Mixed-Criticality Support in a High-Assurance, General-Purpose Microkernel

Anna Lyons, Gernot Heiser
NICTA and UNSW Australia
{anna.lyons,gernot}@nicta.com.au

***Abstract*—We explore a model for mixed-criticality support in seL4, a high-assurance microkernel designed for real-world use. Specifically we investigate how the seL4 model can be extended without compromising its security properties and its general-purpose nature, including high average-case performance. The proposed model introduces reservations, with admission control performed at user level, similar to how seL4 handles spatial resources.**

I. Introduction

seL4 is a high-performance microkernel of unprecedented assurance, with a machine-checked proof of implementation correctness, as well as proofs of spatial isolation properties (integrity and confidentiality) [1]. This makes it an excellent base for security-critical uses, particularly systems where security-critical components share a processor with less critical code, such as a critical crypto service co-located with an untrusted Linux system running in a virtual machine.

Our aim is to evolve seL4 into a platform for supporting similar setups in the safety-critical domain, without compromising the kernel's security properties nor its excellent performance [2]. A first step was a complete and sound analysis of seL4's worst-case execution-time (WCET) latencies [3]. The obvious next step is support for mixed-criticality scheduling, i.e. the ability to guarantee the timely execution of highly critical tasks in the presence of less critical tasks with potentially tighter timeliness requirements.

The cost of formal verification (although less than that of traditional high assurance [1]) provides a strong disincentive to maintaining multiple variants of a verified system. Furthermore, security is increasingly becoming a safety issue, as demonstrated by the recent spate of car-hacking [4]. We are therefore interested in widening the application domain of seL4, without losing any of its existing benefits.

Specifically, we are looking for a design which satisfies the following requirements:

- It preserves seL4's strong spatial isolation properties, its support for transparently interposing security monitors between communicating components, as well as its best/average case performance.
- Criticality (i.e. ability to meet deadlines) must be orthogonal to urgency (proximity of deadline), in that an overcommitted system must meet deadlines of all criticality levels that would be met if none of the lower-critical tasks had been admitted. Admission control (i.e. schedulability analysis) must be possible without making any assumptions on less critical tasks.
- Tasks of different urgency and criticality must be able to share resources.
- There must be no significant (algorithmic or overhead-related) capacity loss, and any slack time must be available for best-effort tasks.
- To support certification re-use, it must be possible to admit black-box components solely based on their criticality, processor utilisation and minimal period.
- Any policy, including admission control, must be implemented at user level, the kernel is only to provide general mechanisms.
- The model must not impose restrictions on the programming model beyond what is required to satisfy all other requirements. In particular, we do not require that all shared resources are multi-threaded.

Clearly, this means that we need to provide asymmetric temporal isolation (lower criticality tasks cannot interfere with the timeliness of higher criticality tasks) enforced by runtime monitoring, with the ability to switch to a higher criticality mode of execution when the system is unable to meet all deadlines. The system should degrade gracefully in such a case, meaning that task of a certain criticality should only miss deadlines if higher-criticality tasks leave insufficient slack (i.e. we should maximize the number of high levels of criticality that meet their deadlines).

We rule out user-level, hierarchical scheduling as it introduces concurrency between user-level and the kernel. The verification of seL4 relies on the kernel remaining single-threaded to avoid the state-space explosion inherent in proofs about concurrent programs. Additionally, the current C semantics of the proof framework do not support concurrent programs [5]. However, the black-box and interposition requirements imply a requirement for delegation of CPU allocation, which we provide by leveraging seL4's capability system.

The requirements for retaining seL4's security and performance properties imply that we retain the basics of the seL4 model, which we summarize in Section II. We explore a model that satisfies the above requirements in Section III and discuss the approach to resource sharing in Section IV. We presently restrict our thinking to uniprocessor systems.

II. seL4 Basics

seL4 is a capability-based microkernel system with strong security and spatial isolation guarantees. Like other L4 microkernels, seL4 adheres to the *minimality principle* which allows features in the kernel only if the required functionality could not be achieved by a user-level implementation [6].

Specifically, device drivers are not part of the kernel, but run as unprivileged processes, the only exceptions being a timer driver and a driver for the interrupt controller.

