Thread Migration for Mixed-Criticality Systems

Alexander Zuepke
RheinMain University of Applied Sciences, Wiesbaden, Germany
Email: alexander.zuepke@hs-rm.de

***Abstract*—This work-in-progress paper presents a thread migrating operating system concept for mixed-criticality systems on multi-core platforms. Thread migration provides fast context switching between isolated software components which handle shared resources. Combined with criticality inheritance protocols and a multi-policy scheduler, the described operating system concept aims to meet the level of determinism and analysability which is required for safety-critical applications.**

I. Introduction

With *Cyber Phyiscal Systems* and the *Internet of Things*, mixed-criticality systems have become a reality in the embedded computing world. Combined with the recent availability of multi-processor systems, it imposes a new challenge on operating systems when different functional units are combined in a single computer system. Similarily, regulatory standards like ISO 26262 require *freedom of interference* between these independent functional units [1]. On the other hand, tight integration of today's hardware technology results in problematic sharing of computational resources like caches and memory bandwidth, and functional resources like I/O devices and buses. An operating system for such scenarios should therefore help to make the side effects of resource sharing *predictable* and enforce the required level of *determinism*.

From a real-time perspective, this means that applications of different *criticality levels* (in the sense of importance to a device's overall function and cost of malfunction) need to be scheduled concurrently. It also requires that access to shared resources and any resulting priority inversion problems need to be solved in a bounded worst-case execution time (WCET) to guarantee that deadlines are met. From a safety perspective, a high degree of separation between different application components and shared components is necessary to guarantee the required freedom of interference and fault isolation. State of the art techniques place applications, drivers, and services into separate address spaces, protected by means of the processor's memory management unit (MMU) [2] [3].

However, while decomposition of a system's components into multiple address spaces helps to fulfill the safety requirements, it entails overhead due to the cost of additional context switches. Therefore, operating system support for mixed criticality systems should include:

- isolation of components in separate address spaces,
- fast context switches between isolated components,
- bounded WCET of all internal operations of the kernel,
- solving of priority inversion problems on shared resources,
- concurrent scheduling of threads of different criticality.

This paper briefly presents the design principles of the WINGERT operating system, which addresses the goals discussed above. Its overall architecure is shown in section II, the benefits of thread migration is described in III, scheduling in IV, resource sharing in V, and related work in VI. We conclude and give an overview of future work in section VII.

II. System Architecture

The WINGERT OS is built upon a small kernel running in the CPU's privileged mode and a hierarchically structured set of isolated address spaces (tasks) of different criticality in user mode, which comprise applications or services like shared drivers. Following the design principle of a small trusted computing base, critical application tasks only depend on the required subset of tasks providing shared services for them. Communication between tasks is implemented by remote procedure calls (RPCs), which are described in detail in the following section. Starting from the application tasks as the leaves of the task tree, the hierarchy of depending tasks down to the kernel as the root node never decreases in the criticality level. This implies that applications can trust the tasks down in the chain.

Further, all system resources like memory, I/O and time budgets are statically assigned to the tasks at startup. This *resource partitioning* approach eliminates the later need to transfer system resources or access permissions via task communication at runtime, keeping the RPC implementation in the kernel fast and simple.

Each task manages its capabilities in its own name space. Capabilities address the user and kernel parts of threads, interrupts, child tasks, child address spaces, and communication channel endpoints. Memory is addressed differently by its implicit virtual address in the task's page tables. Additionally, tasks can freely repurpose their assigned amount of page-sized *kernel memory* to page tables for dynamically created memory mappings or in-kernel stacks for threads. This degree of freedom allows for example a para-virtualized Linux task to reconfigure itself for different use cases at runtime, without violating the static partitioning approach.

III. Thread Migration

The main difference of WINGERT compared to other micro kernels like L4 lies in its low-level abstraction model named *body and soul* instead of threads as the basic entities of execution. The *soul* is a scheduling entity with priority, deadline, and a kernel stack. It migrates synchronously between different *bodies*, which comprise of an entry point and dedicated stack in user space. The invocation of a new body resembles an RPC call to the same or a remote address space, while keeping the calling soul to have a unique entity to control the execution flow and to reduce context switching overhead in the kernel.

For asynchronous communication and decoupling of potentially blocking calls, the kernel provides *fork-join* operations: a soul forks and instructs its forked sibling to issue a synchronous RPC. With a technique named *lazy forking*, the kernel follows the forked path using the original soul first and performs the real fork operation when a blocking point is encountered. Assuming there is no blocking point, the forked one returns the result and joins gracefully without any overhead.

