Codezero® Embedded Hypervisor Documentation

Version 1.0

1 November 2010

Table of Contents

Preface

Codezero Documentation

The Codezero documentation below includes information on getting started with development, tutorials, API reference, architecture descriptions, and more. If you believe you are missing information or a concept is not described clearly enough, please notify us by direct e-mail or ask on our mailing list.

1.) Getting Started with Codezero

You may take a look at the Getting started with Codezero development guide for downloading, installing and running Codezero along with all the tools that are required. Going through this guide, you should be able to build a working environment and get up and running with Codezero Microkernel development from the comfort of your host PC.

2.) Codezero Project Overview

Codezero Microkernel features and applications are discussed in the Codezero project overview section to provide a general background about the software.

3.) Codezero API Reference

Codezero is a new microkernel written from scratch as an interpretation of the earlier line of L4 microkernels. As such, it has a very similar API to other L4-based systems.

Codezero Microkernel development is an ongoing process. Naturally, there may be changes in the API as the software evolves. However the following Codezero API reference document should serve as a good, stable starting point.

4.) Applications Development

In this section, a tutorial is provided for writing Codezero Applications in general.

Essentially, Codezero can be used as a barebones system, or the userspace libraries provided may be used for higher-level software development. The Writing applications using LIBL4 chapter may be used as a starting point for writing both barebones Codezero applications or leveraging the virtualized Linux instance.

Getting started with Codezero development

This step-by-step guide explains on how to download, install, and configure various tools required to build, run, and debug Codezero.

1 Creating the directory structure for Codezero sources and tools

Make the following commands on the shell:

```
% mkdir /opt
% mkdir /opt/archives
% mkdir /opt/tools
```

This will respectively create the directories where the Codezero source code and required build tools will be saved and installed in reference to this guide.

2 Installing Git

The Git SCM tool is needed for downloading and managing Codezero sources.

For Ubuntu, please use the command below for the Git installation:

```
% sudo aptitude install git-core
```

To install Git or any other package from the Ubuntu repository, the sudo password is required. On Ubuntu, the Git installation is done by the step above. For non-Ubuntu users, please refer to Installing from Source guide for the installation of Git from scratch.

3 Installing SCons

Codezero uses SCons as its build software. SCons is a python-based build tool where builds can be manipulated using Python scripts and functions.

For Ubuntu, please use the command below for installation:

```
% sudo aptitude install scons
```

The sudo password is required for this step. For non-Ubuntu users, please refer to the Installing SCons section in the Installing from Source guide.

4 Installing the GCC cross-compiler

The Codezero project uses the Codesourcery ARM cross-compilers for compiling the sources. Other cross-compilers, such as crosstools, may work, but testing only has been done using this toolchain.

Note: The Codesourcery toolchain comes in two types for ARM, namely EABI and GNUEABI. The former is used for building barebones embedded software (such as OS kernels); the latter is for building applications for Linux. In Codezero, a similar choice is made: the microker-

nel is built using the EABI toolchain, and the userland is built using the GNUEABI toolchain. The relevant compiler prefix options are already referred to in the configuration system and may be changed for different toolchains.

For installing the cross-compiler toolchain, please follow the steps below.

1. Create a directory for the cross-compiler toolchain:

```
% mkdir /opt/archives/cc
```

2. Download the latest GCC ARM cross-compiler toolchain (IA32 GNU/Linux Installer) for ARM EABI from: http://www.codesourcery.com/sgpp/lite/arm/portal/subscription3053

A file named "arm-2009q3-68-arm-none-eabi.bin" will be downloaded.

Download latest GCC ARM cross-compiler toolchain (IA32 GNU/Linux Installer) for GNU-LINUX from: http://www.codesourcery.com/sgpp/lite/arm/portal/subscription3057

A file named "arm-2009q3-67-arm-none-linux-gnueabi.bin" will be downloaded.

We have tested 2009q1 and 2009q3 toolchains, so preferably download any of these.

3. Save both files to /opt/archives/cc.
4. Use the following shell commands to configure and install the toolchain:

```
% cd /opt/archives/cc
% chmod 777 arm-2009q3-68-arm-none-eabi.bin
```

The last command would grant execution rights to run the binary, but root permission may be needed to achieve this.

```
% . arm-2009q3-68-arm-none-eabi.bin -i console
```

The command above will start installing the toolchain. Choose appropriate options when prompted during the installation of the toolchain.

Please use an installation path as "`/opt/tools/cc`" when prompted.

Please read the "Getting Started Guide" presently at http://www.codesourcery.com/sgpp/lite/arm/portal/release1033 for more help in the installation of the toolchain.

5. Repeat step (d) above for installing the ARM binary:
   `arm-2009q3-67-arm-none-linux-gnueabi.bin`
6. Add the following lines at the end of the .bashrc file presently in the home directory of the current user (for example `/home/amit` for a fictional user named `amit`):

```
% PATH=$PATH:/opt/tools/cc/bin
% export PATH
```

This will add the path of the installed toolchain binaries to the PATH environment variable. Please note that for the new `.bashrc` file to be effective, a new console should be started. For making changes in `.bashrc` to be effective in the currently running console, please use the following command in the console:

```
% source /home/user/.bashrc
```

To test if the toolchain is installed properly and the paths are updated, please use the following command on the shell:

```
% arm-none-eabi-gcc –help
```

This should display the help section of the installed GCC cross-compiler.

5 Downloading Codezero sources

To download/clone the Codezero Git repository, please use the following shell commands:

```
% cd /opt
% git clone git://www.git.l4dev.org/codezero.git
```

This will start cloning the repository and will create the Codezero source tree under the directory /opt/codezero. The master branch on this repository contains the stable version of the sources. The devel branch includes the latest development sources.

6 Building Codezero sources

Provided here are the minimal instructions needed to build Codezero. Please refer to the Codezero project overview guide for general information about the Codezero directory layout and generated images during a Codezero build.

To build Codezero, a `build.py` script is provided in the project root: `directory /opt/codezero`.

Please issue the following shell commands to run `build.py`:

```
% cd /opt/codezero
% ./build.py --configure
% ./build.py
```

which will first configure Codezero for the build and then start building it.

This will start building the Codezero project; a configuration screen (like “make menuconfig” in Linux) will pop up, prompting the user to choose various configuration options for building Codezero.

The user can either choose the default configuration or decide to use a custom configuration. Please refer to the Codezero Configuration Symbols section for more help on configuration symbols and their meanings. Press the ‘x’ key to save the configuration and use the arrow keys to move up, down, in, and out of the menu and submenu. The ‘h’ key will display help for the individually selected symbol, and the ‘c’ key will provide help on general usage of the configuration system.

This will build the Codezero microkernel and various containers selected, producing an elf image in the respective build folders:
for example: `container0.elf` under `/opt/codezero/build/cont0/`
in case we compile Codezero with one container selected.

During the next step, the build system will combine all the generated elf images to produce a final image called `final.elf` under the `/opt/codezero/build` folder. This final executable contains the external loader with all necessary images embedded inside.

Note: There is a complete set of options available that can be used with the build.py script. Some important options are described below:

- `./build.py -f <config_file>`: Build Codezero using an existing configuration file as given by the <config_file> path.
- `./build.py -r`: Reset the configuration to its default.
- `./build.py -C`: Configure before building.

For more details please check:

```
./build.py -h
```

7 Installing QEMU

QEMU is a freely available virtual platform emulator.

Codezero uses the QEMU/ARM Versatile platform as the reference platform for development. For SMP/Cortex-A9, B Labs also provides a version that emulates this setup over the Versatile Express platform.

The latest version of QEMU needs a small patch in order to work correctly with GDB/Insight. Particularly, the `-s` switch behavior needs to be modified so that QEMU waits for a GDB connection before proceeding.

You may use the patched and prebuilt qemu-system-arm binary provided on this link. If you have any difficulty running this binary, please let us know.

Alternatively, if you prefer to build and install QEMU from its sources, please refer to the QEMU section in Installing from Source guide.

8 Installing Insight

Insight is a GUI front end for GDB with a focus on cross debugging of embedded targets. Insight is not necessary to run Codezero, but it greatly simplifies development by providing step-by-step debugging and inspection opportunities for both the Codezero microkernel and its userland applications.

Using Insight and QEMU together, a complete ICE/HW debugging environment can be emulated that is faster than most real ICE debugger/HW platform solutions. Linux kernel debugging is also possible for precision in stepping through assembler files and symbols.

- Insight 6.8 is known to work well with QEMU, and it is highly recommended to use version 6.8 of Insight with QEMU. Other versions have not been tested when creating this guide.
- Insight needs to be built from its sources, as it needs to be configured for the ARM architecture at the time of compiling Insight.

1. Create a directory for Insight.

```
% mkdir /opt/archives/insight
```

2. Download the Insight tarball
   (for example `insight-6.8.tar.bz2`) from ftp://sourceware.org/pub/insight/releases.
3. Save it to `/opt/archives/insight`.
4. Use the following shell commands to untar the tarball:

```
% cd /opt/archives/insight
% tar -xjvf insight-6.8-1.tar.bz2
```

This will generate an insight-6.8-1 directory under `/opt/archives/insight`.

Note: Insight depends on the X11 Development Libraries and the termcap library. Both of these should be installed before installing Insight.

As an example, in Ubuntu you may install these libraries by:

```
% sudo aptitude search libncurses5
% sudo aptitude install libncurses-dev libncurses5-dev
% sudo aptitude search libx11
% sudo aptitude install libx11-dev
```

The Ubuntu libncurses package contains the termcap library.

Other dependencies that must be installed are as follows:

```
% sudo aptitude search makeinfo
% sudo aptitude install texi2html
% sudo aptitude install texinfo
% sudo aptitude search expat
% sudo aptitude install libexpat-dev libexpat1-dev
```

Before running make, a `-Werror` flag may need to be removed from the `gdb/Makefile` and the `WERROR_CFLAGS` variable left as empty.

5. Please use the following shell commands to configure and install Insight:

```
% mkdir /opt/archives/insight-build
% cd /opt/archives/insight-build
% ../insight-6.8-1/configure --prefix=/opt/tools/insight --target=arm-
none-eabi
    --program-prefix=arm-none- --with-expat
```

- `target=` tells configure that the target to debug using Insight is `arm-none-eabi`. The `program-prefix` is the prefix to be added to all generated executables. For instance, `arm-none-gdb` and `arm-none-insight`. This avoids confusion with Insight/GDB and the cross-compiler gGDB that exists on the host.
- `with-expat` is required for GDB parsing the target XML definition file during connection to QEMU.

```
% ./configure --help
```

Please note that Insight is tested to work with GCC 4.4.1 or higher. It is recommended to use this version of GCC. If not, a 3.x series GCC compiler may give better results. The `CC` command-line option can be used to specify the compiler to be used for compiling Insight.

```
% make
```

followed by:

```
% make install
```

will build and install all the binaries in the system at the prefixed directory given during the configuration stage.

8.1 .gdbinit file

Insight is a GUI front end for GDB. GDB takes its configuration from the ".gdbinit" file, which should be present under the home directory of the current user. (For example: `/home/amit` for the fictional user named `amit`). To connect GDB to QEMU that is running Codezero and load its symbols, the following commands should be written to the `*.gdbinit`, file, line-by-line.

Note the file is broken up below to explain each step.

```
target remote localhost:1234
```

tells GDB to connect to QEMU.

```
load final.elf
```

loads the following image every time GDB connects to QEMU.

```
sym final.elf
```

tells GDB to use the following filename for symbol information.

```
break break_virtual
```

A break_virtual symbol is defined as standard in Codezero builds to mark the execution right after virtual memory is enabled. From this point onwards, it becomes possible to step through the code as desired. This command makes GDB stop right after virtual memory is enabled.

```
continue
stepi
```

These commands tell GDB to start from beginning, step over one instruction after the break_virtual breakpoint is hit. This will allow insight to refresh its screen with the new symbol table.

```
sym kernel.elf
```

This command tells GDB to use the `kernel.elf` file for symbol information that includes Codezero symbol information after virtual memory is enabled.

Please copy this file over to your home directory:

```
% cd codezero
% cp ./tools/gdbinit ~/.gdbinit
```

For more help on these commands, please check the GDB manual available at http://www.gnu.org/software/gdb/documentation.

Add the following lines in the end of the `.bashrc` file present in the home directory of the current user (For example: `/home/amit` for a fictional user named `amit`):

```
% PATH=$PATH:/opt/tools/insight/bin
% export PATH
```

This will add the path of Insight binaries to PATH environment variable. Please note for the new .bashrc file to be effective, a new console should be started. For making changes in `.bashrc` to be effective in the currently running console, the following command should be used:

```
% source /home/username/.bashrc
```

To test if insight is installed properly and paths are updated correctly, run the following command on the shell:

```
% cd codezero/build
% arm-none-insight
```

which should start the Insight debugger with its GUI window.

9 Running Codezero

After all tools have been installed as described in the previous sections and Codezero sources have been successfully built, everything may be put together to run Codezero.

Please refer to the Codezero project overview guide to get an understanding about what is included in the standard build, the executables created, and the directory layout of the project.

After Codezero has been successfully built, a final .elf executable should be present under the `/opt/codezero/build` directory. This image is the external loader executable that has the microkernel and all userspace container images embedded inside.

9.1 /opt/codezero/tools/run-qemu-insight script

The script above contains all the necessary commands to start QEMU, load `final.elf`, start Insight, and initiate execution until the point where virtual memory is enabled inside the Codezero microkernel. Here, some logic is provided by the .gdbinit file under the home directory for loading the symbols and setting the breakpoint during an Insight/GDB startup.

Running the script simply starts everthing and executes Codezero until virtual memory is enabled:

```
% cd /opt/codezero
% ./tools/run-qemu-insight
```

Taking a closer look at the run-qemu-insight script:

```
% cd /opt/codezero/tools
% vim run-qemu-insight
```

opens up the l4-qemu script file, which contains the lines:

```
cd build
qemu-system-arm -s -kernel final.elf -m 128 -M versatilepb -nographic &
arm-none-insight ; pkill qemu-system-arm
cd ..
```

1. As this script is run from the `/opt/codezero` directory, the shell enters the `/opt/codezero/build` directory.
2. In the second line, QEMU's ARM emulation binary is called using "qemu-system-arm".
   - The `-kernel` option tells QEMU to use the `final.elf` image, present in the current working directory as the kernel image.
   - The `-m 128` option provides emulation of 128MB of available RAM.
   - `-M` tells qemu to emulate processor/platform type; as specified here, the *ARM Versatile PB platform* is used. Currently, ARM/Versatile PB926 is the reference platform for running Codezero.
   - The `-s` option tells QEMU to wait for a debugger connection before executing anything. This option is particularly broken on newer versions of QEMU.
   - The `-nographic` option redirects PL011 UART output directly to the console.
3. `arm-none-insight` in the third line starts the Insight debugger.
4. The next command i.e., `pkill qemu-system-arm` is executed by the shell only after the debugger is terminated. This enables termination of QEMU as soon as the debugger is shut down.
5. Finally, at the last line, the shell goes one directory level up (to `/opt/codezero`) and finishes.

9.2 How it looks

The following is a typical screenshot after Codezero is up and running in control of the Insight debugger. If you have come to this stage, it means you have successfully built and run Codezero for the first time.

From this point onwards, you may step through the Codezero initialization code, place breakpoints, and generally do everything possible with the Insight/GDB debugger. In order to debug userspace applications, please refer to the Writing applications using LIBL4 tutorial.

10 Running Codezero-provided examples and templates

Codezero comes with a set of well-designed examples and templates called baremetal Projects, which a user can instantiate to see how the Codezero system works. Also, these examples can be used as templates to develop and generate new projects.

Within the `codezero/config/cml/baremetal` directory, various configuration files are present that can be used to build baremetal projects provided by Codezero.

Any of the example scripts may be invoked for a build by issuing the following commands in the shell:

```
% cd codezero
% ./build.py -f </path/to/config_file>
```

Sources for various baremetal projects are present under: `codezero/conts/baremetal/`.

