TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Complex Lab – Operating Systems 2014 Winter Term

Sessions & Dynamic Memory

General Coding Style

- use one language for your output (avoid special characters)
- use line breaks

- avoid hard-coding, define a constant and use it instead
- make definitions at one single point (not *the same definition* on the top of each c-file)

- document your code (English preferred)
  - specify parameters types and return value
  - keep it short

- delete unused code

String handling

- Examples from your code
  - char str[20];
  - char str[BUF_SIZE];

- max string length 63 * 4 characters for 1 IPC message
  - limitations given by the UTCB structure
  - check string length and send more massages if necessary

- just reading and printing can crash
  - string termination not necessary copied (string to long)
  - str[size] = 0 helps for printing but overwrites the last element

String handling

- bad hard coding examples

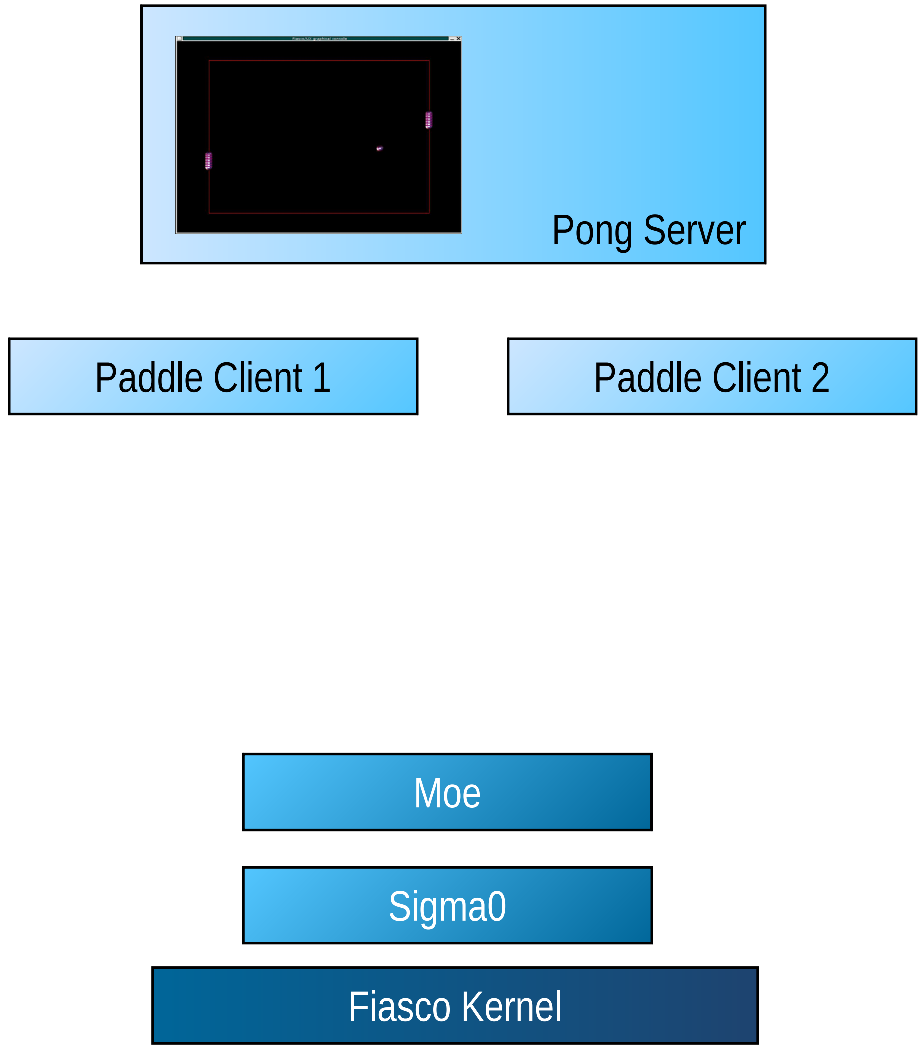
```
char *str = "Hello World"
ios << buf_cp_out(str, 64);
```

