TECHNISCHE UNIVERSITÄT DRESDEN

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

Introduction to Microkernel-Based Operating Systems

Björn Döbel

Lecture Outline

- Microkernels and what we like about them

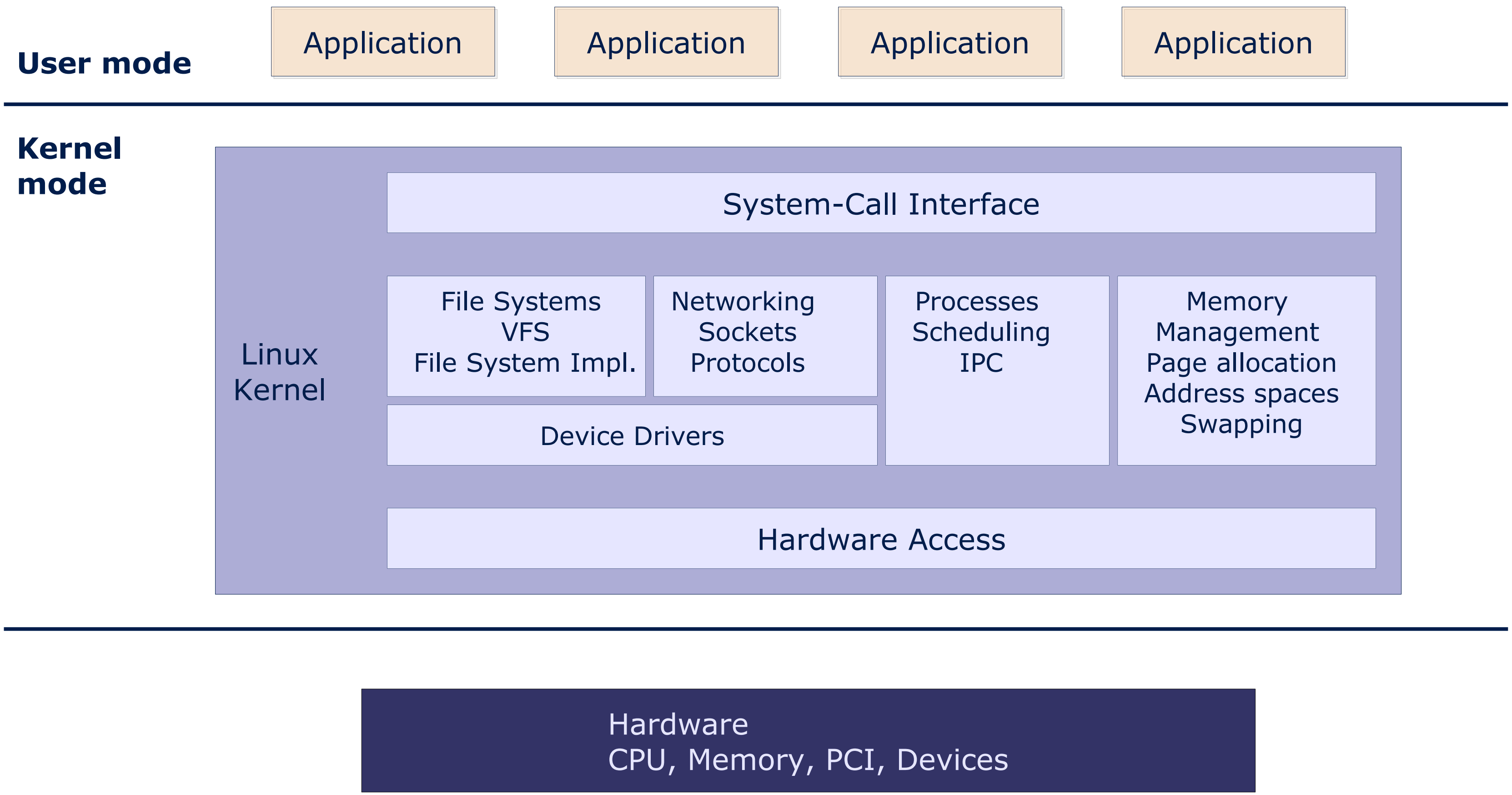
- The Fiasco.OC microkernel
  - Kernel Objects
  - Kernel Mechanisms

- OS Services on top of Fiasco.OC
  - Device Drivers
  - Virtualization

Purpose of Operating Systems

- Manage the available resources
  - Hardware (CPU) and software (file systems)
- Provide users with an easier-to-use interface to access resources
  - Unix: data read/write access to sockets instead of writing TCP packets on your own
- Perform privileged / HW-specific operations
  - x86: ring0 vs. ring3
  - Device drivers
- Provide separation and collaboration
  - Isolate users / processes from each other
  - Allow cooperation if needed (e.g., sending messages between processes)

Monolithic kernels - Linux



What's the problem?

- Security issues
  - All components run in privileged mode.
  - Direct access to all kernel-level data.
  - Module loading → easy living for rootkits.
- Resilience issues
  - Faulty drivers can crash the whole system.
  - 75% of today's OS kernels are drivers.
- Software-level issues
  - Complexity is hard to manage.
  - Custom OS for hardware with scarce resources?