The most significant difference between seL4 and other microkernels is its (spatial) resource-management model: The kernel, after booting up, never allocates any memory. Instead, all memory not needed to boot the kernel is handed to a user-level manager. When performing an operation that requires allocation of kernel data structures, such as creating threads or address spaces, the invoking user-level process must provide the kernel with memory for storing those data structures. Hence, all memory is completely managed by user-level code, subject to policies implemented at user level.

The kernel only supports a small number of abstractions: *threads* as the execution abstraction, *address spaces* for memory management and spatial protection, and *endpoints* for communication. *Synchronous endpoints* are rendezvous points for message-passing communication (synchronous IPC). *Asynchronous endpoints* support non-blocking signalling, they are essentially binary semaphores. Threads are tied to address spaces and communicate via endpoints. In earlier versions of L4, IPC messages were addressed directly to threads rather than endpoints. This model was abandoned as it introduces covert channels [7].

All access rights in seL4 are represented by capabilities [8], unforgeable access tokens protected by the kernel. Capabilities can be transmitted via IPC (subject to appropriate access rights) and support privilege delegation. For example, the initial resource manager can hand control over a partition of memory to another process, which then can manage that memory autonomously.

The delegatable user-level control over memory is the key to the strong, provable spatial isolation properties of seL4 [9], [10]. It is also useful for temporal isolation, as it can be used to partition caches [11], which can reduce WCET bounds [12].

IPC is also authorised by capabilities: a thread needs an endpoint capability in order to send or receive messages. Besides the simple `send()` and `wait()` (i.e. receive) operations, the kernel offers two combined send-receive operations, `call()` and `reply_wait()`.

`call()` is an RPC-like operation typically used by clients to invoke a server; it consists of a send to a specified endpoint, followed by waiting on a reply. It is semantically different from `send()` immediately followed by `wait()` in two respects: (i) the transition between sending and waiting to receive is atomic, and thus non-preemptible,[1] and (ii) instead of specifying an endpoint from which to receive the reply message, the kernel during the send phase creates a temporary one-shot endpoint for the reply, and transfers the corresponding *reply cap* to the server. Similarly, `reply_wait()` combines the reply to the caller (through the reply cap) and waiting for the next request in one atomic system call.

Management of the resource *time* is less developed, and time is in fact considered the last concept for which no satisfactory abstraction has been found to date [2]. Consequently, scheduling is deliberately left underspecified in seL4 [14]; the present implementation uses a fixed-priority round-robin scheduler.[2]

III. Proposed Scheduling Model

In order to support temporal isolation we add *reservations* to seL4. This approach had been introduced by RT-Mach [15], and later deployed in resource kernels [16]. Traditional reservations contain task scheduling parameters enforced by the kernel, specifically a limit on CPU time consumed over some interval. Additionally, the kernel performs an admission test to make sure the set of reservations is schedulable.

Mixed-criticality systems leverage the slack left from conservative WCET estimates of higher criticality tasks to run lower criticality tasks, thus increasing the overall utilisation of a system. This is achieved by allocating the excess budget of high-criticality tasks (from now on called "high tasks" for simplicity) to tasks of lower criticality. Asymmetric protection ensures high tasks meet their deadlines, even if this violates the temporal constraints of low tasks, but not vice versa. Recent models for mixed criticality systems [17] implement this through a *mode change*: if the system is unable to meet its deadlines, it increases the system criticality level, and tasks below that level are no longer guaranteed to meet their deadlines.

Our proposed model differs from traditional reservations in that we only guarantee upper bounds on execution time, and by delegating all admission control to user level.

A. Reservation capabilities

An seL4 reservation is a kernel object, and thus is represented by a *reservation capability* ("resCap"). Like any capabilities, resCaps can be easily delegated to subsystems through existing capability transfer mechanisms. A thread can only run if it is associated with a resCap, and a resCap can only be associated with a single thread at a time. Threads can share resCaps by cooperatively scheduling through IPC, as will be explained in Section IV.

Reservations act as sporadic servers [18], characterized by a budget, period and relative deadline, which encapsulates the processor share and replenishment frequency the reservation entitles. The kernel enforces budgets through a timer interrupt.

B. Scheduling

For now we retain seL4's fixed-priority scheduler, although, in order to experiment with EDF scheduling, we treat the median priority (126) special: threads at this priority use the deadline parameter for EDF scheduling (but only if no threads of a higher fixed priority are runnable), similar to Ada [19]. Reservations of EDF threads are treated as hard CBS [20].