Note:

- Configuration files present under `codezero/config/cml/posix`, may be used to run various examples of POSIX projects/containers.
- Configuration files present under `codezero/config/cml/linux` may be used to invoke the virtualized Linux kernel container.

11 Developing new projects for Codezero

Developers can follow the processes described here to build and integrate new projects to the Codezero build system. During the configuration phase, baremetal projects provide two paths of development; either existing sources can be instantiated on their own or new projects may be created.

The **Hello World** and **Empty** containers automatically create new project directories, ready to be built for integration with the Codezero runttime. New projects would typically reside within the `codezero/conts/<project_name>` directory, where the `<project_name>` value may be selected during configuration. Application development would take place inside these directories, and builds may be invoked via the `build.py` script at the Codezero root. Application testing may then take place on real or emulation hardware by loading and running the final.elf image resulting from these builds.

Once the project is developed to completion, it may be desirable to integrate the project as one of the standard baremetal builds provided during the configuration phase. For this purpose, `codezero/scripts/baremetal/baremetal_add_container.py` script should be run on the ready container project as follows:

```
% cd codezero
% ./scripts/baremetal/baremetal_add_container.py -a -i <description> -s
<source_path>
```

Here, `<description>` would be a short, one-line description for the new project to be added, and `<source_path>` would be a directory containing the source files for new container to be added.

This is to be done only once, and after running the script, the container should appear as one of the baremetal options during configuration.

To delete a recently added container project, the following commands may be used:

```
% cd codezero
% ./scripts/baremetal/baremetal_add_container.py -d -s <source_path>
```

where <source_path> denotes the path where the project resides. For more help on using this script, use:

```
% ./scripts/baremetal/baremetal_add_container.py -h
```

Apart from the **Hello world** and **empty** project types, all other baremetal containers may be directly built from sources that are already resident under codezero/conts/baremetal/<project_name> directories for demonstration purposes.

If you feel you are missing information or a concept is not described clearly enough, please notify us by direct email or ask on our mailing list.

12 The next step

If you plan to develop and help improving the Codezero software, the current setup would be sufficient.

The development cycle would be such that new changes would be made, recompiled, and tested on QEMU/Insight, as described in the steps above.

For further information, please consult the Codezero project overview document, Codezero API reference and join our codezero-devel mailing list, where technical issues about the Codezero microkernel are discussed.

Codezero project overview

1 Microkernel overview

Below is an example figure that succinctly describes the complete Codezero runtime with regard to software components and multiple cores:

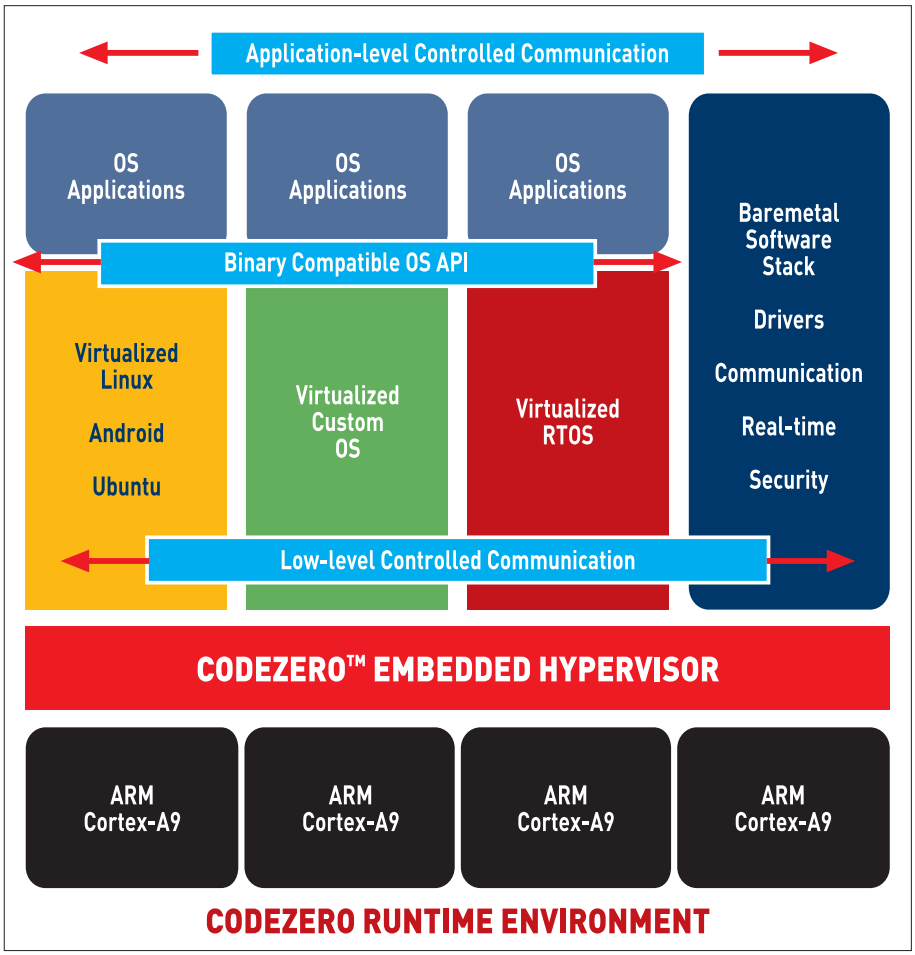


Figure 1: Codezero Runtime Environment

Provided below are sections that divide this runtime into parts, to describe the microkernel architecture and its embedded hypervisor capabilities in detail.

1.1 Codezero architecture overview

L4 microkernel architecture

Codezero is a new L4 microkernel that has been written from scratch, following the latest development and research principles on microkernel design. Codezero aims to carry the L4 microkernel architecture to the next level by actively evolving and through the support of its open community.

Design principles

The Codezero and L4 line of microkernels are founded on a few fundamental design principles. The primary principle is that only the most fundamental and abstract software mechanisms are incorporated into the microkernel, ruling out any policy from the implementation.

Codezero implements only the mechanisms to manage threads, address spaces, and the communication mechanisms between them.

Benefits

In relation to its main founding principle, the microkernel becomes simple, abstract, and flexible. Due to its abstract nature, it may be used for multiple independent purposes, such as a Hardware Abstraction Layer, a Virtualization Platform, or as a basis for implementing new operating systems. By its simple and abstract design, L4 has a distinguished position among other real-time executives.

The microkernel is the only component that runs in privileged CPU mode. Therefore, it is the central point of trust on the platform, responsible for the overall security and stable operation of the system. The microkernel is kept rigorously small, therefore making the system secure and stable.

Since the microkernel has system-wide control, the division of components and resource partitioning are also managed by the microkernel. In this respect, Codezero implements the notion of *Capabilities* to protect and safely multiplex all resources to its run-time components.

Codezero sources are a mere ~10K lines of C code, and its API consists of twelve main system calls that can be grasped within a few days. The size and simplicity of the kernel also reflects positively on its performance.

1.2 Codezero microkernel technical features

General technical features of Codezero are listed below.

General features

- System partitioning with the concept of containers
- Fully capability-checked kernel provides:
  - Flexible and configurable resource management
  - Fine-grain security
- CML2-based kernel and system configuration interface
- Written in C using a familiar open-source coding style
- Support for the ARM architecture, including ARMv7, Cortex-A9
- Multicore enabled
- Portable design and structured layout
- Focus on embedded systems
- Open-source license option and development model

The Codezero system calls provide the following mechanisms on an embedded system:

- Thread creation, destruction, and management of thread execution
- Address-space creation, deletion, and manipulation
- Interprocess communication
- Creation of virtual-to-physical address mappings
- Dynamic management of resource access via capabilities
- Userspace shared-memory synchronization
- Cache and TLB control
- System-on-Chip security, power, and error-recovery management

Real-time features

The microkernel supports *kernel preemption*. This means that even tasks running inside the microkernel may be preempted if their timeslice expires.

All blocking operations are interruptible. A task sleeping on an IPC queue, a lock, or any wait-queue, may be interrupted.

Codezero has a priority-based scheduler. As such, timeslices are distributed based on the priorities.

There are very few locks in the microkernel; consequently, concurrency conflicts are avoided and kernel preemption is enabled most of the time.

Generally, Codezero has been designed from the start to incorporate the necessary infrastructure for real-time performance.

Multithreading

Codezero implements a 1-to-1 thread model, where each thread in the system is known and controlled by the microkernel. As a result, every thread on the system is scheduled by the microkernel scheduler.

Multicore support

Codezero supports multicore. Particularly support for ARM Cortex-A9 has been added and tested on a quad-core SMP platform. Codezero transparently schedules threads to multiple cores. There is no restriction on how the threads can be scheduled to cores, though such restrictions may be introduced, if needed.

Interprocess communication

Codezero supports the conventional IPC methods provided by other L4 microkernels. There are three types of IPC defined: namely **short**, **full**, and **extended** IPC. The message transfer sizes increase in respective order. Extended IPC is particularly flexible such that it has no restrictions on the buffer address and allows page faults to happen.

Shared-memory communication is also provided for no-copy message transfers. Shared-memory access may be protected by high-performance userspace locks. Also, access to shared memory may be defined in ways that limit access to one or more parties during the lifetime of the shared area.

Please refer to the Communication section for further information on interprocess communication methods.

Virtualization features

Codezero provides OS virtualization mechanisms throughout its design. It introduces the notion of *Containers* which are essentially virtual, isolated rooms of execution for running virtualized operating systems or baremetal applications. The isolation aspect of containers provide a plain partitioning mechanism.

The system's overall security and resource management policy are further refined with the concept of *capabilities*. Codezero capabilities allow fine-grain allocation of resources and

strictly define the operations that each software component is allowed to do. Please refer to *Containers* and *capabilities* sections for further details on these concepts.

2 Architecture and runtime components

2.1 Containers

Codezero provides the notion of containers for virtualization. Each container implements a conceptually isolated execution environment with its own set of resources. For example, each container has its own set of threads, address spaces, and memory resources. Initially, a container consists of a single privileged task called the *pager.* The pager produces further children tasks to produce a rich multitasking environment in its container's boundaries. Below is a figure illustrating two containers running on top of Codezero:

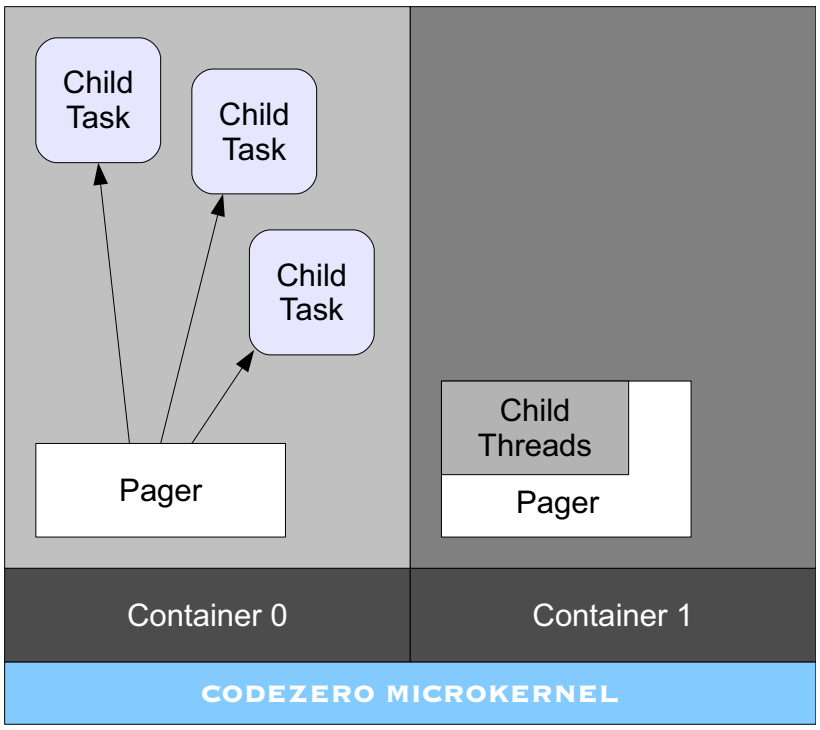


Figure 2: Codezero Containers

Each container provides enough mechanism for development of any software architecture:

- In a client-server model, a designated pager produces child threads and address spaces, and manages them. A virtualized operating system kernel and its applications model this behavior.
- In a standalone-application model, the application is the pager itself, and it creates multiple threads in the same address space. Such standalone multithreaded applications are also possible to develop.

The figure above shows sample containers for both of these scenarios. In **Container 0,** the pager has created multiple children tasks that run in their own virtual memory space. This is

a typical setup for a virtualized operating system, where the guest kernel becomes the pager and guest applications become the children tasks.

In **Container 1**, the pager itself has created many threads in its own address space. This typically corresponds to a baremetal, self-contained application.

It is worth noting that in the above scenarios, each pager has rights to its own container only. This allows for a simple and powerful way to manage isolation between containers.

Below is a code snippet that illustrates how each container is precisely represented inside the microkernel:

```
struct container {
     l4id_t cid;                                  /* Unique container ID */
     int npagers;                                    /* Number of pagers */
     struct link list;                          /* List ref for containers */
     struct address_space_list space_list;     /* List of address spaces */
     char name[CONFIG_CONTAINER_NAMESIZE];          /* Name of container */
     struct ktcb_list ktcb_list;                     /* List of threads */
     struct link pager_list;                          /* List of pagers */

     struct id_pool *thread_id_pool;       /* ID pools for thread/spaces */
     struct id_pool *space_id_pool;

     struct mutex_queue_head mutex_queue_head;  /* Userspace mutex list */
     struct cap_list cap_list;/* Capabilities shared by whole container */
     struct pager *pager;                    /* Boot-time array of pagers */
};
```

2.2 Capabilities

Containers boldly partition the system into isolated environments. Capabilities build up on the notion of containers with fine-grain and flexible resource access management.

Codezero implements capability-based security for all kernel-managed resources. All system calls, expendable resources such as memory regions, memory pools, and interprocess communication mechanisms are protected by capabilities. Below is a figure that illustrates how each container has been controlled by capabilities:

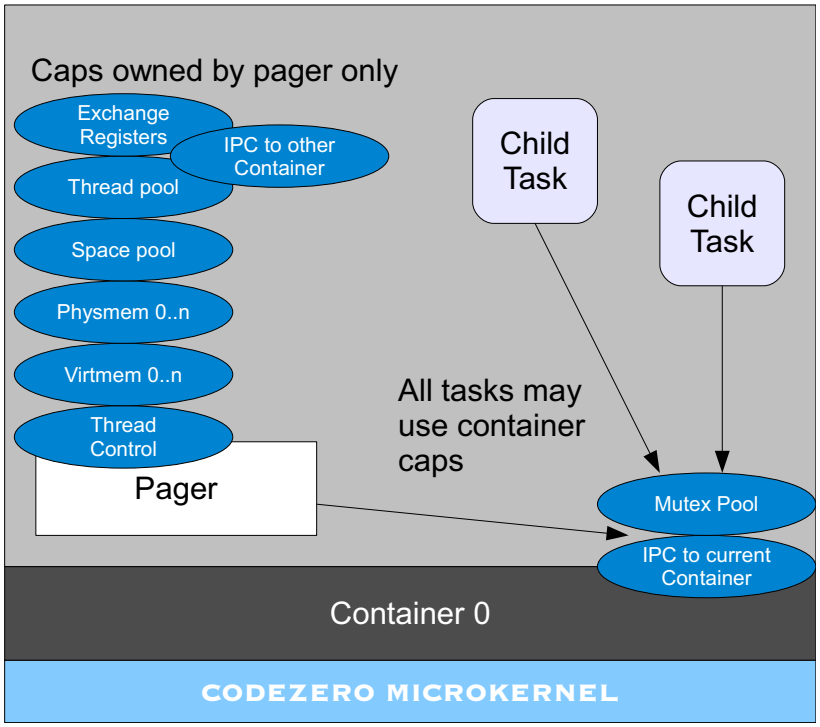


Figure 3: Capability organization

As the above figure illustrates, there are two levels of capability possession in Codezero:

- Capabilities possessed by all the tasks inside the container
- Capabilities possessed by each task (namely, not by each thread but the virtual address space)

In the above scenario, the children tasks have access to a **mutex pool** capability, and they have been allowed to communicate via IPC for any thread inside their container.