IV. Scheduling

The practical challenge of mixed-criticality scheduling lies in reclaiming scheduling reservations of higher critical tasks at run time caused by their overly pessimistic WCET analysis. Inspired by MC^2 [4], the kernel scheduler provides multiple scheduling policies for different levels of criticality. In descending order of criticality, these are:

1) P-FP: partitioned fixed-priority scheduling
2) P-EDF: partitioned earliest deadline first scheduling
3) G-FP: global fixed-priority scheduling
4) G-EDF: global earliest deadline first scheduling
5) BE: best effort scheduling for non-real-time applications
6) IDLE: scheduling of idle threads of the lowest level

All scheduling policies are mapped into the same priority space, but have disjoined priority ranges and different queueing policies (FIFO or deadline ordered). The highest priority level ready queues are kept exclusive per processor to ensure partitioned scheduling, the lower levels share a single set of ready queues. The dispatcher picks the highest eligible thread for scheduling on its CPU. Supporting other scheduling policies, like the ones used by Linux, is not the responsibility of the OS scheduler. On top of this system, a para-virtualized Linux implementation would use its built-in scheduler and dispatch its processes by thread migration.

V. Resource Sharing

WINGERT provides two different mechanisms for synchronization and resource sharing: thread migration across tasks; and mutexes and condition variables shared by threads in the same task. The latter use *Deterministic Futexes* described in [5] as the underlying kernel mechanism. The implementation enters the kernel only on contention and uses atomic operations on variables in user space in the fast path.

As bodies have a single user space stack only and therefore do not support multiple souls inside, migrating souls have to wait when a body is already occupied. With an extension to let souls *wait* outside the body and let them stay there until they are *signalled* again by the body, the body effectively becomes a *Monitor* [6].

On contention, both bodies and futexes need to properly solve priority inversions problems. The standard *priority inheritance protocol* (PIP) [7] solves this issue for the class of highest criticality P-FP scheduling. Additionally, the protocol covers P-EDF by prefering earlier deadlines on priority ties. Finally, with *migratory priority inheritance* [8], the scheduler migrates preempted threads across CPUs and solves priority inversions in global scheduling scenarios. With these extensions for mixed-criticality scheduling, the described protocol effectively becomes a *criticality inheritance protocol*.

VI. Related Work

WINGERT has in common with micro kernels like L4 [9] a similar overall system structure of decomposed software components in isolated address spaces and the use of a synchronous context switch mechanism as a means of communication [2]. However, our approach is more specific to the mixed-criticality use case than the policy-free approach in L4.

Thread migration was previously used in [10], [11], and [12]. Compared to COMPOSITE, which uses thread migration and solves contention on user stacks with PIP and PCP (priority ceiling protocol) [3], the presented approach scales to multi-processor platforms.

VII. Conclusion and Future Work

This paper presented the WINGERT operating system, which aims to exploit thread migration for real-time systems. Using thread migration for strictly hierarchical system designs such as mixed-criticality systems seems to be a good trade-off between software component isolation for safety reasons on the one hand and fast performance on the other hand, while at the same time reducing the number of possibly misbehaving actors and keeping the overall system complexity low.

In future work, we plan to evaluate the system performance and provide an in-depth analysis of the presented criticality inheritance protocol, with a special focus on an implementation with a bounded WCET.

References

[1] ISO 26262, "Road vehicles – Functional safety," 2011.

[2] J. Liedtke, "On μ-Kernel Construction," in *SOSP*, 1995, pp. 237–250.

[3] Q. Wang, J. Song, and G. Parmer, "Execution Stack Management for Hard Real-Time Computation in a Component-Based OS," in *RTSS*, 2011, pp. 78–89.

[4] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "RTOS Support for Multicore Mixed-Criticality Systems," in *RTAS*, 2012, pp. 197–208.

[5] A. Zuepke, "Deterministic Fast User Space Synchronisation," in *OSPERT Workshop*, 2013.

[6] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Commun. ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974.

[7] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[8] B. B. Brandenburg and A. Bastoni, "The case for migratory priority inheritance in linux: Bounded priority inversions on multiprocessors," in *Fourteenth Real-Time Linux Workshop*, 2012.

[9] K. Elphinstone and G. Heiser, "From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?" in *SOSP*, 2013, pp. 133–150.

[10] B. Ford and J. Lepreau, "Evolving Mach 3.0 to A Migrating Thread Model," in *USENIX Winter Conference*, 1994, pp. 97–114.

[11] G. A. Parmer, "Composite: A Component-based Operating System for Predictable and Dependable Computing," Ph.D. dissertation, Boston, MA, USA, 2010.

[12] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "The Pebble Component-based Operating System," in *USENIX ATC*, 1999.