COMPLEX LAB: MICROKERNEL-BASED SYSTEMS

CARSTEN WEINHOLD
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

Starting November 27 will switch time slots:
- Tuesday, 2:50pm: Lecture
- Tuesday, 4:40pm: Exercise / complex lab
Assignment 1

• You have basic understanding of:
  • How to start programs
  • How to configure communication channels
  • Basic L4Re IPC framework usage
• Client and server are communicating
• But everything is static: only one client per server
Sessions

• Scenario:
  • Multiple clients per server
  • Server stores per-client data, needs to distinguish between clients

• Poor man’s solution:
  • Assign dynamic ID to each client
  • Client sends ID whenever it calls the server
  • Problem: IDs can be faked

• Better solution: Sessions
  • One IPC gate per client
  • Clients can be distinguished based on gate label
  • Ideally: clients need not do anything special
Sessions in L4Re

Ned creates IPC gate
Sessions in L4Re

Ned starts server & map IPC gate to it
Sessions in L4Re

Ned calls
Factory::create()
Sessions in L4Re

Server creates IPC gate for client session & maps a capability to Ned.
Sessions in L4Re

Ned starts client & maps received capability to IPC gate to client
Sessions in L4Re

Client can invoke its session IPC gate
local L4 = require("L4");

local loader = L4.default_loader;
local hello2 = loader:new_channel();

L4.default_loader:start(
    { caps = { my_server = hello2:svr() },
      log = { "server", "red" } },
    "rom/hello2");

L4.default_loader:start(
    { caps = { my_server = hello2 },
      log = { "client1", "green" } },
    "rom/hello2_test");
Lua Startup Script for Session Server

```lua
local L4 = require(„L4“);

local loader = L4.default_loader;
local hello2 = loader:new_channel();

L4.default_loader:start(
    { caps = { my_server = hello2:svr() },
      log = { "server", "red" } },
    "rom/hello2");

L4.default_loader:start(
    { caps = { my_server = hello2:create(0, "client1") },
      log = { "client1", "green" } },
    "rom/hello2_test");

L4.default_loader:start(
    { caps = { my_server = hello2:create(0, "client2") },
      log = { "client2", "blue" } },
    "rom/hello2_test");
```
How to Implement Sessions

• Client don’t need to be changed (that’s what we wanted)
• Servers need to handle create call
• Before we look at that ...
• ... a short tour the L4Re IPC server framework
L4Re IPC Server Framework

• **L4::Server** implements basic server loop:

```c
void loop() {
    while (true) {
        m = recv_message();
        ret = dispatch(m, utcb);
        send_reply(m, ret);
    }
}
```

• For each IPC gate there is a **L4::Epiface**, which
  • Keeps the capability to the IPC gate
  • Handles messages from its IPC gate via its `dispatch()` method

• How does the server know which **L4::Epiface** to use?
L4::Epiface Registry

- **L4::Epifaces** are stored in a per-server registry
- The registry can find **Epifaces** by an ID (label of IPC gate)
- **L4::Basic_registry**: ID is pointer to object
- **L4Re::Util::Object_registry** provides a convenient interface:

```cpp
L4::Cap<void> register_obj(L4::Epiface *o,
                            char const *service);

L4::Cap<void> register_obj(L4::Epiface *o);

bool unregister_obj(L4::Epiface *o);
```
Registry Server

- **L4Re::Util::Registry_server** is a **L4::Server** that maintains a **L4Re::Util::Object_registry**

```cpp
static L4Re::Util::Registry_server<> server;

class MyServer : public L4::Epiface_t<MyServer, MyInterface> {
    ... 
};

// When you need a new session object
server.registry()->register_obj(new MyServer());
```
class Session_server : public L4::Epiface_t <Session_server, 
L4::Factory> {

public:
  int op_create(L4::Factory::Rights, 
     L4::Ipc::Cap<void>& res, 
     l4_mword_t type, 
     L4::Ipc::Varg_list<> &&args) {
    if (type != 0)
      return -L4_ENODEV;
    L4::Ipc::Varg tag = args.next();
    if (!tag.is_of<char const *>())
      return -L4_EINVAL;
    auto my_server = 
      new My_server(tag.value<char const *>());
    server.registry()->register_obj(my_server);
    res = L4::Ipc::make_cap_rw(my_server->obj_cap());
    return L4_EOK;
  }
};
Assignment 2.2: Sessions

• With that you can add support for multiple clients in the hello server.

• Assignment 2.2:
  • Make your hello server a logging server that supports multiple clients
  • Client messages should be prefixed with an id string that is passed to the server in the create call

• Problem: Now you need dynamic memory, but malloc and free are missing

• Hence, there is also Assignment 2.1 ...
Assignment 2.1

You build this!
Memory Allocation

• Memory allocation is **(currently not)** implemented in a backend of L4Re's C library:
  `src/l4/pkg/l4re-core/libc_backends/libs/l4re_mem`

• You can get new pages from Moe:
  • Allocate a Dataspaces capability
  • Get a Dataspaces from Moe:
    `L4Re::Env::env() -> mem_alloc() -> alloc(size, ds);`
  • Attach dataspace to local address space:
    `L4Re::Env::env() -> rm() -> attach(&addr, size, flags, ds);`

• To free unused pages:
  • `L4Re::Env::env() -> rm() -> detach(addr, nullptr);
    L4Re::Env::env() -> mem_alloc() -> free(ds);`
How **NOT** to Allocate Memory

- What I don’t want to see: Allocating aDataspace for each an every call to malloc()

```c
void *malloc(unsigned size) {
    L4::Cap<L4Re::Dataspace> ds =
        L4Re::Util::cap_alloc.alloc<L4Re::Dataspace>();
    if (!ds.is_valid())
        return 0;
    long err =
        L4Re::Env::env()->mem_alloc()->alloc(size, ds);
    if (err)
        return 0;
    void *addr = 0;
    err = L4Re::Env::env()->rm()->attach(&addr, size,
        L4Re::Rm::Search_addr, ds);
    if (err)
        return 0;
    return addr;
}
```
Memory Management Done Right

• Idea:
  • Keep list of (address, size) pairs
  • In malloc, search for an appropriate entry
• Problem: You'd need dynamic memory for that list
• Typical Solution: Inlining
  • Put size and next-pointer directly into your memory
  • Do not hand out the memory where size is stored; it's needed for free
  • That's what most libC-implementations do
• Or use a bitmap to store available chunks
Memory Management: Problems

• You will need some initial memory. You can use L4Re's memory allocator for that
• As soon as you have multiple threads (you will), you need proper locking
• There are more options for the implementation. Come up with something yourself, or have a look in some book or the internet
Assignment Summary

• Implement `malloc()`, `free()`, `realloc()`, `calloc()`
  • `malloc()` has to work first, but the other functions will be needed later on
  • Try not to waste too much memory

• Implement session support in your logging server:
  • It must support multiple clients
  • Have server print a string prefix that identifies the client who sent the message

• Next meeting: Let’s see something!
  8 January 2019