



Björn Döbel

Microkernel-Based Operating Systems

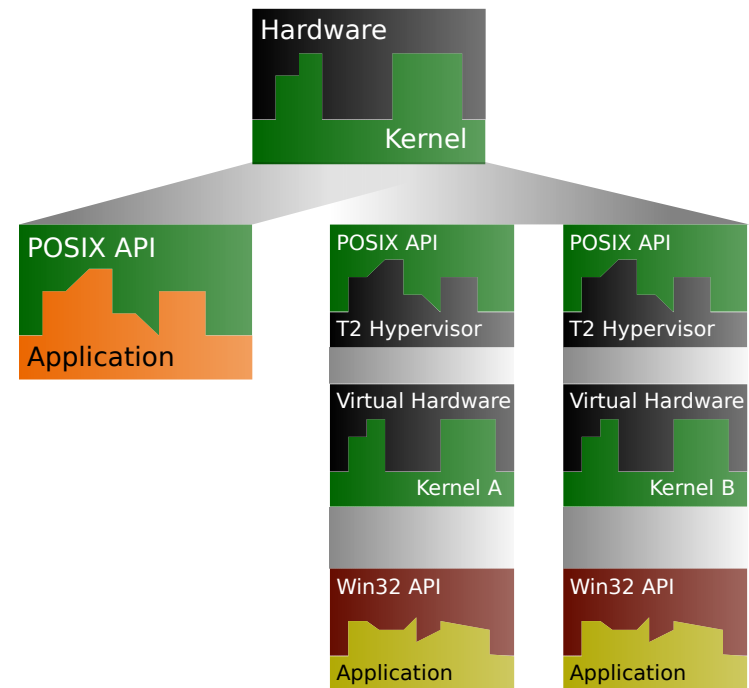
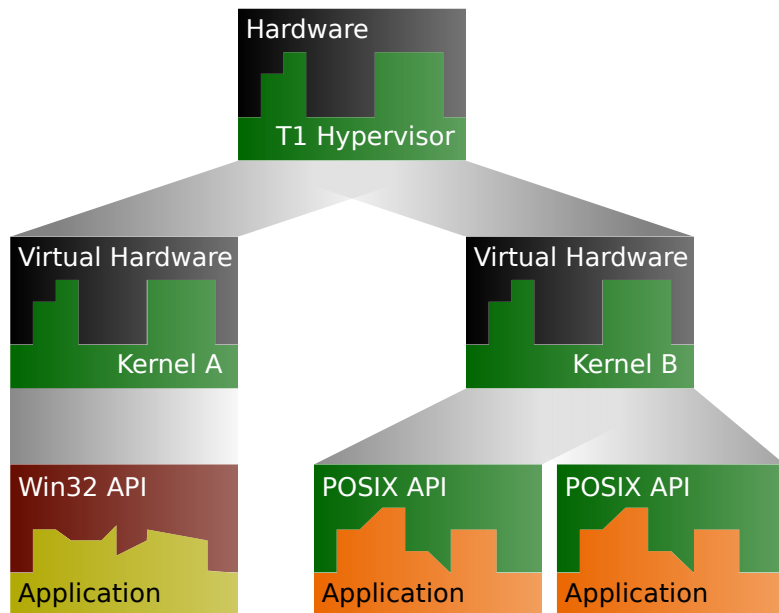
Exercise 3: Virtualization

- Vast amount of computing resources in modern machines
 - Increase utilization
 - Consolidate services
- Strong architectural isolation
 - System partitioning
 - Architectural protection
 - Non-intrusive monitoring
- Support for old (legacy) software
 - Will your grandsons be able to watch your crazy “Let's play Minecraft” videos?

