School of Computer Science & Engineering
Trustworthy Systems Group

Evolving seL4CP Into a Dynamic OS

Ben Leslie[1] Gernot Heiser[2]

[1]Breakaway Consulting [2]UNSW Sydney

March 2022

1 Introduction: seL4 Core Platform

1.1 Purpose

The **seL4 Core Platform** (seL4CP) provides an easy-to-deploy operating system for embedded systems, based on seL4, the world's most highly assured OS kernel. It is intentionally kept simple and lean. This is in contrast to the `CAmkES` framework, which is presently the recommended base for the construction of seL4-based systems. CAmkES brings significant complexity to the development and configuration of seL4-based systems that provide a steep learning curve. CAmkES internals are hard to understand and harder to extend, and the framework also imposes significant performance overheads.

In contrast, seL4CP aims to remain approachable for all embedded-systems developers in industry and to impose low performance overheads. seL4CP was co-developed by Breakaway Consulting with a device called Laot that protects critical infrastructure (such as remotely-managed power stations) from cyber attacks. The Laot device was built using seL4CP by an engineer without prior seL4 systems development experience, which indicates that the usability aim had been successfully achieved.

1.2 Concepts

seL4CP introduces a small number of very simple abstractions that are close to seL4 objects, lending themselves to a lean implementation. Specifically, an seL4CP system consists of *protection domains* (PDs) and *memory regions* (MRs). Interactions between PDs is authorised by the precense of a channel link two PDs. Interaction can be through through *notifications* which provides a singalling mechanism, and *protected procedure calls* (PPCs) support the application-controlled switch to a function executing in a different PR.

A PD runs an seL4CP program, which is an ELF file containing code and data, both of which are exposed as memory regions and mapped into the PD.

1.3 Present state

1.3.1 Fully static system

seL4PD systems are presently fully configured at build time: The configuration of PDs, MRs and channels is performed at build time, and embedded into the seL4CP system image. There are no mechanisms that enable runtime modification of this configuration.

Present seL4CP systems are *static* in two ways: the have a *static architecture* (structure of PRs and their interactions) and run *static code* (i.e. the executed code cannot change).[1]

1.3.2 Fault Handling

Fault handing in current seL4CP is simplistic: The system will simply stop a PD that faults (by using an invalid capability, or accessing an invalid virtual memory address), while other protection domains continue to run. Of course, if a faulting PD is providing a service to other PDs then it is unlikely that these PDs will continue to operate once the server PD is stopped.

1.4 Proposed improvements

The fully static design is simple and efficient, but is seriously limited in the use cases it supports. The aim of this investigation is to explore options and approaches for supporting a much wider class of use cases, as they present themselves in embedded and cyberphysical systems.

We do not currently propose to move away from the *static architecture*, as this is the key to simplicity, efficient implementation, and enables reasoning about security properties. Secure dynamic architectures require much more research and are years away from practical deployment. However, we will see that the static architecture approach can scale to satisfy the requirements of most cyberphysical systems.

Hence we propose retaining the static architecture while stepwise removing the static code property. Specifically, we explore the possibility of incrementally extending dynamic functionality, in order to address increasingly broader classes of use cases. Specifically, we look at five small increments:

- supporting restartable PDs (Section 2);
- introducing a PD hierarchy (Section 3);
- introducing *empty* PDs (Section 4);
- dynamic loading of code (Section 5.1);
- loading and unloading new applications at run time (Section 5).

We observe that small increments will significantly broaden the application domain of seL4CP.

2 Restartable PDs

The first improvement to seL4CP is to add a second built-in fault handling technique, *restart*, to the existing *stop*.

[1] CAmkES has the same limitations.

2.1 Model

We add a *fault handler* attribute to PDs, which can be either *stop* (the current, and default behaviour) or *restart*.

If *restart* is specified, then on a fault the PD shall restart execution from its initial entry point. This enables a PD that experiences a transient fault to continue to provide a service.

A challenge with restart is that the memory in a program's `.data` and `.bss` segments is likely to have been modified, and no longer contain the correct initial values. To support this the PD must correctly initialise these segments as part of the C runtime startup code. An implementation of this logic shall be provided in `libsel4cp.a`. To support this approach, a read-only copy of the `.data` segment will be included in the program image.