- use L4::String and Buf_cp_out / Buf_cp_in

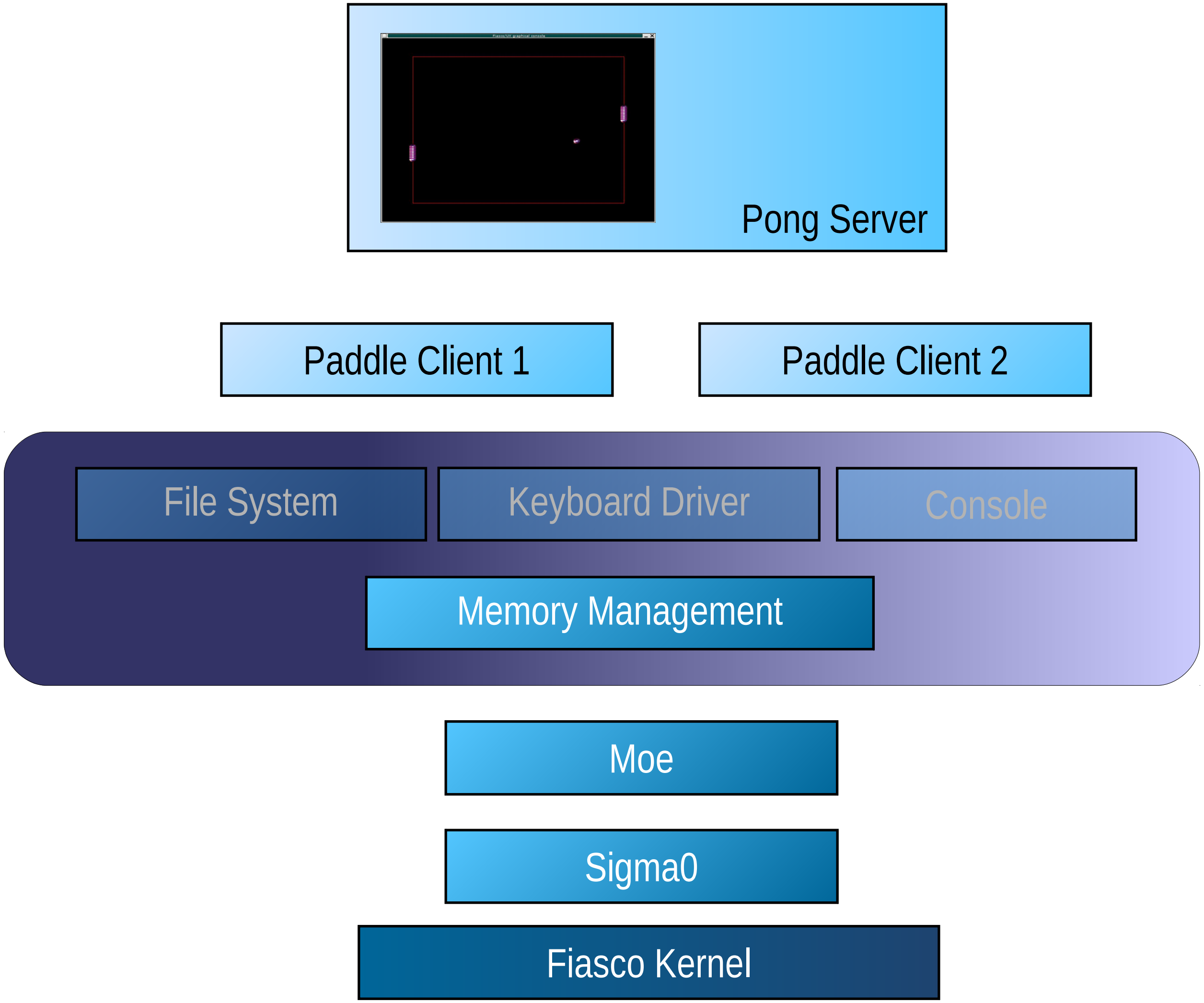
```
L4::String *str = new L4::String("Hello World");
ios << Buf_cp_out<char const>(str.p_str(), str.length());
ios >> Buf_cp_in<char const>(str, length);
```

- `Buf_in<char const>`
  - avoids copy
  - be careful: a new IPC message overwrites your buffer!