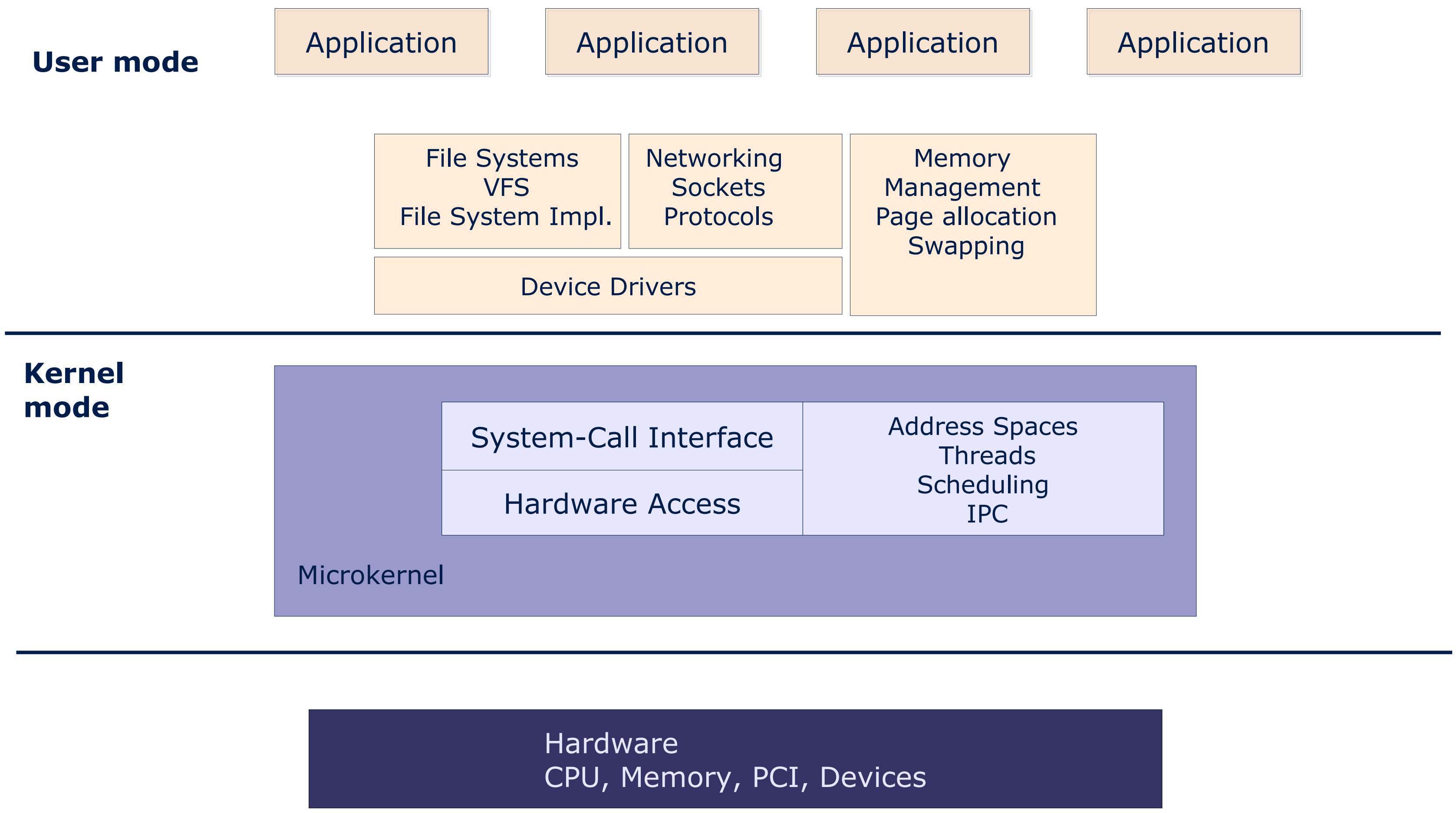
One vision - microkernels

- Minimal OS kernel
  - less error prone
  - small *Trusted Computing Base*
  - suitable for verification

- System services in user-level *servers*
  - flexible and extensible

- Protection between individual components
  - systems get
    - More secure – inter-component protection
    - More resilient – crashing component does not (necessarily...) crash the whole system