When the current reservation's budget is depleted, it is placed into a waiting queue ordered by replenishment time, unless the reservation is a *full* reservation (100%, i.e. budget

[1] The kernel executes with interrupts disabled and limits interrupt latencies through strategically placed preemption points; none are in performance-critical IPC code [13].

[2] For security-oriented temporal isolation the scheduler is configurable with multiple non-preemtible scheduling domains, which are scheduled for a fixed time slice. These domains are unsuitable for real-time use due to the large algorithmic capacity loss and the high interrupt latencies.

= period), in which case the thread is appended to the end of its priority's scheduling queue. Full reservations preserve L4's traditional round-robin scheduling.

Obviously a thread with a full reservation should have a low priority, unless it is *trusted* not to overrun its budget, in which case a full reservation with a long period can be used to avoid the overhead of run-time monitoring.

Our model of reservations enforcing upper bounds of CPU usage encourages overcommitting, round-robin threads being an example. Schedulability analysis is a user-level concern. In fact, the kernel lacks the information to determine schedulability, as this would require locating and examining all resCaps that are associated with some thread.

C. Admission testing

Admission testing implements a particular policy, eg. on-line vs off-line, dynamic vs static, the degree of overloading allowed, and whom to trust not to overrun their reservations. According to the minimality principle it should therefore be performed at user level. Admission tests can also be very complex and hard to formally verify.

The basic safety mechanism is control over creation of reservations. We restrict this to the holder of the special `sched_control` capability, who is in complete control over time allocation in the system. The holder is trusted to perform an admission test upon a request for a reservation. seL4's startup protocol provides the `sched_control` capability to the initial process, which may then transfer it to a dedicated time manager. It may also split the total available bandwidth and delegate partitions to individual managers, which achieves most of the benefits of hierarchical scheduling without its cost.

This approach is analogous to seL4's mechanism for controlling memory, where the initial process obtains rights to all free memory. It is also similar to how seL4 manages access to devices: the holder of a special `IRQ_control` capability grants device drivers the rights to specific interrupts. On seL4, all resource management is performed by trusted user-level servers, and time is no longer an exception.

Schedulability depends on priorities as well as reservations. The system provides a safety mechanism by associating each thread with a *maximum controlled priority* (MCP). While a holder of a thread capability can control that thread's priority, the kernel will not allow it to raise any thread's priority (including its own) to a value higher than its own MCP.[3]

D. Task Model

We adopt the sporadic task model, where tasks are an infinite series of jobs. A task is represented by an seL4 thread, and a job is the release of a thread by the kernel.

A thread has an optional asynchronous *trigger endpoint*; by waiting on that endpoint, the thread indicates *job completion*. A thread that does not complete is rate-limited by its reservation.

Job release happens by signalling that endpoint, thus resuming the thread's execution. The kernel signals the endpoint when the thread's budget is recharged, thus supporting time-triggered tasks. Alternatively the endpoint can be signalled by some event, e.g. an interrupt or another thread, resulting in an event-triggered task. Such a thread does not actually become runnable until its recharge time has passed (until that occurs, it has no budget to run).

The kernel has no concept of threads being real-time or not: whether a thread is able to meet its deadlines solely depends on whether the thread's budget is sufficient for its WCET, and whether the system is over-committed at the thread's priority.

E. Criticality

We add a criticality field to seL4 threads, and track a global kernel criticality level. The criticality level is changed at user-level by invoking the `sched_control` capability. Threads whose criticality is less than the global kernel criticality will not be scheduled: instead, they are post-poned by the period of their reservation, at which point the criticality level may have changed. This approach maintains the preemption level of the lower criticality workload, but allows threads to come back online automatically once the criticality level is restored.

F. Mode changes

To enable the mode change required for mixed-criticality support, we introduce a simple, policy-free mechanism: *temporal exceptions*. This extends the existing seL4 exception handling approach, which associates an exception endpoint with each thread. When a thread triggers an exception, the kernel sends a message to the exception endpoint. A handler thread waiting on that endpoint can then handle the exception. In a practical system, many threads share the same exception endpoint (and thus handler), typically the responsible operating-system personality.

For temporal exceptions we introduce a second, optional, temporal exception endpoint. The kernel sends a message to this endpoint if the thread exceeds its budget or overruns its deadline. If the thread has no temporal exception endpoint, it is silently rate-limited. The handler, assumed to be a highly-privileged thread, can then transition the system into high-criticality mode.

