<Slides download>
https://www.pf.is.s.u-tokyo.ac.jp/classes

Advanced Operating Systems

#1

Shinpei Kato
Associate Professor

Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo

Room 007

Course Plan

- Multi-core Resource Management
- Many-core Resource Management
- GPU Resource Management
- Virtual Machines
- Distributed File Systems
- High-performance Networking
- Memory Management
- Network on a Chip
- Embedded Real-time OS
- Device Drivers
- Linux Kernel

Schedule

1. 2018.9.28 Introduction + Linux Kernel (Kato)
2. 2018.10.5 Linux Kernel (Chishiro)
3. 2018.10.12 Linux Kernel (Kato)
4. 2018.10.19 Linux Kernel (Kato)
5. 2018.10.26 Linux Kernel (Kato)
6. 2018.11.2 Advanced Research (Chishiro)
7. 2018.11.9 Advanced Research (Chishiro)
8. 2018.11.16 (No Class)
9. 2018.11.23 (Holiday)
10. 2018.11.30 Advanced Research (Kato)
11. 2018.12.7 Advanced Research (Kato)
12. 2018.12.14 Advanced Research (Chishiro)
13. 2018.12.21 Linux Kernel
14. 2019.1.11 Linux Kernel
15. 2019.1.18 (No Class)
16. 2019.1.25 (No Class)

Introduction

Generalities of the operating system

/* The case for Linux */

Acknowledgement:

Prof. Pierre Olivier, ECE 4984, Linux Kernel Programming, Virginia Tech

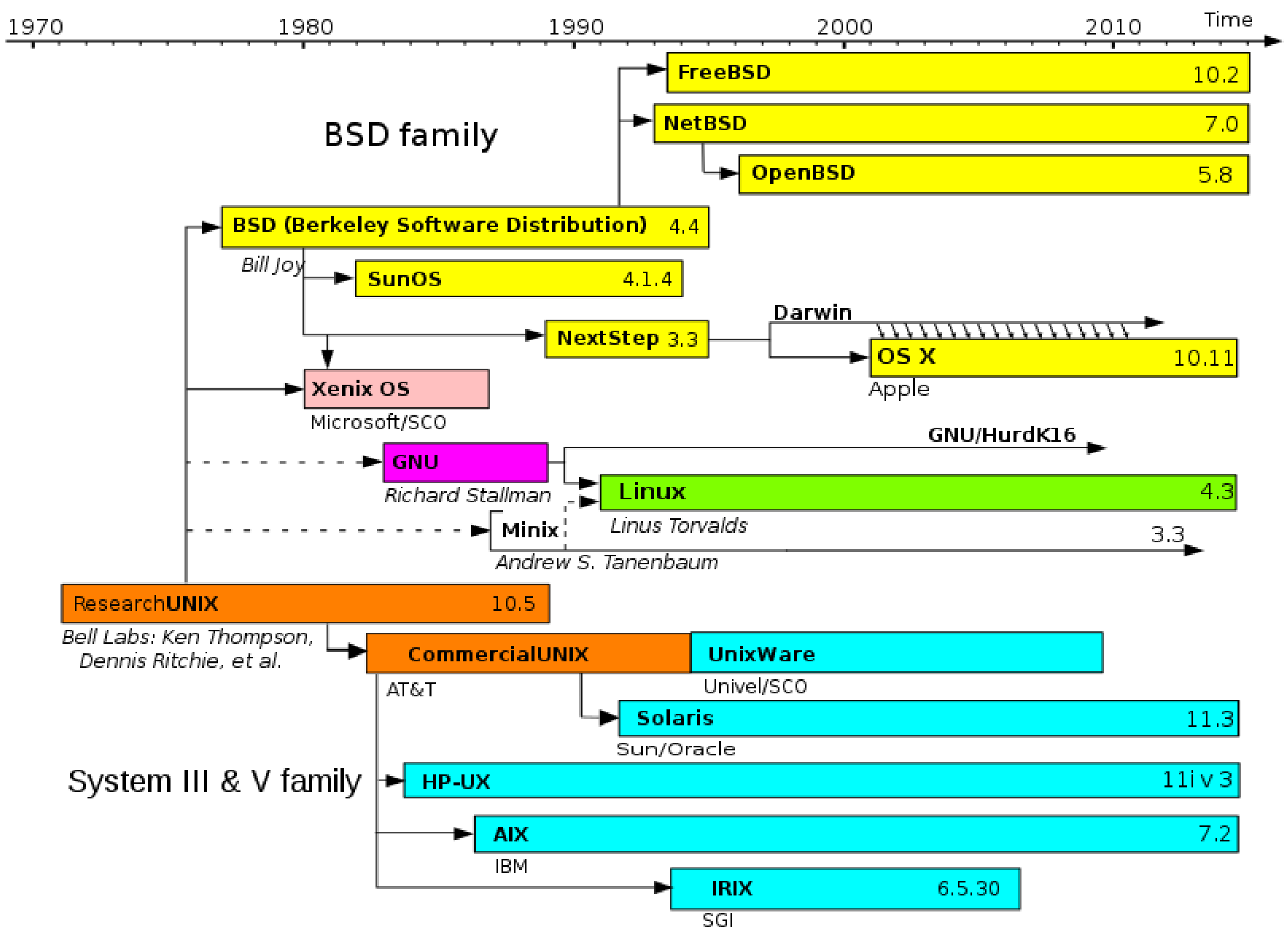
Im doing a (free) operating system (just a hobby, wont be big and professional like gnu) for 386(486) AT clones.

Linus Torvalds 08-25-1991

Outline

Outline

The Unix OS family



Source: [9]

The popularity of Unix

- Unix is very popular and ported to multiple architectures
  - This is due to its **simple design** and ease of use

- **Simplicity:**
  - Small number of system calls with clearly defined design
  - *Everything is a file*
  - Written in C for **portability**
  - Easy and fast process creation (`fork()`)
  - Simple and efficient Inter-Process Communication mechanisms (IPC)

Enters Linux

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki

Hello everybody out there using minix

I’m doing a (free) operating system (just a hobby, won’t be big and
professional like gnu) for 386(486) AT clones. This has been brewing
since april, and is starting to get ready. I’d like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I’ve currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I’ll get something practical within a few months, and
Id like to know what features most people would want. Any suggestions
are welcome, but I won’t promise I’ll implement them