The microkernel vision



Microkernel-Based Systems

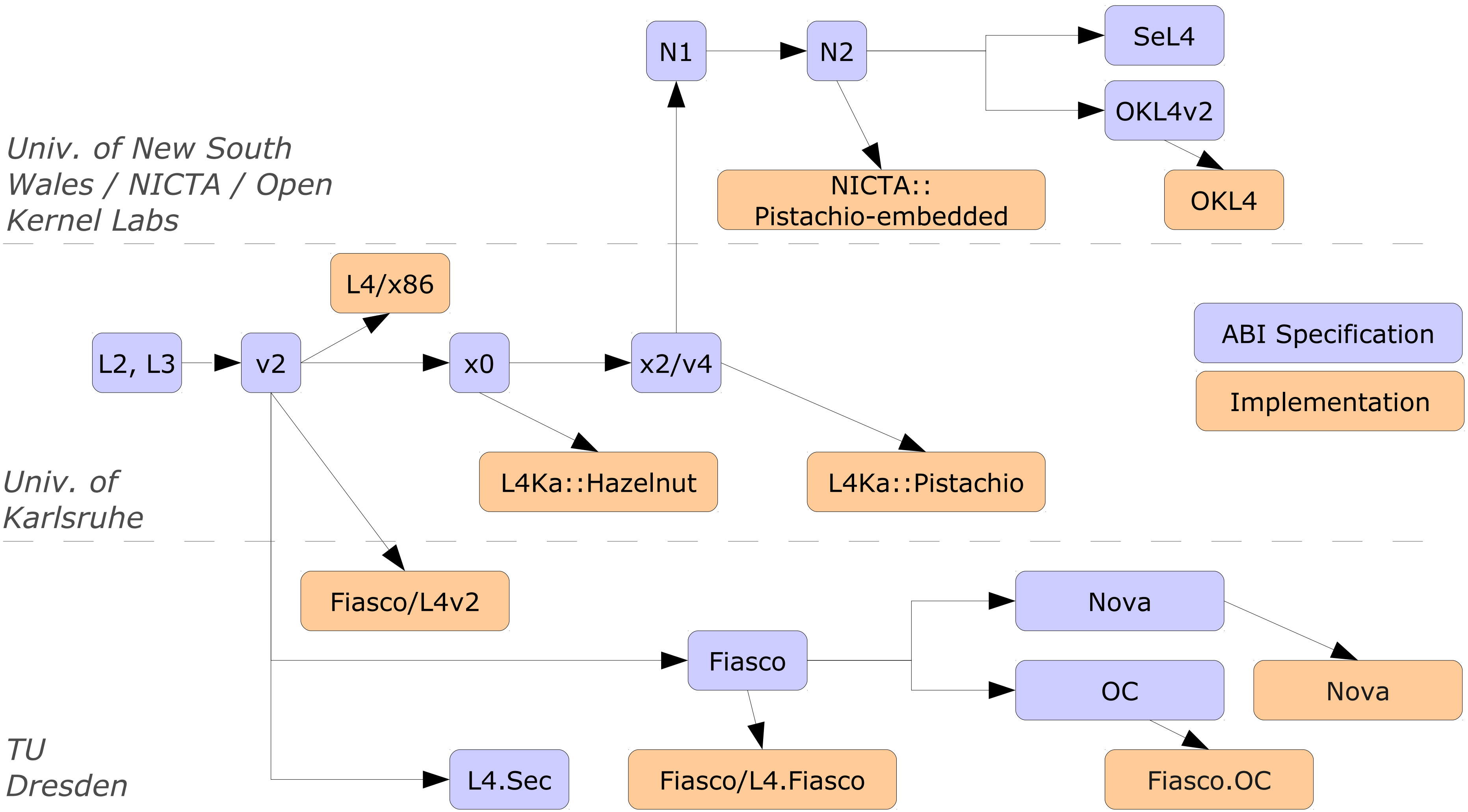
- 1st generation: Mach
  - developed at CMU, 1985 - 1994
  - Foundation for several real systems (e.g., NextOS → Mac OS X)

- 2nd generation: Minix3
  - Andrew Tanenbaum @ VU Amsterdam
  - Focus on restartability

- 2nd/3rd generation:
  - Various kernels of the L4 microkernel family

TECHNISCHE UNIVERSITÄT DRESDEN

The L4 family – a timeline



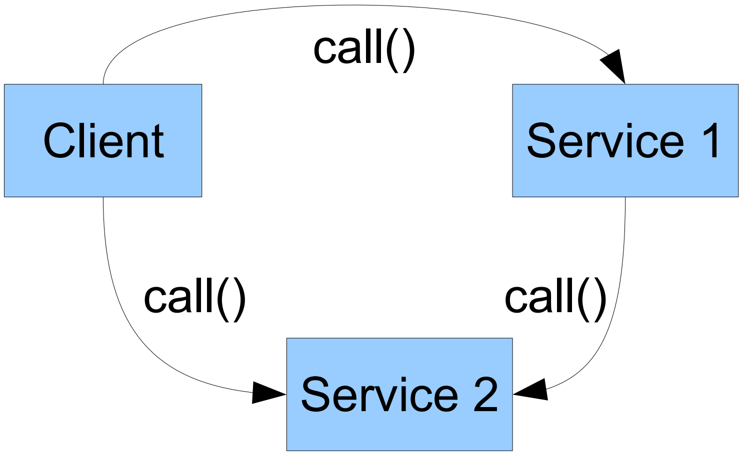
L4 concepts

- Jochen Liedtke:
  "A microkernel does no real work."
  - Kernel only provides inevitable mechanisms.
  - Kernel does not enforce policies.