Whether a specific PD should be configured as *stop* or *restart* is the choice of the system designer.

2.2 Design Analysis

2.2.1 Implementation Approach

The system description XML schema shall be extended to allow PDs to specify the fault handling approach. This is a small change to the existing XML parsing and tooling data model. The fault handling setting shall be embedded into the seL4CP monitor data structures directly by the build tool.

The seL4CP monitor (which is the system initial task) shall be updated so that it:

1. Determine the fault handling approach by examining a data structure (which is populated by the build tool).
2. Perform the appropriate action, which is simply an `seL4_TCB_Suspend`, or an `seL4_TCB_WriteRegisters` as appropriate.

To enable resetting the `.data` segment appropriately the linker script and C runtime provided by `libsel4cp` shall be updated as required.

2.2.2 Complexity

The changes described in the implementation approach are minor in nature and represent a very incremental complexity to the existing system. The runtime monitor is effectively adding an `if` statement over the current implementation.

The most complexity in this change is around the linker script. The tooling and linker script language syntax and smenatics can present some complexity here, even though the overall change is minimal.

2.2.3 Security

There are no significant risks introduced by this change.

A perceived risk may be that a constantly faulting on startup PD introduces a denial of service. However, the PD's access to the CPU is restricted by its scheduling budget (in the new MCS variant of the kernel), which should be set properly to prevent the PD from

monopolising the processor. This is no worse then a PD having a bug that leads to an infinite loop rather than a crash.

2.2.4 Performance

This change does not create any significant performance overhead. At runtime there is now the possibility that, on faults of specifically configured PDs, there may be a small amount of extra code run for the new fault handling method.

3 PD Hierarchy

While supporting recovery from some transient faults, restarting will not help much for systematic faults, especially many software bugs, and may lead to livelock. To mitigate this limitation we allow a more privileged PD to handle fault recovery dynamically.

3.1 Model

Currently seL4CP is a flat system, where all PDs are considered equal by the system (different priorities notwithstanding). One consequence of this is that all PDs have the single built-in fault handler, which only provides a limited set of fault handling options.

To address this restriction, we introduce a PD hierarchy: Every PD in the system can have zero of more *child PDs*. Consequently, a PD *may* have a *parent PD*. A PD without a parent PD is known as a *root PD*.

When a child PD faults, the system invokes a fault entry point in the parent:

```
void fault(sel4cp_pd pd, sel4cp_msginfo msginfo);
```

The arguments to the fault invocation are `pd`, the identifier of the faulting child, and a message info data structure that allows the parent to determine the cause of the fault.

The parent is responsible for determining the appropriate fault handling approach for its children; it may implement whatever specific policy it considers appropriate. For example, it might log an error and then shut down the whole system if the faulting child provides a critical service. Alternatively, it may decide to restart the PD, but only if it has not been restarted it in the last 5 seconds.

A fault in a root PD is handled by the built-in fault handlers as either *stop* or *restart* according to the fault handling attribute configured for the PD.

To implement any fault handling or recovery, the parent PD needs a set of APIs which allow it some form of control on child PDs. These are:

```
void sel4cp_pd_stop(sel4cp_pd pd)

void sel4cp_pd_restart(sel4cp_pd pd, uintptr_t entry_point)
```

For all APIs the child PD is identified via a protection domain identifier. This is the same identifier as passed to the `fault` entry point. Every child PD has a unique identifier within the context of its parent. The identifier is an integer in the range 0 - 255. The PD identifier need only be unique for the given parent, for example parent PD A and parent PD B may both have a child with identifier 5.

The `sel4cp_pd_stop` API shall stop execution of the PD; it won't be scheduled again until started. The `sel4cp_pd_restart` API shall start execution of the PD at the specified entry point.

At this point the API interface is intentionally kept narrow. There are potentially more general APIs (such as read/write arbitrary register) which could be provided. However such general-purpose APIs are not needed to solve the immediate use-cases and so are left as potential areas for future expansion if and when a solid use case is identified.

3.2 Design Analysis

3.2.1 Implementation Approach

An seL4 endpoint shall be created for each parent PD. In situations where the parent PD also provides protected procedures, a single endpoint is used for both faults and protected procedure calls.