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes     it’s free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never
will support anything other than AT-harddisks, as that’s all I have :-(.
```

Enters Linux (2)

- 1991: First apparition, author: *Linus Torvalds*
- 1992: GPL License, first Linux distributions
- 1994: Linux v1.0 - Single CPU for i386, then quickly ported to Alpha, Sparc, Mips architectures
- 1996: v2.0 - *Symmetric multiprocessing* (SMP) support
- 1999: v2.2 - *Big Kernel Lock* removed
- 2001: v2.4 - USB, later: RAID, Bluetooth, etc.
- 2003: v2.6 - Lots of features: Physical Address Expansion (PAE), new architectures, new file systems, etc.
- 2011: v3.0 - ”Nothing” [1]
- 2015: v4.0 - Livepatch, Kernel Address Space Sanitizer (KASAN)
- Today stable version: *Let’s check it out* → `www.kernel.org`

Sources: [10, 5]

Outline

1. A bit of history
2. Linux usage today
3. Linux open source model & the community
4. Global overview of the kernel
5. Kernel debugging (Qemu)
6. System calls

A Few Numbers

Linux usage in today's computer systems landscape

Embedded systems

- Smartphones, tablets, etc.: **65%** market share (Android)
  - vs iOS (24%)
- Embedded systems in general: **29%** (Android, embedded Linux solutions)

Servers

- **36%** market share (top 10M web servers)
  - vs other Unix-like OS (30%) and Windows (30%)
- **98%** (top 1M)

Source: [11]

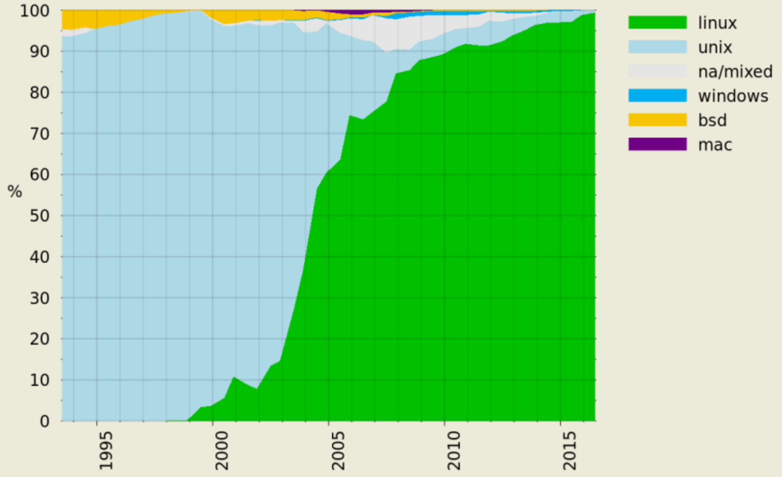
A Few Numbers (2)

Linux usage in today's computer systems landscape

Source: [11]

High-performance computing

- **99%** market share



Desktop Computers

- **2%** market share
- vs Microsoft Windows (90%) and Mac OSX (8%)

- **Indisputably one of the most popular operating systems over the whole spectrum of today's computer systems**

Jobs & Linux

- *Linux Jobs Report* released by the Linux foundation in 2014 [3] and 2015 [4]
  - 1000+ hiring managers polled as well as 3400+ Linux professionals
    - From large corporations to medium/small businesses, and government organizations
- **97% of the managers report they will hire Linux talent in the next six month**
  - Up from 46% in 2014