- But what is inevitable?
  - Abstractions
    - Threads
    - Address spaces (tasks)
  - Mechanisms
    - Communication
    - Resource Mapping
    - (Scheduling)

Fiasco.OC – Objects

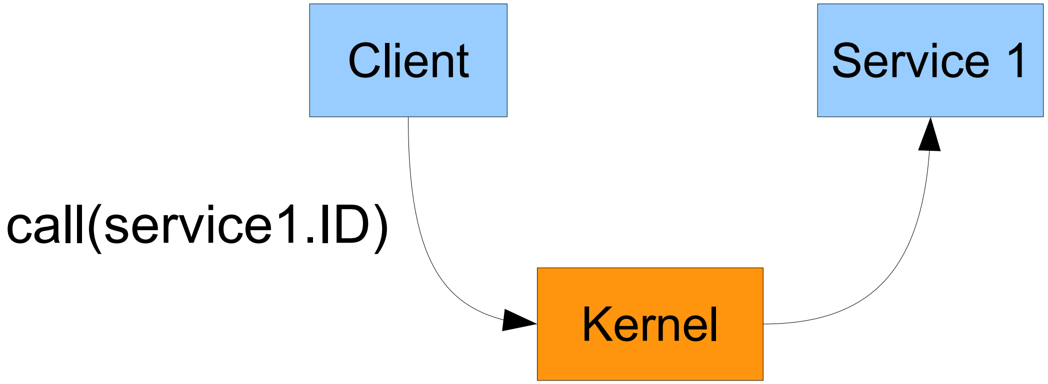
- OC – Object-Capability system
- System designed around objects providing services:



- Kernel provides
  - Object creation/management
  - Object interaction: Inter-Process Communication (IPC)

Fiasco.OC – Calling objects

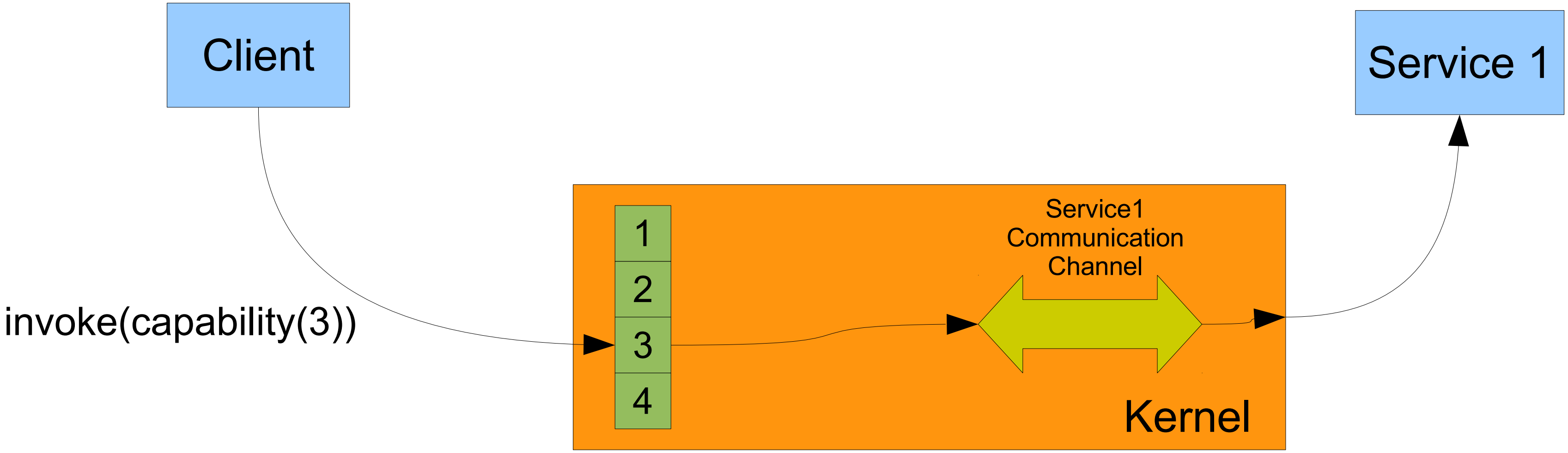
- To call an object, we need an address:
  - Telephone number
  - Postal address
  - IP address



- Kernel returns ENOTEXISTENT if ID is wrong.
- Security issues:
  - Client could simply “guess” IDs brute-force.
  - Existence/non-existence can be used as a covert channel

Fiasco.OC – Capabilities

- Capability:
  - Reference to an object
  - Protected by the Fiasco.OC kernel
    - Kernel knows all capability-object mappings.
    - Managed as a per-process capability table.
    - User processes only use indexes into this table.



Fiasco.OC: System Calls

- "Everything is an object."