- **Emulation / Simulation**
 - Complete software implementation of a hardware platform
 - Includes emulated processor, devices, ...
 - Slow: everything is done in software
 - Flexible: can emulate any architecture on any host
- **Virtualization**
 - Exploit real hardware if possible
 - Device pass-through
 - Instruction set (ISA) requirements:
 - Non-sensitive instructions run on native hardware
 - Security-sensitive instructions need to be emulated

- **Virtual Machine Monitor**

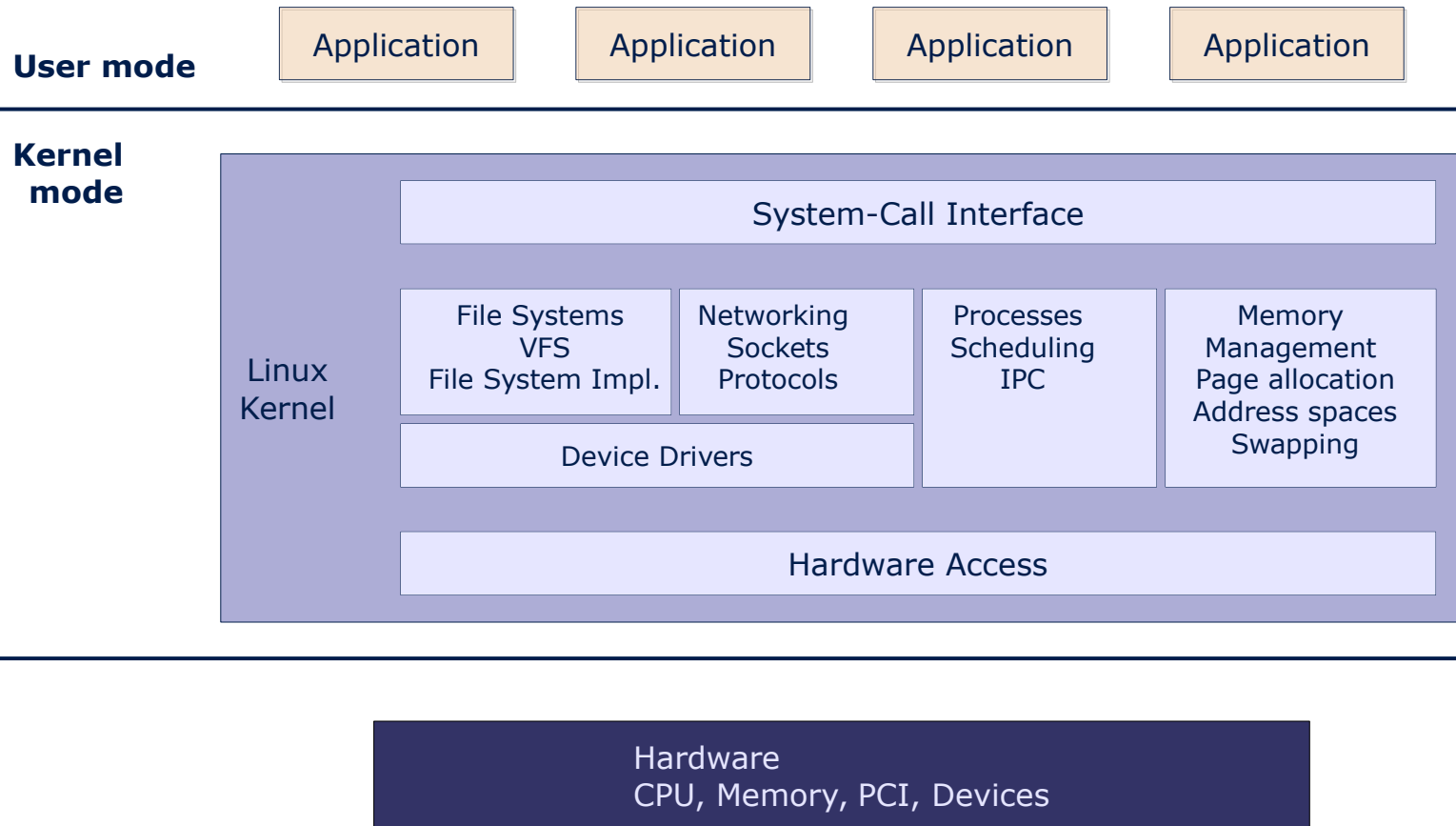
- Manages virtualized guest(s)
- Performs emulation of sensitive instructions
- Multiplexes devices