- **86% of Linux professionals report knowing Linux has given them more career opportunities**
  - **64% says they chose to work with Linux because it is pervasive**

Outline

Linux open source model

- Linux is licensed under **GPLv2**
  - Sources are freely available (https://www.kernel.org/)

Quick summary of the license terms:

You may copy, distribute and modify the software as long as you track changes/dates in source files. Any modifications to a software including (via compiler) GPL-licensed code must also be made available under the GPL along with build & install instructions.

https://tldrlegal.com/license/gnu-general-public-license-v2

- Third party can distribute modules as proprietary binaries
  - Compliance with GPLv2 is still an outstanding question, but Linus allows this

Linux open source model (2)

Benefits from the open source model

Benefits of open source vs closed development:

- **Security, stability**: *Given enough eyeballs, all bugs are shallow* [12]
  - Not a panacea though (OpenSSL Heartbleed, Bash Shellshock, etc.)
- **Code quality**
- **Freely modifiable** by anyone having an interest to do so
  - Hardware and systems vendors, distributors, end users
  - Enable innovation !
- **Education, research**
  - That's us !
  - We can `PEEK` at the code to understand how it works
  - And we can `POKE` it to experiment and build new systems

The Linux community

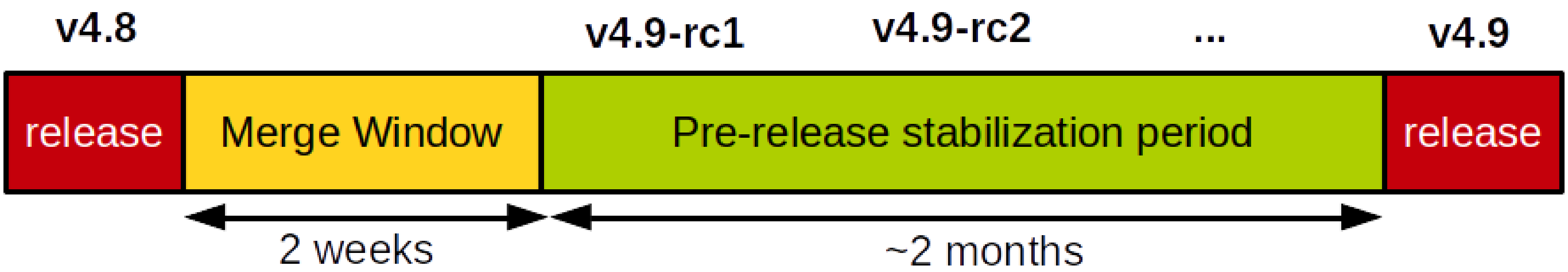
- **Developers**:
  - Anybody can propose modifications (*patches*)
    - Patches posted on **mailing lists**
    - Reviewed and commented by the subscribers
    - Then by **kernel maintainers**
    - More info here: [7]
  - Most of Linux contributors are actually employed by companies producing hardware (ex: Intel), systems (ex: Google), or doing consulting (ex: Red Hat)
- **Linux foundation** [2]
- **Conventions & Symposiums**
- **Software ecosystem** - very few examples:

etc.

Linux open source model & the community

Kernel version numbering system

- **<major>.<minor>.<stable>**
  - ex: 4.8.17



- **Stable version numbers** (bugs fixes) are added after release
- **Long Term Support** for a subset of releases:
  - Ex: 4.4.43 or 3.2.84 vs 4.8.17 [EOL]

Outline

1 A bit of history

2 Linux usage today

3 Linux open source model & the community

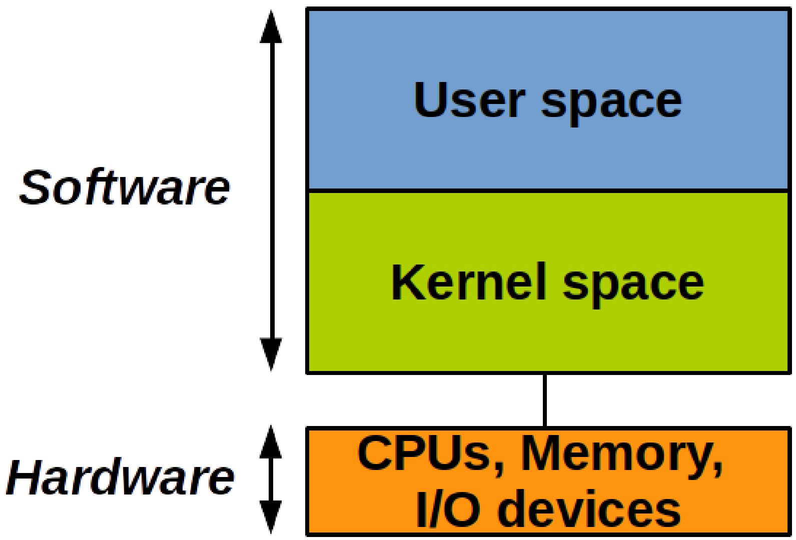
4 Global overview of the kernel

5 Kernel debugging (Qemu)

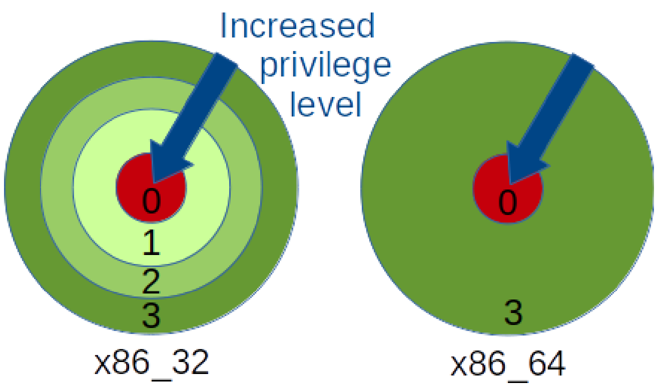
6 System calls

Global overview of the kernel

User vs Kernel space

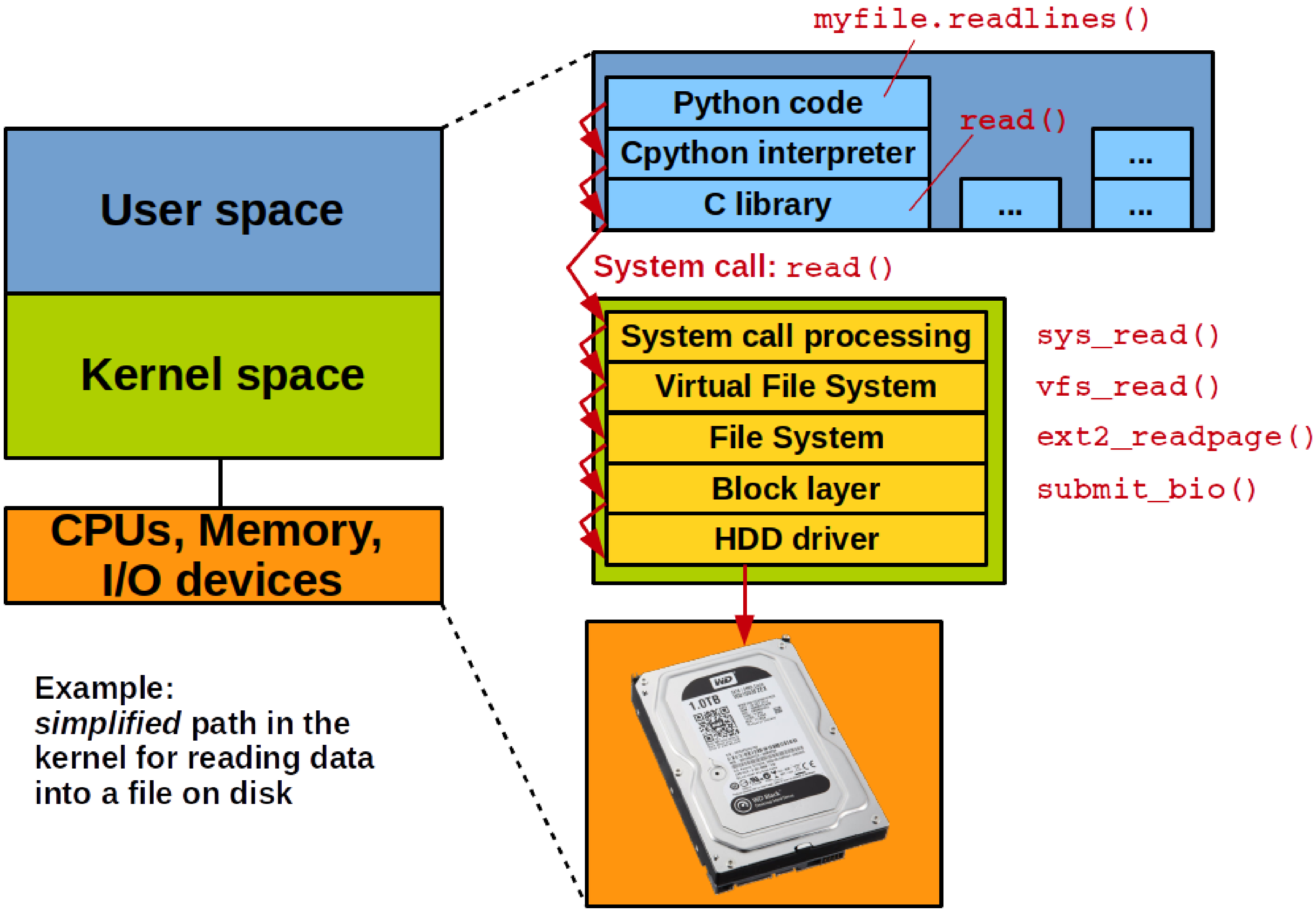


- At any given moment a CPU is executing in **user** *or* in **kernel** mode
- Only the kernel is allowed to perform privileged operations and access I/O devices



Global overview of the kernel

User vs Kernel space (2)



Example:
simplified path in the kernel for reading data into a file on disk

Global overview of the kernel

Monolithic Model

- Linux is **monolithic**
  - **All the OS services run in kernel, privileged mode**
    - In the **same address space**
- Opposed to the *microkernel* model:
  - Only the core services run in full privileged mode
    - Scheduling, interrupt handling, etc.
  - Other services run with reduced privileged in their own address space
  - Communication through message passing
  - Ex: Minix, L4, etc.
  - More secure and modular, but communication brings performance down
    - See the *Tanenbaum-Torvalds Debate* [8]
  - Other classes of OS: Exokernel, MultiKernel, etc.

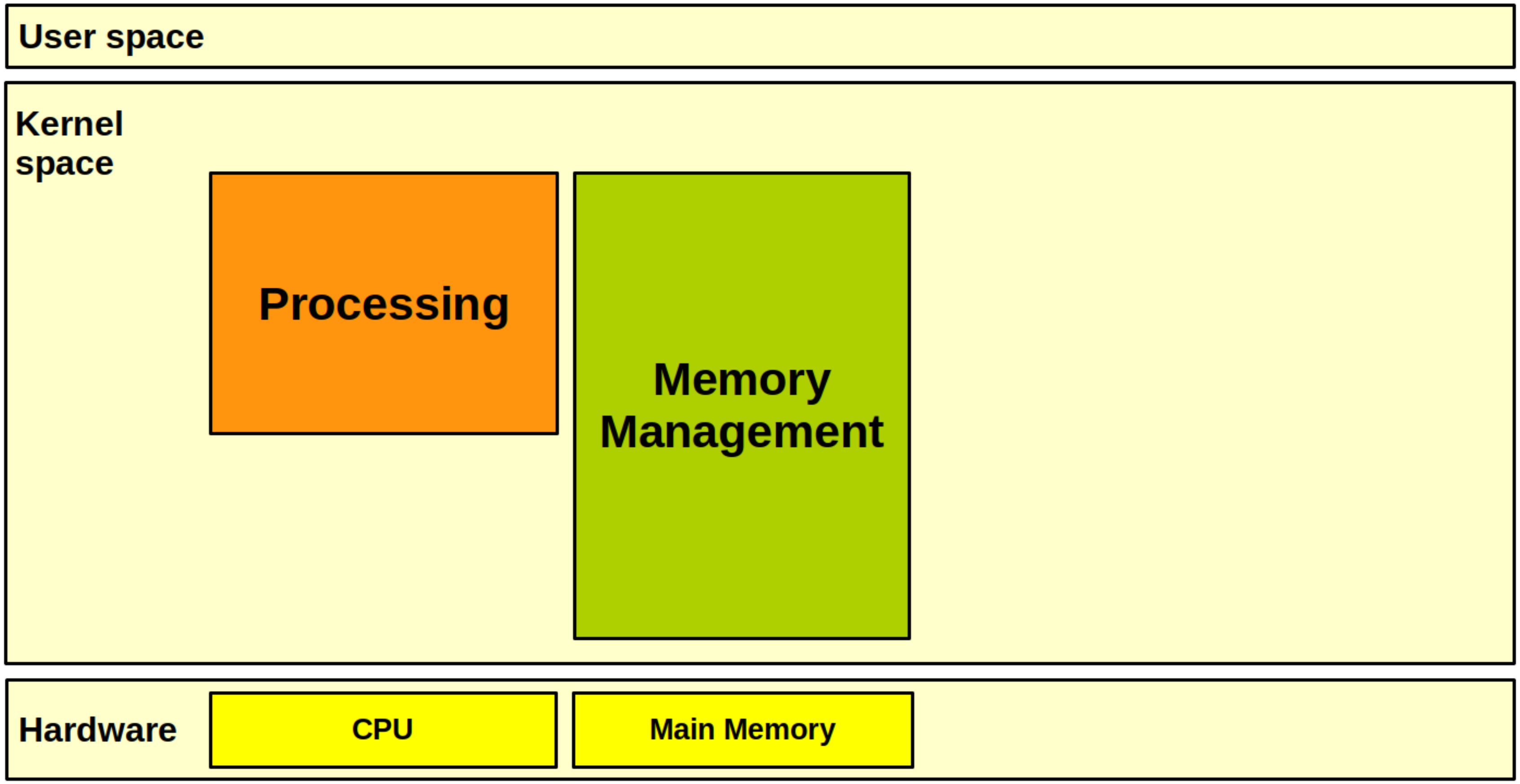
Global overview of the kernel

Kernel vs user level programming

- **No C library** mostly because of speed and size
  - However, the kernel implements lots of usefull functions from the C library
    - Ex: `strcat`, `vmalloc`, `ssleep`, etc.
    - `printf` → `printk`
- **2% assembly, the rest in GNU C**
  - GCC extensions
- **No memory protection**
- **Highly concurrent**
  - Preemption, interrupts, running on SMP
  - Race conditions without proper synchronization

Global overview of the kernel

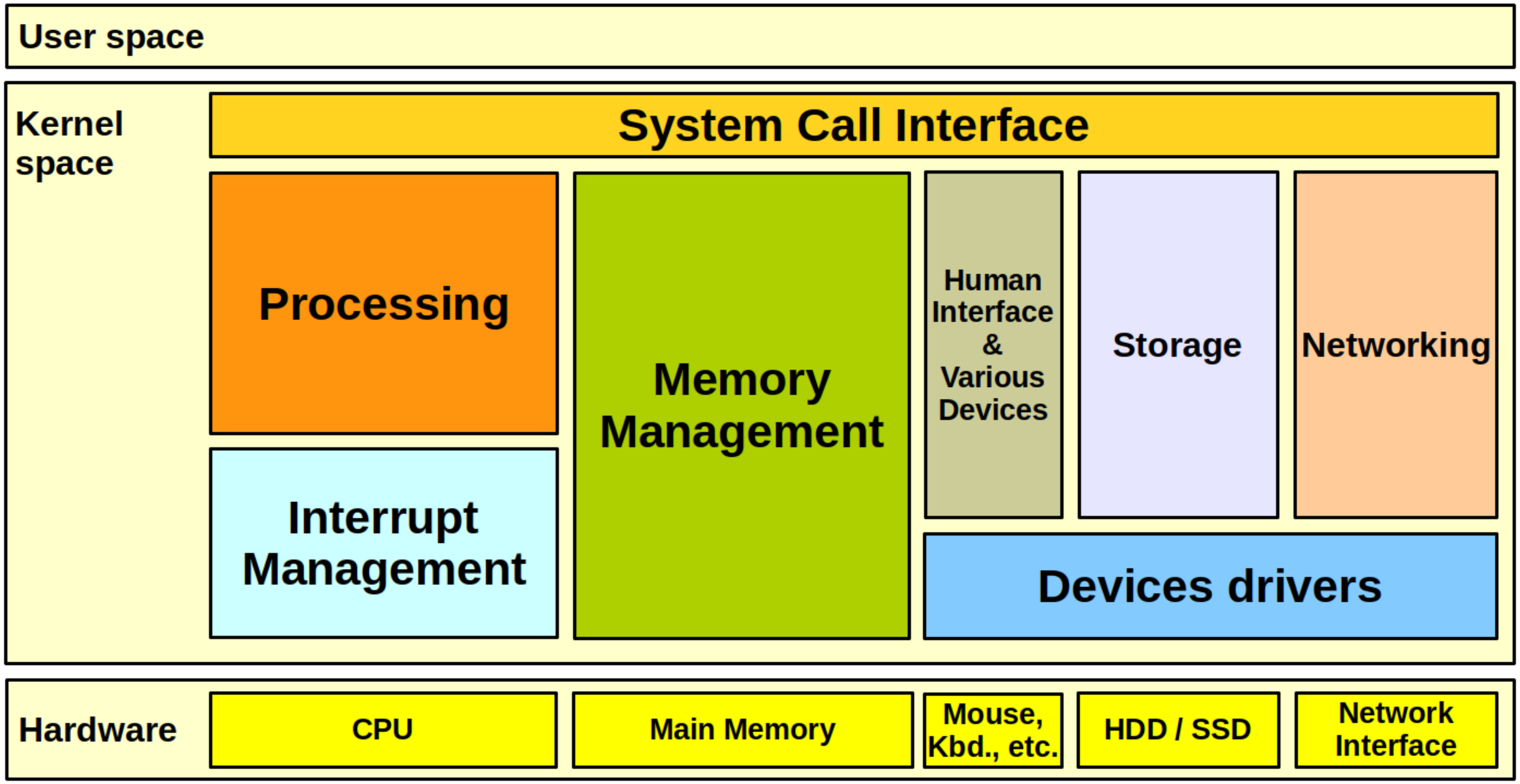
Kernel & course map



Inspired from [6].

Global overview of the kernel

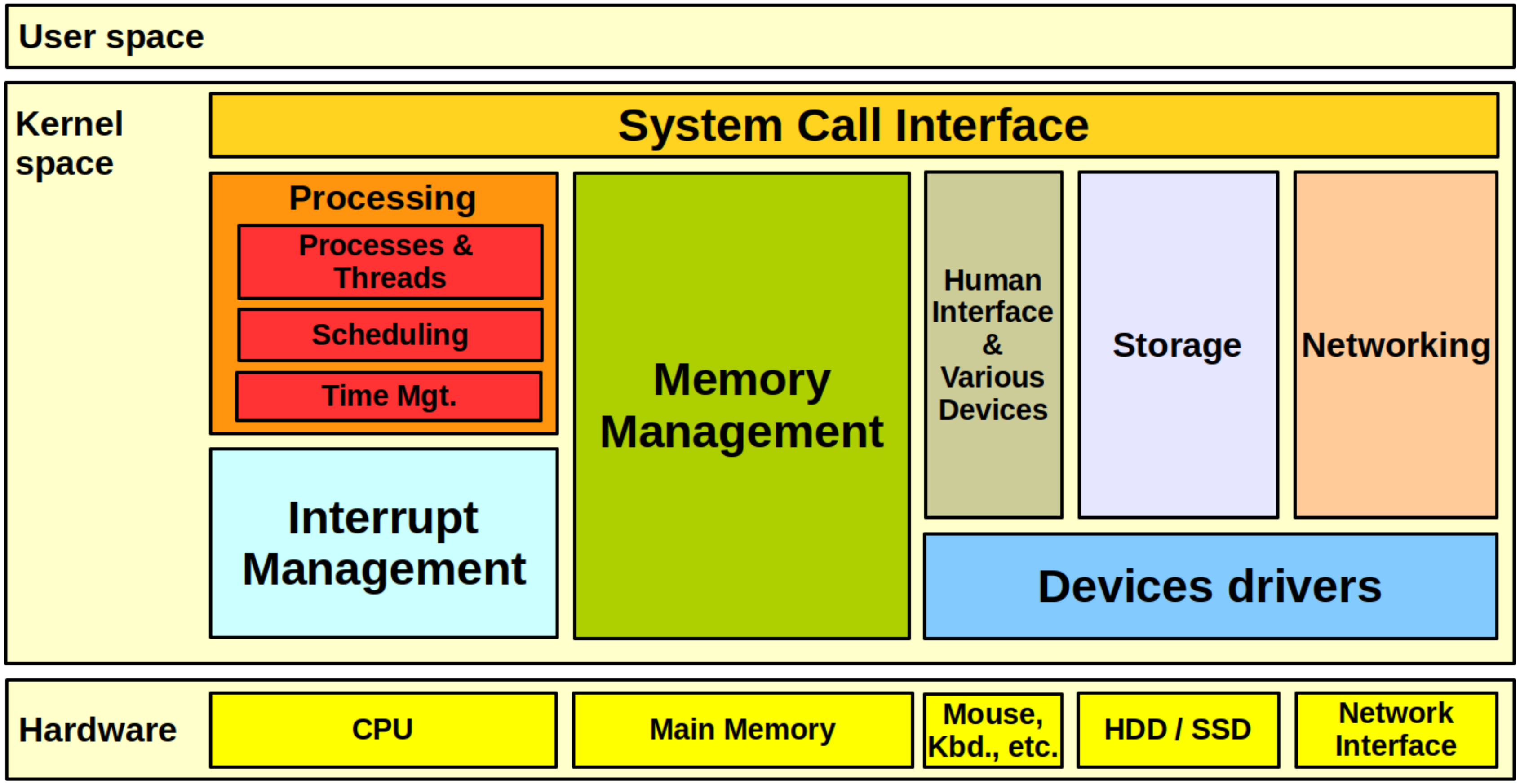
Kernel & course map



Inspired from [6].

Global overview of the kernel

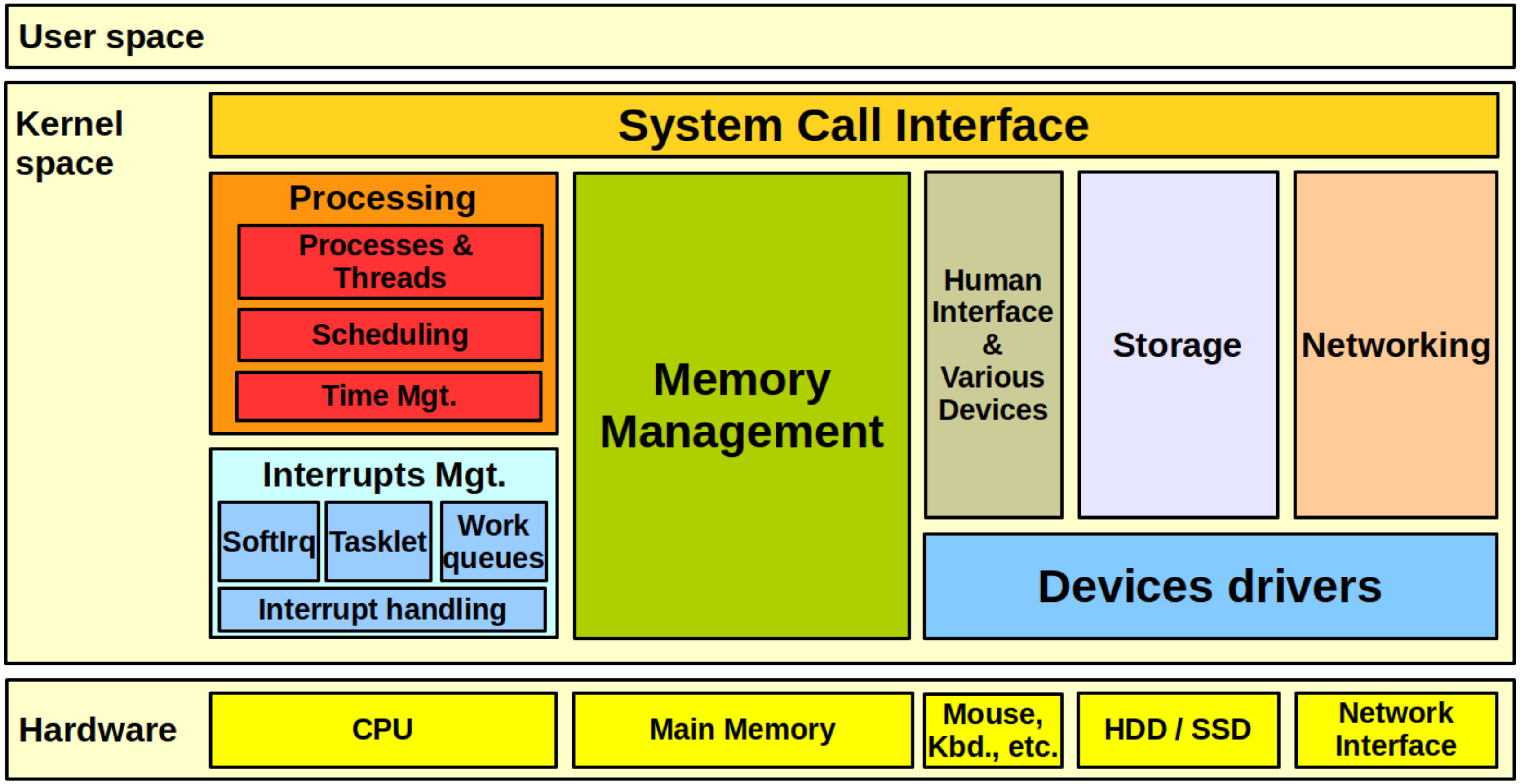
Kernel & course map



Inspired from [6].

Global overview of the kernel