A badged capability to the parent endpoint is created for each child PD. This badged capability is copied to the child PD's TCB via the `seL4_TCB_SetSpace` API.

The fault endpoint capability has the 2^{nd} top bit set to one which allows the incoming IPC to be identified as a fault message (as opposed to a protected procedure call). This bit is used in the IPC decoding loop to detect fault messages, and call the fault entry point.

Parent PDs are provided with capabilities to the TCB of all child PDs. The `sel4cp_pd_stop` is implemented as a simple wrapper around `seL4_TCB_Suspend`, while `sel4cp_pd_restart` is a simple wrapper around `seL4_TCB_WriteRegisters`.

3.2.2 Complexity

The design inherently adds a new level of complexity to seL4CP. We move from a flat list of PDs in the system to a tree of PDs. Additionally, the concept of PD identifiers is introduced.

However, should a system designer wish to continue with a simple flat system design this is still available. In such a case no runtime complexity is added to the system as compared to the current system design.

For system designers that opt-in to using the PD hierarchy features the amount of complexity added is commensurate with the inherent complexity of their fault handling approach.

In terms of implementation complexity the implementation of the 'libsel4cp.a' runtime library adds two new API functions, which are simple (less than 10 LoC) wrappers around existing seL4 APIs. There is a very small amount of complexity added in the IPC handling loop to detect and dispatch calls the the `fault` entry point when necessary.

The most implementation complexity comes to the seL4CP image building tool. There is a reasonable amount of complexity in correctly parsing and creating a data model for the tree internally. Thankfully, it is possible to mostly limit this complexity to the parsing phase of the tool. The inherent tree structure that comes from the system description can effectively be flattened into a list of PDs so that the existing algorithms used to generate the start up code remain mostly unaffected. There are small changes required to the generation of the startup sequence, however this is mostly limited to:

1. creating slightly different capabilities for the fault_ep capability

2. adding capabilities to TCB into the parents CSpace.

3.2.3 Security

This design limits the amount of authority that is required in the system.

To allow control of child PD, parent PDs are now provided with TCB capabilities for their children (and only their children).

Unfortunately seL4 TCB capabilities are not fine-grained. Specifically, this means that an seL4 TCB capabilities provides full authority over the TCB. Ideally the authority in this case would be limited to only those invocations that are required to implement the `sel4cp_pd_`‘ functions.

Additional analysis is needed to determine if this provides a parent with too much authority over a child TCB. If so options for remediation would be:

1. To not provide capabilities directly to parents, and instead have the monitor as the only entity with full authority over TCBs. Implementation of `sel4cp_pd_*` would be via calls to the monitor to perform on the parents behalf.
2. Changes to seL4 so that capability rights can be specific on TCB capabilities which limit to specific subset of invocations.

An additional concern could be that a child PD effectively creates a denial-of-service on the parent by intentionally faulting. MCS budget limits should be used to ensure this can not cause a problem.

3.2.4 Performance

There are no performance implications of this. Current performance is maintained. Existing functionality is limited only by seL4 performance.

4 Empty PDs

In the current implementation of seL4CP every PD must be configured with a *program image*, which is configured in with the `program_image` tag for the PD in the system description. To provide more flexibility to seL4CP, we remove the requirement for a PD to specify a `program_image` at build time.

4.1 Model

Before this change, when a program image is configured the build system configures the system so that the program is automatically started at boot. To achieve this the following actions are performed:

1. Convert the ELF program segments into seL4CP memory regions which are then embedded into the system image.
2. Ensure the memory regions are mapped with appropriate permissions into the PD’s address space.
3. Automatically set up an IPC memory buffer region for the PD.

4. Perform symbol patching, to set specific data values in the ELF segments to the appropriate values. This includes any explicitly configured patching (done with `setvar`) or built-in patching (specifically of the `__sel4_ipc_buffer_obj` symbol).
5. Configure the TCB's thread program counter register based on the ELF file entry point, and resumes the TCB.

When a program image is not specified the steps mentioned above will no longer take place. All other protection domain creation remain unchanged. The thread control block (TCB) of such an empty PD will be left in a stopped state. The PD's parent can then start the PD using `sel4cp_pd_restart` API. In this case a system design must create appropriate memory regions into which a program image can be loaded at runtime.

Note: All Root PDs must have a `program_image` supplied.