- 1 system call: *invoke_object()*
  - Parameters passed in UTCB
  - Types of parameters depend on type of object

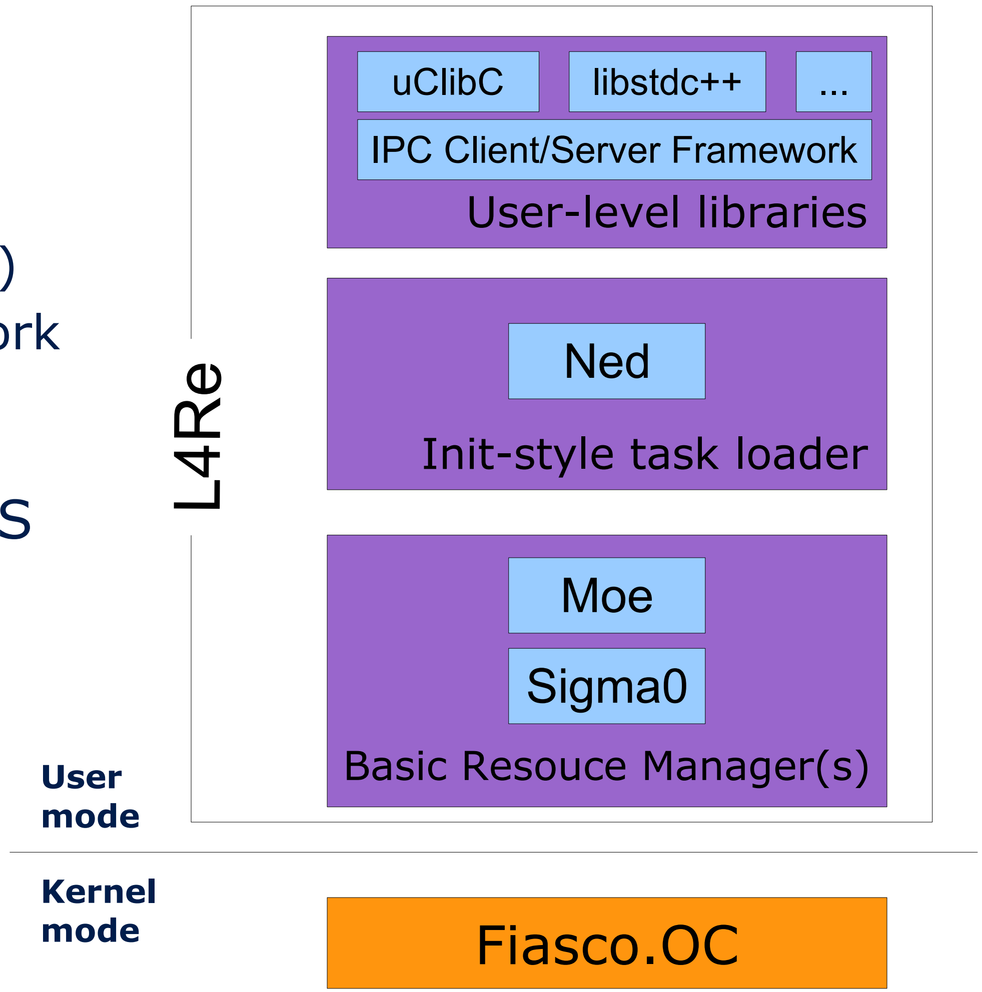
- Kernel-provided objects
  - Threads / Tasks / IRQs / …

- Generic communication object: IPC gate
  - Send message from sender to receiver
  - Used to implement new objects in user-level applications

Kernel vs. Operating System

- Fiasco.OC is not a full operating system!
  - No device drivers (except UART + timer)
  - No file system / network stack / …
- A microkernel-based OS needs to add these services as user-level components

→ L4 Runtime Environment (L4Re)



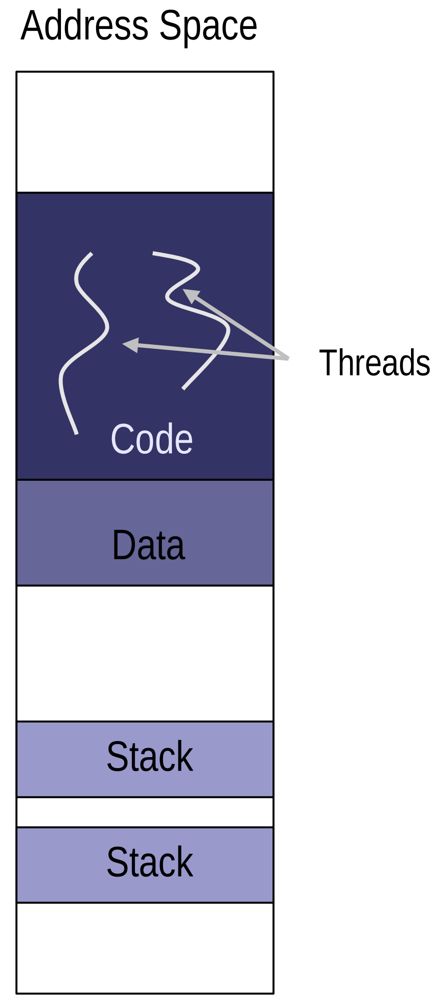
Outline for the Next Lectures

- Fiasco.OC's mapping from managed resources to kernel objects:
  - CPU → threads
  - Memory → tasks (address spaces)
  - Communication → Inter-Process Communication (IPC)

- L4 Runtime Environment
  - Device Drivers
  - L^4Linux

L4 - Threads

- Thread ::= abstraction of execution
  - Unit of CPU scheduling
  - Threads are temporally isolated
- Properties managed by the kernel:
  - Instruction Pointer (EIP)
  - Stack Pointer (ESP)
  - CPU Registers / flags
  - (User-level) TCB
- User-level applications need to
  - allocate stack memory
  - provide memory for application binary
  - find entry point
  - ...