You are here


Pong Server
Paddle Client 1
Paddle Client 2
Moe
Sigma0
Fiasco Kernel

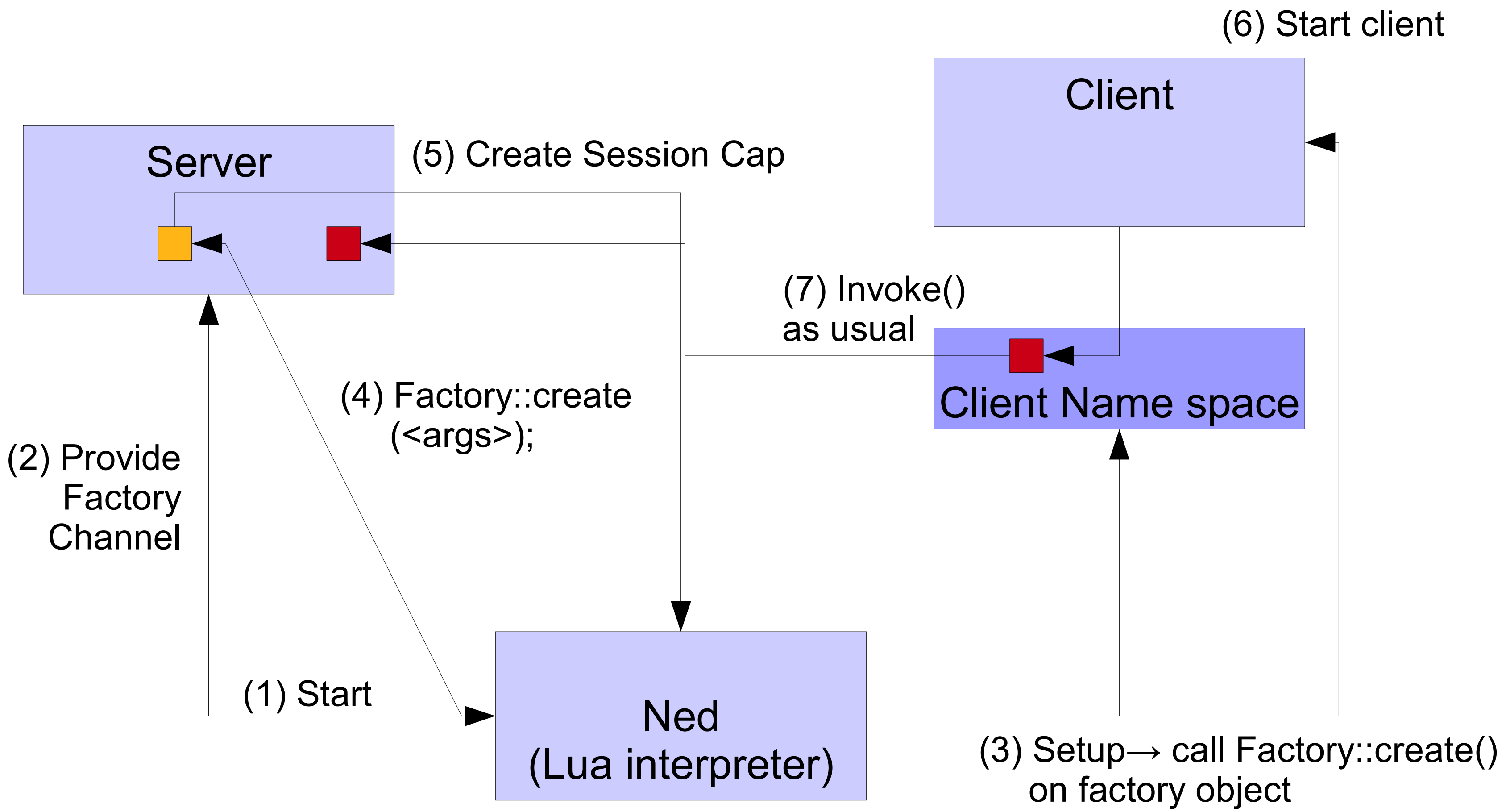
Today's Goal


Pong Server
Paddle Client 1
Paddle Client 2
File System
Keyboard Driver
Console
Memory Management
Moe
Sigma0
Fiasco Kernel

Sessions

- Scenario:
  - Multiple clients per server
  - Server stores per-client data
  - Need to distinguish clients
- Poor man's solution:
  - Assign dynamic ID
  - Client provides ID (+ data) upon each call
  - Problem: faking IDs possible
- Better solution **Sessions**:
  - One IPC gate per client
  - Distinguish client using gate label
- Even better:
  - Client should not know about sessions (embedded graphics driver vs. fully-featured windowing system)

Sessions in L4Re



L4/Lua: Connecting Services

- Lua interpreter establishes communication channels and passes them through the environment.

```
local ld = L4.default_loader;
local the_server = ld:new_channel();

ld:start( { caps = { calc_server = the_server:svr() },
            log    = { "server", "blue" } },
          "rom/server" );

ld:start( { caps = { calc_server = the_server },
            log = { "client", "green" } },
           "rom/client" );
```

new_channel() creates an IPC gate. No thread is bound to it yet.

L4/Lua: Connecting Services

caps = {} defines additional capabilities that are passed to the application on startup. The syntax is <name> = <object>, where name is later used to query this capability at runtime and object is a reference to an object.

For IPC channels, 'object' is the sender side of the channel, while 'object:svr()' refers to the server/ receiver side.

ication channels
nment.

);

```
ld:start( { caps = { calc_server = the_server:svr() },
            log = { "server", "blue" } },
          "rom/server" );

ld:start( { caps = { calc_server = the_server },
            log = { "client", "green" } },
          "rom/client" );
```

- Lua interpreter establishes communication channels and passes th

log = { <tag>, <color> }