Typically, the pager task possesses all privileged capabilities inside the container. For example, the pager is allowed to create and manipulate threads via its **thread control** and **exchange registers** capabilities. The pager may also consume kernel resources, such as thread and address space pools via these capabilities.

Finally, the pager has capabilities to a set of virtual and physical memory resources. By having possession to these resources, a pager may **map** memory to its own address space, as well as the address space of its children.

For every resource access, the microkernel is concerned whether the accessing party is in possession of a corresponding capability. This design makes the overall security architecture elegant and simple for multiple reasons:

- Each new type of operation or resource requires only a new capability definition to be added.
- The access rights can be partitioned among software components flexibly, since each granted access merely depends on the possession of a capability.

The complete capability architecture in Codezero is implemented with less than 1K lines of C code. Capabilities are a key security aspect as they provide the fine-grain resource management privileges on top of the abstract notion of containers.

Please refer to the Capability section in the API reference manual chapter for further information on capabilities.

2.3 Communication

In Codezero, communication is provided in the form of IPC and shared memory. Most common method of IPC requires the notion of a *User Thread Control Block* as a message buffer, shortly referred as the UTCB. A UTCB is defined for every unique thread on the system. This block of memory can be used as thread-local storage as well as a buffer for interthread communication.

UTCB memory regions are optionally predefined in the virtual memory address space. Unlike other L4 microkernels, the task of allocating UTCBs to threads is left to the userspace pagers. This relieves the microkernel from extra policy.

On the ARM architecture, UTCBs are memory blocks that consist of 64 words (256 bytes).

IPC communication is done synchronously via a *rendez-vous* implementation in the kernel. Asynchronous message-passing methods are also possible. However, synchronous communication is encouraged for maintaining predictable software behavior. Particularly, synchronous IPC is considered as a critical building block for deterministic concurrency on multithreaded systems.

Three types of IPC are provided on Codezero:
1. Short IPC
2. Full IPC
3. Extended (long) IPC

Short IPC

This is the most common method of IPC between userspace threads. On short IPC, two threads that communicate only transfer their *primary message registers*. The primary message registers consist of those registers that are capable of mapping onto real registers on the system.

For example on an ARM system, short IPC would cover only the primary message registers MR0–MR5, which would map onto R3–R8 on real hardware registers. It is strongly advised not to rely on the register mappings, as this is a notion that is transparently managed by the microkernel.

Full IPC

In full IPC, the complete UTCB buffer is transferred among threads. In the ARM case, this would correspond to 64 machine words (256 bytes).

Extended (long) IPC

Extended IPC is the most flexible method of IPC on Codezero. Using extended IPC, a thread may transfer a message of up to 2 Kilobytes, and the message buffer may be located at **any** address on the caller's address space. The buffer may freely page fault, as any page faults

would be handled by the pager during IPC. The downside of using extended IPC is that since it may page fault, there is an aspect of nondeterminism. Also, it may be slightly slower due to extra copying that it requires.

Communication of larger-sized buffers are encouraged to be implemented via shared-page mappings.

The figure below illustrates possible IPC paths that can take place in a Codezero system:

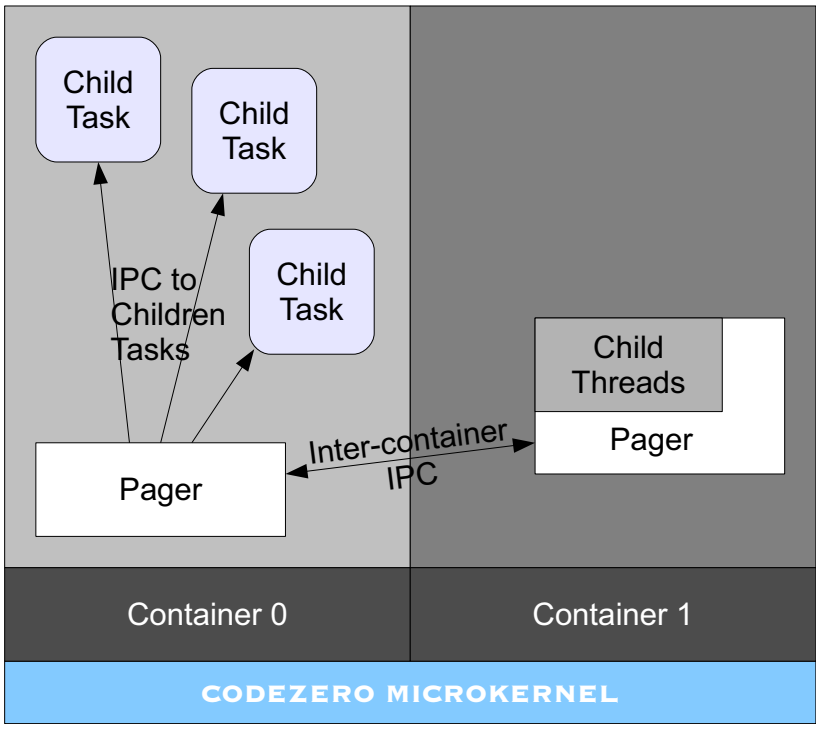


Figure 4: IPC inside and between containers

As seen in the figure, each child task in a container may communicate with their pager via IPC. Children tasks may also communicate between each other. Furthermore, if appropriate capabilities are defined, each child task on a container may communicate with tasks in other containers. The type and allowed domain of communication is dependent on whether the relevant capabilities are possessed by the communicator. For example, if a container-wide capability has been defined for **container 0** that targets **container 1**, all tasks in **container 0** may send messages to **container 1**. Similarly, communication may be restricted by making them possessed by each task, and/or targeting only one task.

Shared memory

Similar to IPC, the extent of shared-memory communication between tasks and containers are dependent on the possession of relevant physical and virtual memory capabilities. For example, two pagers that reside in different containers may have capabilities to the same physical memory region. As a result, they may map the same physical area allowing them to communicate.

2.4 Virtualization

Virtualization of operating system kernels are typically based on *para-virtualization* methodology on Codezero. In para-virtualization, an operating system kernel is abstracted away from the hardware details by manual inspection of source code. Para-virtualization is the most powerful method of virtualization since there is a dedicated engineering effort involved in the process. As such, the virtualized kernel runs with high performance, and yet with a maintainable software layer for future enhancements.

In the Codezero runtime environment, a para-virtualized OS kernel typically takes the place of the pager in a container. The kernel runs as a userspace component with capabilities to create applications. The kernel is limited in its access by the boundaries of the container that it is running inside. The figure below illustrates how a para-virtualized kernel takes place in the Codezero runtime environment:

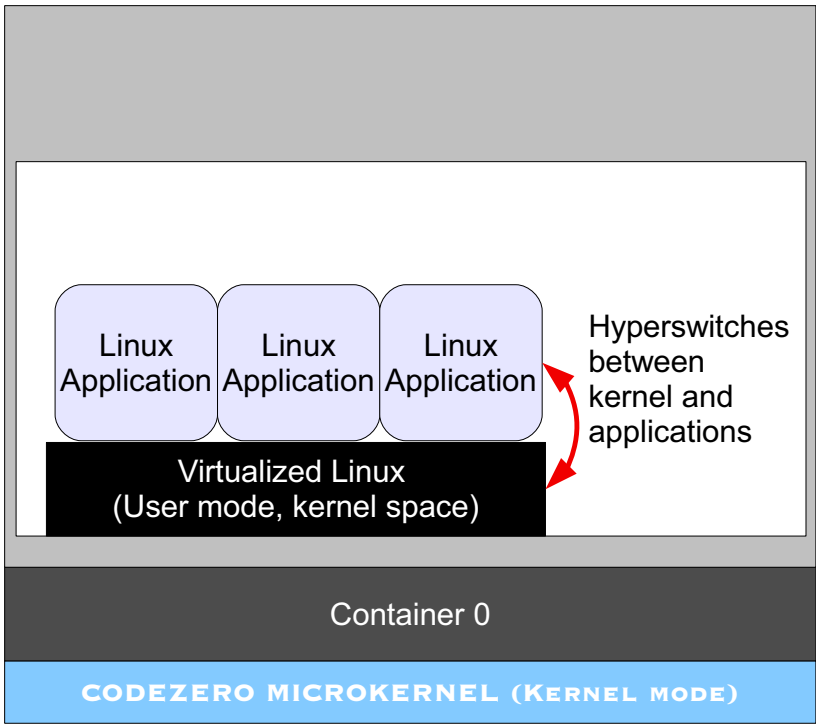


Figure 5: Paravirtualized Linux on Codezero

In the example figure, the Linux kernel is running as a privileged pager inside its own container. The kernel is initiating para-virtualized exception handling services and serves Linux applications by IPC.

2.5 Existing Paravirtualization Methods

As previously mentioned, paravirtualization typically involves modification of guest kernel operations and inserting of replacement calls into the hypervisor. One of the other significant challenges in paravirtualization is the way guest kernel services are provided to applications in the virtualized environment. Often, new virtualization constructs and extra privileged threads are introduced inside the guest kernel space, in order to provide a mech-

anism to make applications dispatch and conclude system calls and exceptions. An example mechanism is shown below:

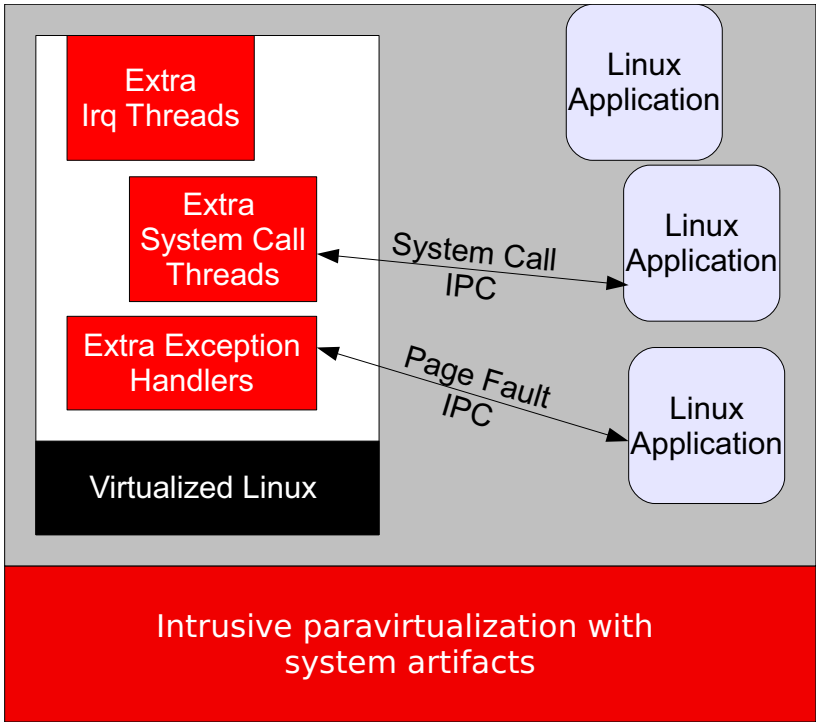


Figure 6: Intrusive paravirtualization with extra threads and system artifacts

In this paravirtualization methodology, guest application system calls are converted to an ipc mechanism inside the hypervisor. The system call ipc requests are then redirected to dedicated service threads inside the guest kernel. Similarly real irqs are redirected to dedicated irq threads inside the guest kernel space. There are multiple problems with this approach. Introduction of new virtualization threads creates extra overhead on the system. Furthermore, the original method that the cpu handles these operations are modified in a fundamental way. This results in further requirements of changes in the system. As an example, a dedicated timer irq thread would now have to figure out which guest process was running before the interrupt, in order to do process accounting on that thread.

2.6 Codezero HyperSwitch Paravirtualization

Codezero implements a different approach to the virtualization problem which have ultimately eliminated all of the artifacts introduced above. The new method named as HyperSwitch may be seen as below.

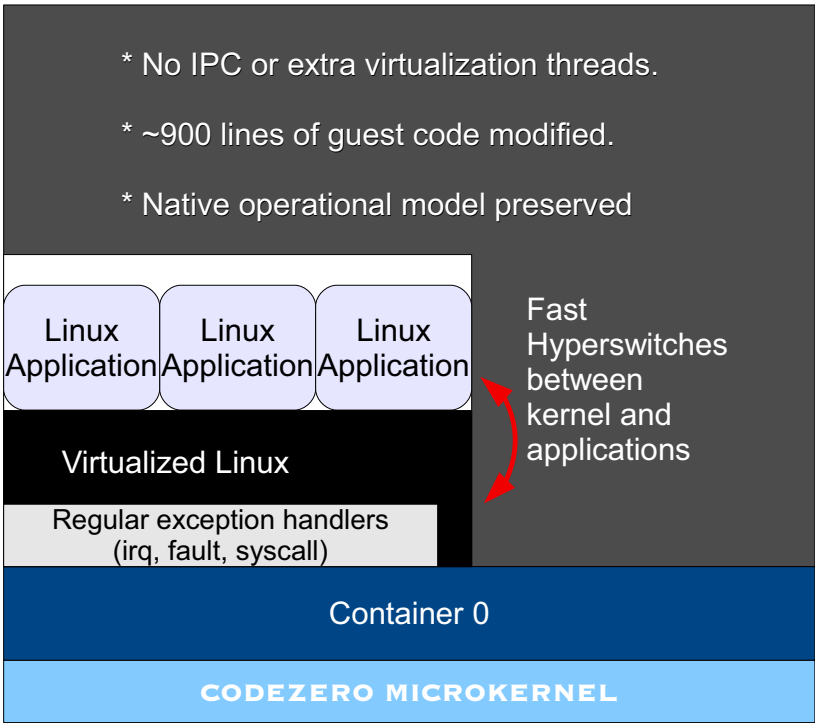


Figure 7: Codezero Hyperswitch Paravirtualization

In HyperSwitch methodology, the existing method of cpu exception handling is preserved. In the same way that a Linux application thread switches between its user mode and kernel mode, Codezero enables a user application thread to switch to its kernel context in user mode and continue its execution there.

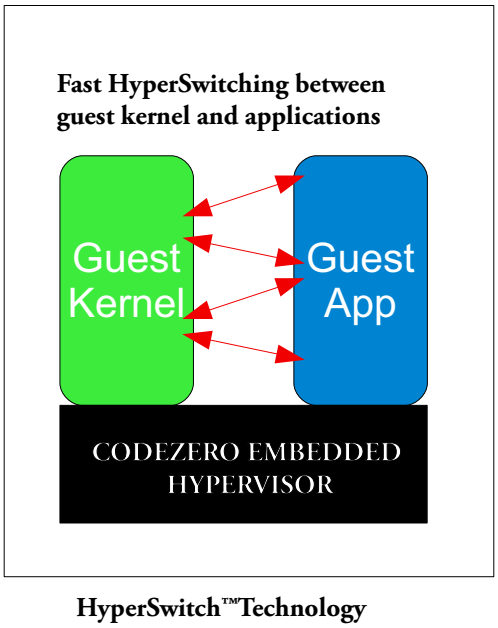


Figure 8: Fast HyperSwitches between guest kernel and applications

Harnessing the HyperSwitch methodology, Codezero preserves the existing infrastructure built inside the guest kernel for handling privileged services. As a result, guest modification requirements are significantly reduced (approximately 900 lines of arch-specific C code for the 2.6.34 Linux kernel). Furthermore, native kernel and application development practices such as GDB debugging and kernel patching may take place in identical fashion in the guest environment.