L4 Threads and the Kernel

- Threads run in userland and enter the kernel
  - Through a system call (sysenter / INT 0x30)
  - Forced by HW interrupts or CPU exceptions

- Kernel Info Page
  - Magic memory page mapped into every task
  - Contains kernel-related information
    - Kernel version
    - Configured kernel features
    - System call entry code (allows the kernel to decide whether sysenter or INT 0x30 are better for a specific platform)

Thread Control Block (TCB)

- Kernel storage for thread-related information
- One TCB per thread
- Stores user state while thread is inactive
- Extension: User-level Thread Control Block (UTCB)
  - Holds data the kernel does not need to trust
  - Mapped into address space
  - Most prominent use: system call parameters

Thread Scheduling

- Whenever a thread enters the kernel, a scheduling decision is made.

- Fiasco.OC: priority- based round-robbin
  - Every thread has a priority assigned.
  - The thread with the highest priority runs until
    - Its time quantum runs out (timer interrupt),
    - Thread blocks (e.g., in a system call), or
    - A higher-priority thread becomes ready
  - Then, the next thread is selected.

L4Re and Threads

- Fiasco provides thread-related system calls
  - `thread_control` → modify properties
  - `thread_stats_time` → get thread runtime
  - `thread_ex_regs` → modify EIP and ESP

- But most L4Re applications don't need to bother:
  - L4Re provides full libpthread including
    - `pthread_create`
    - `pthread_mutex_*`
    - `pthread_cond_*`
    - `...`

L4Re Applications

- Every L4Re application starts with
  - An empty address space
    - Memory managed by parent
  - One initial thread
    - EIP set to binary's entry point by ELF loader
  - An initial set of capabilities – the **environment**
    - Parent
    - Memory allocator
    - Main thread
    - Log
    - ...

Performing System Calls

- All Fiasco.OC system calls are performed using IPC with different sets of parameters.
  - Functions are called l4_ipc_*()
  - Colloquially: *invoke*
- Generic parameters (in registers):
  - Capability to invoke
  - Timeout (how long do I want to block at most? – let's assume L4_IPC_NEVER for now.)
  - Message tag describing the rest of the message
    - Protocol
    - Number of words in UTCB
- Message-specific parameters in UTCB message registers

Writing Output

- L4Re environment passes a LOG capability
  - Implements the L4_PROTO_LOG protocol
    - By default implemented in kernel and printed out to serial console
  - UTCB content:
    - Message reg 0: log operation to perform (e.g., L4_VCON_WRITE_OP)
    - Message reg 1: number of characters
    - Message reg 2...: characters to write

Writing Output: The Code

```
#include <l4/re/env.h>
#include <l4/sys/ipc.h>

[..]

l4re_env_t *env    = l4re_env();    // get environment
l4_msg_regs_t *mr = l4_utcb_mr(); // get msg regs

mr->mr[0] = L4_VCON_WRITE_OP;
mr->mr[1] = 7; // 'hello\n' = 6 chars + \0 char
memcpy(&mr->mr[2], “hello\n”, 7);

l4_msgtag_t tag, ret;
tag = l4_msgtag(L4_PROTO_LOG, 4, /* 4 msg words /
                     0, L4_IPC_NEVER);

ret = l4_ipc_send(env->log, l4_utcb(), tag); // System Call!

if (l4_msgtag_has_error(ret)) {
  /* error handling */
}
```

Writing Output: The Code

```
#include <l4/re/env.h>
#include <l4/sys/ipc.h>

[..]

l4re_env_t *env  = l4re_env();    // get environment
l4_msg_regs_t mr = l4_utcb_mr(); // get msg regs

mr->mr[0] = L4_VCON_WRITE_OP;
mr->mr[1] = 7; // 'hello\n' = 6 chars + \0 char
memcpy(&mr->mr[2], “hello\n”, 7);

l4_msgtag_t tag, ret;
tag = l4_msgtag(L4_PROTO_LOG, 4, /* 4 msg words /
                0, 0, L4_IPC_NEVER);

ret = l4_ipc_send(env->log, l4_utcb(), tag); // System Call!

if (l4_msgtag_has_error(ret)) {
  /* error handling */
}
```

In real code, please just do

```
puts(“hello”);
```

Multithreading

- Fiasco.OC allows multithreading
  - Many threads sharing the same address space
  - Spread across multiple physical CPUs

- Classical Problem: critical sections

global: `int i = 0;`

Thread 1

```
for (unsigned j = 0; j < 10;
     ++j)
   i += 1;
```

Thread 2

```
for (unsigned j = 0; j < 10;
     ++j)
       i += 1;
```

- The result is rarely i == 20!