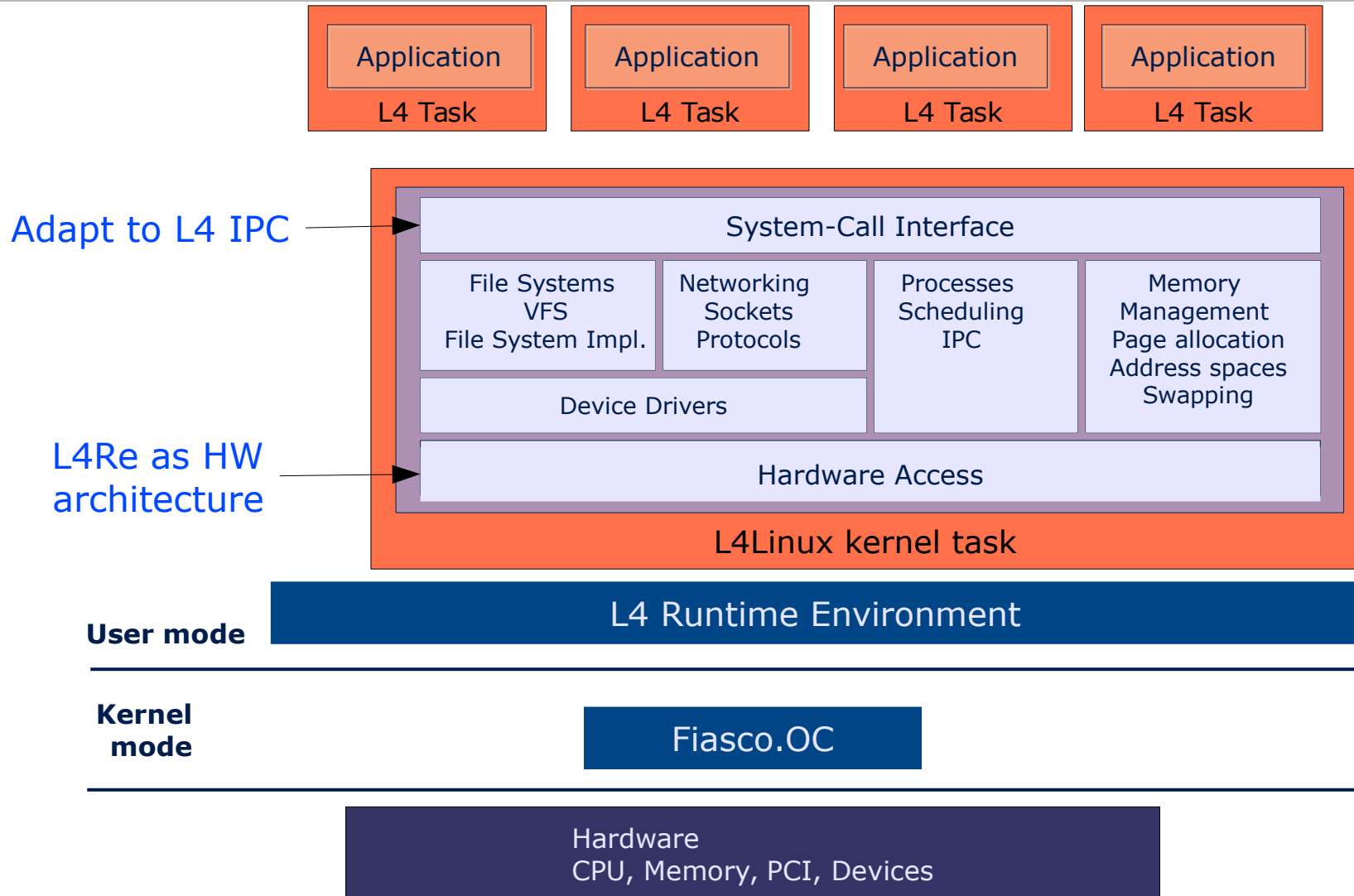


- x86 is not easy to virtualize:
 - Not all privileged instructions cause a trap
 - Example: `popf`
- Approach 1: Binary Rewriting
 - Scan kernel binary for critical instructions
 - Recompile instructions to trap
 - Examples: early VMware products
 - Optimizations: combine multiple traps into one
- Approach 2: Faithful virtualization
 - Hardware extensions that allow trapping everything the VMM might want to monitor
 - Examples: Intel VT, AMD V, ARM Virt. Extensions
 - Optimizations: nested paging

- Type 1 hypervisors:
 - Don't bother fighting with HW support
 - (Slightly) Adapt guest OS to directly run on hypervisor interface → paravirtualization
- Benefit: theoretically runs on any platform the hypervisor runs on
 - Practice: need adaptations for specific hardware
- Drawback: requires source code modifications
 - Hard to para-virtualize Windows
 - Constant development effort