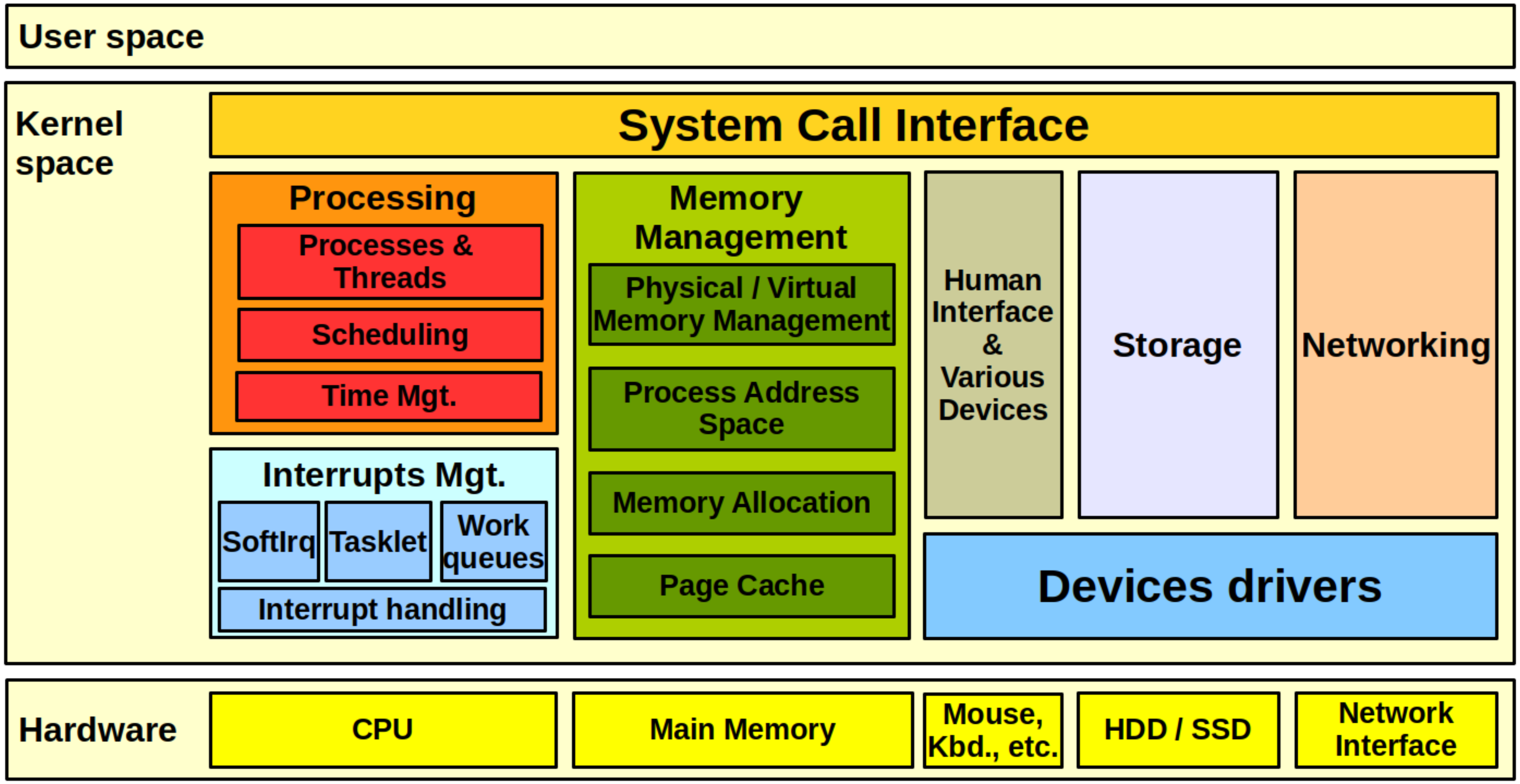
Kernel & course map



Inspired from [6].

Global overview of the kernel

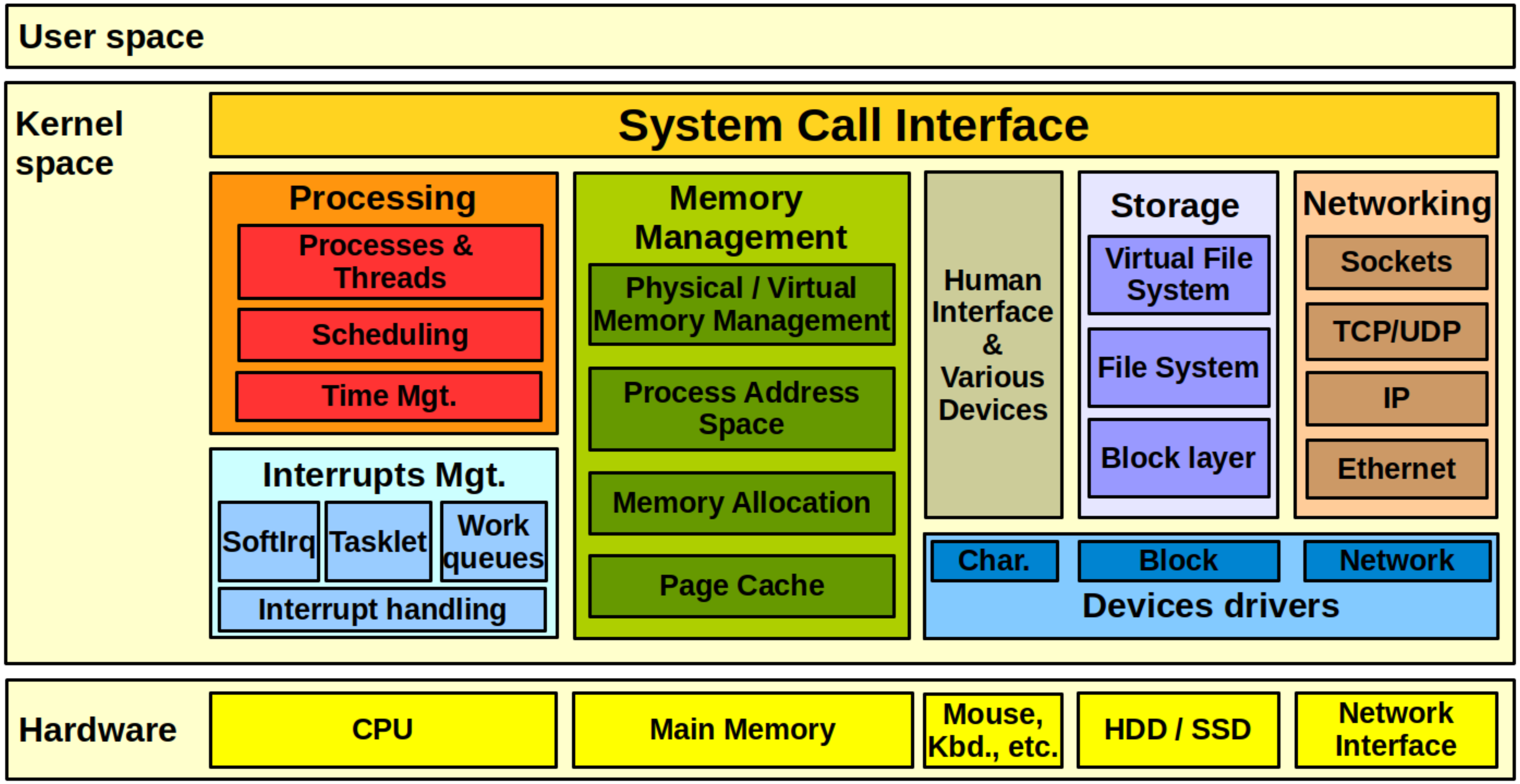
Kernel & course map



Inspired from [6].

Global overview of the kernel

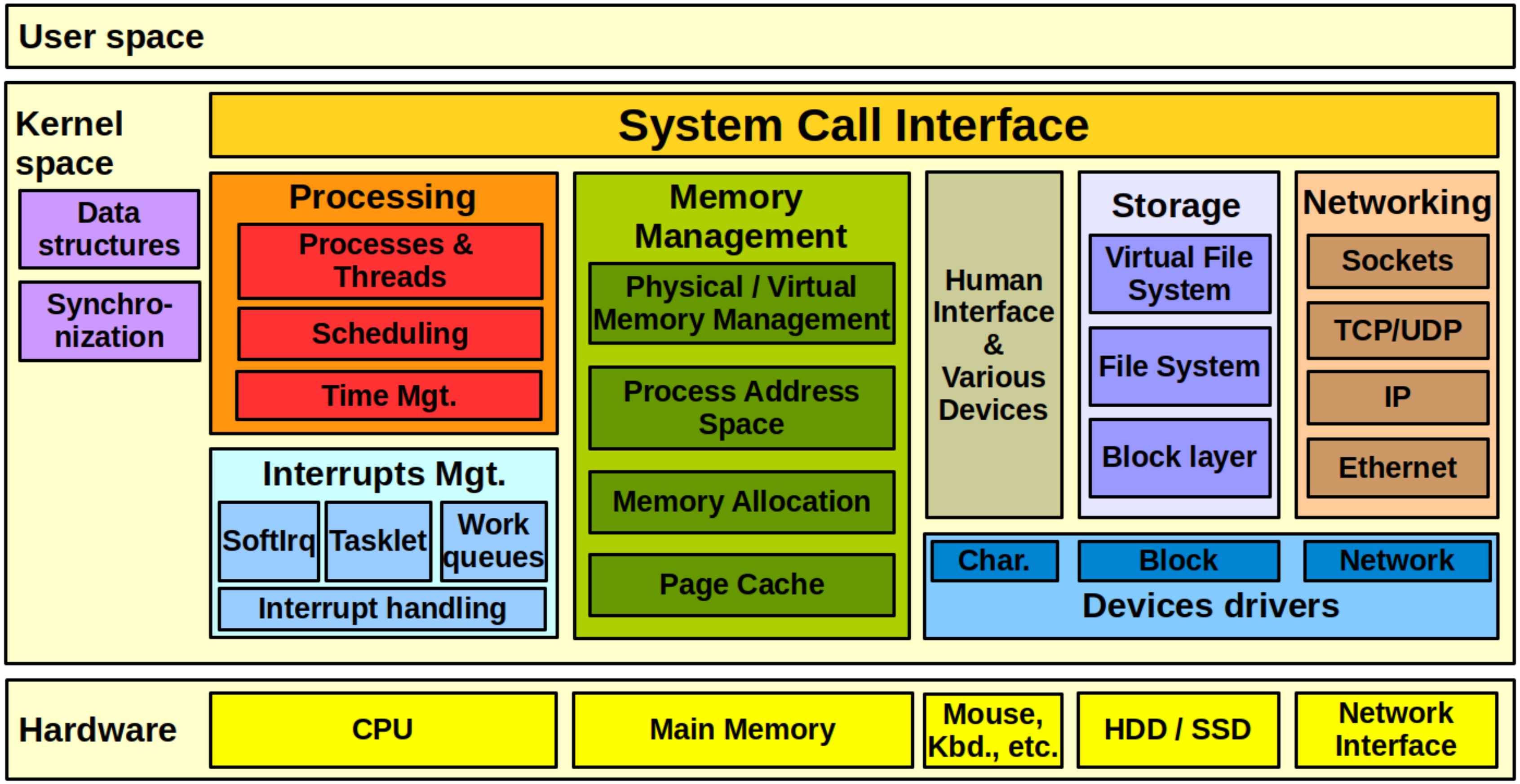
Kernel & course map



Inspired from [6].

Global overview of the kernel

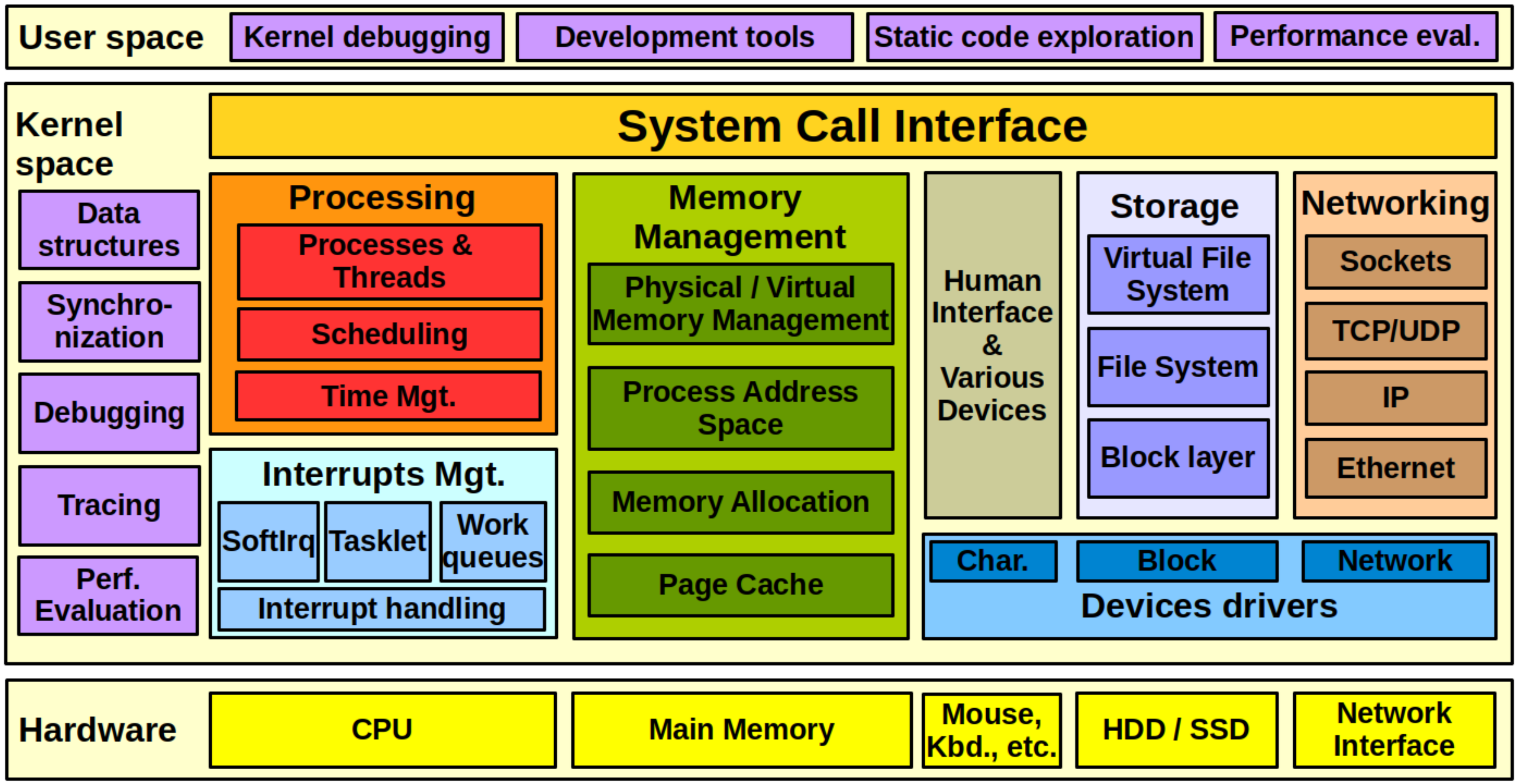
Kernel & course map



Inspired from [6].

Global overview of the kernel

Kernel & course map



Inspired from [6].

Outline

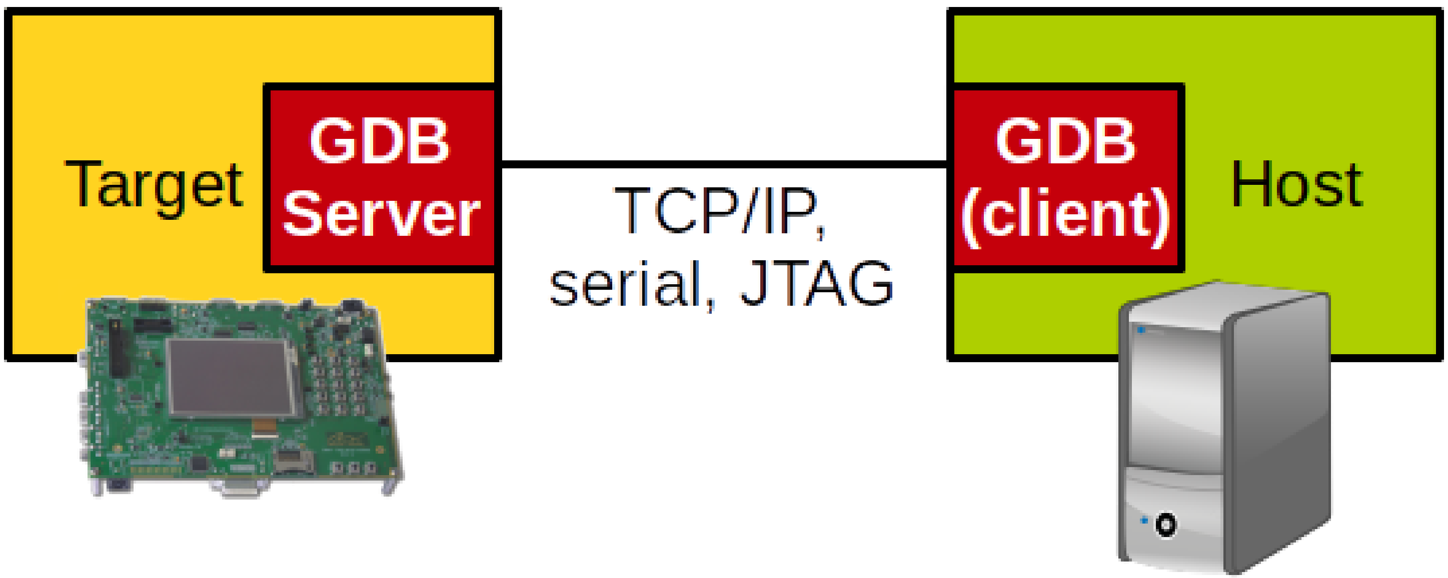
Qemu quick presentation

- **Full system emulator:** emulates an entire virtual machine
  - Using a software model for the CPU, memory, devices
  - Emulation is slow
- Can also be used in conjunction with hardware virtualization extensions to provide high performance virtualization
  - **KVM**
    - In-kernel support for virtualization + extensions to Qemu

Qemu and kernel development

GDB server

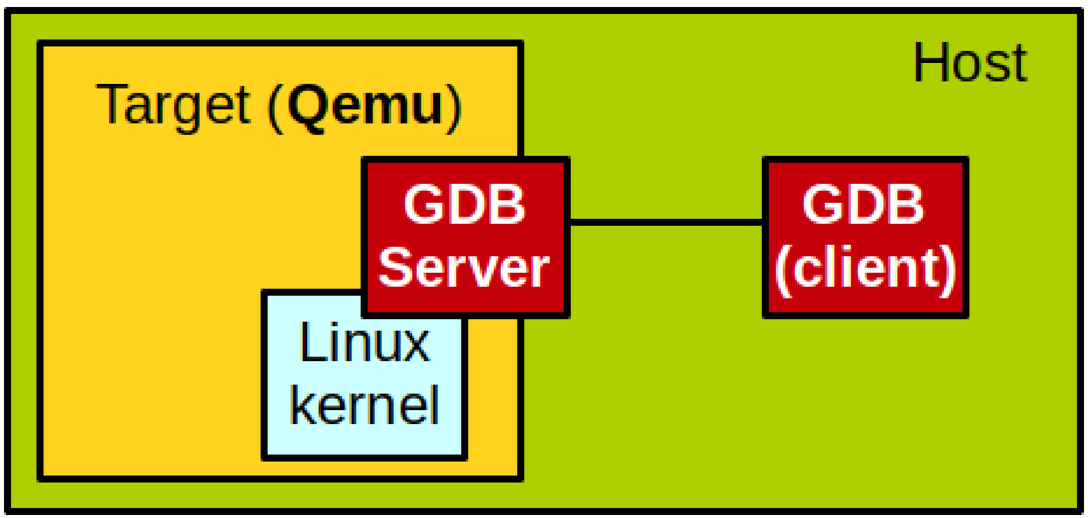
- **GDB server**
  - Originally used to debug a program executing on a remote machine
  - For example when GDB is not available on that remote machine
    - Ex: low performance embedded systems



Qemu and kernel development

Qemu & GDB sever

- Qemu is capable of running a kernel in an emulated machine with an associated root file system ...
- ... and **act as a GDB server for the kernel itself**



Qemu and kernel development

Qemu & GDB server: benefits

- Benefits:
  1. **Debugging**
  2. **Runtime code exploration**

Using Qemu and GDB server

Requirements

- **Requirements**:
  - Linux should be compiled with debug symbols:
    - `make menuconfig` › Kernel hacking › Compile the kernel with debug info (old kernels)
    - `make menuconfig` › Kernel hacking › Compile-time checks and compiler options › Compile the kernel with debug info
  - Qemu options:
    - **`-kernel path/to/bzImage`**: path to the `bzImage` of the kernel we want to execute and debug
    - **`-s`**: enable the GDB server
    - **`-S`**: (optional): pause on the first kernel instruction waiting for a GDB client connection order to continue
- **Usage (client side)**:

```
gdb path/to/vmlinux
(gdb) target remote:1234
```

Using Qemu and GDB server

Requirements (2)

- GDB usage:
  http://www.dirac.org/linux/gdb/

Miscellaneous information

Remote GDB bug on 64 bits

- Error when connecting to the remote target: `Remote 'g' packet reply is too long`
  - You need to patch GDB (client)
  - Patch for old versions of GDB sources: http://www.cygwin.com/ml/gdb-patches/2012-03/msg00116.html
  - Last version (7.11+): https://github.com/olivierpierre/gdb-remote-patch
- Compiling GDB:
  1. Grab the sources:
  2. https://www.sourceware.org/gdb/download/
  3. Patch it using `patch -p1 < patch-name.patch`

```
./configure # Might notify for missing dependencies
make
sudo mv /usr/bin/gdb /usr/bin/gdb.old # Backup the old version
make install
```

Miscellaneous information

Optimized values

```
(gdb) p some_variable
$1 = <value optimized out>
```

- **It is not possible to disable optimization for the entire kernel**
- Needs to be done on a per-file basis
  1. Identify the file containing the variable declaration
  2. Update the corresponding makefile (example with fs/ext4/Makefile):

```
obj-$(CONFIG_EXT4_FS) += ext4.o

CFLAGS_bitmap.o = -O0

ext4-y := balloc.o bitmap.o dir.o file.o fsync.o ialloc.o inode.o page-io.o ¥
ioctl.o namei.o super.o symlink.o hash.o resize.o extents.o ¥
ext4_jbd2.o migrate.o mballoc.o block_validity.o move_extent.o ¥
mmp.o indirect.o extents_status.o xattr.o xattr_user.o ¥
xattr_trusted.o inline.o readpage.o sysfs.o
# ...
```

Miscellaneous information

Mounting a virtual disk

- With Qemu, the root filesystem is generally present on a virtual disk (disk image)
- What if there is a crash at boot time that prevent the emulated machine from booting?
- You can mount the virtual disk on the host to try to fix the problem from there
  - **Mounting depends on the image format**
- Check the format using `file`:

```
file <disk image file path>

# QCOW2 format:
debian7.qcow2: QEMU QCOW Image (v3), 21474836480 bytes

# RAW format:
hd.img: x86 boot sector; partition 1: ID=0x83, active, starthead 32, startsector 2048,
     40134656 sectors; partition 2: ID=0x5, starthead 254, startsector 40138750,
     1802242 sectors, code offset 0x63
```

Miscellaneous information

Mounting a virtual disk (2)

- **Raw format:**

```
file hd.img
hd.img: x86 boot sector; partition
      1: ID=0x83, active, starthead
      32, startsector 2048, 40134656
      sectors; partition 2: ID=0x5,
      starthead 254, startsector
      40138750, 1802242 sectors,
      code offset 0x63

# 1048576 == 2048 * 512
sudo mount -o loop,offset=1048576 hd
      .img /mnt/image

# work on the mounted filesystem ...

sudo umount /mnt/image
```

- **Qcow2 format:**

```
sudo modprobe nbd max_part=63
sudo qemu-nbd -c /dev/nbd0 image.qcow2
sudo mount /dev/nbd0p1 /mnt/image

# work on the mounted filesystem ...

sudo umount /mnt/image
sudo qemu-nbd -d /dev/nbd0
sudo rmmod nbd
```

- **Do not launch the VM while the root filesystem is mounted on the host**

Miscellaneous information

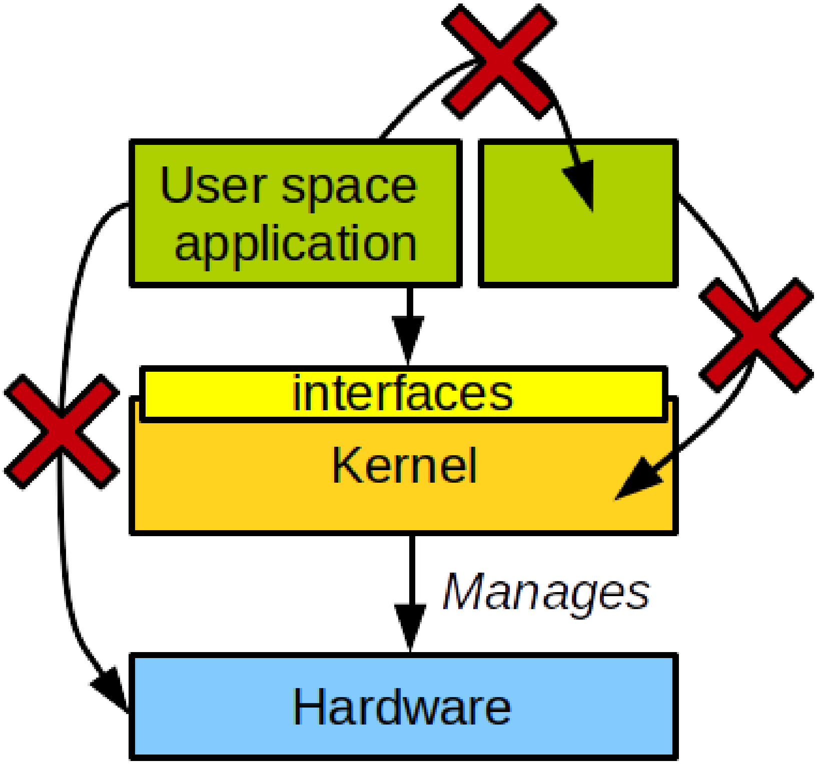
Additional info

- Cursor disappears in qemu window?
  - Ctrl + Alt (right)
- **Do not close Qemu without issuing the `halt` command to the (Qemu) VM**
  - Risks leading to inconsistent filesystem state (data loss, VM unable to boot ...)
  - This is true for all VMs
- Qemu is too slow
  - Update Qemu version
  - Try KVM but you need a native installation

Outline

System calls: general notions

Kernel entry point from user space



- The kernel:
  - Manages the hardware
  - Provides **interfaces** or user space processes to access the hardware and perform privileged operations
- **User space cannot access HW/perform privilege operations directly**

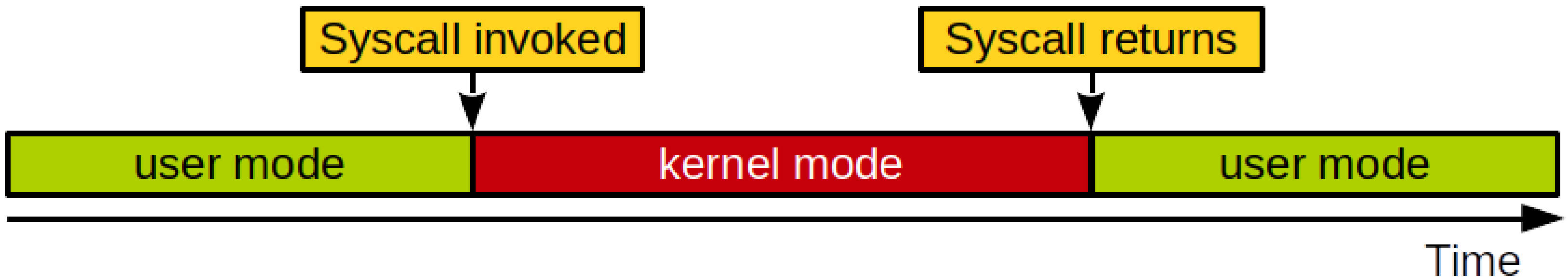
Interfaces + user space privileges restriction: **the key to stability and security in the system**

System calls: general notions

User/kernel mode

System calls (*syscalls*) are the one and only way an application can enter the kernel to request OS services and privileged operations such as accessing the hardware

- Examples of privileged/restricted operations:
  - **Privileged CPU instructions** (x86 examples): `HLT`, `INVLPLG`, `MOV` to control registers, etc.
    - Including IO related instructions (`IN`/`OUT`)
  - **Access to all memory areas**
    - Including areas mapping device registers



System calls: general notions

Examples of syscalls

Syscalls can be classified into groups:

- **Process management/scheduling:** `fork`, `exit`, `execve`, `nice`, {`get|set`}`priority`, {`get|set`}`pid`, etc.
- **Memory management:** `brk`, `mmap`, `swap`{`on|off`}, etc.
- **File system**: `open`, `read`, `write`, `lseek`, `stat`, etc.
- **Inter-Process Communication**: `pipe`, `shmget`, `semget`, etc.
- **Time management**: {`get|set`}`timeofday`, `time`, `timer_create`, etc.
- Others: {`get|set`}`uid`, `syslog`, `connect`, etc.

System calls: general notions

System calls table syscall identifier

- For x86_64, the syscall list is present in `arch/x86/syscalls/syscall_64.tbl` (4.0, location changes with versions)
  - Text file translated to c source code by a script during the compilation process

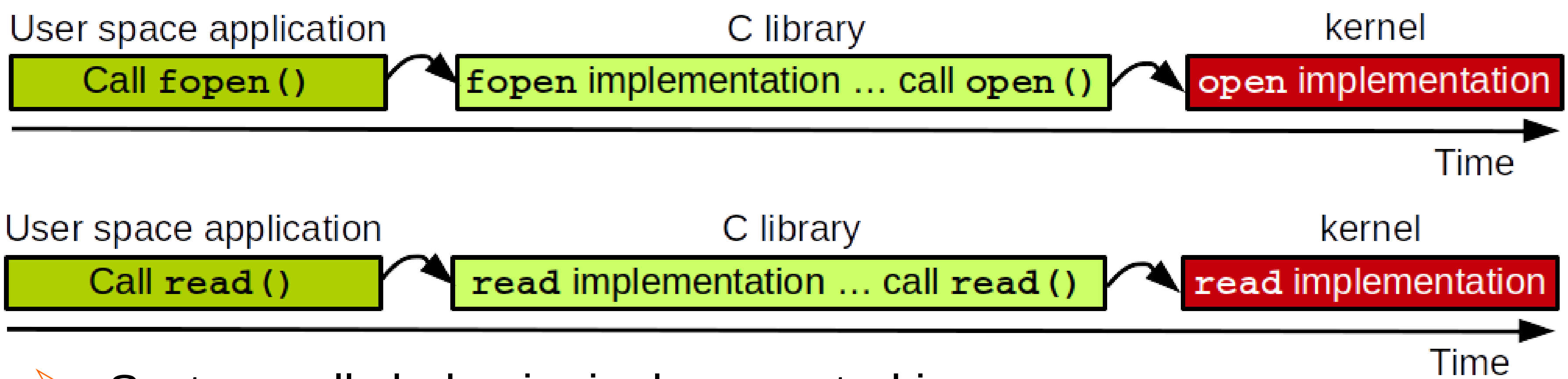
```
## 64-bit system call numbers and entry vectors
#
  # The format is:
  # <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0 common   read       sys_read
1 common   write      sys_write
2 common   open       sys_open
3 common   close      sys_close
# ...
```

- Syscall identifier: unique integer
  - Currently 352 (linux 4.9) for x86_64
  - New syscalls identifiers are given sequentially

Syscall invocation: user space side

C library

- Syscalls are rarely invoked directly
  - Most of them are wrapped by the C library
  - The programmer uses the C library **Application Programming Interface (*API*)**



- System calls behavior is documented in man pages

```
man <syscall name>
```

Syscall invocation: user space side

C library: Invocation without wrapper

- Some syscalls does not have a wrapper in the C library
- A syscall can be called directly through `syscall`
  - `man syscall`

```
#include <unistd.h>
#include <sys/syscall.h>   /* For SYS_xxx definitions */

int main(void)
{
  char message[] = "hello, world!¥n";
  int bytes_written = -42;