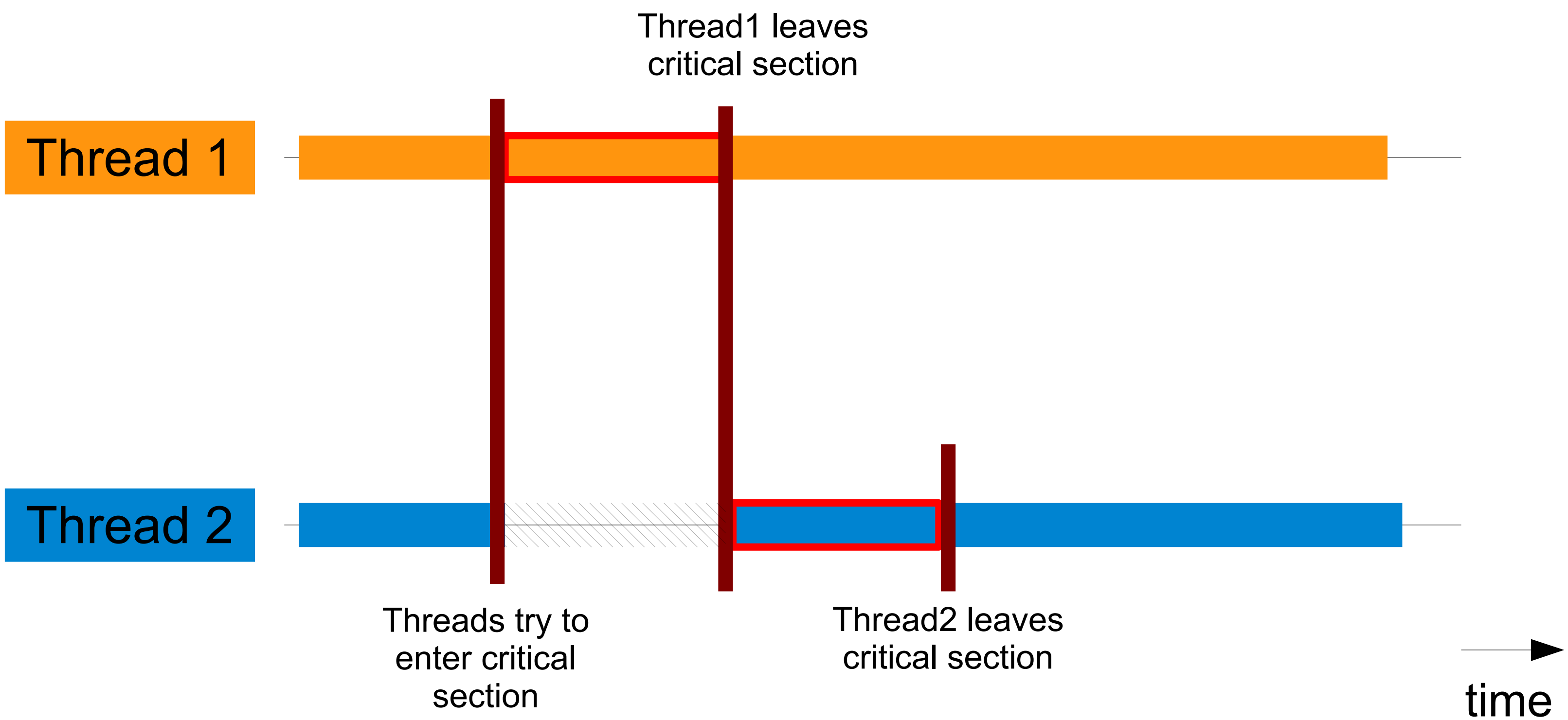
Synchronization

```
for (unsigned j = 0; j < 10; ++j)
        i += 1;
```

Critical Section

- Critical Sections need to be protected
  - Disable interrupts → infeasible for user space
  - Spinning → burns CPU / energy / time quanta
- What we want: blocking lock
  - Thread tests flag: critical section free yes/no
  - waits (sleeping) until section is free

