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

MyThOS D3.3 Softwareentwicklungsplan

Stefan Bonfert, Vladimir Nikolov, Robert Kuban, Randolf Rotta

28. März 2017

Zusammenfassung

Dieses Dokument umfasst den Strategieplan für die Weiterentwicklung des Prototypen und Anleitungen für Systementwickler. Es fasst somit nützliche und geplante 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.

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

This document recaps a strategy plan for the further development of the MyThOS operating system and its prototype. It summarizes a bunch of useful and nice-to-have improvements, like for example for error and performance analysis, interfaces for the configuration of the interrupt handling and capability transfers between communication portals. It also outlines the development strategy of the partners in terms of future academic activities and projects, that will help to sustainably improve the system.

In the second phase of the project the MyThOS system and its architecture were completely redesigned and re-implemented from scratch. The lessons learned from the previous version helped us to streamline the kernel into a more consistent shape and operation. Logical and functional gaps are not pushed up anymore to the responsibility of the application developers, but were filled and closed with clear definitions, requirements and architectural elements of the system. In the course of that development various nice-to-have functionalities were left aside or postponed, as the focus was initially set on the development of the new version of the base kernel. Such functionalities and features are presented and discussed later in Section 2.

MyThOS is planned to serve as a basis for various further research projects and initiatives. Thereby, aspects like different scheduling policies, task models, resource management strategies (CPU, energy, network), fault tolerance and support for time-critical applications will be investigated. The fields of application and research range from heterogeneous and embedded systems, over large scale infrastructures to time-critical and real-time applications and systems. Concrete developments and projects as well as cooperations with industrial and academic partners will be announced and referenced on MyThOS's website (https://manythreads.github.io/mythos/).

Furthermore, we aim to extend the code-base with further (example) applications and to port the system to different hardware architectures (e.g. ARM, PPC, particular Microcontrollers, etc). Also modules for physical storage, network communication, graphical display and device drivers for further peripherial hardware are still on the roadmap.

Besides this, theoretical and practical aspects of MyThOS are already and will be further integrated in various lectures, seminars, student thesis and labs at the Ulm University and at the BTU Cottbus.

MyThOS was also published on GitHub with the aim to build an open source developer community and to provide a well-known platform for further developments and improvements.

In the following Section 2 we now summarize some of the directions for technical improvements of the kernel.

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.