  /* the first "1" is the "write" syscall identifier */
  /* the second "1" is the standard output file descriptor */
  /* the remaining arguments are the "write" syscall arguments */
  bytes_written = syscall(1, 1, message, 14);

  /* or */

  bytes_written = syscall(SYS_write, 1, message, 14);

  return 0;
}
```

syscall.c

Syscall invocation: user space side

Invocation without the C library

- On x86_64, syscalls can be used directly through the `syscall` assembly instruction
  - Usage example: disabling the C library considerably reduces the size of a program

```
        .global _start

        .text
_start:
        # write(1, message, 14)
        mov     $1, %rax
        mov     $1, %rdi
        mov     $message, %rsi
        mov     $14, %rdx
        syscall

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi
        syscall
message:
        .ascii "Hello, world!¥n"
```

syscall asm.s

- Compilation & execution:

```
gcc -c syscall_asm.s
    -o syscall_asm.o
ld syscall_asm.o
    -o syscall_asm
./syscall_asm
hello, world!
```

- Parameters are passed in registers

Syscall execution: kernel space side

User/kernel space transition

- **User space applications cannot call kernel code directly**
  - For security and stability, kernel code resides in a memory space that cannot be accessed from user space
  - *So how is a syscall invoked from user space ?*

Syscall execution: kernel space side

User/kernel space transition (2)

- A few words about interrupts:
  1. Asynchronous: **hardware interrupts**, issued from devices
     - Ex: keyboard indicating that a key has been pressed
  2. Synchronous: **exceptions**, triggered involuntarily by the program Itself
     - Ex: divide by zero, page fault, etc.
  3. Synchronous, programmed exceptions: **software interrupts**, issued voluntarily by the code of the program itself
     - `INT` instruction for x86
- When an interrupt is received by the CPU, it stops whatever it is doing and **the kernel executes the interrupt handler**

Syscall execution: kernel space side

User/kernel space transition (3)

- ***So how is a syscall invoked from user space ?***
  - User space put the **syscall identifier and parameters values into registers** (x86)
    - Identifier in `rax`
    - x86 64: parameters in `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9`
  - Then issues a **software interrupt**
    - On x86, interrupt 128 was used:

```
int $0x80
```

    - Now `sysenter` (x86 32) and `syscall` (x86 64)
  - The kernel executes the interrupt handler, **system call handler**
    - Puts the registers values into a data structure placed on the stack
    - Checks the validity of the syscall (number of arguments)
    - Then execute the system call implementation:

```
call sys_call_table(, %rax, 8)
```

Syscall execution: kernel space side

Syscall implementation execution: example with `gettimeofday`

- Example: **gettimeofday**
  - implementation in `sys gettimeofday`

```
NAME
       gettimeofday, settimeofday - get / set time

SYNOPSIS
       #include <sys/time.h>

       int gettimeofday(struct timeval *tv, struct timezone *tz);

       int settimeofday(const struct timeval *tv, const struct timezone *tz);

     DESCRIPTION
     The functions gettimeofday() and settimeofday() can get and set the time as well as a
     timezone. The tv argument is a struct timeval (as specified in <sys/time.h>):

           struct timeval {
               time_t      tv_sec;     /* seconds */
               suseconds_t tv_usec;    /* microseconds */
           };

       and gives the number of seconds and microseconds since the Epoch.
```

Syscall execution: kernel space side

Syscall implementation execution: example with `gettimeofday` (2)

- Usage example:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(void)
{
  struct timeval tv;
  int ret;

  ret = gettimeofday(&tv, NULL);
  if(ret == -1)
  {
    perror("gettimeofday");
    return EXIT_FAILURE;
  }

  printf("Local time:¥n");
  printf(" sec:%lu¥n", tv.tv_sec);
  printf(" usec:%lu¥n", tv.tv_usec);

  return EXIT_SUCCESS;
}
```

```
./gettimeofday
Local time:
sec:1485214886
 usec:523511
```

Syscall execution: kernel space side

Syscall implementation execution: example with `gettimeofday` (3)

- Let's check it out using vim code indexing features

```c
SYSCALL_DEFINE2(gettimeofday, struct timeval_user
        *, tv, struct timezone_user *, tz)
{
  if (likely(tv != NULL)) {
    struct timeval ktv;
    do_gettimeofday(&ktv);
    if (copy_to_user(tv, &ktv, sizeof(ktv)))
      return -EFAULT;
  }
  if (unlikely(tz != NULL)) {
    if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
      return -EFAULT;
  }
  return 0;
}
```

- `SYSCALL_DEFINE2`
  - Macro to define `sys_gettimeofday` (2 parameters)
- **`likely/unlikely`**
  - Compiler assisted branch predictor hints
- **`user` pointers and `copy_{to|from}_user`**
  - Kernel / user space memory management

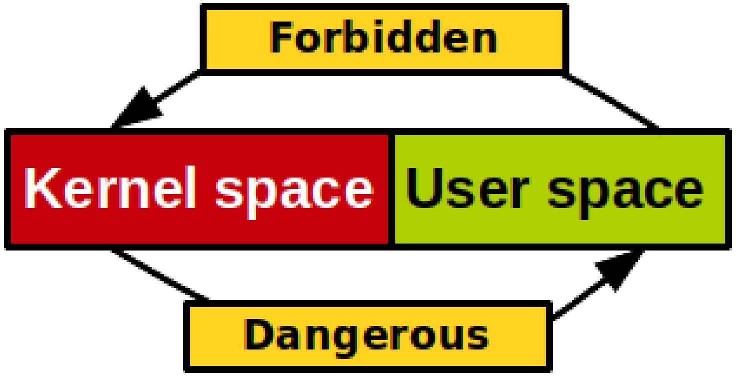
Syscall execution: kernel space side

`likely`/`unlikely` and kernel/user memory transfers

- **`likely/unlikely`**
  - `include/linux/compiler.h`:

```
#define likely(x)    (__builtin_expect(!!(x), 1)) /* !! convert to int and */
#define unlikely(x)  (__builtin_expect(!!(x), 0)) /* into actual 0 or 1     */
```

- **User vs kernel memory areas**



- User space cannot access kernel memory
- Kernel code should not directly access user memory
- How to exchange data with pointers ?

Syscall execution: kernel space side

`likely`/`unlikely` and kernel/user memory transfers (2)

- The `user` attribute
  - `include/linux/compiler.h`:

```
#define  user       __attribute_((noderef, address_space(1)))
#define  kernel     __attribute_((address_space(0)))
```

  - Used for static code security analysis (sparse)
- `copy {to|from} user`

```
static inline long copy_from_user(void *to, const void_user *from, unsigned long n);
static inline long copy_to_user(void_user *to, const void *from, unsigned long n);
```

  - When a kernel function gets a pointer to some memory in user space it needs to use:
    - The kernel copies it into its memory area (`copy from user`)
  - When the kernel wants to write in a user space buffer:
    - It uses `copy to user`
- These functions perform checks for security and stability

Implementing a new system call

Basic steps

1. **Write your syscall function**
   1. In an existing file if it makes sense
      - Is it related to time management ? → `kernel/time/time.c`
   2. Or, if the implementation is large and self-contained: in a new file
      - You will have to edit the kernel `Makefiles` to integrate it in the compilation process
2. **Add it to the syscall table and give it an identifier**
   - `arch/x86/syscalls/syscall_64.tbl` for Linux 4.0
3. **Add the prototype in `include/linux/syscalls.h`:**

```
asmlinkage long sys_gettimeofday(struct timeval_user *tv,
                                 struct timezone_user *tz);
```

4. Recompile, reboot and run
   - Touching the syscall table will trigger the entire kernel compilation

Implementing a new system call

Editing the kernel Makefiles (2)

- Example: syscall implemented in linux sources in `my_syscall/my_syscall.c`

  1. `my_syscall/Makefile`:

     ```
     obj-y += my_syscall.o
     ```

  2. Linux root `Makefile`:

     ```
     # ...
      core-y          += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ my_syscall/
     # ...
     ```

Implementing a new system call

Do you really need it?

- **Pros:** Easy to implement and use, fast
- **Cons:**
  - Needs an official syscall number
  - Interface cannot change after implementation
  - Must be registered for each architecture
  - Probably too much work for small exchanges of information
- **Alternative:**
  - Device or virtual file:
    - User/kernel space communication through `read`, `write`, `ioctl`

Bibliography I

1 Linux 3.0-rc1.
https://lkml.org/lkml/2011/5/29/204.
Accessed: 2016-09-30.

2 Linux foundation website.
https://www.linuxfoundation.org/.
Accessed: 2016-09-30.

3 Linux jobs report 2014.
https://www.linux.com/publications/linux-jobs-report-2014.
Accessed: 2017-01-12.

4 Linux jobs report 2015.
https://www.linux.com/publications/linux-jobs-report-2015.
Accessed: 2017-01-12.

5 Linux kernel newbies - kernel versions.
https://kernelnewbies.org/LinuxVersions.
Accessed: 2016-09-30.

6 makelinux.net - linux kernel map.
http://www.makelinux.net/kernel_map/.
Accessed: 2016-12-26.

7 A survivor's guide to contributing to the linux kernel.
http://events.linuxfoundation.org/sites/events/files/slides/klf2015_slides_javier_martinez_0.pdf.
Accessed: 2017-01-16.

8 The tanenbaum-torvalds debate.
http://www.oreilly.com/openbook/opensources/book/appa.html.
Accessed: 2017-01-12.

Bibliography II

9 Wikipedia - darwin os.
https://en.wikipedia.org/wiki/Darwin_(operating_system).
Accessed: 2016-09-30.

10 Wikipedia - linux history.
https://en.wikipedia.org/wiki/History_of_Linux.
Accessed: 2016-09-30.

11 Wikipedia - usage share of operating systems.
https://en.wikipedia.org/wiki/Usage_share_of_operating_systems.

12 RAYMOND, E.
The cathedral and the bazaar, vol. 12.
Springer, 1999.