configures this application's log output. All log messages will be sent to the serial line by default. Every line for the application will start with “<tag> | “ and will be colored in the respective VT100 terminal color.

```
local ld = L4.
local the_ser

ld:start( { caps = { calc_server = the_server:svr() },
            log = { "server", "blue" } },
          "rom/server" );

ld:start( { caps = { calc_server = the_server },
            log = { "client", "green" } },
          "rom/client" );
```

Session Channels in Lua

```
local ldr = L4.default_loader;
factory = ldr:new_channel();

-- start server as on last slide
-- [...]

-- start the client and implicitly create new session
ldr:startv( {   caps = { server = factory:create(0,
                                "config parameters")
                     },
                     log = {"client", "blue"}
                 },
             "rom/client" );
```

factory:create() lets ned call the server side's factory function to create a new server object.

Requirements:
1. The server is already running.
2. The server implements the Factory and Meta protocols.

- Client: no change at all
- Server: gets a bit more complicated

A short tour of the L4Re IPC server framework

(or: Things you didn't want to know, but now need to)

Server Class

L4::Server

- Implements a basic server loop

```
while (1) {
      m = recv_message();
      ret = dispatch(m.tag(), ios);
      reply(m, ret);
}
```

- Users need to provide a `dispatch()` function.
  - If curious, see `L4::Ipc_svr::Direct_dispatch`
- Plus: Loop hooks
  - Allows you to specify callbacks for
    - IPC error
    - Timeout
    - Prepare to wait
  - You should get away with `L4::Default_look_hooks`

Server objects

L4::Server_object

- Servers may provide
  - Different services
  - To different clients
  - Through different server objects
- A server object is basically a wrapper for a capability: `L4::Server_object::obj_cap()`
- Server object implementations may vary
  - Need a dedicated `dispatch()` function per object
- But how does the server decide at runtime which function to call?

