09] Ulrich Drepper. Futexes are tricky. red hat. *Inc., August*, 2009.

[FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, page 479, 2002.

[GZAR16] Jana Giceva, Gerd Zellweger, Gustavo Alonso, and Timothy Roscoe. Customized OS support for data-processing. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 2:1–2:6. ACM, 2016.

[MBZ+15] Manuel Mohr, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel, and Daniel Lohmann. Cutting out the middleman: Os-level support for x10 activities. In *Proceedings of the ACM SIGPLAN Workshop on X10*, X10 2015, pages 13–18, New York, NY, USA, 2015. ACM.

[RLM13] Eli Rosenthal, Edgar A León, and Adam T Moody. Mitigating system noise with simultaneous multi-threading. *Proceedings of SC13, poster session*, 2013.

[SFL+11] S. Seelam, L. Fong, J. Lewars, J. Divirgilio, B. F. Veale, and K. Gildea. Characterization of system services and their performance impact in multi-core nodes. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 104–117, May 2011.

[ZGKR14] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 17–31, Berkeley, CA, USA, 2014. USENIX Association.