How the handler responds to the exception depends on the policy of the system. Some systems may have infrequent and short mode changes, where all lower criticality threads should be briefly suspended until the system returns to normal. In this case, using the kernels criticality mechanism is suitable: the overruning thread's budget can be increased to parameters for a higher criticality mode, and the kernel criticality level increased. Alternatively, if the system requires that lower criticality threads remain runnable but with weaker or no guarantees, the exception handler can reduce the priorities of lower criticality tasks [21], or give high tasks full reservations and boost their priorities. Under any mode switch policy, the exception handler needs its own (high-priority) reservation, which must be factored into the cost of the mode switch.

The opportunity to return to a lower criticality level can be detected by using a dedicated thread running at a priority below that of all threads at or above the current criticality level, but above the (down-graded) priority of all low threads (should

[3]Note that a thread's actual priority can exceed its MCP, provided it has been set by another thread with a sufficiently high MCP.

they be runnable). When the kernel schedules this thread, it is an indication that there is slack in the system, and the thread can move the system toward normality by restoring scheduling parameters or increasing the kernel criticality level.

IV. Resource Sharing

The frequently made assumption of no sharing across criticality levels is unrealistic [21]. For example, the low-level flight control of a unmanned aircraft (UAV) is highly critical, as it ensures the vehicle remains stable and on track, its failure would lead to loss of the UAV. The UAV's mission control determines, in communication with the ground station or based on analyzing senor input, where the vehicle is to go next. It is less critical, as ground control can re-transmit commands or the analysis can be repeated. But, in order to be effective, mission control must share resources with flight control, e.g. the way points updated by mission control and used by flight control.

By definition, sharing implies that a high task may be blocked while a low task is holding a resource. A shared resource must therefore be considered to have the same criticality as its highest client, including a WCET certified at the level required for that client. We furthermore require a mechanism that allows the high task to progress if the low task runs out of budget while holding the resource.

In seL4 we model shared resources as *resource servers* accessed via synchronous IPC [22]. We distinguish between *active* servers, which have their own reservation, and *passive* servers, which do not. A passive server can only execute by another thread *transferring* its reservation to the server. Such a transfer happens during synchronous IPC: when a client invokes a server (via a `call()` IPC operation), its reservation is transferred to the receiver, and the server returns it when completing (via the `reply_wait` operation), see Figure 1. Such a server is said to execute on a *borrowed* reservation.

This is similar to time-slice donation in earlier L4 versions [23], with one crucial difference: a reservation will only transfer if the receiver does not already have a reservation (a passive server or a thread which has transferred away its reservation). That way, all of a passive server's execution time is forced to be accounted against a client-provided reservation, while an active server will always execute on its own reservation. Both cases enforce temporal isolation between clients.

Reservation transfer avoids invoking the scheduler or updating accounting parameters, key properties for maintaining seL4's highly-efficient IPC. But we obviously need to consider budget expiry and mode changes.

A. Priority Inversion

Resource servers are critical sections, which means to maintain system schedulability we must provide a mechanism to avoid unbounded priority inversion. Priority inheritance (besides its other drawbacks such as implementation complexity and long worst-case blocking times) is infeasible to implement in a security-oriented model of IPC being mediated by endpoints: the kernel has no knowledge of who will be receiving messages sent to a specific endpoint, and thus cannot determine which thread should inherit the priority of the sender thread blocked on the endpoint. Similar comments apply to the original priority-ceiling protocol.

Instead we provide the means for user-level code to implement basic priority ceilings, following highest locker's protocol (HLP), where resources are assigned ceiling priorities and tasks that acquire a resource run at the ceiling priority immediately. HLP is used in POSIX for `PRIO_PROTECT` with one key difference, while POSIX runs the task at the highest priority of any resources held, our model assumes that nested resource access will be in ascending priority order. The kernel mechanism for this is simple: even a passive server has a defined priority, at which it executes irrespective of the priority of the thread whose reservation the server borrowed. A correct system configuration then requires that resource servers are given the correct ceiling priority. (Note that user-level can, in principle, do this assignment automatically: only clients who have a `send` capability on the server's request endpoint can invoke the server. The resource manager which distributes these capabilities can adjust the server priority to the maximum of the priorities of all clients to which it hands the server's request endpoint capability.)