2.7 Multicore

Codezero supports multicore SMP and optionally AMP designs. Particularly support for the **ARMv7/Cortex-A9** CPUs have been added and demonstrated on a quad-core SMP design.

Codezero transparently schedules threads onto cores and imposes no restriction on which thread is running on which core. For example, the formerly mentioned concept of containers are not obliged to run on any one core each. This allows for flexible and fully transparent utilization of cores.

It is possible, however, that certain applications are tied to certain cores, for performance and real-time purposes. For example, IRQ handling threads may be tied to one core this way. Also, the CPU timeslice of each thread may be adjusted, if needed.

3 Project layout

3.1 Software overview

The Codezero runtime consists of two main components: the microkernel and containers.

The microkernel is the only privileged component in the Codezero runtime. It is considered as the single point of trust, namely the trusted computing base. It exposes a minimal system call API to userspace containers. It is also responsible for security checking of resource accesses issued by each container, through the use of its system call API.

Containers are independent userspace projects that may consist of a single baremetal application, or a full-blown virtualized operating system with many applications. There may be any number of containers in a Codezero build, and each container may be developed independent of others.

3.2 Build system overview

The build system consists of the **SCons** build tool and a set of Python scripts. **SCons** is a Python-based build tool that has similar functionality to **Make**. The main difference is that build rules are written using Python functions.

All container-build scripts are located under the scripts/ directory. The microkernel is built by the top-level SConstruct file and with help of many SConscript files spread across the kernel build directories. The build system may be invoked via the `./build.py` script at the top-level directory.

3.3 Configuration system and CML2

The Codezero project is configured using the Python-based **CML2** kernel configuration system. CML2 allows a menu-based configuration of build parameters before building. Typically, configuration data produced by CML2 are used for configuring the microkernel, as well as kernel-related parameters of each container included in the build. In essence, the CML2 configuration is used for a complete userspace and microkernel build.

The source code for CML2 itself lies under the `tools/cml2-tools` directory. Codezero project configuration data and scripts reside under **scripts/config** directory.

CML2 uses configuration files with a .cml extension for configuration. Various predefined configuration files reside under the `scripts/config/examples/cml` directory.

For reference to CML2 syntax and internals, please refer to the CML2 Language Reference Manual.

3.4 Directory layout

loader

The loader directory contains sources for an ELF library and the boot loader. The boot loader is a simple ELF program that is responsible for scatter-loading the kernel and container images.

The loader is the final project built during a Codezero system build. After all containers and the kernel are built, the loader project bundles all resulting ELF files under a single `final.elf` image, which also contains the ELF loader program.

The loader/libs directory contains two libraries called `C` and `elf`. The `elf` library is used for loading the ELF files during boot. The `C` library is a partial C library that is used by the bootloader during boot.

tools

This directory contains various tools and scripts that are used as part of the project. The `tools/tagsgen` directory contains scripts to produce ctags and cscope index data for the microkernel and any other project included in the build. For instance, `./tools/tagsgen/tagsgen_kernel.py` may be used to create a kernel index. Similarly, `./tools/tagsgen/tagsgen_linux.py` may be used for creating an index for the Linux kernel. These scripts assume an existing architecture and platform configuration in order to filter out only the relevant architecture and platform directories for an index generation. `tools/pyelf` is a host-side Python library for inspecting ELF elements of a file inside the host. For instance, it is possible to extract fields from ELF headers or make calculations using this library. The `tools/run-qemu-insight` script is an autogenerated script that allows the `final.elf` image to be instantiated on QEMU and Insight/GDB after a build.

docs

This directory includes various self-contained documents about peculiarities of the Codezero runtime. The `docs/man` directory includes reference manual pages for each Codezero microkernel system call. They may be invoked by:
`% man -M docs/man <man_page_name>`.

In the future, more ad-hoc documentation will be added here.

src, include

These directories contain the main sources for the Codezero microkernel.

conts

The `conts` directory contains all the userspace sources.

conts/baremetal

Contains example userspace servers and applications For example, `Hello World` and a template for user space development.

conts/linux

This directory is used for building Virtualized Linux on top of Codezero. It does not exist in original Codezero sources but should be created if Linux virtualization functionality is going to be used. The typical instantiation of this directory includes copying Linux environment files from a separate repository, which includes the Linux filesystem and boot parameters. Finally, the virtualized Linux kernel should be copied or cloned from a separate kernel repository into this directory. The contents of this directory must be set up in order to initiate a Linux build via the Codezero build system.

conts/uboot

This directory is used for building virtualized u-boot instances on top of Codezero. It does not exist in original Codezero sources, but should be created if u-boot functionality is going to be used at runtime. The typical instantiation of this directory includes copying or Git-cloning virtualized u-boot sources from a separate repository into this directory, before initiating a u-boot build via the Codezero build system.

conts/userlibs

This directory contains all userspace libraries that are available for any userspace application. These libraries consist of `libl4`, `libc`, `libmem`, and `libdev`.

conts/userlibs/libl4

`libl4` is the most fundamental library provided for userspace. This library presents raw Codezero system calls in a more user-friendly format. It is appropriate to link with this library from every project, including virtualized operating system kernels. It is recommended that system calls to Codezero are issued via this library.

conts/userlibs/libc

`libc` is the userspace C runtime library. It is particularly useful for referring to C library functionality that does not require an operating system. For example, string and basic IO functionality is included.

conts/userlibs/libdev

`libdev` is a helper library for clcd, timer, and UART devices for different platforms. Device services that reside under conts/baremetal make use of this library.

conts/userlibs/libmem

`libmem` contains three different memory allocators. A fixed-size *memory cache* implementation, a *kmalloc* implementation, and a *page allocator* implementation for page-size granularity memory allocation. Out of the three, memcache is the most frequently used because it provides a simple and deterministic method of memory allocation.

conts/posix

The POSIX sources consist of a partially complete POSIX implementation on top of Codezero. Advanced capabilities such as demand paging, memory, and process management are provided. As of this writing, this is an experimental container for investigating advanced pager behavior on top of the microkernel.

4 Debugging and runtime

4.1 Codezero debugging overview

As described in the Getting started with Codezero development guide, the standard debugging method on Codezero is based on a Insight/GDB debugger connected to a QEMU emulator instance. This setup provides an experience that is very close to an In-Circuit-Emulator debugging of a real hardware platform.

Using Insight, any debug capability may be leveraged such as placing breakpoints, creating watchpoints, inspecting local variables, registers, function call stack, and memory.

Emulation environment can be easily set up on any Linux-hosted PC, and debug-fix-reboot cycle is several times faster than a real platform. Therefore, emulation environment is recommended for general software development. It is also recommended that the QEMU instance used for emulation is backed by a real hardware environment, and that software is checked for real hardware boot up on regular intervals.

For the purpose of debugging Codezero, the emulation tools have been optimized to provide enhanced debugging capabilities. QEMU has been modified to support the Versatile Express quad-core Cortex-A9 platform. Virtualized Linux kernel environment has been optimized so that Linux boots can be instantiated on the debugger within a few seconds.

Before starting with debugging and development of source code, it is recommended that below sections are read for an understanding of how individual components of a typical Codezero build are laid out in the final executable image, physical memory, and virtual memory spaces.

4.2 Codezero boot image layout

The typical Codezero build involves creation of multiple ELF images that are bundled together in a cascaded fashion. All images are standard ELF files that can be manipulated and inspected using standard GNU ELF tools such as `readelf` or `objdump`.

As formerly mentioned, the typical Codezero runtime consists of the microkernel and multiple containers. After all these components are built, the ELF loader build is initiated. ELF loader is built into the `final.elf` image, with every container image and the kernel image embedded inside. At runtime, the loader would traverse its own sections to find each ELF image and load them to their respective physical memory addresses.

Below is a summary of how images are cascaded inside final.elf:

```
final.elf:
   Section cont.0  (Container0.elf)
   Section cont.1  (Container1.elf)
   ...
   Section cont.X  (ContainerX.elf)
   Section .kernel (Kernel.elf)
```

where each `containerX.elf` image is further cascaded as follows:

```
container0.elf:
   Section img.0 (image0.elf)
   Section img.1 (image_XYZ.elf)
   ...
   Section img.X (imageX.elf)
```

When building each container project, note that the builder picks up **any** file with a .elf extension from the project directories and places them in a `.img.X` section for that container. This allows support for containers with multiple ELF images.

During boot-time, the ELF loader unpacks each `.cont.X` section, and further unpacks and loads the ELF files contained in `.img.X` sections. Finally, it loads the `kernel.elf` from the .kernel section. It is essential that every image has been linked to be loaded at a nonclashing physical address with respect to other images. The build system automatically ensures this during the configuration phase.

Future enhancements

In the future, support will be added for optionally loading images at runtime via the network or flash.

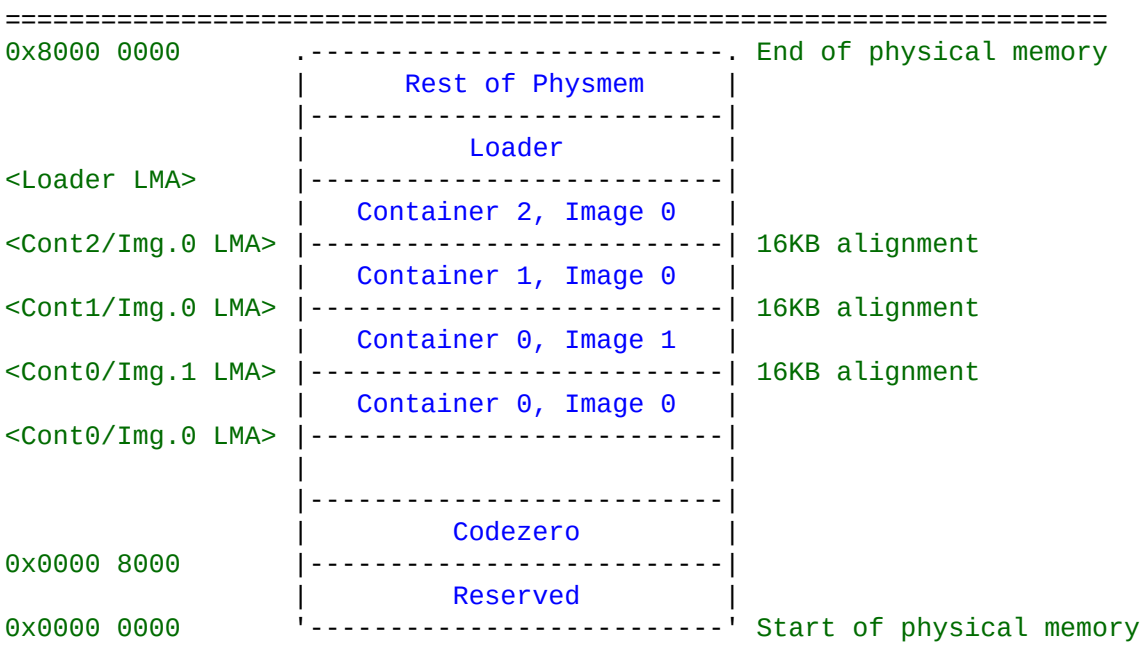
Physical memory layout

During boot up, the ELF loader placed into the `final.elf` image starts executing. It loads each image according to its defined physical memory locations.

A container image may be loaded at any possible physical memory address during load time. The load information is is obtained from its LMA (logical memory address) value defined in its linker script. It is essential that this LMA matches with any one of the physical memory regions (**PHYSMEM** capabilities) defined for the corresponding container. Otherwise, the kernel would refuse to execute the image. It is also worth noting that load addresses of images must not overlap. Generally, parameters such as load address and physical memory regions are defined during kernel container configuration. The configuration process automatically ensures that the setup is sane.

Even though the location of container images in physical memory is highly configurable, below is an example setup for physical memory layout of images:

Table 1: Physical memory layout on Codezero/ARM Versatile PB926



In the example above, there are three containers defined where **container 0** has two images, and **container 1** and **container 2** have one image each. It is expected that each LMA address is within the physical memory regions defined for its respective container. It is also worth noting that the loader image itself is placed at a higher physical memory address that is calculated during build so that it is placed at an address that will not overlap with any one of the container images.

4.3 Virtual memory layout

After all images are loaded to their physical memory regions, the ELF loader starts running the Codezero microkernel. Codezero enables virtual memory and starts running at the fixed virtual address of `0xF0000000`.

After kernel initialization, knowing LMA (logical memory address), VMA (virtual memory address), and total image size for each pager, the microkernel maps the pager for each container to its runtime virtual address. Similar to a pager's LMA, the VMA value must lie inside one of the virtual memory capabilities defined for the pager. Otherwise, this would be a permission violation, and Codezero would refuse to run the pager for that container.

Unlike the physical LMA regions, it is possible that a virtual memory region may be allocated to more than one pager. Since each pager has a different set of page tables, it may be a valid setup that they run in the same virtual memory address. This is discouraged, however, as there may be implications with regard to caching, in case any two such pager wishes to communicate via shared memory or IPC.

Even though the virtual memory layout of a Codezero runtime is highly dependent on the configuration, below is a fictional example for the virtual memory layout of containers that were listed earlier:

Table 2: Runtime virtual-memory layout of a Codezero/ARM system:

```
=========================================================
0xFFFF FFFF    .----------------. End of virtual memory
               |  Syscall page  |
0xFFFF F000    |----------------|
               |    Reserved    |
0xFFFF 1000    |----------------|
               |   Vector page  |
0xFFFF 0000    |----------------|
               |    Reserved    |
0xFF00 1000    |----------------|
               |       KIP      |
0xFF00 0000    |----------------|
               |    Kernel IO   |
0xF900 0000    |----------------| UTCB area ends (optional)
               |       ...      |
               |    ---------   |
               |    UTCB page   |
               |    ---------   |
               |    UTCB page   |
0xF800 0000    |----------------| UTCB area starts (optional)
               |    Codezero    |
               |   Microkernel  |
0xF000 0000    |----------------| Container 2 Task/Pager area ends
               |   Container 2  |
               |   Pager/Tasks  |
CONT2_VIRTMEM0 |----------------| Container 2 Task/Pager area starts
               |   Container 1  |
               |      Pager     |
CONT1_VIRTMEM1 |----------------| Container 1 User task area ends
               |   Container 1  |
               |      Task      |
               | Address Space  |
CONT1_VIRTMEM0 |----------------| Container 0 Pager area ends
               |   Container 0  |
               |      Pager     |
```

```
CONT0_VIRTMEM1 |----------------| Container 0 User task area ends
               |  Container 0   |
               |      Task      |
               | Address Space  |
CONT0_VIRTMEM0 |----------------| Container 0 User task area starts
               |     Empty      |
0x0000 8000    |----------------|
               |    Reserved    |
0x0            '----------------' Start of virtual memory
```

As seen from the table above, the containers have a highly configurable virtual runtime environment. Each container may be placed at a different portion of virtual memory. In case the container consists of a virtualized OS kernel and its applications, they may be optionally placed in different or same virtual regions. The configuration of virtual spaces are ultimately determined by the kernel configuration system before the build phase.

In the ARM architecture, there are a few fixed virtual address regions defined. The system call page is placed at the highest virtual memory page. The exception vectors are placed at the architecture-defined high vector address of `0xFFFF0000`. The kernel accesses very few devices during its lifetime, such as the timer and UART devices on the platform. To access any such device region with flexibility, a kernel IO region has been defined at `0xF800000`. Finally, the lowest few pages of virtual memory space have been reserved, in order to catch null pointer dereference errors and avoid confusion.

One notable feature of the Codezero environment is the notion of UTCBs. UTCBs denote the per-thread storage spaces that are used during an IPC. For performance and simplicity reasons, it is a strict requirement that any UTCB allocated to a thread does not overlap with another UTCB in the virtual address space. This ensures that the kernel does not need to remap UTCB regions or issue other complex cache maintenance operations during the course of the IPC.

