TECHNISCHE UNIVERSITÄT DRESDEN

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

Björn Döbel

Microkernel-Based Operating Systems

Exercise 3: Capability-based systems

Before we start

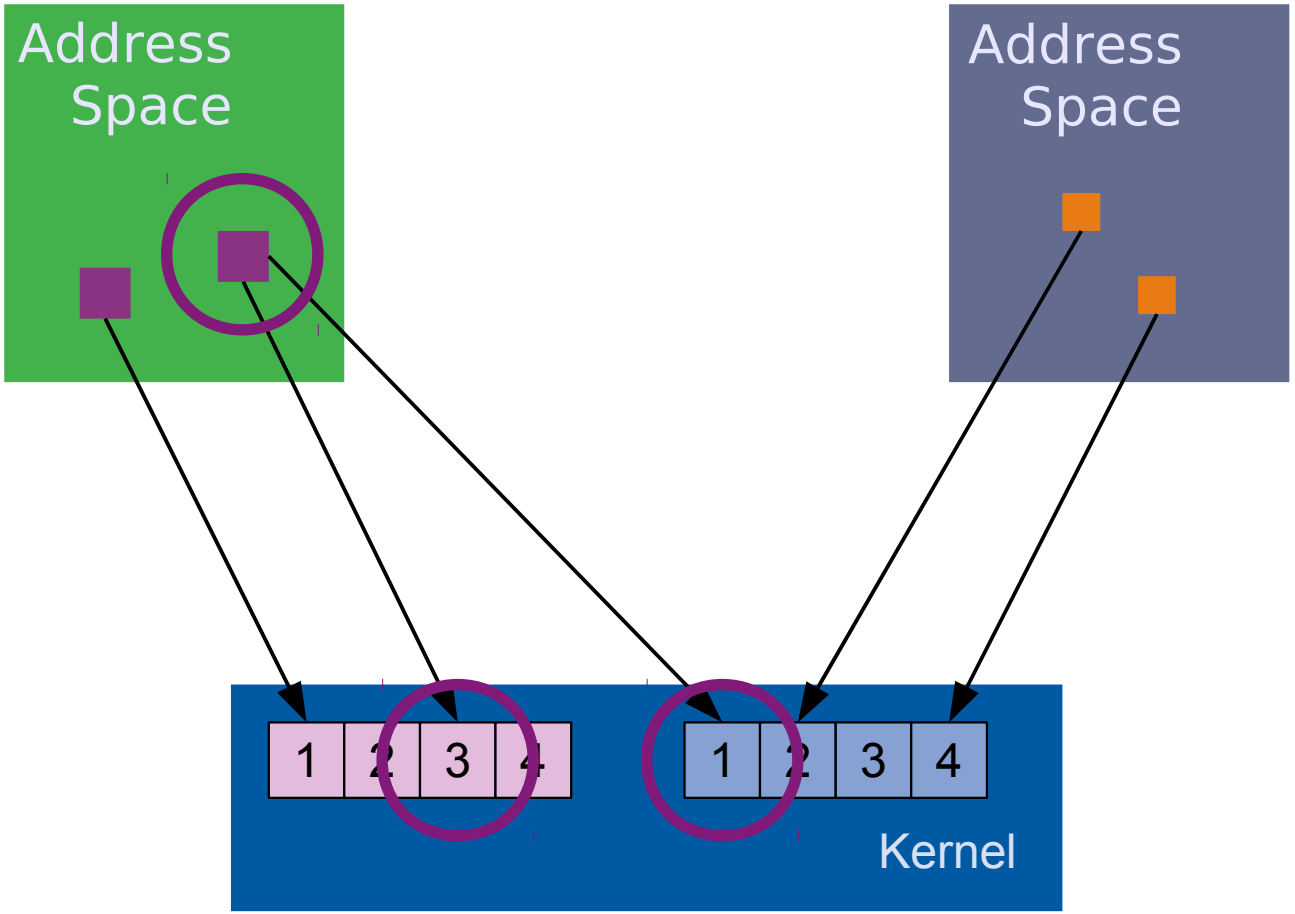
- Yet another build...

```
$> wget
http://os.inf.tu-dresden.de/Studium/KMB/WS2010/exercise3.tar.bz2

$> tar xjf exercise3.tar.bz2

$> cd mos3

$> make setup

$> make
```