B. Budget Expiry

If the budget of a server's borrowed reservation expires before the server completes the request, the server is left in a state where it cannot serve other client's requests until the borrowed reservation is replenished. This constitutes a potential criticality inversion, where a high thread must trust that any low thread invoking the server does it with sufficient budget, obviously not an acceptable situation.

The *helping* approach taken by Fiasco [23], where clients donate budget to the blocked thread to get it out of the server, does not work in the security-oriented IPC endpoint design: The kernel has no way of knowing on which endpoint the server will attempt to receive next, and thus cannot determine the helper.

Temporal exceptions are a suitable mechanism for recovering from this situation. When the reservation expires, the kernel sends an exception message to the *owner* of the reservation (i.e. the thread to which the reservation was allocated, ignoring any borrowing). The temporal exception handler is then responsible for the recovery action. Possible actions include giving the faulter an emergency budget or resetting the server back to a defined state (ready to receive further requests) and sending an error replying to the client on the server's behalf.

The exception handler has its own reservation, which must be sufficient to implement the policy required by that server. Note that the required budget can be quite large, if the number of a server's low clients is large, and it must be replenished at the highest rate of all clients. Clearly, cross-criticality resource-sharing must be done wisely. seL4's protection mechanisms help limit such sharing, by controlling the distribution of capabilities to server request endpoints.

C. Mode change

Mode changes can occur while a shared resource is being accessed, specifically while threads are enqueued on the resource endpoint or actively using the resource. We lazily detect

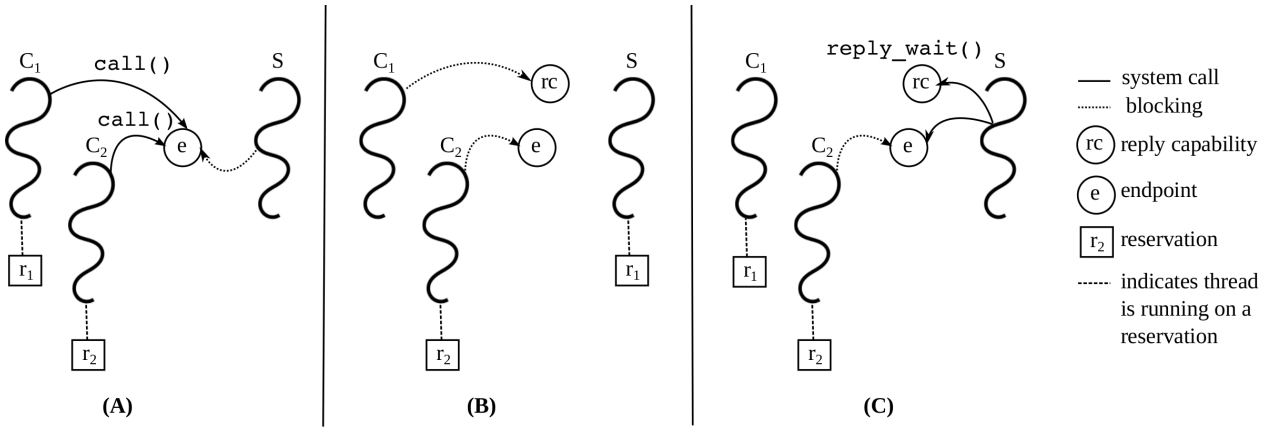

Fig. 1. Client threads invoke a passive server via IPC on endpoint e. In **(A)**, two clients, (C_1 and C_2 with reservations r_1, r_2) both send requests to the server S via `call()`. In **(B)**, C_1's message is processed first: the kernel generates a one-shot endpoint (rc) that C_1 blocks on, and the server borrows C_1's reservation r_1 while running on C_1's behalf, while C_2 remains blocked on the endpoint. **(C)** shows S completing the invocation using `reply_wait()` on e, transferring r_1 back to C_1 over rc. Note that the system is strictly speaking never in the state shown in **(C)**, as the `reply_wait()` operation is atomic, so S switches directly from the reply to C_1 (through rc) to receiving the message and reservation from C_2.

if threads queued on an endpoint have sufficient criticality: if a high-criticality server attempts to receive a message and the client has insufficient criticality, it will be removed from the endpoint queue and post-poned. The IPC operation will restart when the client is scheduled after the kernel criticality level has been raised. Threads actively using a resource during a criticality change are detected when they are next scheduled: the kernel detects that the server is running on a reservation belonging to a thread with an insufficient criticality level, and sends a temporal exception to the servers temporal exception handler, which can reset the server.