For simplicity reasons, Codezero does not maintain UTCB address regions inside the microkernel. In the above example, an optional UTCB region has been allocated between `0xF8000000–0xF9000000` so that pagers may possess and allocate virtual areas from this region for their children threads or themselves. Note, that this is only a suggested allocation, as any address in the virtual address space may be used for this purpose.

4.4 Debugging tips

This section provides a brief introduction to debugging userspace applications and the Codezero microkernel in general.

After successfully building all containers, the microkernel and the loader final.elf image is ready for running under the build/ directory. In an emulation environment, this can be invoked by:

```
% ./tools/run-qemu-insight
```

In a real hardware environment, final.elf image may be loaded to RAM via a remote debugger that has capabilities to load ELF files, such as RVD.

Inspecting and loading symbols

Any one of container images or the microkernel may be debugged while running the system.

In order to load symbols of the desired debug executable, the relevant symbols must be loaded. To debug the microkernel as an example, the following GDB command must be executed in the GDB console:

```
% sym /opt/codezero/build/kernel.elf
```

In order to debug a baremetal application, its symbols must be loaded:

```
% sym /opt/codezero/build/cont0/baremetal_app/app.elf
```

In order to debug the virtualized Linux kernel, Linux symbols must be loaded:

```
% sym /opt/codezero/build/cont0/linux0/kernel-2.6.34/vmlinux
```

The above commands may be placed into the `*.gdbinit` file in the current user home directory. This would allow the user to start debugging at a particular state of execution in automated fashion. Using such a debug script may significantly speed up startup times of a debug session. As an example, the following .gdbinit file is used for debugging the Linux kernel:

```
target remote localhost:1234
load final.elf
sym final.elf
break break_virtual
continue
sym kernel.elf
stepi
set $var = 0xa0008000
break *$var
continue
sym cont0/linux/kernel-2.6.34/vmlinux
stepi
```

where the initial `head.S` entry point of the Linux kernel is located at virtual address `0xa0008000` in this occasion.

Useful debugging tip

A notable point to keep in mind here is that the debugger catches breakpoints on an executable by matching the symbol addresses loaded to the real object code executing at the particular moment. In that respect, there may be ambiguities as to which executable is being debugged if two executables are running at the same virtual address range.

As an example, if two applications are running at virtual address `0×1000 0000` and a breakpoint is set at an address in one of them, the debugger may naturally fall into the fallacy of breaking on the other task, if it hits the same address. On disjoint virtual memory addresses (e.g., the microkernel virtual address range) no problem occurs.

Another useful debugging tip

A frequently needed operation is to set a breakpoint at an address where there is no symbol defined. GDB does not allow setting a breakpoint at a raw address this way. As a workaround, the following GDB convention helps:

```
% set $var = [hex address to break]
% break *$var
```

Codezero API reference

This reference guide provides an insight to the Codezero Microkernel's raw system call interface and system-related data structures. In practice, system calls and system-related data structures are managed via a thin userspace library called `libl4`.

As a result, the semantics of each system call is only briefly described here. Each system call is presented with accompanying `libl4` convenience functions in order to quicken the learning process. If you have any questions on the specifics of each system call, please ask on our mailing list.

Codezero system call interface

Codezero Embedded Hypervisor provides 12 API calls with purposefully minimized function signatures to provide a unified management interface of the underlying hardware with a greatly simplified approach. Using 12 generic API calls, one can virtualize complete operating systems such as Linux, create a multitasking environment, create communication between software components, control caches, page tables and manage multiple threads running on multiple cores.

L4_IPC

Name

l4_ipc - Interprocess communication

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_ipc(l4id_t to, l4id_t from, unsigned int flags)
```

Description

The **l4_ipc()** call provides the main interprocess communication mechanism on Codezero. Upon an IPC call between two threads, message registers that are stored in one thread's UTCB block are transferred to the other thread's UTCB block, in synchronous fashion. The amount and nature of the data transferred depends on the type of IPC call. See below for details.

The UTCB is a memory block that is private to each thread's address space and its purpose is to provide thread-local storage to each thread. See UTCB for more details.

The *to* and *from* fields jointly determine whether the IPC is going to be a **send** or a **receive.** A valid thread id in the *to* field and an **L4_NILTHREAD** value in the *from* field denotes the IPC is a **send**, and when the fields are vice versa, it denotes the IPC is a **receive**.

If both *from* and *to* fields have valid thread IDs, the IPC becomes a joint **send** and **receive**, i.e., a **send** operation is followed by a **receive** operation during a single call. This particular variant is useful for calling server-like services, where the request would be received, processed, and a result reply would be sent back.

There is also an **L4_ANYTHREAD** parameter that may be used in the *from* field. This parameter allows a receiver to receive from any thread in the system. This notion is useful for implementing server-like services, where any sender may be accepted and served on a first-come, first-served basis. It is also worth noting that in such a scenario, the sender must possess the capabilities to call the receiver. For more information on capabilities, see CAPABILITY.

The *flags* field determines what type of IPC is going to be initiated. In Codezero, there are three types of IPC, defined as shown below:

Short IPC

This is the shortest and simplest form of IPC, where only the **Primary Message Registers** are transferred. Primary message registers are a handful of registers that are resident on the first few slots of the UTCB. During an IPC, these registers may be optionally mapped to real registers on the CPU as an optimization, but this notion is transparent of the API. It is worth noting that the number of these registers may be variable, depending on the configuration and the platform. See the section Message registers below for a detailed explanation of the possible set of roles each register may take in an IPC.

Full IPC

A full IPC transfers the complete set of message registers resident on the UTCB of the thread, including the primary MRs. The total number of these registers may be variable depending on the configuration and the platform, although a good estimate would be the total size of the UTCB minus the size of the thread-local fields defined on the UTCB.

Extended IPC

This is a special type of IPC where the UTCB is not involved at all. It is the most flexible type of IPC provided by Codezero. On an extended IPC, *any* buffer in a thread's address space may be transferred to *any* buffer on another thread's address space. Page faults are also allowed to occur on the buffers, provided that a third-party thread not involved in the IPC is ready to handle the faults. An extended IPC also defines a variable size for the transfer, and transfers may be of up to 2KB in size. An extended IPC is SMP-safe and brings no negative timing burden on system in overall, although the parties involved in the IPC may block for an unpredictable amount of time due to the possibility of page faults.

It is recommended that for transfers larger than 2KB in size, shared memory is used.

Message registers

On **short** and **full** IPC, all primary message registers are available for transferring raw data. Particularly, the kernel imposes no restriction on these registers, and has no interference with their contents. However, almost always some of these registers may be used for designated purposes in userspace. Some of them are listed below:

MR_TAG

The tag register allows the caller to define a label for the IPC. It defines the primary reason for an IPC call.

MR_SENDER

The sender register is only meaningful for a thread that has done a receive on any thread, via the **L4_ANTHREAD** special thread ID. Also, this is the only exception where the kernel interferes with the contents of a message register, such that the kernel indicates the thread ID of the sender to the caller. This exception must be made, since otherwise a sender could disguise its identity by placing arbitrary values into one of the MRs that the receiver expects to find the sender information.

MR_RETURN

This register is used for sending back a result when handling a joint **send/receive** request. Since it is only relevant on the return phase of an IPC call, it may well be the same register that was designated for the receive phase of a call.

The above registers may also be called as **system registers**. Note that they are solely defined by the **LIBL4** library, and their designated labels have no meaning outside of the userspace context. Since they are used by their label on most IPC calls, further registers have been defined to identify a set that lies outside the above set. These are the system call argument registers, and they are labeled as **L4SYS_ARG0**, **L4SYS_ARG1**, **L4SYS_ARG2**, and **L4SYS_ARG3**, respectively.

L4 userspace library

```
/* Short IPC calls */
static inline int l4_send(l4id_t to, unsigned int tag);
static inline int l4_receive(l4id_t from);
static inline int l4_sendrecv(l4id_t to, l4id_t from, unsigned int tag);

/* Full IPC calls */
static inline int l4_send_full(l4id_t to, unsigned int tag);
static inline int l4_receive_full(l4id_t from);
static inline int l4_sendrecv_full(l4id_t to, l4id_t from, unsigned int
tag);

/* Extended IPC calls */
static inline int l4_send_extended(l4id_t to, unsigned int tag, unsigned
int size, void *buf);
static inline int l4_receive_extended(l4id_t from, unsigned int size, void
*buf);
```

L4_MAP

Name

```
l4_map - Set up a range of virtual-to-physical address mappings.
```

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_map(void *p, void *v, u32 npages, u32 flags, l4id_t tid)
```

Description

l4_map() sets up a virtual-to-physical mapping for the thread identified by *tid*, from virtual address *v* to physical address *p*, spanning for the number of pages defined in *npages* and using the map flags provided in *flags*.

A thread that calls this function must possess capabilities associated with the physical and virtual memory ranges to be mapped. In particular, the range and access permissions for one or more physical and one or more virtual memory capabilities must match the request.

Due to the hierarchical nature of page table organization on some CPU architectures, an **l4_map()** request may result in a middle-level page-table expenditure. The caller is also expected to have enough capability resources, namely **mappool** capabilities, to cover these cases. All formerly mentioned capability parameters may be configured at kernel configuration time. For more information see CAPABILITY.

The *flags* field determines different mapping permissions and caching behavior, as defined below:

MAP_USR_RW

Map with read/write permissions for userspace as cacheable/bufferable.

MAP_USR_RWX

Map with read/write/exec permissions for userspace as cacheable/bufferable.

MAP_USR_RX

Map with read/exec permissions for userspace as cacheable/bufferable.

MAP_USR_RO

Map with read-only permissions for userspace as cacheable/bufferable.

MAP_USR_IO

Map with read/write permissions for userspace as noncacheable/nonbufferable.

MAP_USR_DEFAULT

An alias for MAP_USR_RW (See above). This is the default for userspace mappings.

MAP_IO_DEFAULT

An alias for MAP_USR_IO (See above). This is the default for userspace device mappings.

L4 userspace library

```
/*
 * Helper that maps given physical address to a virtual address
 * in the current task spanning npages in size.
 * Returns mapped virtual address on success, negative value on error.
 */
static inline void *l4_map_helper(void *phys, int npages);
```

Return value

l4_map() returns 0 on success, and negative value on failure. See below for error codes.

Errors

-ESRCH

The thread to make the mapping could not be found on the system.

-ENOCAP

Capabilities required don't exist or do not have sufficient privileges.

See also

CAPABILITY

L4_UNMAP

Name

l4_unmap - Removes virtual-to-physical address mappings.

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_unmap(void *virtual, unsigned long npages, l4id_t tid)
```

Description

l4_unmap() removes a virtual-to-physical mapping from the address space of the thread identified by *tid* starting from virtual address *virtual* and spanning *npages* number of pages.

L4 userspace library

```
/*
 * Unmaps the given virtual address from current address space
 * spanning npages number of pages. Returns a negative value
 * if the address was already unmapped.
 */
static inline void *l4_unmap_helper(void *virt, int npages);
```

Return value

l4_unmap() returns 0 on success and negative value on failure. See below for error codes.

Errors

-ESRCH

The thread to remove the mapping could not be found on the system.

-ENOCAP

Capabilities required don't exist or caller does not have sufficient privileges.

-ENOMAP

There is already an unmapped area in one of the pages that were unmapped. This may or may not be an error, depending on what state the caller expects the unmapped area to be in.

See also

L4_MAP

L4_THREAD_CONTROL

Name

l4_thread_control - Create, destroy, suspend, resume, recycle, and wait on threads.

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_thread_control(unsigned int action, struct task_ids *ids)
```

Description

The **l4_thread_control()** system call manipulates threads in the system. Pagers may create, destroy, recycle, suspend, and resume threads via this call. While the Codezero microkernel aims to provide dynamic privilege and resource management in the form of capabilities, this system call inherently assumes a hierarchical parent-child relationship between the caller and the target thread, such that the caller should be the pager of the targeted thread. See the Thread relationships subsection below for a detailed explanation of the matter.

The *ids* field specifies the thread, address space, and thread group IDs of the targeted thread. Below is the declaration for this structure:

```
struct task_ids {
  int tid;   /* Fully qualified thread ID */
  int spid;  /* Address space ID (local to container) */
  int tgid;  /* Thread group ID (local, defined by userspace protocol) */
};
```

The *tid* argument may have different meanings for different thread control actions. For an existing thread, this argument specifies the thread on which the action is to be performed. On a thread creation request, this argument would specify the thread whose context is to be copied from for creating the new thread. See *actions* below for a more detailed explanation.

The *spid* field has meaning only on a **THREAD_CREATE** request, in conjunction with one of **TC_SHARE_SPACE**, **TC_NEW_SPACE**, or **TC_COPY_SPACE** flags.

The *tgid* field is provided as an extra ID slot for the thread. The pager of the thread may designate a group of threads to be in the same thread group, defining the group by a userspace protocol. This field has no meaning from the kernel's perspective, and may be removed in future releases.

The *action* field is the main action specifier, where one of the following may be supplied as valid actions:

THREAD_CREATE

Creates a new thread in a new or existing space, depending on the provided space flags. A thread create request requires valid *tid* and *spid* fields in order to specify which thread con-

text and address space to copy from or use. Following are the action flags that are associated with a **THREAD_CREATE** request.

Note: these must be bitwise OR'ed with the **THREAD_CREATE** flag:

TC_SHARE_UTCB

Sets the new thread's UTCB as the creator's UTCB. Threads may validly share UTCBs by making sure that they don't initiate the IPC at the same time or ensure synchronized access to UTCB fields.

TC_SHARE_GROUP

Sets the new thread's thread group ID as the ID specified by the *tgid field.*

TC_SHARE_SPACE

Places the new thread into the address space specified by *spid* and copies the thread context from thread specified by *tid* field. The thread represented by the *tid* argument is said to represent a **parent** relationship to the newly created thread.

TC_COPY_SPACE

Copies all page tables of the address space specified by *spid* to the new thread's newly created address space. Also copies the thread context from thread specified by the *tid* field, which is said to represent a *parent* relationship to the newly created thread. This flag is particularly useful for implementing the **POSIX fork()** system call.

TC_NEW_SPACE

Creates the new thread in a brand new address space.

THREAD_DESTROY

Destroys a thread and its address space if the thread destroyed is the only thread left in that address space.

THREAD_SUSPEND

Suspends execution of a thread. The thread goes into a dormant state.

THREAD_RUN

Runs or resumes execution of a thread.

THREAD_RECYCLE

Clears all existing state of a thread but does not deallocate the thread, leaving it dormant. The only information retained is the existing thread IDs of the original thread. This is particularly useful for implementing the execve() POSIX system call.

THREAD_WAIT

Waits on a thread to exit with exit status.

On a system setup where a pager is responsible for creating threads in separate address spaces and communicating with them via an IPC, the children may send an exit IPC message

to their pager. This way, a pager may synchronously receive exit status of a child in the form of an IPC and take action to destroy it as part of handling the IPC. However, on systems where the application is a multithreaded, single address-space application, a thread wait call provides a simple synchronous channel for the parent to wait on its child's exit status, without requiring any extra set up for IPC handling.

See L4_GETID for more details on resource IDs in Codezero.

Thread relationships

Codezero aims to provide fine-grain privilege levels to threads in the system in the form of capabilities. Capabilities enable privileges of threads over each other to become highly configurable, resulting in the hierarchical relationship between them to become blurry. However, even though such a relationship is not enforced by the architecture, often it comes natural that threads are created by other threads. As a result, a thread hierarchy exists and each thread has a *pager* field inside the kernel, to denote the relationship that a thread has create or destroy rights on another thread.

Future

In the future, it may be possible to blur the hierarchical relationship by introducing new flags and create a flat hierarchy. E.g., during thread creation a **TC_AS_PAGER** flag would create a new thread that is self-paging and independent of its creator. Also, a **TC_SHARE_PAGER** flag would allow creation of threads whose parents are the same as the caller's parents. Currently, there is no potential use case for these flags.

L4 userspace library

N/A

Return value

l4_thread_control() Returns 0 on success, and negative value on failure. See below for error codes.