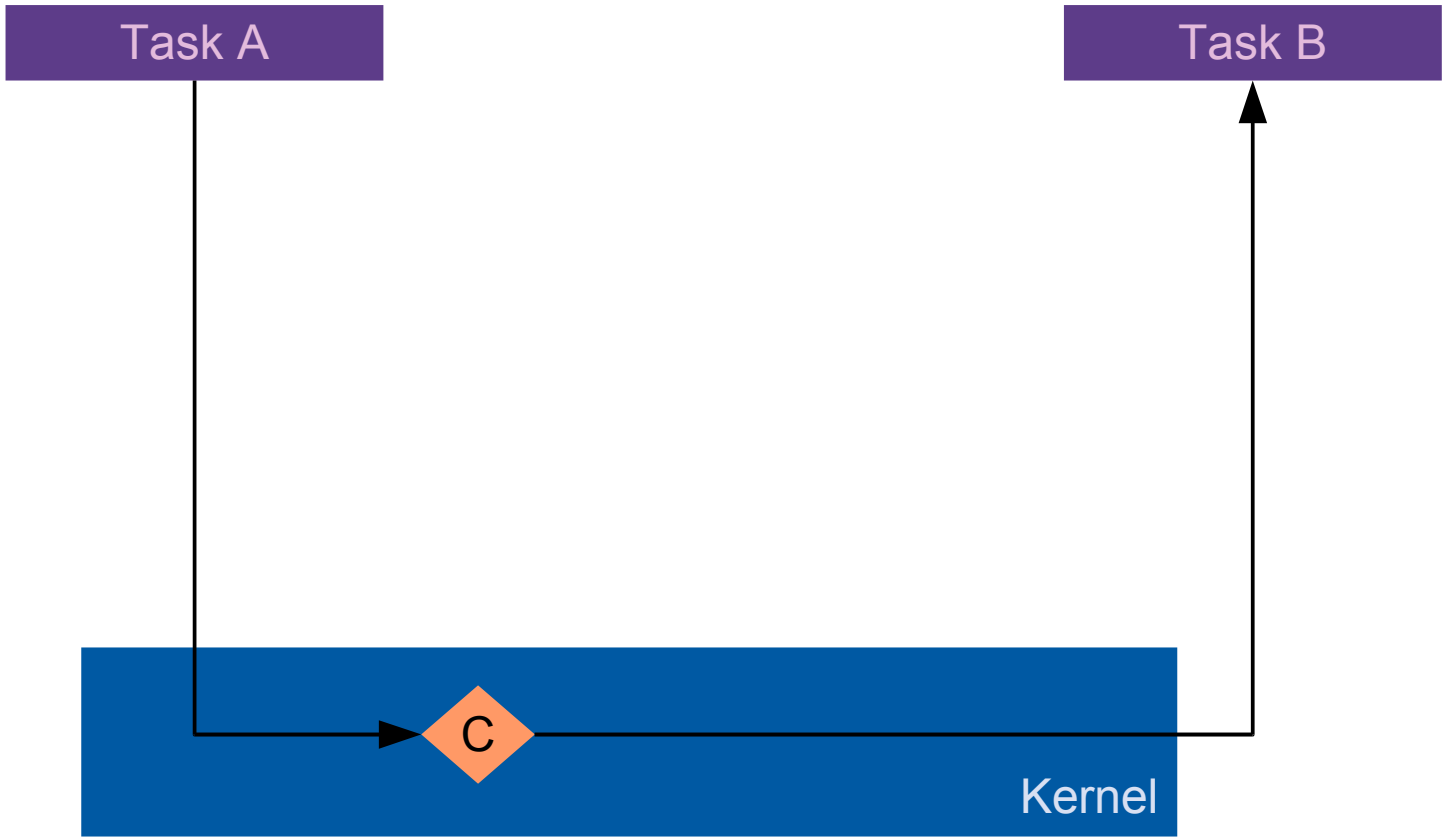
Capabilities

- Kernel-protected
  - Per-task capability table
  - Mapped using IPC
- Security
  - Local name spaces
  - Explicit delegation of communication rights
- Flexibility
  - *"Every problem in computer science can be solved by adding another layer of indirection."*
    (David Wheeler – not David A. Wheeler!)
    "... except the problem of having too many layers of indirection."