Today's example: L⁴Linux





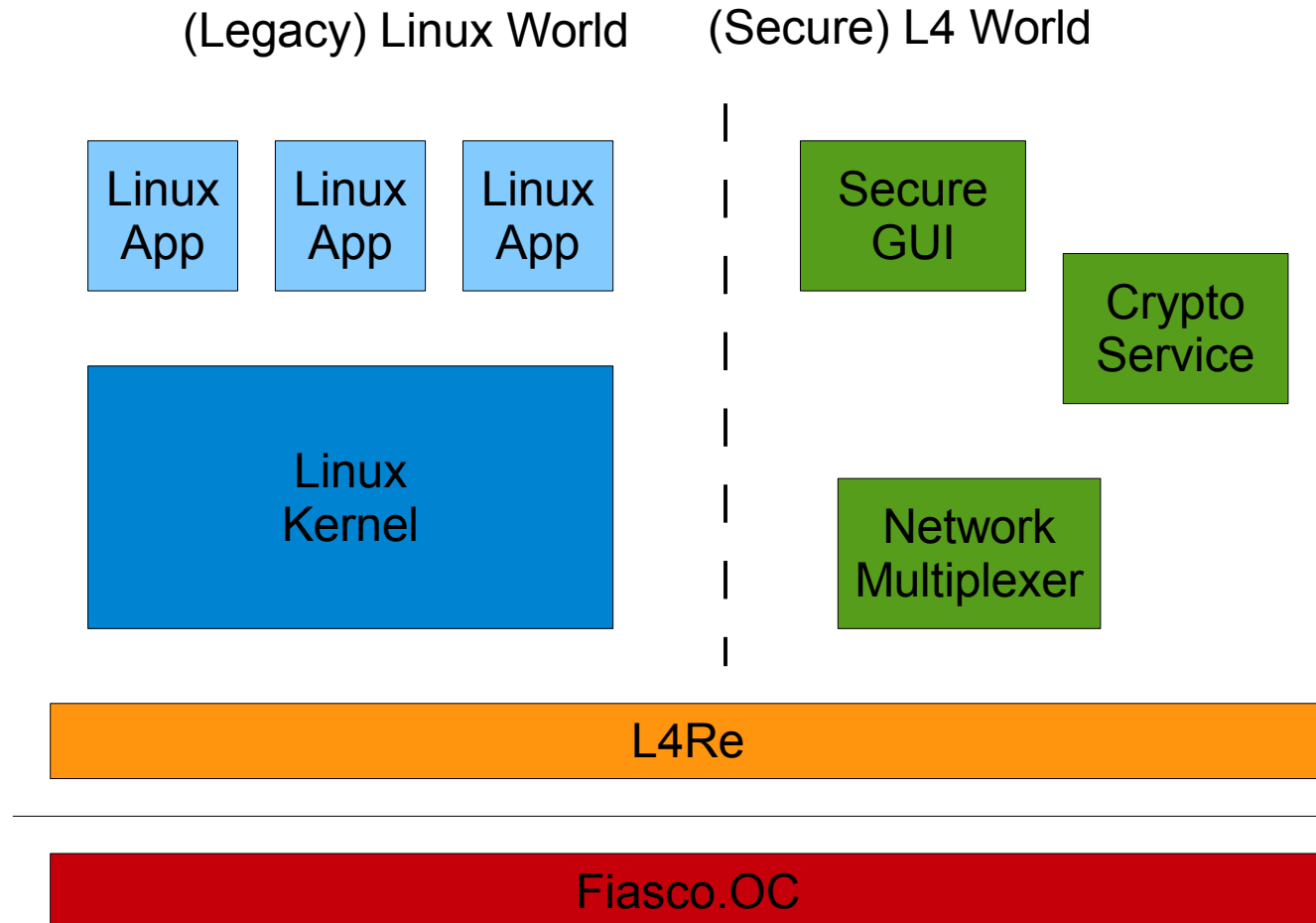
```
$> cd /tmp
$> mkdir l4
$> cd l4
$> mkdir build
$> svn cat
http://svn.tudos.org/repos/oc/tudos/trunk/repomgr | perl
- init http://svn.tudos.org/repos/oc/tudos fiasco l4re
$> cd src/l4/pkg
$> svn up acpica drivers examples fb-drv input io
libevent libio-io libgfortran libirq libvcpu lxfuxlibc
mag mag-gfx rtc shmc x86emu zlib
$> cd ..
$> make O=/tmp/l4/build config