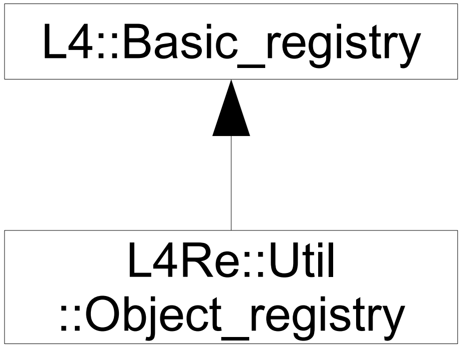
Object registries

L4::Basic_registry

- Server objects are stored in per-server object registry
- Registry provides a function to find an object by an ID

```
L4::Basic_registry::Value
   L4::Basic_registry::find(l4_umword_t id);
```

- ID is the label of the IPC gate through which a message arrived.
- Basic_registry is **basic**:
  - No ID management
  - ID is simply interpreted as a pointer to a local object



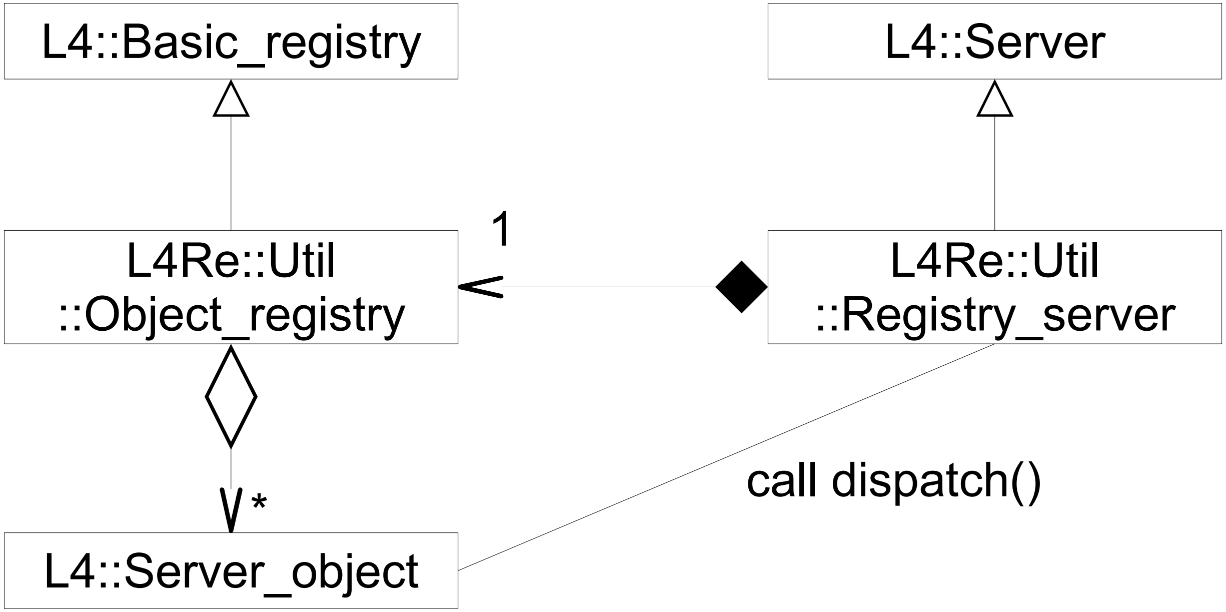
- Advanced servers want to bind objects to
  - IPC gates or
  - Names specified in the Lua init script
- Use object pointer as label for IPC gate

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

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

bool unregister_obj(L4::Server_obj *o);
```

Using Object_registry in a server

- There's a utility class representing a server with an object registry:

`L4Re::Util::Registry_server`



Session Implementation for Beginners

- Declare a server:

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

- Implement a session factory object:

```
class MySessionServer
    : public L4::Server_object
{ .. };
```

- Implement a dispatch function:

```
int MySessionServer::dispatch(l4_umword_t o,
              L4::Ipc_iostream& ios)
{ .. };
```

Session dispatch

- Factory object implements 2 functions:

  1. Meta service capability

```
#include <l4/re/util/meta>

... dispatch(tag, ios) ...
{
  ..
  switch(tag.label()) {
    case L4::Meta::Protocol:
      return L4Re::Util::handle_meta_request<L4::Factory>(ios);
  ..
}
```

During session creation, ned will send a request to your server asking whether it supports the Meta protocol. This protocol allows querying some information about a channel – just copy these 2 lines into your dispatch() implementation.

Session dispatch

- Factory object implements 2 functions:
  2. Factory protocol handler

```
#include <l4/re/util/meta>

... dispatch(tag, ios) …
{
  ..
  switch(tag.label()) {
    case L4::Factory::Protocol: {
      unsigned op;
      ios >> op;
      if (op != 0) return -L4_EINVAL;

      /* Create new server object */
      server_obj = new ServerObject();
      server.registry()->register_obj(server_obj);
      ios << server_obj->obj_cap();
      return L4_EOK;
    }
    ..
  }
}
```

This handles real creation of a server object.

register_obj() enters the new object into the registry. Afterwards, obj->obj_cap contains the new object's capability slot.

This slot is then returned to the client.

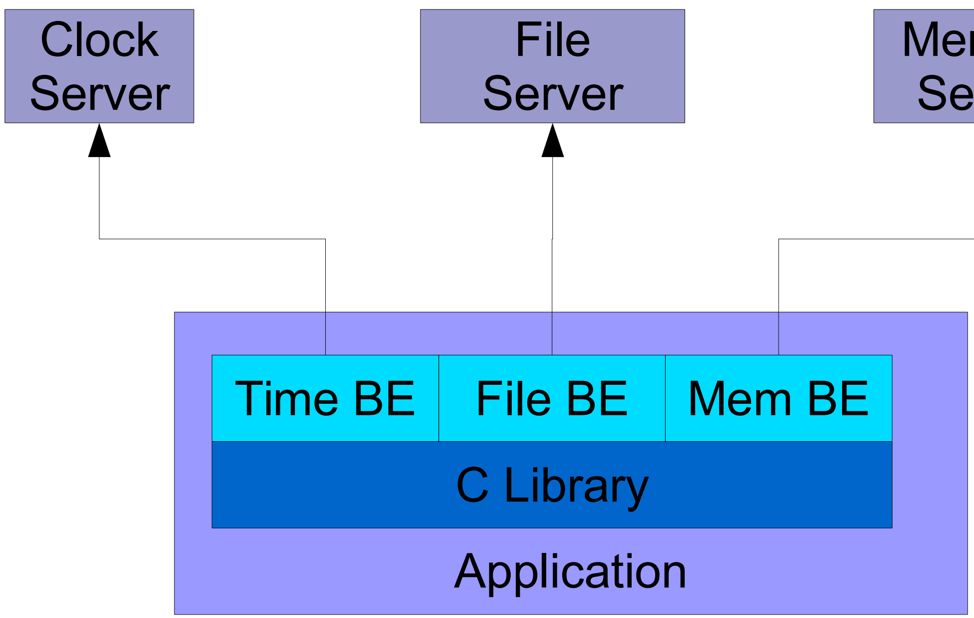
Break

- You should now be able to rewrite your hello server so that it supports multiple client sessions and prefixes every client with a specific tag.
  → Assignment 1.5
  - Let the tag be a parameter that's passed during session creation.

- Problem: Now you'd need to dynamically allocate server objects, but malloc and free (new and delete) are missing.
  → Assignment 2

C Library + backends

- L4Re comes with a C library
- System-dependent functionality is implemented in **backends** (l4/pkg/libc_backends/*)



Memory in L4Re

- Each task initially gets a Memory Allocator capability (usually implemented by Moe)
  - L4Re::Env::env()->mem_alloc()
- Obtaining memory:
  - Allocate capability for dataspace
  - Get a dataspace from mem allocator
    - L4Re::Env::env()->mem_alloc()->alloc(size, ds)
    - Fills dataspace cap
  - Attach dataspace to local address space
    - L4Re::Env::env()->rm()->attach(addr, size, ..., ds)
  - Use memory
  - Detach and release
    - ds = L4Re::Env::env()->rm()->detach(...);
    - L4Re::Env::env()->mem_alloc()->free(ds);

malloc() - wrong solution

