COMPLEX LAB: MICROKERNEL-BASED SYSTEMS

CARSTEN WEINHOLD
Organization

• Website:
  https://os.inf.tu-dresden.de/Studium/Praktikum/

• Mailinglist:
  kpr2018@os.inf.tu-dresden.de
  Subscribe at:
  https://os.inf.tu-dresden.de/mailman/listinfo/kpr2018

• Repository:
  https://os.inf.tu-dresden.de/repo/git/kpr.git

• Send Code to:
  carsten.weinhold@tu-dresden.de
Important Dates & Deadlines

• 23 October 2018: Introduction
• 18 November 2018: 1st assignment due
• 20 November 2018: Sessions & memory
• 8 January 2019: Graphical console
• 29 January 2019: Keyboard driver
• 31 March 2019: Final assignment due
Requirements

• Basic OS knowledge:
  • threads
  • processes
  • virtual memory
  • drivers
  • ...

• Experience with Linux:
  • editor, shell, development tools (make, gcc, ...)

• Experience with C/C++
Goal of this course
Goal of this course

- Pong Server
- Paddle Client 1
- Paddle Client 2
- Keyboard Driver
- Console
- Memory Management
- Sigma0
- Moe

You build this!
Developer environment

- A Linux system (preferably Debian/Ubuntu)
- Qemu
- build-essentials:
  - make
  - gcc (>=6), g++ (>=6),
  - gcc-multilib, g++-multilib
  - ncurses
  - dialog
  - ...

The repository

• Directories:
  - doc/source/ documentation
  - obj/l4/$ARCH/ build directory
  - src/kernel/ Fiasco kernel sources
  - src/l4/pkg/ sources for all user-space packages
  - src/l4/conf/ boot configuration

• To build everything in the beginning:
  
  $ make setup
  
  $ make [-j X]

  Handy: add OBJ_BASE=<full-path-to-build-dir>
  to build from shell in source tree

| To override the default compiler: |
| add CC=gcc-8 CXX=g++-8 to make setup, if your default compiler is gcc < 6 or >= 9 |
| Also add the following to src/l4/Makeconf.local: |
| CC=gcc-8 |
| CXX=g++-8 |
Build infrastructure

• simple Makefile:

PKGDIR ?= ../..
L4DIR ?= $(PKGDIR)/../..
SRC_C  = main.c
SRC_CC  = other_file.cc
TARGET  = my_program
REQUIRED_LIBS = <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
• Package headers are in:
  kpr/src/l4/pkg/my_pkg/include/my_header.h

• They are installed to:
  kpr/obj/l4/x86/include/l4/my_pkg/my_header.h
Compiling

• Compiling stuff
  
  # build everything (e.g. in the beginning)
  kpr$ make [-j X]
  
  # build single pkg from build dir
  kpr/obj/l4/x86/pkg/my_pkg$ make
  
  # build single pkg from source dir
  kpr/src/l4/pkg/my_pkg$ make [O=<BUILDDIR>]

• Run:
  
  kpr/obj/l4/x86$ make qemu [E=<entry>]
Qemu

- QEMU configuration in kpr/obj/l4/amd64/conf/Makeconf.local
  
  **MODULE_SEARCH_PATH**: binaries, cong files, etc.
  **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 <module 1>
  module <module 2>
  module ...
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
IPC overview: startup config in Lua

• Example Ned cong 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
IPC Overview: clients & servers

• 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 UT CB (lecture). We use a high-level interface on top
IPC interface

• 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> Rcps;
};
```

• Functions return error code
• All functions are listed in type `Rcps`
IPC interface

- The server implements a class that inherits from \texttt{L4::Epiface}, which 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 \texttt{L4::Kobject_t} needs a fourth template argument of type \texttt{L4::Type_info::Demand_t<NUM>}, with \texttt{NUM} being the maximum number of caps per call
- See \texttt{src/l4/pkg/examples/clntsrv} for an example
- See documentation in \texttt{doc/source/html/index.html} and \texttt{doc/source/html/l4_cxx_ipc_iface.html}
Questions?

• The source is the ultimate documentation (and it includes all the normal documentation).

• If you have questions, use the kpr2018 mailing list or send me an email
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 18 November 2018

• Next meeting: Let’s start the real project!
  20 November 2018