Operating Systems

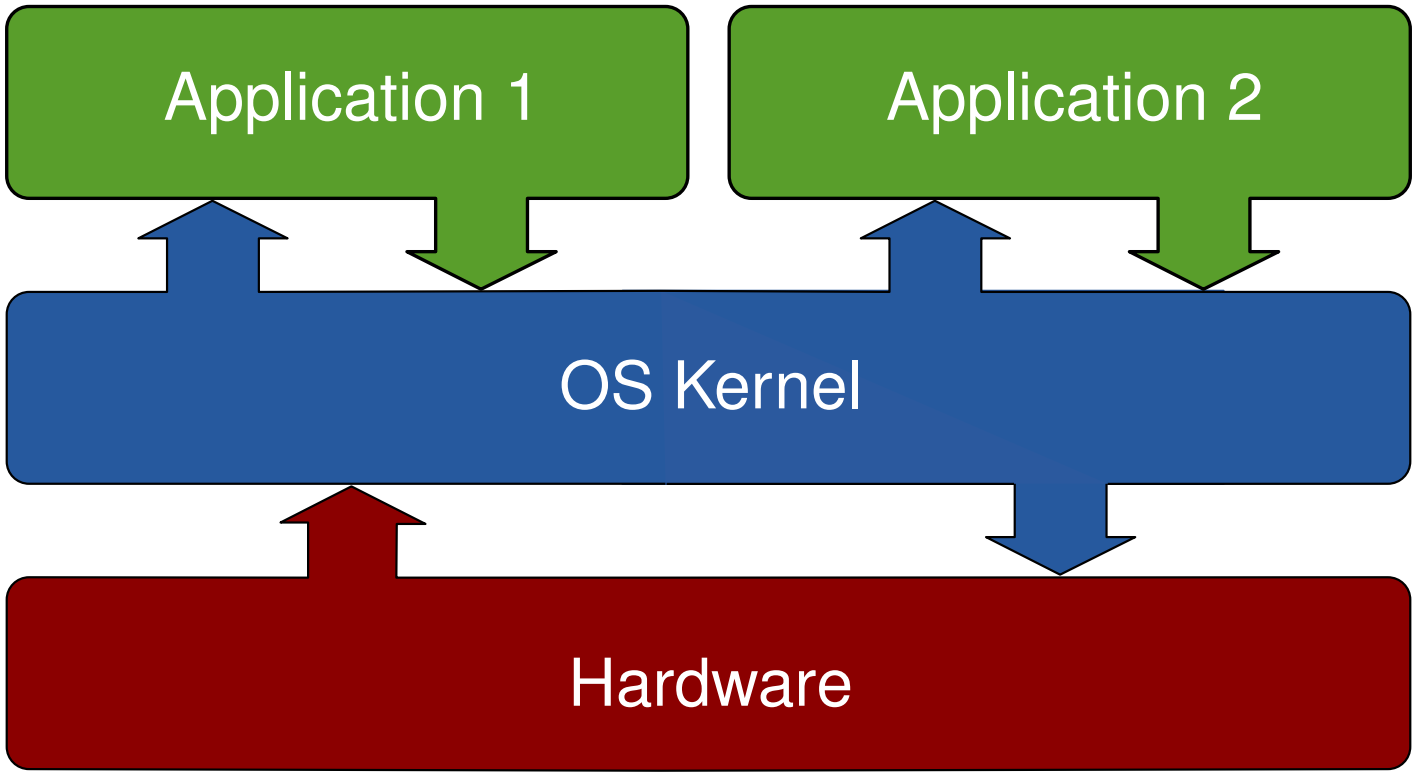
System calls

Guillaume Salagnac

Insa-Lyon – IST Semester

Fall 2019

Previously on IST-OPS

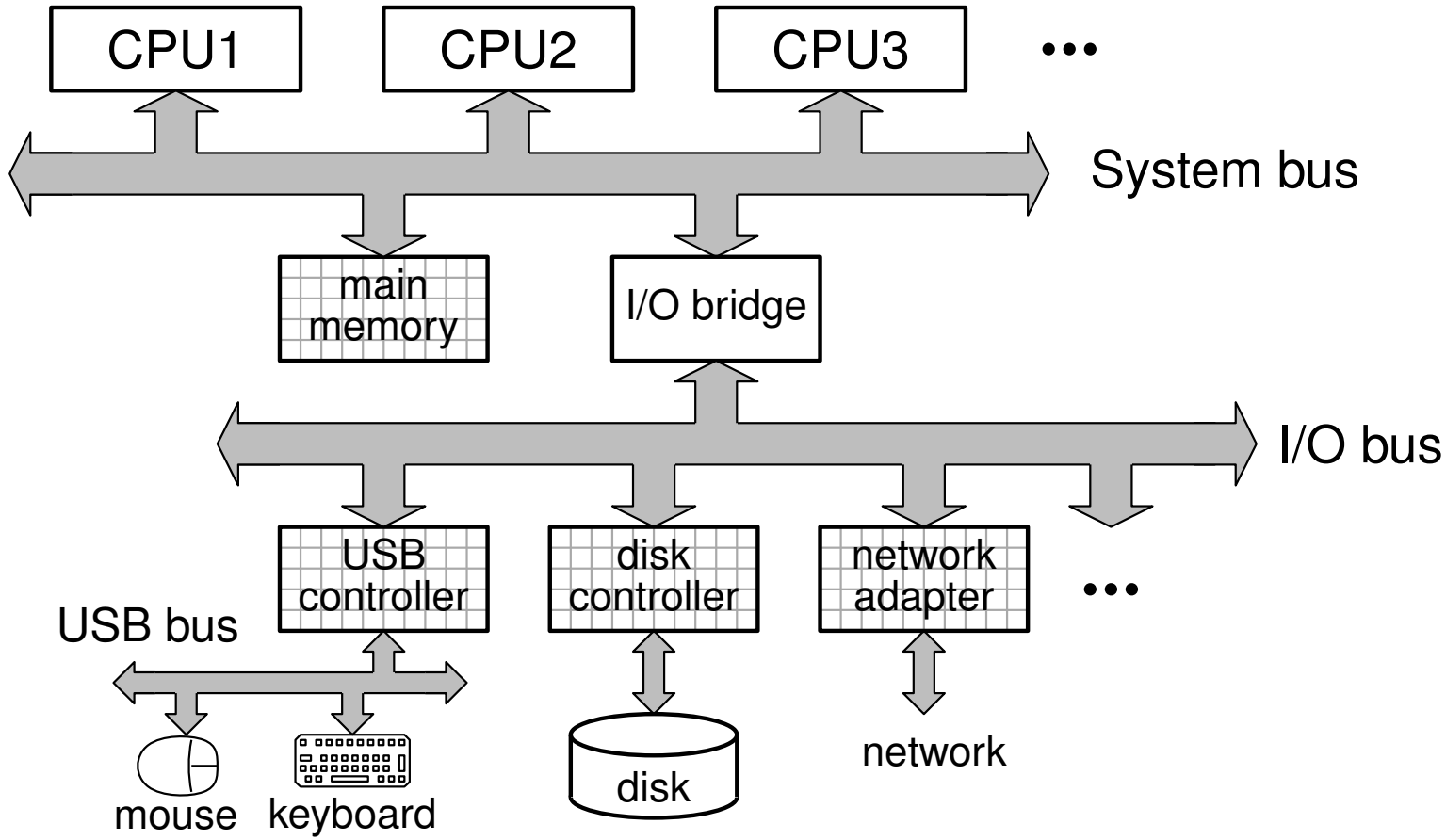


- The CPU implements the Von Neumann cycle
  - executes instructions one after the other
- The kernel is the part of the OS which is not an application
  - but what is it then ?

Outline

1. Interface between software and hardware peripherals
2. Interface between the kernel and applications: syscalls
3. A few important UNIX syscalls

Hardware architecture of your average computer



Warning: peripheral device $\neq$ device controller $\neq$ device driver !