4.2 Design Analysis

4.2.1 Implementation Approach

The implementation of this approach is all within the build tool itself.

The system description parsing shall be changed to allow for `program_image` to remain unspecified.

The image build step shall be modified to check if a `program_image` is available for the PD and act accordingly.

4.2.2 Complexity

There is a small amount of complexity added to the build tool to handle a `program_image` now being optional. This is minimal and not expected to cause any issues.

4.2.3 Security

This feature does not introduce any additional security considerations.

4.2.4 Performance

There are no negative performance implications due to this feature.

5 Dynamic System Use Cases

This section describes how the improvements described in the prior sections can be applied to dymamic system use cases.

5.1 Dynamic Code Loading

By adding the APIs required to support the improved fault handling it is now also possible ot support dynamic code loading at runtime. This will enable software upgrades on a running system.

A child PD can be reloaded by a parent PD by:

1. Using `sel4cp_pd_stop` to stop the current execution of the PD.
2. Copying the new code into the child PDs MRs. It is necessary for the parent PD to have the child PDs MR mapped into its own address space to enable this.
3. Using `sel4cp_pd_restart` to start the newly loaded program.

It is a system specific problem as to how the program data is loaded into the memory, or where the originating ELF file is stored.

5.2 Late loading

Having enabled dynamic loading, it is now possible to load at run time applications that were not part of the initial system configuration. This can be achieved by *empty PDs* as described in the previous section.

Setting this attribute on a PD means it operates as before: The system will start it at boot time. If the attribute is unset, the system will not attempt to locate an ELF image for the PD, will not initialise it other than setting up the specified channels and MRs, and will not start it.

To start the PD, its parent must then run a *dynamic loader* program to locate and load an ELF file into the target PD and then start it.

5.3 App-specific access rights

The late-loading approach just described opens up further use cases, but still has some restrictions. In particular, it requires that the template PD is configured with the correct channels and MRs before it is known what app to run. Inevitably this means that the PD needs to be configured with generous access, which is likely to violate the *principle of least authority* (POLA).

POLA can be maintained is possible to:

- remove excessive rights prior to starting the dynamically loaded program, and
- restore the empty PD to its original state after stopping the dynamically loaded program.

This means that the dynamic loader needs to be given access to the template's capability space (Cspace). The loader can then read the authorised access rights from the app's signed ELF file, remove all unauthorised capabilities for the template's Cspace and then load the app as normal. At deactivation time, the loader will reset that target PD's memory and re-insert the original set of capabilities into its Cspace. (There are some obvious variants of this general scheme that need to investigated further.)

5.4 Limited dynamic memory management

While the above extension supports app-specific access rights, it still has a completely static memory assignment. This is largely not a problem, as cyberphysical systems requires a high degree of predictability and therefore cannot allow over-allocation of resources.

However, there is a case for limited dynamism, where a reserve of memory can be temporarily loaned to a PD, e.g. to improve performance by caching bulk data in memory, or

otherwise improve performance by providing a temporary memory boost. This is safe as long as there is a protocol that allows the system to revoke the loan. A standard approach is the use of “ballooning drivers” in virtual machines, which can dynamically reclaim memory from the Linux guest OS (Waldspurger, 2002).

This level of dynamism can be achieved by giving a parent PD the right to add the capability of one of its MRs to a child’s Cspace, and remove it at a later time.

5.5 Summary

The combination of late loading, app-specific access rights and MR loaning will address all identified needs of cyberphysical systems, showing that the seL4CP can scale to a really large space of embedded systems use cases.

However, designing the required mechanism in detail is premature at this stage, as we first need to gain experience with the previous steps. We therefore leave the design and detailed analysis of generalised app loading to future work.

6 Proof of Concept

A (somewhat poorly named) branch that prototypes the functionality described in this document is available in the pd_hierarchy branch on Github. The status is:

- Restartable PDs, the pd hierarchy and empty pds are implemented
- A demonstration of a simple policy for restarting / stopping a PD is provided.
- Empty PDs are untested.
- The restartable root PD is not yet implemented (but easy to do).
- The advanced features are not yet implemented, including rights reductions for empty PDs, rights reinitialisation for empty PDs, and dynamic memory regions.

References

Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–194, Boston, MA, US, 2002. USENIX.