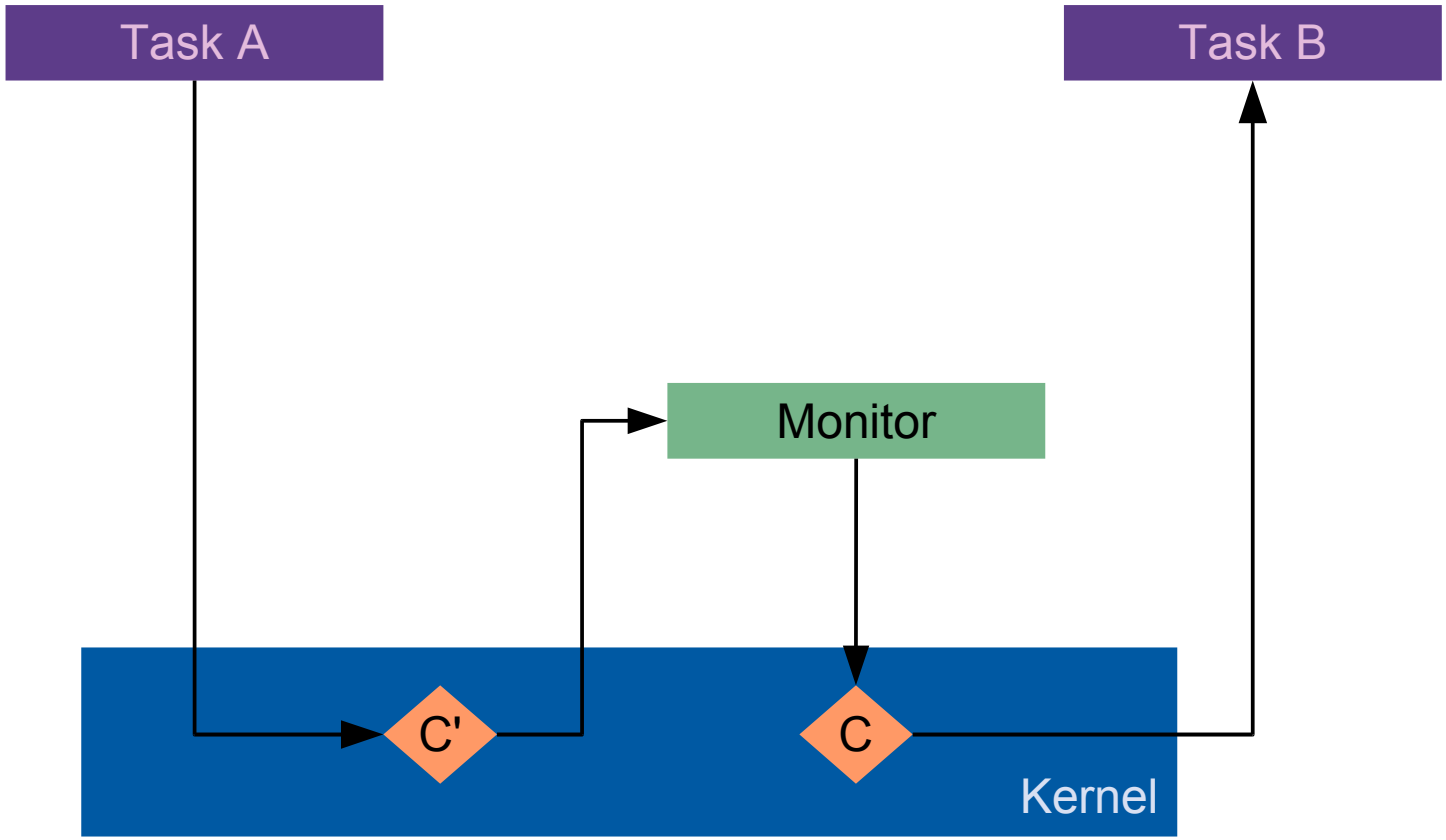
Local Names



Flexibility through indirection

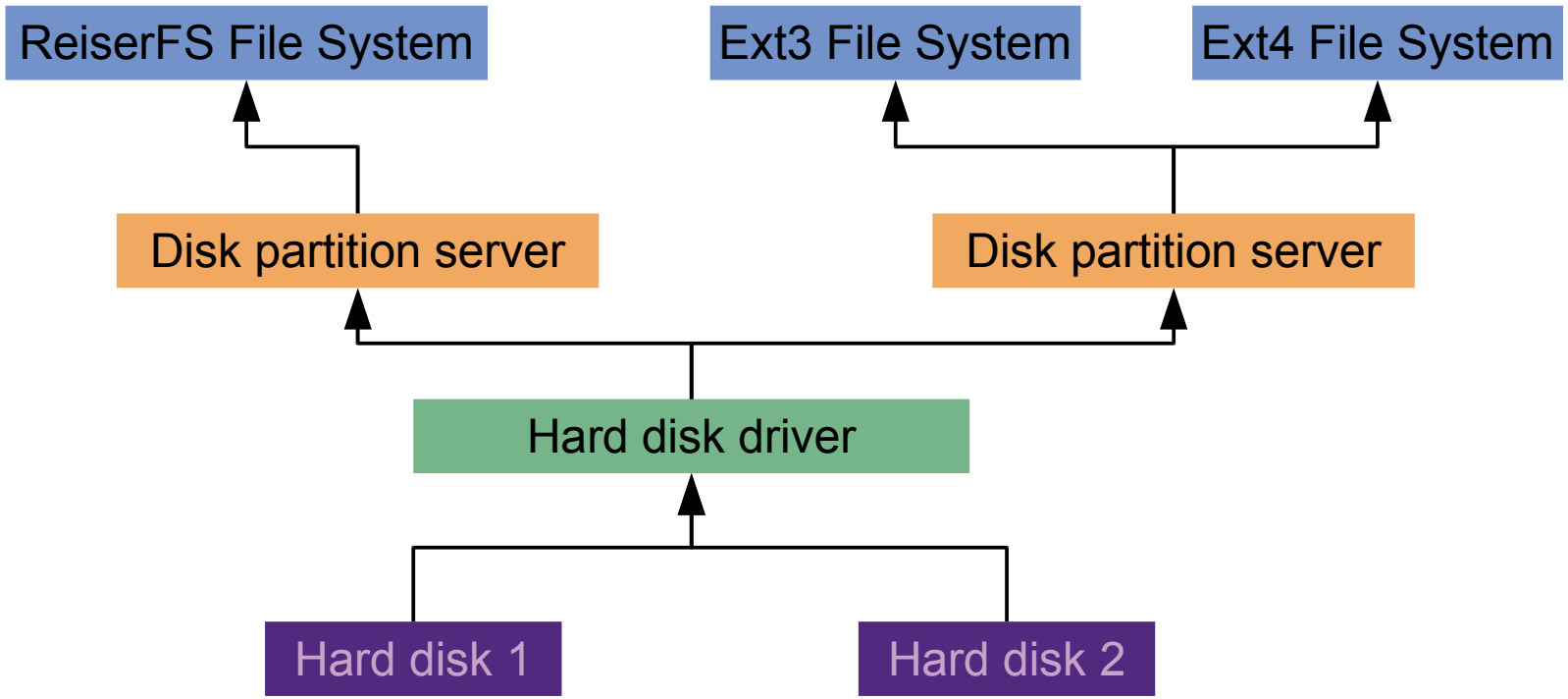


Flexibility through indirection



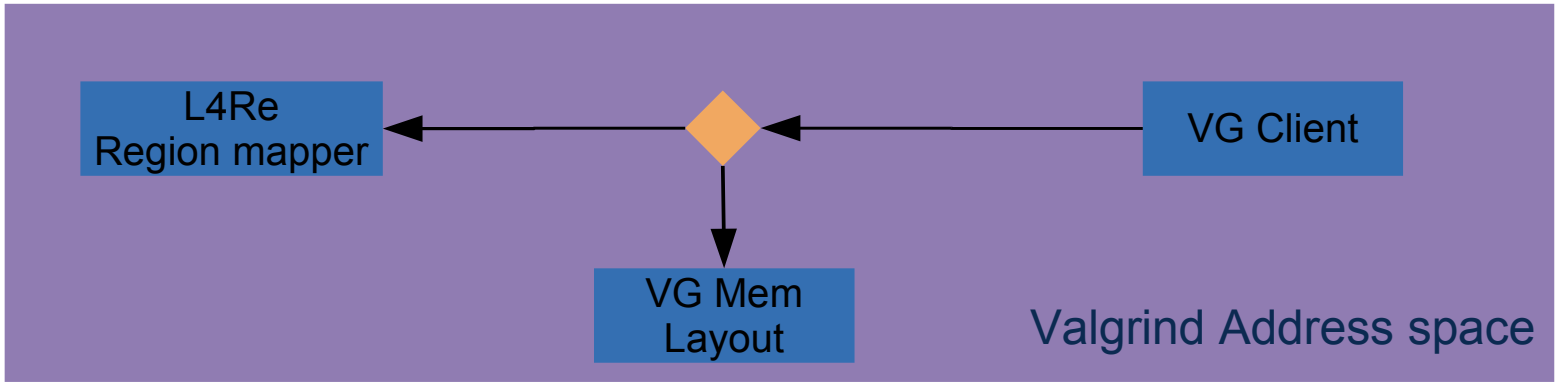
Indirection: Use Cases

- Communication tracing
- Stackable security policies



Not so obvious: Valgrind

- Dynamic binary analysis framework
  - Memory leak detector
  - Keeps track of app's memory layout
    - In Linux
      - Initially parse /proc/self/maps
      - Monitor all mmap() system calls
    - In L4Re:
      - Memory can come through arbitrary IPC
        - » Check all system calls for mappings → inefficient



Virtualizing objects

- L4Re comprises objects
  - IRQs
  - Data spaces
  - Log facility
  - …
- Referenced through capabilities
- Interception works by replacing objects with a redirector object: IPC gate

Example: Hello world

```
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

results in invocation of the Log capability:

```
puts() → putc_unlocked()
       → __stdio_wcommit()
       → __stdio_WRITE()
       → write()
       → default_stdout_ops::write()
       → L4Re::Log::printn(...)
       → l4_vcon_write(...)
```

Starting hello on L4Re

- L4Re is configured through Lua scripts
- Starting hello (mos3/src/l4/conf/hello.lua):

```
require("L4");
local ldr = L4.default_loader;
ldr:start( { }, "rom/hello" );
```

- Go to mos3/obj/l4/x86

- Run

```
$> make qemu E=hello
```

Virtualizing the LOG interface

- Required:
  - An own implementation of a LOG server
    - mos3/src/l4/pkg/logger
  - A communication channel for the logger created in the Lua startup script:

```
local log_chan = ldr:new_channel();

ldr:start( { caps =
             { logger = log_chan:svr() } },
           "rom/logger" );
```

  - An environment for "hello" to use the new log channel

```
local env = L4.App_env.new();
env.log = log_chan;
ldr:startv(env, "rom/hello");
```

  - See mos3/src/l4/conf/hello-complex.lua

What do we need to do?

- Edit mos3/src/l4/pkg/logger/server/src/main.cc
  - Implement the dispatch function
    - Log Message is completely in the UTCB message registers
    - mr[0] → opcode (should be 0 for log::write)
    - mr[1] → length of message in characters
    - mr[2...] → message
  - First take:
    - Simply implement a server that prints messages once it receives them.
  - Later:
    - Add extended functionality

Extending the LOG server

- Some ideas:
  - Keyword highlighting
    - Keep a list of keywords
    - When a keyword is encountered in a log message, make it appear bold or colored
  - Text formatting
    - Support log messages containing some markup
      - e.g., a subset of HTML
      - Or some own sequences for making things colored or bold
  - Log indexing
    - Keep track of log messages
    - Add a function that allows to re-print all messages that contain a certain word
    - Support regular expression queries