Errors

-EINVAL

Returned when the *req* field has an invalid value.

-ENOCAP

Returned when capabilities required don't exist or do not have sufficient privileges.

-EFAULT

Returned when the *ids* argument would cause a page fault.

-ESRCH

Returned when a given thread ID has not been found in the container.

See also

CAPABILITY,L4_EXCHANGE_REGISTERS, L4_GETID

L4_EXCHANGE_REGISTERS

Name

l4_exchange_registers - Modifies the context of a suspended thread.

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>
int l4_exchange_registers(void *exregs_struct, l4id_t tid);
```

Description

l4_exchange_registers() Reads and modifies the context of a suspended thread.

By this call, pagers can read and modify any register or other crucial information about a thread, such as the pager id or utcb virtual address. An architecture specific **struct** *exregs_data* is passed to the kernel for modifying the targeted thread's context. See below for a detailed description of this structure and the default context structure for the ARM architecture.

```
/* Exchange registers context structure for the ARM architecture */
typedef struct arm_exregs_context {
  u32 r0;           /* 0x4 */
  u32 r1;           /* 0x8 */
  u32 r2;           /* 0xC */
  u32 r3;           /* 0x10 */
  u32 r4;           /* 0x14 */
  u32 r5;           /* 0x18 */
  u32 r6;           /* 0x1C */
  u32 r7;           /* 0x20 */
  u32 r8;           /* 0x24 */
  u32 r9;           /* 0x28 */
  u32 r10;          /* 0x2C */
  u32 r11;          /* 0x30 */
  u32 r12;          /* 0x34 */
  u32 sp;           /* 0x38 */
  u32 lr;           /* 0x3C */
  u32 pc;           /* 0x40 */
} __attribute__((__packed__)) exregs_context_t;
/*
 * Generic structure passed by userspace pagers
 * for exchanging registers
 */
struct exregs_data {
  exregs_context_t context;
  u32 valid_vect;
  u32 flags;
  l4id_t pagerid;
  unsigned long utcb_address;
};
```

Each bit in the *valid_vect* field determines which register offsets are going to be modified.

The *flags* field determines whether to set or read the pager values provided and affects the *pagerid* and *utcb_address* fields of the thread:

EXREGS_SET_PAGER

This flag is available as read-only. When it is set together with **EXREGS_READ**, pagerid value of the targeted thread is read back.

EXREGS_SET_UTCB

Sets or reads the utcb virtual address of the targeted thread.

EXREGS_READ

Enables read operation; all set fields are read back from the targeted thread's context instead of modifying it.

L4 userspace library

```
/*
 * Exchange register library calls to modify program counter,
 * stack, pager ID, UTCB address, and any hardware register.
 */
void exregs_set_stack(struct exregs_data *s, unsigned long sp);
void exregs_set_mr(struct exregs_data *s, int offset, unsigned long val);
void exregs_set_pc(struct exregs_data *s, unsigned long pc);
void exregs_set_pager(struct exregs_data *s, l4id_t pagerid);
void exregs_set_utcb(struct exregs_data *s ,unsigned long virt );
void exregs_set_read(struct exregs_data *s );
```

Above functions may be used for convenient manipulation of the **struct** *exregs_data* structure.

Return value

l4_exchange_registers() Returns 0 on success and negative value on failure. See below for error codes.

Errors

-ESRCH

Target thread was not found in the system.

-ENOCAP

Capabilities required don't exist or do not have sufficient privileges.

-EACTIVE

Target thread has not suspended yet.

-EAGAIN

Target thread is busy holding a mutex.

See also

L4_THREAD_CONTROL

L4_GETID

Name

l4_getid - Returns thread ID, thread group ID, and space ID of a thread.

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_getid(struct task_ids *ids);
```

Description

l4_getid() returns thread ID, thread group ID, and space ID of a thread in a **struct task_ids** structure, as shown below.

```
struct task_ids {
  int tid;   /* Fully qualified thread ID */
  int spid;  /* Address space ID (local to container) */
  int tgid;  /* Thread group ID (local, defined by userspace protocol) */
};
```

Every thread in the system has a thread ID, space ID, and a thread group ID associated with it. Each thread and space ID is globally unique across the system. Thread group IDs are available for grouping threads in arbitrary groups, via a user-defined protocol. A newly created thread may join an existing thread group or create a new group. This behavior is defined by the thread's pager. Such a user-defined thread group allocation protocol may be useful for implementing groups of threads by higher-level OS services.

Even though both thread and space IDs are globally unique across the system, there is an addressability difference between them. Each thread ID is a fully qualified ID, carrying its container ID information with it. Upon a system call that targets a thread ID, the system allows addressing threads that reside in other containers. In contrast, space IDs are not fully qualified. They are local to a container, and any system call addressing a space ID cannot target a space in another container. For thread IDs, the **__cid(tid)** macro extracts the container ID information from the fully qualified thread ID, whereas the **__raw_tid(tid)** macro provides the raw thread ID, which omits the container ID information from the thread ID. Such a raw ID still uniquely identifies the thread across containers, i.e., there is one such raw ID per thread across the system.

Future

The **l4_getid()** call is currently not subject to capability checking, as every thread has a natural right to discover their IDs. In the future, it is possible that this system call is used for naming discovery for other addressable entities. If such a role is given to this call, it may also become subject to capability checking, as access control would prove beneficial over naming discovery services.

L4 userspace library

```
#include <l4lib/arch/syslib.h>

/*
 * Returns thread ID of current thread
 */
static inline l4id_t self_tid(void)
```

Return value

l4_getid() always succeeds with a return value of 0.

L4_THREAD_SWITCH

Name

```
l4_context_switch - Yields the CPU
```

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_context_switch(u32 dest, u32 flags, s32 retval);
```

Description

l4_context_switch() yields the CPU. The current thread yields the CPU donating its left timeslice to thread with id specified in the dest field.

The **flags** field contains various behavior modifiers for this call as shown below.

retval the value to be returned to the *dest* thread.

CSWITCH_STOP

This flag stops the caller thread temporarily until another thread resumes it. Note that a thread stopped this way is not eligible for destruction as it may be still sleeping in the kernel. This state rather guarantees that the thread is never going to be scheduled until it is resumed, but the thread's sleep state is not modified (i.e., the thread is not interrupted if it is asleep). For interrupted destruction, a **thread suspend** would be more appropriate. This may be issued via the **l4_thread_control** system call.

CSWITCH_PREV_CONTEXT

Return to previous context.

CSWITCH_SPACE_SWITCH

Space switch request. Switch to space with id **dest.**

CSWITCH_DOMAIN_SWITCH

Domain switch request. Return to USER domain.

CSWITCH_STOP_EXIT

Stop the thread and put it in suspended mode.

Return value

l4_context_switch() Returns 0 on success, or negative value on failure. See below for error codes.

ERRORS

-ESRCH returned when a given thread id has not been found in the container.

SEE ALSO

l4_thread_control(7)

L4_TIME

Name

`l4_time - Sets or reads system time.`

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_time(struct timeval *timeval, int set);
```

Description

l4_time() sets or reads system time.

The *timeval* field is the representation of time since the system has started running. See below for details.

The *set* field instructs whether the time is to be read or set to the values specified in the *timeval* field supplied.

```
struct timeval {
  int tv_sec;    /* Time value in seconds */
  int tv_usec;   /* Time value in microseconds */
};
```

Limitations

Currently, setting the time feature is disabled.

Currently, the system time is returned since the system has started rather than a set date.

Currently, this system call is not subject to capabilities.

Errors

-EFAULT

The *timeval* field would cause an unhandled page fault.

-EBUSY

Timer structures in the system are currently being used.

-ENOSYS

Operation of given type is not supported.

L4_MUTEX_CONTROL

Name

`l4_mutex_control - Userspace locking and synchronization for shared memory`

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_mutex_control(void *mutex_word, int op);
```

Description

l4_mutex_control() provides userspace locking and synchronization for shared memory. The *mutex_word* argument contains the virtual address for the lock, and is typically placed on a shared memory segment. It would still be valid if the lock virtual address is different on different address spaces, as the kernel uses the actual physical address for uniquely identifying the lock.

Any virtual address on an address space may be used as a mutex lock, and mutexes are locked and unlocked with no system call assistance if there is no lock contention. If a lock contention occurs, lock holder and lock contender synchronize by the assistance of this system call. In its current state, this system call represents a **binary semaphore** implementation. Note that the userspace threads involved with locking must cooperate, as a noncooperative thread may cause one or more of the threads to block indefinitely on the lock.

As with all other system calls, the success of this call is subject to the caller having sufficient capabilities. See CAPABILITY for more information. On the matter of capabilities, mutex allocation inside the kernel by this call is subject to the caller having the **mutexpool** typed memory pool capability. The individual mutex instances, however, are not first-class objects, as capability tracking of each mutex would put too much memory burden on the system.

The *op* argument defines whether the operation is a **lock** or an **unlock** operation:

L4_MUTEX_LOCK

Locks a contended mutex.

L4_MUTEX_UNLOCK

Unlocks a contended mutex.

The *mutex_word* argument specifies the userspace virtual address for the lock. See below for a userspace representation of the mutex structure.

L4 userspace library

```
struct l4_mutex {
  unsigned int lock;
} __attribute__((aligned(sizeof(int))));
```

```
void l4_mutex_init(struct l4_mutex *m);
int l4_mutex_lock(struct l4_mutex *m);
int l4_mutex_unlock(struct l4_mutex *m);
```

Return value

l4_mutex_control() returns 0 on success and a negative value on failure. See below for error codes.

Errors

-ENOMEM

The caller has no sufficient capability to allocate a mutex structure in the kernel.

-EINVAL

Invalid value in the *op* field.

-ENOCAP

Caller has no capability to make this system call.

See also

CAPABILITY

L4_CAPABILITY_CONTROL

Name

`l4_capability_control - Capability inspection and manipulation`

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_capability_control (unsigned int req, unsigned int flags, void
*buf);
```

Description

l4_capability_control() enables a thread to read and manipulate the list of capabilities that it possesses. Capabilities may be shared, granted to other threads, or they may be replicated, destroyed, reduced in privileges, or split into parts—effectively enabling a dynamically configurable resource management architecture. The thread calling this system call must possess relevant capabilities, as any operation done by this call are also subject to capability checking.

req denotes the type of request. See below for a full list.

flags denotes additional flags for the given request. See below for a list of flags.

buf almost always contains a capability structure that describes the request with regard to given *req* and *flags.*

CAP_CONTROL_NCAPS

Get capability count. This is the sum of thread-private capabilities, address space capabilities, and container capabilities.

CAP_CONTROL_READ

Returns an array of **struct** *capability* structures in *buf.* The number of capabilities in the array should be first obtained by the **CAP_CONTROL_NCAPS** request.

Return value

l4_capability_control() Returns 0 on success, and negative value on failure. See below for error codes.

Errors

-EINVAL

When a capability struct is passed in *buf* but with invalid fields.

-ENOCAP

When capabilities required don't exist or do not have sufficient privileges.

See also

CAPABILITY

L4_CACHE_CONTROL

Name

`l4_cache_control - Cache/TLB manipulation`

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_cache_control (void * start, void * end, unsigned int flags);
```

Description

l4_cache_control() enables a thread to invalidate and clean the cache memory.

start denotes the start address(virtual memory address) of the memory region to be invalidated/cleaned. This is not used in case of armv5.

end denotes the end address(virtual memory address) of the memory region to be invalidated/cleaned. This is not used in case of armv5.

flags denotes the operation to be performed.

L4_INVALIDATE_ICACHE_ENTIRELY

Invalidate/flush the entire instruction cache.

L4_INVALIDATE_ICACHE

Invalidate/flush the instruction cache occupied by region from *start* to *end*.

L4_INVALIDATE_DCACHE

Invalidate/flush the data cache occupied by region from *start* to *end*.

L4_CLEAN_DCACHE

Clean the data cache occupied by region from *start* to *end*.

L4_CLEAN_INVALIDATE_DCACHE

Clean and invalidate the data cache occupied by region from *start* to *end*.

L4_SYNC_CACHES

Clean the data cache to the point of unification and invalidates the icache for the same region.

L4_INVALIDATE_TLB

Invalidate/flush unified TLB occupied by region from *start* to *end*. On a system with separate TLBs, this flushes both of them.

Return value

l4_cache_control() Returns 0 on success and negative value on failure. See below for error codes.

Errors

-EINVAL

When a *flag* is passed with invalid fields.

L4_IRQ_CONTROL

Name

`l4_irq_control - Register/unregister device IRQs.`

Synopsis

```
#include <l4lib/arch/syscalls.h>
#include <l4lib/arch/syslib.h>

int l4_irq_control (unsigned int req, unsigned int flags, l4id_t id);
```

Description

l4_irq_control() enables a thread to register/unregister device IRQs. Caller of this system call has an option to choose between synchronous or asynchronous IRQ handling.

req denotes the type of operation to be performed.

IRQ_CONTROL_REGISTER

Register a specific irq **id** either as threaded or with user handling.

IRQ_CONTROL_RELEASE

Release a specific irq **id** which was registered as threaded or with user handler.

IRQ_CONTROL_WAIT

Wait for irq to happen. This flag is used by the irq handler to block and wait for irq to happen.

IRQ_CONTROL_ENABLE

Enable all irqs on this domain (for user handling guests).

IRQ_CONTROL_DISABLE

Disable all irqs on this domain (for user handling guests).

IRQ_CONTROL_ACK_MASK

Ack the irq with index **id**.

flags denotes the slot number representing the irq handler; this is the identifier that distinguishes various handlers registered for the same device.

IRQ_USER_HANDLER

Flag to indicate irq **id** is not a threaded irq, rather has a userspace handler.

IRQ_WAIT_THREADED

Flag to indicate wait for irq **id** as a thread.

IRQ_WAIT_IDLE

Flag to Wait for any interrupt.

IRQ_ENABLE_GLOBAL

Flag used with IRQ_CONTROL_ENABLE to enable irqs globally on the guest.

id denotes the platform-specific IRQ index of the concerned device.

Return value

l4_irq_control() Returns 0 on success and a negative value on failure. See below for error codes.

Errors

-EINVAL

When the *flags* passed does not have valid information.

-ENOUTCB

In case the thread making the system call does not have a valid UTCB.

-EFAULT

In case the UTCB of the caller thread is not mapped.

-ENOIRQ

In case the device represented by IRQ index **id** is invalid or kernel does not allow user space tasks to avail this device.

Codezero system structures

Codezero Embedded Hypervisor provides two primary system structures for managing system-wide and thread-wide information. A Kernel Interface Page provides global information to upper software layers about the system they are running on and the User Thread Control Block provides a per-thread information for userspace management of threads. Finally a capability structure is used internally by the hypervisor for flexible management of system security. The internals of this structure are also provided in this section.

KIP

Name

Kernel Interface Page (KIP) - Overview of KIP in Codezero

Synopsis

```
#include <l4/api/kip.h>
```

Description

The kernel interface page acts as a read-only identification structure for system-level information. Information about the microkernel version, API, and various fundamental configuration flags may be obtained via this page. Another purpose of this page is to supply system-call addresses to userspace during initialization. Since the system call offsets are discovered at runtime this way, the system reserves the flexibility of changing system call offsets in the future.

KIP also provides the address of the thread-local UTCB address, and the value of this field dynamically changes as each thread becomes runnable. The KIP structure is defined as follows:

```
struct kip {
  /* System descriptions */
  u32 magic;
  u16 version_rsrv;
  u8  api_subversion;
  u8  api_version;
  u32 api_flags;

  /* Addresses of various available system calls */
  u32 container_control;
  u32 time;
  u32 irq_control;
  u32 thread_control;
  u32 ipc_control;
  u32 map;
  u32 ipc;
  u32 capability_control;
  u32 unmap;
  u32 exchange_registers;
  u32 thread_switch;
  u32 schedule;
  u32 getid;
  u32 mutex_control;
  u32 arch_syscall0;
  u32 arch_syscall1;
  u32 arch_syscall2;