Expected behavior



Synchronization - pthreads

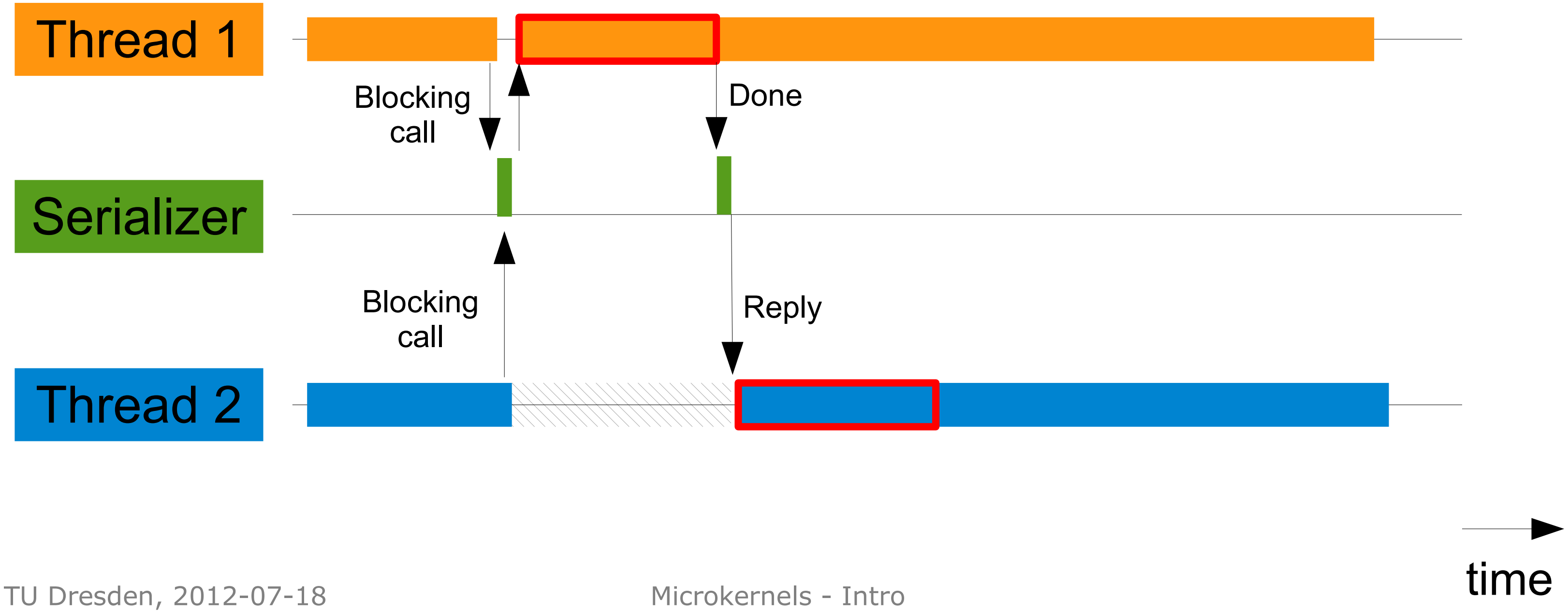
- L4Re provides libpthread, so we can simply use pthread_mutex operations:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

[..]

for (unsigned j = 0; j < 10; ++j) {
    pthread_mutex_lock(&mtx);
    i += 1;
    pthread_mutex_unlock(&mtx);
}
```

- Fiasco.OC's IPC primitives allow for another solution, though.

Synchronization: Serializer Thread

- IPC operations are synchronous by default:
  - Sender and receiver both need to be in an IPC system call
- There's a combination of sending and receiving a message: `l4_ipc_call()`.
- This allows synchronization using a serializer thread:



Downloading and Compiling

- Fiasco.OC and L4Re are available from http://os.inf.tu-dresden.de/L4Re

- There are download and build instructions.
  - We will use the 32bit versions for this course
    → simply leave all configuration settings at their defaults
  - Note, you have to do 2 separate builds: one for Fiasco.OC and one for the L4Re.
  - GCC-4.7 did not work for me at the moment.

L4Re directory structure

- src/l4
- Important subdirectories: pkg/, conf/
- pkg/contains all applications (each in its own package)
  - Packages have subdirs again:
    - server/ → the application program
    - lib/ → library to be used by clients
    - include/ → header files shared between server and clients

Running Fiasco.OC/L4Re

- We'll use QEMU to run our setups.
- L4Re's build system has QEMU support integrated, which is configured through files in src/l4/conf:
  - modules.lst → contains multiboot setup info, similar to a GRUB menu.lst
  - Makeconf.boot → contains overall settings (where to search for binaries, qemu, ...)

modules.lst

```
modaddr 0x01100000
```

Have this once in your modules.lst file.

Each entry has a name

```
entry hello
roottask moe --init=rom/hello
module l4re
module hello
```

roottask is the initial task to boot. --init rom/hello asks it to load the hello binary from the ROM file system

modules are additional files. They are loaded into memory and can then be accessed through the ROM file system under the name rom/<filename>.

Makeconf.boot

- Start from the example in src/l4/conf (rename it to Makeconf.boot)

- At least set:
  - MODULE_SEARCH_PATH (have it include the path to your Fiasco.OC build directory)

Booting QEMU

- Go to L4Re build directory
- Run “make qemu”
  - Select 'hello' entry from the dialog
    - If there's no dialog, you need to install the 'dialog' package.
    - You can also circument the dialog:
      `make qemu E=<entry>`
      where entry is the name of a modules.lst entry.

Assignments

- Download and compile Fiasco.OC and L4Re.
- Run the hello world example in QEMU.
- Modify the hello world example (it is in l4/pkg/hello/server/src):
  - Replace the puts() call with a manual system call to the log object.
  - You can use the example code from these slides.

Further Reading

- P. Brinch-Hansen: *The Nucleus of a Multiprogramming System*
  http://brinch-hansen.net/papers/1970a.pdf
  Microkernels were invented in 1969!

- J. Liedtke: *On microkernel construction*
  http://os.inf.tu-dresden.de/papers_ps/jochen/Mikern.ps
  Shaping the ideas found in L4 microkernels.

- D. Engler et al.: *Exokernel – An operating system architecture for application-level resource management*
  http://pdos.csail.mit.edu/6.828/2008/readings/engler95exokernel.pdf
  Taking user-level policy implementation to the extreme.