Of course, the approach described above works only for systems using the kernel criticality level to implement mode changes. Other mode change policies involve client priorities being lowered or raised, and/or reservation parameters changing. Endpoint queues are reordered on priority change, and tasks that are suspended have pending IPC messages cancelled, while changing reservation parameters has no effect on the endpoint queues, but will result in an exception triggering the budget expiry handler if a thread no longer has budget to complete a resource request.

A server's borrowed reservation may run out of budget after a mode change, resulting in a temporal exception. As the server runs at the ceiling priority, which should be unaffected by the mode change, a change of the client priority will not take effect until the server replies to the client. This increases the worst-case cost of the mode change.

We observe that handling of a temporal exception depends greatly on circumstances: An exception triggered by a low thread may simply be ignored, resulting in rate-limiting. If the low thread's budget expires while borrowed by a server, a reset action may be required. If, however, a high thread's budget expires, this may require a mode switch. This means that the handler needs sufficient information to determine the course of action. To solve this, we allow a data word to be set in each scheduling context which is delivered with the temporal fault message. Systems can set this data word to identify the client, or the clients criticality, within the temporal fault handler.

D. Summary

Our kernel changes in total account for a 2045 LoC patch[4] This includes the addition of a release queue of pending and rate-limited jobs, reservations, criticalities, improved timer driver and modifications to the IPC path.

V. Related Work

Traditional resource kernels [24] support slack reuse but do not guarantee deadlines of low-criticality tasks even if this does not prevent high tasks from timely execution. Burns and Davis [17] present a detailed survey of mixed-criticality systems research. The systems closest to ours in their aims are COMPOSITE and Fiasco.

COMPOSITE [25] completely frees the kernel from any scheduling policy by providing mechanisms for hierarchical user-level scheduling. It reduces overhead-related capacity loss by configuration buffers shared between user-level and the kernel. Some capacity loss remains as timer interrupts must be delivered down the scheduling hierarchy. This approach does not suit seL4, as the required reasoning about concurrent access (by kernel and user-level) to those buffers would drastically increase verification overhead [1]. Unlike all L4 microkernels, COMPOSITE implements a migrating thread model [26]. This implies that access to shared resources does not block, thus avoiding priority inversion, although at the cost of requiring all server code to be re-entrant, a requirement we do not want to impose.

A version of Fiasco [23] uses bandwidth inheritance [27] over IPC, which is analogous to priority inheritance. For security reasons, Fiasco has also moved to IPC mediated through endpoints, so this approach does not work in later versions of the kernel.

Brandenburg introduces an IPC protocol for clustered multicore mixed criticality systems using EDF and CBS, using multiple IPC queues to separate critical real-time and non-critical background tasks [22]. As it uses unmediated IPC,

[4]Counted by David A. Wheeler's "SLOCCount".

their approach does not directly apply to seL4. They avoid mode changes by servers prioritizing high clients irrespective of scheduling priority, and resetting a server on budget expiry.

Quest-V [28] is a separation kernel which can be used to sandbox tasks of different criticalities, allowing them to safely share hardware, however has no support for mode changes and thus offers no utilisation increase. Lackorzynski showed that to virtualise multiple mixed criticality RTOSes, information must be passed between the guest and host about mode changes to avoid violating the schedulability guarantees of either guest, and implemented this in Fiasco.OC [29]. An implementation of mixed criticality systems in Ada, demonstrates reordering of priorities on mode change [30].

Recent proposals adapt the original priority-ceiling protocol to mixed criticality [31], [32], but are unsuitable for us as explained in Section IV-A.

VI. Conclusions & Future Work

We have outlined a model for supporting mixed-criticality scheduling in seL4. The model supports cross-criticality resource sharing and mode switches, while retaining seL4's security properties and high average-case performance.

We have a mostly complete implementation and are presently working on evaluating it by building practical mixed-criticality systems on top, including a UAV and a space satellite. This will be the real test of the practicality of the proposed approach. In particular, we need this practical experience to determine the best approach to the (user-level) implementation of mode switches and temporal exception handling.

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *Trans. Comp. Syst.*, vol. 32, pp. 2:1–2:70, Feb 2014.

[2] K. Elphinstone and G. Heiser, "From L3 to seL4 – what have we learnt in 20 years of L4 microkernels?," in *SOSP*, (Farmington, PA, USA), pp. 133–150, Nov 2013.