  /* UTCB address field */
  u32 utcb;
```

```
  /* Brief description of kernel */
  struct kernel_descriptor kdesc;
};
```

where:

- *magic*—denotes the magic key of this kernel build.
- *version_rsrv*—reserved for future use.
- *api_subversion*—denotes the sub-version of the API in use.
- *api_version*—denotes the version of the API in use.
- *api_flags*—denotes the flags corresponding to this API verison.
- *container_control*—address of the l4_container_control system call.
- *time*—address of the l4_time system call.
- *irq_control*—address of the l4_irq_control system call.
- *thread_control*—address of the l4_thread_control system call.
- *ipc_control*—address of the l4_ipc_control system call.
- *map*—address of the l4_map system call.
- *ipc*—address of the l4_ipc system call.
- *capability_control*—address of the l4_capability_control system call.
- *unmap*—address of the l4_unmap system call.
- *exchange_registers*—address of the l4_exchange_registers system call.
- *thread_switch*—address of the l4_thread_switch system call.
- *schedule*—address of the l4_schedule system call.
- *getid*—address of the l4_getid system call.
- *mutex_control*—address of the l4_mutex_control system call.
- *arch_syscall0*—address of the l4_arch_syscall0 system call, a template for adding new system call to Codezero.
- *arch_syscall1*—address of the l4_arch_syscall1 system call, a template for adding new system call to Codezero.
- *arch_syscall2*—address of the l4_arch_syscall2 system call, a template for adding new system call to Codezero.
- *utcb*—address of the UTCB for this thread.

L4 userspace library

```
/* Functions to get the address of the kernel interface page */
static inline void *
l4_kernel_interface(unsigned int *api_version, unsigned int *api_flags,
unsigned int *kernel_id);
```

UTCB

Name

UTCB - Userspace Thread Control Block

Synopsis

```
#include <l4lib/arch/utcb.h>
```

Description

UTCB is a per-thread data structure designated as thread local storage for IPC message registers and private data. The UTCB area in a thread's address space is a virtual address region that is unique for each thread available on the system. It is discovered at runtime by reading the **Kernel Interface Page** utcb field.

The UTCB stores message registers that are transferred between threads during an IPC. Depending on whether the IPC is a send or a receive, the message register fields are either transferred to other threads, or overwritten by message registers of other threads. For details on IPC behavior, please refer to the **l4_ipc()** system call reference page.

UTCB may also be used for any thread-local information that is private to each thread in an address space. For example on stacked IPCs where a new IPC is initiated before the current IPC has been completed, the *saved_tag* and *saved_sender* fields serve the purpose of saving the unfinished IPC information. For a full description of each field, please refer below.

```
struct utcb {
  u32 mr[MR_TOTAL]; /* MRs that are mapped to real registers */
  u32 saved_tag;        /* Saved tag field for a stacked IPC */
  u32 saved_sender;  /* Saved sender field for a stacked IPC */
  u32 mr_rest[MR_REST];     /* Complete the UTCB to 64 words */
};
```

- mr[MR_TOTAL]
  Primary message registers. On the ARM architecture, there are six of these registers, named as **MR0–MR5**. As an optimization, these registers may be mapped to real registers by the kernel during an IPC. However, this behavior is not warranted by the kernel API.
- saved_tag
  Saved IPC tag field on a stacked IPC.
- saved_sender
  Saved sender ID on a stacked IPC.
- mr_rest[MR_REST]
  Rest of the message registers located on the UTCB. These registers are transferred upon an IPC only if the IPC type is a **full IPC**. See L4_IPC for more details.

UTCB allocation

UTCB address and memory allocation are not maintained by the microkernel. Both UTCB allocation and maintenance are expected to be handled in userspace.

There are two possible scenarios for management of UTCB addresses. A pager that maintains children tasks with separate address spaces would need to implement UTCB management functions to manage UTCBs that reside in multiple address spaces. This would mean that the pager would need to distinguish between address spaces when allocating UTCBs. For example, if a UTCB area has a size that is less than the page size granularity, same threads in the same address space could legitimately be assigned UTCBs on the same page, but threads in different spaces would require their UTCBs to reside on separate pages for protection.

The other scenario is simpler where a pager is a multithreaded application with all of its children residing in the pager's address space. In this case, since all UTCBs reside on a shared address space, the pager may simply use a pool of memory and virtual addresses to manage per-thread UTCB areas.

Note, there are library functions available in *libl4* to manage both of the scenarios above.

UTCB addresses may include any virtual address; however, there is a restriction that the UTCB address region must always lie in a disjoint virtual-memory region. This is required because when an IPC is established, the kernel does a direct copy between UTCB regions, without the need for page-table manipulation. In other words, two threads that will communicate via an IPC must have their UTCBs in different virtual-address areas for the kernel to be able to do direct copying.

The simplest solution to create a UTCB is to simply declare a UTCB size aligned array of UTCB structures statically by the macro below:

```
#define DECLARE_UTCB(name) struct utcb name ALIGN(sizeof(struct utcb))
```

and use it by:

```
DECLARE_UTCB(utcb);
```

While this works as the simplest solution, it usually allocates the UTCB on a spontaneous virtual address in the program address space that is not especially disjoint. As previously mentioned, the kernel requires the statically allocated structure to lie in a virtual-address area that is disjoint from any other UTCB address in the system—or at least disjoint from those UTCBs whose owners would be two possible parties for an IPC call. To achieve this, the above declaration may be placed in such a special section using a section attribute on the declaration.

Pagers may set a thread's UTCB address by the **l4_exchange_registers()** system call. Please see L4_EXCHANGE_REGISTERS for more details.

The UTCB structure may be of variable size and may change by configuration or by different machine architecture. It has been set to a total of 256 bytes on an ARM system. The UTCB structure may be subject to change. New fields may be reserved on the UTCB, as needed.

L4 userspace library

```
/* Functions to read/write UTCB registers */
static inline unsigned int read_mr(int offset);
static inline void write_mr(unsigned int offset, unsigned int val)
```

See also

L4_IPC, L4_EXCHANGE_REGISTERS

CAPABILITY

Name

Capability - Overview of capabilities in Codezero

Synopsis

```
#include <l4/api/capability.h>
#include <l4/generic/cap-types.h>
```

Description

A capability is a unique representation of security qualifiers on a particular resource. Each thread, address space, and container is associated with its own capability list represented by the structure below.

The capability structure is defined as follows:

```
struct capability {
  struct link list;

  /* Capability identifiers */
  l4id_t capid;         /* Unique capability ID */
  l4id_t owner;         /* Capability owner ID */
  l4id_t resid;         /* Targeted resource ID */
  unsigned int type;    /* Capability and target resource type */

  /* Capability permissions */
  u32 access;           /* Permitted operations */

  /* Other Limits/Attributes of the resource */
  unsigned long start;  /* Resource start value */
  unsigned long end;    /* Resource end value */
  unsigned long size;   /* Resource size */
  unsigned long used;   /* Resource used size */
};
```

capid—denotes the unique capability ID.

resid —denotes the unique ID of a targeted resource. The smallest resource targetable by a capability is a thread. There are also collections of targetable resources, such as an address space or a container. An address space target resource implies all threads inside that address space, and a container target resource implies all threads inside that container. Quantitative capabilities, such as typed memory pools, do not possess a target and, therefore, have an invalid resource ID.

owner—denotes the unique ID of the one and only capability owner. This is always a thread ID. The ownership of a capability determines who has the right to practice the capability modification privileges available over the capability, such as sharing, granting, spliting, reducing, or destruction of the capability.

The *type* field contains the capability type or targeted resource type. The capability type determines the generic operations that the capability describes. For example, a capability describing a system call would likely have a type name that resembles the name of that system call. See the Capability types section below, for a list of valid capability types. The resource type denotes the type of targeted resources. In case this is a thread or a collection of threads, the type may be one of thread, address space, or container. Quantitative resources also have different types, describing the resource. Since quantitative resources, such as memory pools or memory regions, are not associated with a target ID, the resource types have no meaning for the resid field for these capabilities. See the Capability resource types section below for the full list of valid resource types.

The *access* field denotes the fine-grain operations available on a particular resource. The meaning of each bitfield differs according to the type of the capability. For example, for a capability type thread_control, the bitfields may mean suspend, resume, create, delete, etc. See below for the full list of capability bits.

Capability types

CAP_TYPE_TCTRL

Defines the capability to make the **l4_thread_control** system call. It is usually owned by the pager and targets the container so that the pager can issue the call on all threads in the container.

CAP_TYPE_EXREGS

Defines the capability to make the **l4_exchange_registers** system call. The *rtype* field is usually expected to target the complete container, similar to **CAP_TYPE_TCTRL** capabilities.

CAP_TYPE_MAP_PHYSMEM

Defines a physical memory region. A thread that owns this capability would be always expected to use the **l4_map** system call, to map the physical area to its address space. As an optimization, there is no separate mapping capability defined for the **l4_map** system call. As a physical memory capability, it may also define fields for devices, such as device types, numbers, and IRQs.

CAP_TYPE_MAP_VIRTMEM

Similar to the **CAP_TYPE_MAP_PHYSMEM** capability, it defines a virtual memory range that can be mapped using the **l4_map** system call.

CAP_TYPE_IPC

This is the most fundamentally used capability in the system. **CAP_TYPE_IPC** defines the ability to make IPC calls to threads in the system. By its *rtype* field, it may be defined such that it enables inter-container IPC, i.e., the ability to send messages to a thread or all threads in another container. IPC operations always have a valid target if it is a **send** operation. By this fact, IPC capability checks are done only during the send phase.

CAP_TYPE_IRQCTRL

Defines a thread's privilege to set up and handle IRQs. A separate **CAP_TYPE_MAP_PHYSMEM** capability would also be necessary on each individual device, to gain access to its IRQs.

CAP_TYPE_UMUTEX

Defines a thread's privilege to use kernel-supported userspace mutexes. A thread that has access to a **mutexpool** would have to have this capability to use it. In future versions, this capability may be removed, assuming the **mutexpool** capability is in itself sufficient for having access to mutexes.

CAP_TYPE_QUANTITY

There are various typed, fixed-size memory pools that have this capability type. Fixed-size memory pools such as **mappool**, **cappool**, **threadpool**, and similar capabilities are some of the examples. A quantitative capability has no valid target resource id type because they are themselves resources to be consumed by their owner. Since they already have a unique capability ID, the target resource id does not provide any better identification. As a result, normally the *resid* field is set to **CAP_RESID_NONE** on quantitative capabilities.

Quantitative capabilities have been introduced for allocation of structures that are fundamentally and minimally needed in the system. Any further abstraction of memory resources would make the design too generic, requiring much effort from userspace. The choice of which mechanisms need to be kept inside and out of the kernel is a subtle one. In this particular case, it was decided that a minimal set of typed resources would be always useful to keep in the kernel.

CAP_TYPE_CAP

Defines the capability to manipulate existing capabilities. Any thread that attempts to share, grant, modify, or replicate its capabilities must make a call to the **l4_capability_control** system call. This capability defines the operations available making this call. A caller must own this capability and also own all other capabilities that are going to be modified.

Capability resource types

Capability resource types define the type of resource id stored in the *resid* field.

CAP_RTYPE_THREAD

Threads are the smallest resource entities in the system, targetable by a system call. A capability with this resource type defines the ability to manipulate a single thread. E.g., **l4_thread_control** or **l4_ipc** syscalls could only operate on the single thread, whose ID is defined by the *resid* field.

CAP_RTYPE_SPACE

Address spaces contain one or more threads. A capability with this resource type may act on any thread inside the defined address space, defined by the *resid* field of the capability.

CAP_RTYPE_CONTAINER

Containers provide the outermost isolation level on the system. A capability with this resource type would have the most comprehensive privileges, since a container defines the largest collection of entities, containing threads and address spaces. As an example, a thread having a capability with a container resource type could issue that system call on all the address spaces and threads that exist in that container.

The rest of the resources in the system are defined as quantitative resources, and they consist of different types of memory pools. As mentioned earlier, their *resid* fields are invalid, and they get used and checked implicitly as part of other capability operations.

CAP_RTYPE_CPUPOOL

Defines the CPU resources of a thread. Depending on the underlying scheduler, it may mean CPU time percentage or a priority. Also, real-time threads may invalidate the value of these capabilities.

CAP_RTYPE_THREADPOOL

Defines the maximum number of threads that may be created by its possessor. Implicitly used and checked as part of the **l4_thread_control** system call.

CAP_RTYPE_SPACEPOOL

Defines the maximum number of address spaces (e.g., page tables and any other related structures) that may be created by its possessor. Similarly affects success of **l4_thread_control** system call by providing address-space accounting.

CAP_RTYPE_MUTEXPOOL

Defines the maximum number of mutexes that may be contended and get temporarily created inside the kernel at any one time. Normally userspace mutex operations are resolved in userspace, but on contended mutexes, kernel internally creates and consumes mutex structures for the userspace.

CAP_RTYPE_MAPPOOL

On some CPU architectures, such as ARM, a virtual to physical memory mapping may require the kernel to allocate intermediate page table structures. This capability defines and enables resource accounting for the allocation of such structures.

CAP_RTYPE_CAPPOOL

When capabilities are manipulated at run-time, some operations may result in allocation of new capability structures. For example, a **replicate** or a **split** operation may create new capabilities in the system. This capability accounts for such operations that result in creation of a new capability.

See also

L4_CAPABILITY_CONTROL

Writing applications using LIBL4

LIBL4 is the generic userspace library that provides a unified interface to the Codezero microkernel. Codezero pagers and applications use this library to initiate system calls. LIBL4 is a plain and abstract library. It implements a thin layer of glue logic for every system call. Since the microkernel does not define any policy for the IPC protocol, LIBL4 also provides certain helpers to form up a userspace protocol for IPC. Finally LIBL4 includes a small set of API helpers for creation of threads and management of capabilities.

Generally the library has been kept small so that it does not enforce complicated API conventions to userspace applications.

Note, the Codezero API Reference chapter already includes exact the API elements that LIBL4 provides for each system call or structure. This guide builds up on the API reference and provides examples on how to use various LIBL4 interfaces.

1 Codezero standalone application and library examples

1.1 Hello World application

The Codezero system configuration provides two baremetal container types named *hello_world* and ***empty***. These container types typically produce new project directories under the *conts/<project_name>*, where <project_name> denotes the name chosen as part

of the configuration. This directory contains all source code, linker, and build scripts that are necessary to build a complete standalone container. The *empty* container contains the bare minimum sources required, and the *hello_world* container includes a few additions for the container to print a *Hello World!* message on the console.

Below is a source code snippet that describes the bare minimum required sources to create a new container, for demonstration purposes. For the methodology of building, advancing, and integration of a new container to the Codezero build system, please refer to the Getting Started chapter.

```
/*
 * Main function for this container
 */

#include <l4lib/macros.h>
#include L4LIB_INC_ARCH(syslib.h)
#include L4LIB_INC_ARCH(syscalls.h)
#include <l4/api/space.h>

void __container_init(void)
{
        /* Generic L4 initialization */
        __l4_init();

        /* Entry to main */
        main();
}

int print_hello_world(void)
{
        printf("%s: Hello world from %s!\n", __CONTAINER__,
__CONTAINER_NAME__);
        return 0;
}

int main(void)
{
        print_hello_world();

        return 0;
}
```

Typically, the _ _container_init() function is called by the LIBL4 library, before calling *main()*.

1.2 Thread library demo

LIBL4 provides a small multithreading library with a handful of API calls. Using this library, a microkernel application may create multiple threads in its own address space. Most LIBL4 helper functions are direct wrappers around actual system calls. Unlike those helpers, this library includes a small and plain runtime for dynamically managing the allocation of thread structures, thread stack, and UTCB areas.

Note that API functions for this library are not listed as part of **L4_USERSPACE_LIBRARY** sections of the microkernel system call API pages. The elements of this API are listed below. The complete list may be found at: *codezero/conts/userlibs/libl4/include/l4lib/lib/thread.h*.