$> make O=/tmp/l4/build -j 8
$> cd ../kernel/fiasco
$> make config && make -j 8
```

```
$> cd /tmp/l4
$> svn co
http://svn.tudos.org/repos/oc/l4linux/trunk l4linux
$> cd l4linux
$> mkdir build
$> wget
http://www.tudos.org/Studium/KMB/WS2013/Exercise3-
L4Linux.config -O build/.config
$> make O=build oldconfig
$> make O=build -j 8

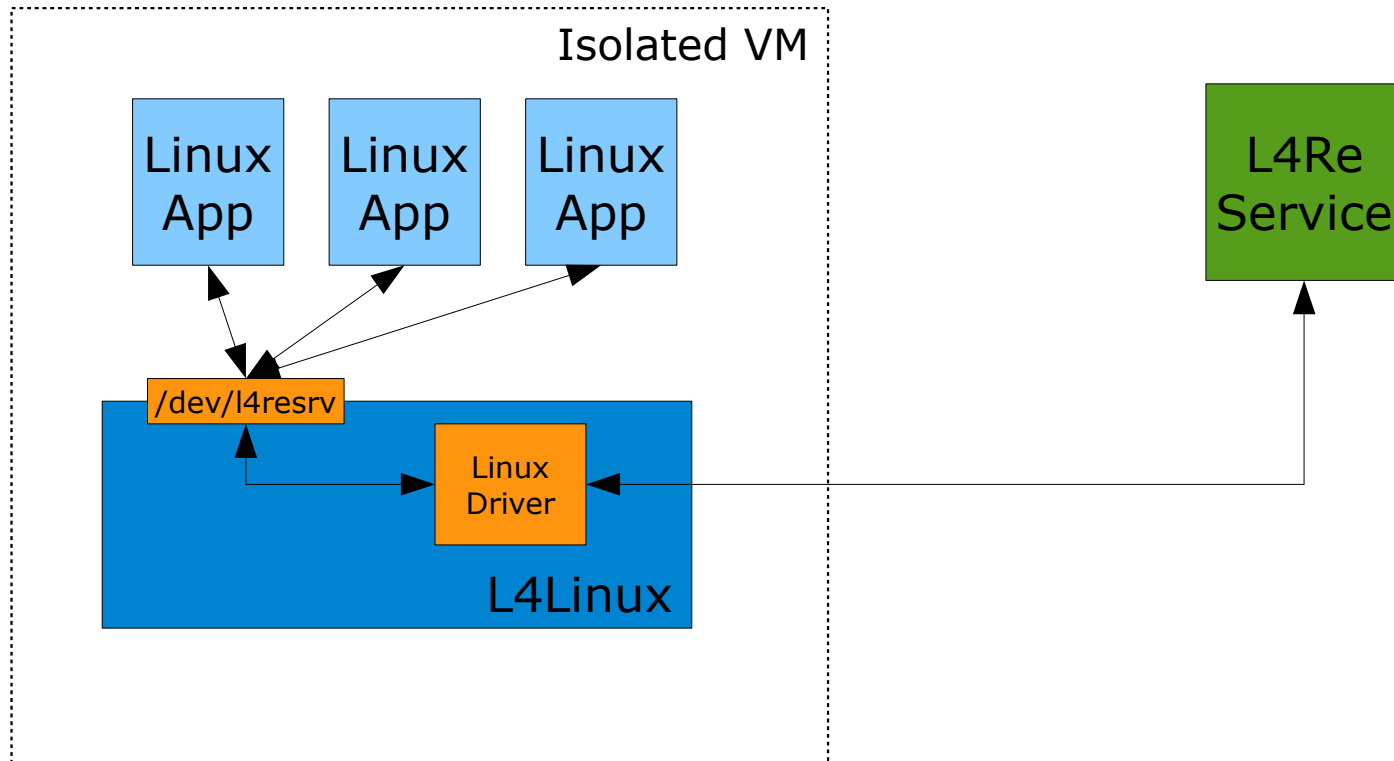
# and get a RAMdisk
$> wget http://www.tudos.org/download/ramdisk-
x86.rd
```

