Complex Lab – Operating Systems

Introduction

Martin Küttler
Organisation

- Website (e.g. source repository): https://os.inf.tu-dresden.de/Studium/Praktikum/
- Mailinglist: kpr2019@os.inf.tu-dresden.de
  Subscribe at https://os.inf.tu-dresden.de/mailman/listinfo/kpr2019
- Repository: https://os.inf.tu-dresden.de/repo/git/kpr.git
- Send Code to: martin.kuettler@os.inf.tu-dresden.de
Organisation – Dates

12.11.19 Introduction
24.11.19 1st Assignment due
26.11.19 Sessions & Memory
07.01.20 Graphical Console
04.02.20 Keyboard Driver
31.03.20 Final Assignment due
Requirements

- Basic OS knowledge: threads, processes, virtual memory, drivers, ...
- Experience with Linux: editor, shell, development (make, gcc)
- Experience with C/C++
Goal of this course

Multi-user Pong game
Goal of this course

- Pong Server
- Paddle Client 1
- Paddle Client 2
- Moe
- Sigma0
- Fiasco Kernel
Goal of this course

- Pong Server
- Paddle Client 1
- Paddle Client 2
- Keyboard Driver
- Console
- Memory Management
- Moe
- Sigma0
- Fiasco Kernel
Required Software

- A Linux system (preferably Debian/Ubuntu)
- qemu
- build-essentials: gcc (<9), g++ (<9), make
- gcc-multilib, g++-multilib
- device-tree-compiler
The Repository

doc/source/ documentation
obj/l4/$ARCH/ build directory
src/kernel/ Fiasco kernel sources
src/l4/pkg/ sources for all userspace packages
src/l4/conf/ boot configuration

To build everything in the beginning:

```
$ make setup
$ make [-j X]
```

Add the following to src/l4/Makeconf.local and the first two lines to src/kernel/fiasco/src/Makeconf.local if your default compiler is gcc >8.

```
CC=gcc -8
CXX=g++ -8
OBJ_BASE=<full-path-to-build-dir>
```
Build Infrastructure

simple Makefile:

```
PKGDIR  ?=  ../..
L4DIR   ?=  $(PKGDIR)/../..

SRC_C   =  main.c
SRC_CC  =  file.cc

TARGET  =  my_program
REQUIRED_LIBS  =

include $(L4DIR)/mk/prog.mk
```

Generate template Makefiles and file structure:

```
kpr/src/l4/pkg$  mkdir my_pkg
kpr/src/l4/pkg$  cd  my_pkg
kpr/src/l4/pkg/my_pkg$  ../../mk/tmpl/inst
```
Headers

Package headers are in:

\texttt{kpr/src/l4/pkg/my_pkg/include/my_header.h}

They are installed to

\texttt{kpr/obj/l4/$ARCH/include/l4/my_pkg/my_header.h}
Compiling stuff

# build everything (e.g. in the beginning)
kpr$ make [-j X]

# build single pkg from build dir
kpr/obj/l4/amd64/pkg/my_pkg$ make

# build single pkg from source dir
kpr/src/l4/pkg/my_pkg$ make [O=<BUILDDIR>]

Run:

kpr/obj/l4/amd64$ make qemu [E=<entry>]

Qemu

Qemu configuration in kpr/obj/l4/amd64/Makeconf.local

- MODULE_SEARCH_PATH: Where binaries, config files, etc. are
- QEMU_OPTIONS: command line parameters

Boot configurations in kpr/src/l4/conf/modules.list

- entry name
  - kernel fiasco <parameters>
  - roottask moe <program/startup script>
  - module modules to load (one per module-line)
IPC overview

- External resources are accessed through capabilities.
- Applications have a namespace containing capabilities.
- Some magically appear at startup, others can be added in startup script.
- Communication requires capability to an IPC gate.
Example Ned config scripts can be found in

- src/l4/conf/examples
- src/l4/pkg/l4re-core/ned/doc/
- src/l4/pkg/examples/clntsrv

General idea for our simple usecase:

- Programs are started with `L4.default_loader.start`
- They take capabilities to channels (aka IPC gates) that are created with `L4.default_loader.new_channel`
- Programs can get a client or a server (`:svr()`) capability
In the program, these capabilities can be used to send and receive messages.

Server side:
- Register Server object (that implements handler) with the named cap.
- Run server loop
- Handle requests

Client side:
- Query IPC gate capability at name service.
- Invoke cap with arguments.

Sending data works with the UTCB (lecture). We use a high-level interface on top.
Define a class that inherits from `L4::Kobject_t` that has a member function for each message type.

```cpp
struct MyClass : public L4::Kobject_t<MyClass, L4::Kobject, MYPROTO_NUM>
{
    L4_INLINE_RPC(int, foo, (int arg1, int arg2));
    typedef L4::typeid::Rpcs<foo_t> Rpc;
};
```

Functions return error code.

All functions are listed in type `Rpc`.
IPC interface

- The server implements a class inheriting `L4::Epiface` that implements the methods.
- Types are converted between interface and implementation; special types for inout/out parameters, capabilities, arrays, etc. are available.
- If you need to send caps, the `L4::Kobject_t` needs a fourth template argument of type `L4::TypeInfo::Demand_t<NUM>`, with `NUM` being the maximum number of caps per call.
- See `src/l4/pkg/examples/clntsrv` for an example.
- See documentation `doc/source/html/index.html` (specifically e.g. `doc/source/html/l4_cxx_ipc_iface.html`)
- The source is the ultimate documentation (and it includes all the normal documentation).
- When you have questions, use the mailing list (kpr2019) or send me an email directly.
Assignment 1

- Download the repo, set it up and compile everything.
- Try it in Qemu, make sure that hello works.
- Build a client-server version of hello: The client should send a string to the server via ipc.
- As a second step, also consider large strings.
- Deadline on 24.11.19.
Next meeting

We meet next on 26.11.19, where we start the real project.