```
struct l4_thread {
  struct task_ids ids;  /* Thread IDs */
  struct l4_mutex lock; /* Lock for thread struct */
  struct link list;     /* Link to list of threads */
  unsigned long *stack; /* Stack (grows downwards) */
  struct utcb *utcb;    /* UTCB address */
};

/*
 * These are thread calls that are meant to be called by library users.
 */
int thread_create(int (*func)(void *), void *args, unsigned int flags,
struct l4_thread **tptr);
int thread_wait(struct l4_thread *t);
void thread_exit(int exitcode);

/*
 * This is to be called only if the to-be-destroyed thread is in a sane
condition for destruction.
 */
int thread_destroy(struct l4_thread *thread);

/* Library init function called by __container_init */
void __l4_threadlib_init(void);
```

Typically, the *__l4_threadlib_init()* is used for library initialization before calling **main()**.

Parent threads may use the *thread_create()* function to create new threads in the current address space.

Parent threads may wait for a thread's own destruction by a *thread_wait()* library call.

Parent threads may optionally destroy children, using a *thread_destroy()* library call. This option should be exercised with caution, as it may cause problems if the thread was manipulating shared data during its destruction.

Alternatively, children threads may exit using the *thread_exit()* library call. Essentially, this call should be paired up with a thread_wait() call by the parent thread.

Below is an example demonstration of how the library may be instantiated:

```
#include <l4lib/macros.h>
#include L4LIB_INC_ARCH(syslib.h)
#include L4LIB_INC_ARCH(syscalls.h)
#include <l4lib/lib/thread.h>

#define NTHREADS      6
#define dbg_printf printf

int thread_test_func1(void *arg)
{
```

```
  /* Wait for a while before exiting */
  int j = 0x400000;
  while (j--)
                ;

  return tid;
}

int thread_demo()
{
  struct l4_thread *thread[NTHREADS];
  int err;

  /* Create threads */
  for (int i = 0; i < NTHREADS; i++)
    err = thread_create(thread_test_func1,0,TC_SHARE_SPACE,&thread[i]);

  /*
   * Wait for all threads to exit successfully
   */
  for (int i = 0; i < NTHREADS; i++)
    if ((err = thread_wait(thread[i])) < 0)
      return err;
  return 0;
}
```

In the above demonstration, a child thread is created by the main thread. The child thread implicitly exits using the *thread_exit()* library call as soon as it returns from its main function. The parent thread waits for the child destruction using the *thread_wait()* library call.

The full source code for the demo may be located at *codezero/conts/baremetal/threads_demo*. The demo may be instantiated by selecting its type as the baremetal container type during the system configuration.

Mutex library demo

Codezero provides the notion of userspace mutexes for multithreaded applications to synchronize. Userspace mutexes consist of architecture-specific synchronization primitives supported by kernel-based mutex wait queues.

Codezero userspace mutexes have been designed such that threads are blocked inside the kernel only upon contention. For the noncontending case, threads simply continue execution without the intervention of the microkernel.

When one or more threads contend, they go to sleep inside the kernel, waiting for a rendezvous to occur for wake up. This design allows for fast userspace locking, while reducing the load on kernel with regard to mutexes.

Below is an example demonstration of how userspace mutexes may be used in a multithreaded application:

```
int mutex_thread_contending(void *arg) {
  struct mutex_test_data *data = (struct mutex_test_data *)arg;
  l4id_t tid = self_tid();
  int err = tid;
```

```
  for (int i = 0; i < MUTEX_INCREMENTS; i++) {
  /* Lock the data structure */
  if ((err = l4_mutex_lock(&data->lock)) < 0)
    return -err;

  /* Sleep some time to have some threads blocked on the mutex */
  for (int j = 0; j < 3; j++)
    l4_thread_switch(0);

  /* Increment and release lock */
  data->val++;

  /* Unlock the data structure */
  if ((err = l4_mutex_unlock(&data->lock)) < 0)
    return -err;
  }
  return 0;
}

int test_mutex(int (*mutex_thread)(void *)) {
  struct l4_thread *thread[MUTEX_NTHREADS];
  int err;

  /* Init mutex data */
  init_test_data(&tdata);

  /* Lock the mutex so nobody starts working */
  if ((err = l4_mutex_lock(&tdata.lock)) < 0)
  return err;

  /* Create threads */
  for (int i = 0; i < MUTEX_NTHREADS; i++) {
    if ((err = thread_create(mutex_thread, &tdata,TC_SHARE_SPACE,
    &thread[i])) < 0)
      return err;

  /* Unlock the mutex and initiate all workers */
  if ((err =   l4_mutex_unlock(&tdata.lock)) < 0)
    return -err;
  return 0;
}
```

Operational model

In this demonstration, the main thread creates multiple threads that will contend on the mutex, namely the *mutex_thread_contending* threads.

The main thread locks the mutex until all threads are created and started.

Each child tries to acquire the mutex.

Once any one of them acquires the mutex, it makes an *l4_thread_switch()* call, effectively giving execution control to other threads.

The other threads, having found out the mutex is locked, declare contention and call *l4_mutex_control* system call for a rendezvous.

When the unlocker releases the lock, it finds out about contention and calls *l4_mutex_control* to wake up any contended threads.

In order to have further insight into userspace mutexes, the *codezero/conts/baremetal/mutex_demo* project is selectable from the Codezero configuration system under baremetal containers.

2 Codezero pager application examples

2.1 Building a service using IPC

IPC is the core method of communication in a virtualization system, based on microkernels. In such a system, often IPC takes place between two parties that are involved in a client-server relationship.

In Codezero, client-server communication is kept simple and lightweight. There is no mechanism to create autogenerated client and server stubs, as this methodology is known to create notoriously complicated and heavyweight implementations. Instead, any client server communication is formed manually using the IPC messaging protocol provided by LIBL4 helper functions.

In this section you may find fictional examples on how to create services that serve requests from potential applications using IPC. Typically, a virtualized operating system kernel serves requests from its client applications this way.

Handling requests

On a typical Codezero service, a request-handling pattern involves the code snippet, as described below.

```
void handle_requests(void) {
  /* Generic IPC data */
  u32 mr[MR_UNUSED_TOTAL];
  l4id_t senderid;
  struct tcb *sender;
  u32 tag;
  int ret;

  /* Receive request from any thread */
  if ((ret = l4_receive(L4_ANYTHREAD)) < 0)
    goto out_err;

  /* Read the tag that identifies a request */
  tag = l4_get_tag();

  /* Read the sender ID, set by the microkernel */
  senderid = l4_get_sender();

  /* Retrieve the information stored on the service about the sender */
  if (!(sender = find_task(senderid))) {
```

```
    l4_ipc_return(-ESRCH);
    return;
}

    /* Read message registers */
    for (int i = 0; i < MR_UNUSED_TOTAL; i++)
    mr[i] = read_mr(MR_UNUSED_START + i);

    /* Handle the request according to the given tag */
    switch(tag) {
      case L4_IPC_REQUEST_NO_RETURN: {
        ret = handle_no_return_request(sender, (char *)mr[0], mr[1], mr[2]);
        if (ret < 0)
          break; /* We only return for errors. */
        else
          return; /* Otherwise, we don't return; a one way request. */
    }
      case L4_IPC_REQUEST_WITH_RETURN:
        ret = handle_returning_request(sender, (void *)mr[0]);
        break;
      default:
    }

/* Send return message back to the client. */
    if ((ret = l4_ipc_return(ret)) < 0) {
      printf("%s: L4 IPC Error: %d.\n", __FUNCTION__, ret);
      BUG();
    }

    out_err:
      printf("IPC Error occured: %d\n", err);
}

void main(void) {
    /* Initialize service */
    initialise();

    while (1) {
    handle_requests();
    }
}
```

Operational model

After initialization, the server continuously asks for requests.

A general server typically accepts requests from any thread on the system, using the **L4_ANYTHREAD** special value.

Each IPC contains crucial information about the request such as the **request tag** and the **sender ID**.

The **sender ID** is set by the microkernel since the receiver service cannot trust this information if it came from a sender party. The microkernel imposes no structure on the content of message fields. This is the only exception for security reasons.

The **tag** in the message identifies the type of request, and it is only relevant to IPC parties in userspace. It bears no significance for the microkernel.

Both the **tag** and **sender ID** are received on preallocated message slots, defined by the protocol between the service and the client. Typically, these message slots are defined by **LIBL4**.

The rest of the message registers contain actual arguments about the request. On the ARM architecture, LIBL4 defines four user-defined slots for sending system call arguments. These are defined by the **L4SYS_ARG0–L4SYS_ARG3** symbols, which denote message-register offset values. These slots typically take place after the **tag** and **senderid** slots.

A typical request involves a **send** and a **receive** phase. For instance, a client makes a system call by the **send** operation and receives the return value of the system call through a **receive**. The return value is also returned in a preallocated message register, typically defined by **LIBL4**.

On requests that do not require a return value (e.g., imagine an **exit** system call that do not return), the service moves on to the next request without a **send** phase, after the first **receive**.

Blocking IPC

A note worth mentioning here is that the communication in this example is synchronous. In other words, both the client and the server tasks block during IPC. This may create complications in those cases where one of the parties involved in the IPC is unresponsive due to a bug. For example, a service in its return phase may block indefinitely if the client does not adhere to the protocol and issue a receive. This problem may be solved by using multithreaded pagers.

This problem is completely avoided when virtualized Linux user threads switch to kernel context because HyperSwitch methodology switches the context of the same thread directly into its kernel context without requiring a synchronous ipc call.

2.2 Management of children threads

Pagers are responsible for creating, destroying, and managing the execution of threads that they are associated with. As a general rule, each pager is responsible for the set of all threads inside a particular container.

Threads may be created in an existing address space, on a brand new, clean address space, or an address space that has been created as a copy of an existing address space.

Below are example code snippets that achieve various thread manipulation operations.

Thread creation

Creating a brand new thread:

```
/* Create a new thread in a new address space */
void thread_new(void) {
  struct task_ids ids;
  int err;
```

```
  ids.tid = TASK_ID_INVALID;
  ids.spid = TASK_ID_INVALID;
  ids.tgid = TASK_ID_INVALID;

  if ((err=l4_thread_control(THREAD_CREATE | THREAD_NEW_SPACE,&ids)) < 0){
    printf("l4_thread_control failed: %d\n", err);
  }
}
```

Creating a new thread in existing address space (e.g., virtualized Linux kernel handling a clone() syscall):

```
/* Create a new thread in an existing address space */
void thread_new(struct task_ids *parent) {
  struct task_ids ids;
  int err;

  /* Specify parent ids */
    ids.tid = parent->tid;
    ids.spid = parent->spid;
    ids.tgid = TASK_ID_INVALID;

  if ((err=l4_thread_control(THREAD_CREATE | THREAD_SAME_SPACE,&ids)) <0){
    printf("l4_thread_control failed: %d\n", err);
  }
}
```

Creating a new thread in a new address space that is a copy of another address space (e.g., virtualized Linux kernel handling a fork() system call):

```
/* Create a new thread in a new, copied space */
void thread_new(struct task_ids *parent) {
  struct task_ids ids;
  int err;

  /* Specify parent ids */
  ids.tid = parent->tid;
  ids.spid = parent->spid;
  ids.tgid = TASK_ID_INVALID;

  if((err=l4_thread_control(THREAD_CREATE | THREAD_COPY_SPACE,&ids)) <0) {
    printf("l4_thread_control failed: %d\n", err);
  }
}
```

Thread context manipulation

Thread context is manipulated via the *l4_exchange_registers()* system call. For example, on a newly created thread in an existing address space, thread context may be modified in order to initiate new thread execution on the relevant context. Typically, any one of thread's registers or UTCB address may be changed or read by this system call. Below is a fictional example on how this call may be used:

```
void thread_manipulate(struct task_ids *new_ids, unsigned long new_stack,
                                    unsigned long utcb_address)
{
        struct exregs_data exregs;
        int err;

        memset(&exregs, 0, sizeof(exregs));

        /* Set new stack for child */
        exregs_set_stack(&exregs, new_stack);

        /* Set child return value to 0 */
        exregs_set_mr(&exregs, MR_RETURN, 0);

        /* Set child utcb */
        exregs_set_utcb(&exregs, utcb_address);

        /* Do the actual exchange registers call to microkernel */
        if ((err = l4_exchange_registers(&exregs, new_ids->tid)) < 0)
                printf("Exchange registers error: %d\n", err);
}
```

Operational model

The Codezero capability system is flexible enough to allocate privileges to make any system call to any particular task. However, as a convention *l4_thread_control* and *l4_exchange_registers()* are privileged system calls that are only meant to be executed by pagers. As an example, the virtualized Linux kernel may receive a system-call request that may involve one of these operations. The operational model in the above system calls are as follows:

A client thread requests a new thread is created in its own address space via IPC.

The pager handles the request by creating a new thread via *l4_thread_control* and modifying its context as requested via *l4_exchange_registers*.

Pager replies back to client that new thread is ready for execution.

Depending on the implementation, the pager may initiate a new thread's execution if requested by the client.

In conclusion, the above code snippets are used usually as part of an IPC request/reply pair between the pager on the system and its client.

2.3 Manipulating children address spaces

Pagers manipulate the address space of their children using privileged address space manipulation functions.

Address spaces are created, cleared, and destroyed by a **l4_thread_control** system call during thread creation. However, the modification of existing address spaces are done by the **l4_map** and **l4_unmap** system calls.

Below are the code snippets for typical address-space manipulation operations by pagers.

Mapping a new page

Below is the microkernel's architecture-specific structure for describing a page fault:

```
/* Kernel's data about the fault */
typedef struct fault_kdata {
  u32 faulty_pc;  /* In DABT: Aborting PC, In PABT: Same as FAR */
  u32 fsr;        /* In DABT: DFSR, In PABT: IFSR */
  u32 far;        /* In DABT: DFAR, in PABT: IFAR */
  pte_t pte;      /* Faulty page table entry */
} __attribute__ ((__packed__)) fault_kdata_t;
```

Below is the code snippet taken from a pager during the handling of a page fault from a task:

```
...
/* Map the new page to faulting task */
l4_map((void *)page_to_phys(page),
       (void *)page_align(fault->address), 1,
       (reason & VM_READ) ? MAP_USR_RO : MAP_USR_RW,
       fault->task->tid);
dprintf("%s: Mapped 0x%x as writable to tid %d.\n", __TASKNAME__,
page_align(fault->address), fault->task->tid);

return 0;
}
```

Operational model

In the above example, a pager (namely a virtualized kernel) is handling a page fault IPC generated by the microkernel in response to a memory exception (e.g., a data abort or a prefetch abort exception on ARM).

The exception data is delivered in the form of IPC, using the **fault_kdata_t** structure in case of the ARM architecture.

Using the information provided by the IPC, the pager determines which physical page to map to its client and with what permission flags.

Usually pagers manipulate client address spaces as a result of page faults. On other occasions, pagers may proactively prefault various pages in a client's address space if this is necessary.

Unmapping a range of pages

Below is the code snippet taken from a pager during the unmapping of a virtual memory address range from a client task:

```
...
/*
* Unmap the whole VMA address range. Note that this
* may return -1 if the area was already faulted, which
* means the area was unmapped before being touched.
*/
l4_unmap((void *)__pfn_to_addr(vma->pfn_start),
vma->pfn_end - vma->pfn_start, task->tid);
  return 0;
}
```

Operational model

In the above example, the pager removes the mapping for a virtual address range from a task in response to a memory unmap IPC request.

2.4 Other examples

For other API usage examples such as IPC, thread management, or capability management, please refer to baremetal container sources provided under *codezero/conts/baremetal*. Under this directory, each individual project is a good starting point for understanding how the API may be used in L4 applications.

If you have questions about other application scenarios, or if a concept is not described clearly enough, please notify us by direct email or ask on our mailing list.