[3] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *32nd RTSS*, (Vienna, Austria), pp. 339–348, Nov 2011.

[4] C. Smith, *Car Hacker's Handbook*. 2014.

[5] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an operating system kernel," *CACM*, vol. 53, pp. 107–115, Jun 2010.

[6] J. Liedtke, "On μ-kernel construction," in *15th SOSP*, (Copper Mountain, CO, USA), pp. 237–250, Dec 1995.

[7] J. S. Shapiro, "Vulnerabilities in synchronous IPC designs," in *IEEE Symp. Security & Privacy*, (Oakland, CA, USA), May 2003.

[8] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *CACM*, vol. 9, pp. 143–155, 1966.

[9] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *2nd ITP*, vol. 6898 of *LNCS*, (Nijmegen, The Netherlands), pp. 325–340, Aug 2011.

[10] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *IEEE Symp. Security & Privacy*, (San Francisco, CA), pp. 415–429, May 2013.

[11] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of some timing channels on seL4," in *ACM Conference on Computer and Communications Security (CCS)*, (Scottsdale, Arizona, USA), Nov 2014.

[12] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *19th RTAS*, (Philadelphia, PA, USA), pp. 45–54, Apr 2013.

[13] B. Blackham, Y. Shi, and G. Heiser, "Improving interrupt response time in a verifiable protected microkernel," in *7th EuroSys*, (Bern, Switzerland), pp. 323–336, Apr 2012.

[14] S. M. Petters, K. Elphinstone, and G. Heiser, *Trustworthy Real-Time Systems*, pp. 191–206. Signals & Communication, Jan 2012.

[15] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: An abstraction for managing processor usage," in *Proceedings of the 4th Workshop on Workstation Operating Systems*, pp. 129–134, 1993.

[16] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: a resource-centric approach to real-time and multimedia systems," in *Readings in multimedia computing and networking*, pp. 476–490, 2001.

[17] A. Burns and R. Davis, "Mixed criticality systems – a review." http://www-users.cs.york.ac.uk/~burns/review.pdf, Jun 2014. Online; accessed 29-Sept-2014.

[18] B. Sprunt, L. Sha, and J. Lehoczky, "Scheduling sporadic and aperiodic tasks in a hard real-time system," technical report CMU/SEU-89-TR-011, Carnegie Mellon University, Software Engineering Institute, Apr 1989. URL resources.sei.cmu.edu/library/asset-view.cfm?assetid=10919.

[19] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada*. 2007.

[20] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *J. Real–Time Syst.*, vol. 27, no. 2, pp. 123–167, 2004.

[21] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Proceedings of the 1st Workshop on Mixed Criticality Systems*, pp. 1–6, 2013.

[22] B. B. Brandenburg, "A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems," in *35th RTSS*, (Rome, Italy), Dec 2014. To appear.

[23] U. Steinberg, A. Böttcher, and B. Kauer, "Timeslice donation in component-based systems," in *OSPERT*, (Brussels, Belgium), Jul 2010.

[24] S. Oikawa and R. Rajkumar, "Linux/RK: A portable resource kernel in Linux," in *19th RTSS*, 1998.

[25] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *29th RTSS*, (Barcelona, Spain), Nov 2008.

[26] G. Parmer, "The case for thread migration: Predictable IPC in a customizable and reliable OS," in *OSPERT*, (Brussels, Belgium), Jul 2010.

[27] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *Trans. Computers*, vol. 53, pp. 1591–1601, Dec 2004.

[28] Y. Li, R. West, and E. S. Missimer, "The Quest-V separation kernel for mixed criticality systems," in *1st WMC*, pp. 31–36, Dec 2013.

[29] A. Lackorzynski, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *EMSOFT*, (Tampere, Finland), pp. 93–102, Oct 2012.

[30] S. Baruah and A. Burns, "Implementing mixed criticality systems in Ada," in *Proceedings of Reliable Software Technologies – Ada-Europe*, pp. 174–188, 2011.

[31] A. Burns, "The application of the original priority ceiling protocol to mixed criticality systems," pp. 7–11, 2013.

[32] Q. Zhao, Z. Gu, and H. Zeng, "HLC-PCP: A resource synchronization protocol for certifiable mixed criticality scheduling," *Embedded Systems Letters, IEEE*, vol. 6, pp. 8 – 11, Jul 2013.