- Copy `src/l4/pkg/io/config/x86-legacy.devs` to `src/l4/conf`
- Adjust QEMU config (`src/l4/conf/Makeconf.boot`)
 - `MODULE_SEARCH_PATH` should include
 - Fiasco build dir
 - L4Linux build dir
 - `src/l4/conf` → location of the config scripts
- Then there is already a working example setup:

```
$> cd /tmp/l4/build
$> make qemu E=L4Linux-mag-x86
```



- Linux applications are plain Linux apps
 - Unaware of being run on L4
 - Can not directly perform L4 system calls
- L4Linux kernel is an L4Re app
 - Can perform system calls
- Connecting Linux applications and L4Re servers:
 - Add communication interface to L4Linux kernel (e.g., kernel module + /proc interface)



- Build kernel module as any other LKM
 - See an example e.g., at TLDP.org: *The Linux kernel module programming guide*
 - L4Linux build system: we can use L4Re include files etc.
- How do I get access to this module?
 - On your own computer: mount the L4Linux ramdisk as root and copy the module into it
 - In the computer lab: use L4Linux block device server
 - In `src/l4/conf/examples/l4lx-gfx.cfg` add a kernel parameter: `l4bdds.add=rom/mymodule.ko`
 - After booting, the file is available as `/dev/l4bdds0`:
`$> cat /dev/l4bdds0 >mymodule.ko`
`$> insmod mymodule.ko`

- 1) Build a “hello world” kernel module and run it on L4Linux.
- 2) Provide a char device in L4Linux that allows a Linux application (e.g., the shell's `echo`) to send a string to your kernel module
- 3) From the kernel module, print the string to the L4Re serial console, e.g. using `l4re_log_print()` from `<l4/re/c/log.h>`