```
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, 0);
   if (err) return 0;

   return addr;
}
```

TECHNISCHE UNIVERSITÄT DRESDEN

Dynamic memory management

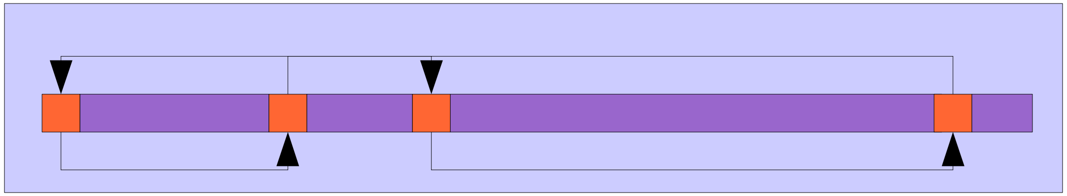
- Previous solution uses one dataspace (>= 4 kB) for each allocation
  - That's wasting memory.
  - You're going to run out of capability slots.

- Better solution:
  - Manage a heap manually
  - Grow if out of memory

- Internal management:
  - Lists
  - Bitmaps

Memory Management - Lists

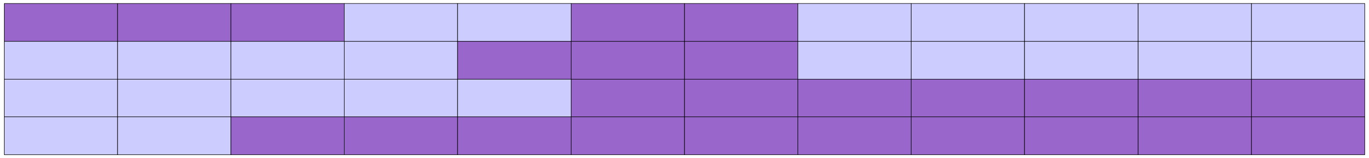
- Keep list of (address, size) entries
- Upon allocation search for fitting entry
  - FIFO, LIFO, Best fit, …
- Problem: chicken vs. egg
  - managing the list would need dynamic memory, too
- Solution: inlining
  - Instead of keeping a separate list, put list pointers into the allocated chunks
  - That's what LibC's malloc does



Memory pools, bitmaps

- Optimization: Manage memory in pre-defined bucket sizes (memory pool)
- Can manage a bitmap of available/used buckets
  - Uses less memory
  - Bit-accesses are a mess, but see l4/pkg/l4util/include/bitops.h
- Somewhat similar to what Linux' SLAB/SLOB/SLUB allocators do

1	1	1			1	1					
				1	1	1					
					1	1	1	1	1	1	1
		1	1	1	1	1	1	1	1	1	1



Dynamic MM problems

- Bootstrapping
  - Need a certain amount of initial memory
  - You may use L4Re's memory allocator for that
- Fragmentation
  - Defragmentation is hard: need to update all existing references into memory area
- Threads
  - Remember locking once you start using multiple threads

Assignment

- Final goal: Session-capable hello server
  - Which you may use as a logging facility for later assignments.
- Intermediate step: implement malloc() and free() using a memory management policy of your choice
  - Try reading on some strategies first – Google should help you a lot (keywords: dynamic memory management, memory allocator, …)
  - Add implementation in **pkg/libc_backends/lib/l4re_mem**
- No need to do garbage collection, but once you're at it... ;)
- You should thereafter also be able to use C++' STL.

JDB - The Fiasco Kernel Debugger

- Start Fiasco with `-serial_esc` command line option and Qemu with `-serial stdio`
  - Redirect's L4 output (serial console) to your terminal
  - Allows you to enter the kernel debugger by hitting ESC
- You can also enter JDB from code:

```
#include <l4/sys/kdebug.h>
[..]
enter_kdebug("message or JDB command");
[..]
```

- JDB is command-driven
  - case-sensitive!
- Most important command: **h** (== help)

Kernel object list

- **Q** – list all kernel objects
  - Navigate with cursor keys (vim bindings hjkl also work)
  - Hit enter on certain objects (e.g., threads) to get more info

  - Layout:
    - Debug ID
    - Object address
    - Object type
    - Additional info
      - *Tasks & threads*: names, S == corresponding address space, C == cpu, R == reference count
      - *IPC gates*: L == label, D = owning thread

Thread info

- List present (all) threads: **lp**
- List ready threads: **lr**
- Show a thread's capability space:
  o on thread in kobject list

- Entering the thread view:
  - **t<debug ID>** (no space in between!)
  - ENTER on a thread in kobject list
  - ENTER on a thread in present/ready lists

- Thread info contains:
  - Kernel state, kernel stack
  - General-purpose registers

What code is my thread executing?

- Get EIP from thread view
  - Tools to help you: objdump, addr2line
- More sophisticated: getting a backtrace
  - **btt<debug ID>**
    - Lists user-level (first) and kernel-level (second) backtraces
    - Numbers only
    - There's a tool: kpr/src/kernel/fiasco/tool/backtrace
      - Give the binary as a parameter
      - Paste the back trace (as it is) into the window

Memory dumps

- **dt<debug ID><virtual address>**
  - Scroll with PgUp/PgDown
  - SPACE to switch between view modes (big-endian, little-endian, ASCII)
- Disassembling
  - In thread view or memory dump
  - Move cursor to address
  - Press **u** in mem dump
  - Press SPACE in thread view

IPC tracing

- JDB can collect information on IPC and results
- This is basically system call logging

- Needs to be turned on:
  - **I*** turn on IPC tracing
  - **IR+** turn on tracing of IPCs and results (← you'll mostly want this one)

- Then run as normal

- Later press **T** to enter the trace buffer view

- Output

  ipc: ***THR_ID TYPE***->[C:***CAP_DEST***] DID=***DEST.debug*** \
  L=***LABEL*** [***TAG***] (***MSG1***, ***MSG2***) TO=***TIMEOUT***

  THR_ID answ [***TAG***] L=***FROM*** err=***ERR.no*** \
  (***ERR.str***) (***MSG1***,***MSG2***)

Other stuff

- **JS** – adapt debugger size to console size
- You can always press
  - ESC to cancel your current input
  - **g** to continue execution
- Related Fiasco command line options
  - -serial_esc
  - -jdb_cmd=<cmd> executes a command on startup
    → I mostly use -jdb_cmd=JS
  - -wait tells the kernel to wait at bootup, so you can setup tracing etc. before running your applications

Even More Debugging

- JDB does not give you the full convenience of GDB-style debugging
- But: Qemu comes with a GDB server stub!
  - Launch Qemu with the `-s` option
  - GDB stub available through TCP on port 1234 by default
  - Connect from GDB using
    `target remote localhost:1234`
- Some issues to keep in mind:
  - You will end up stepping through kernel code with no debug information (EIP > 0xC0000000)
  - You might not be viewing the address space for your binary (Qemu won't tell you)
  - Don't even try to switch binaries while GDB is running. :(

Next meeting

- All assignments from now on due **Mar 31st 2014**
- Next meeting: Jan 6th 2015
- Topic: graphical console