How software talks to to hardware

PMIO = port-mapped input/output

- program uses special-purpose instructions

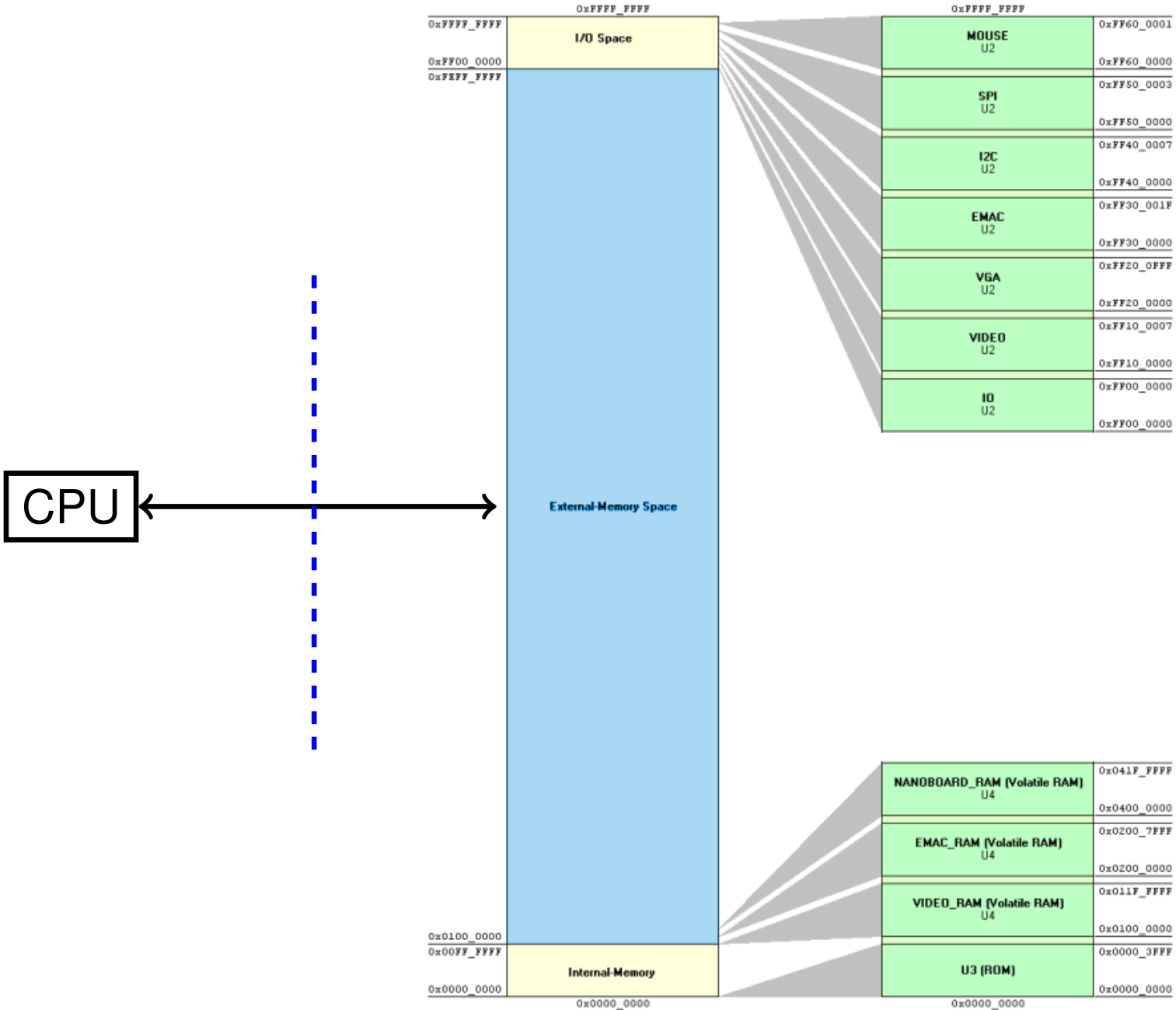
VS

MMIO = memory-mapped input/output

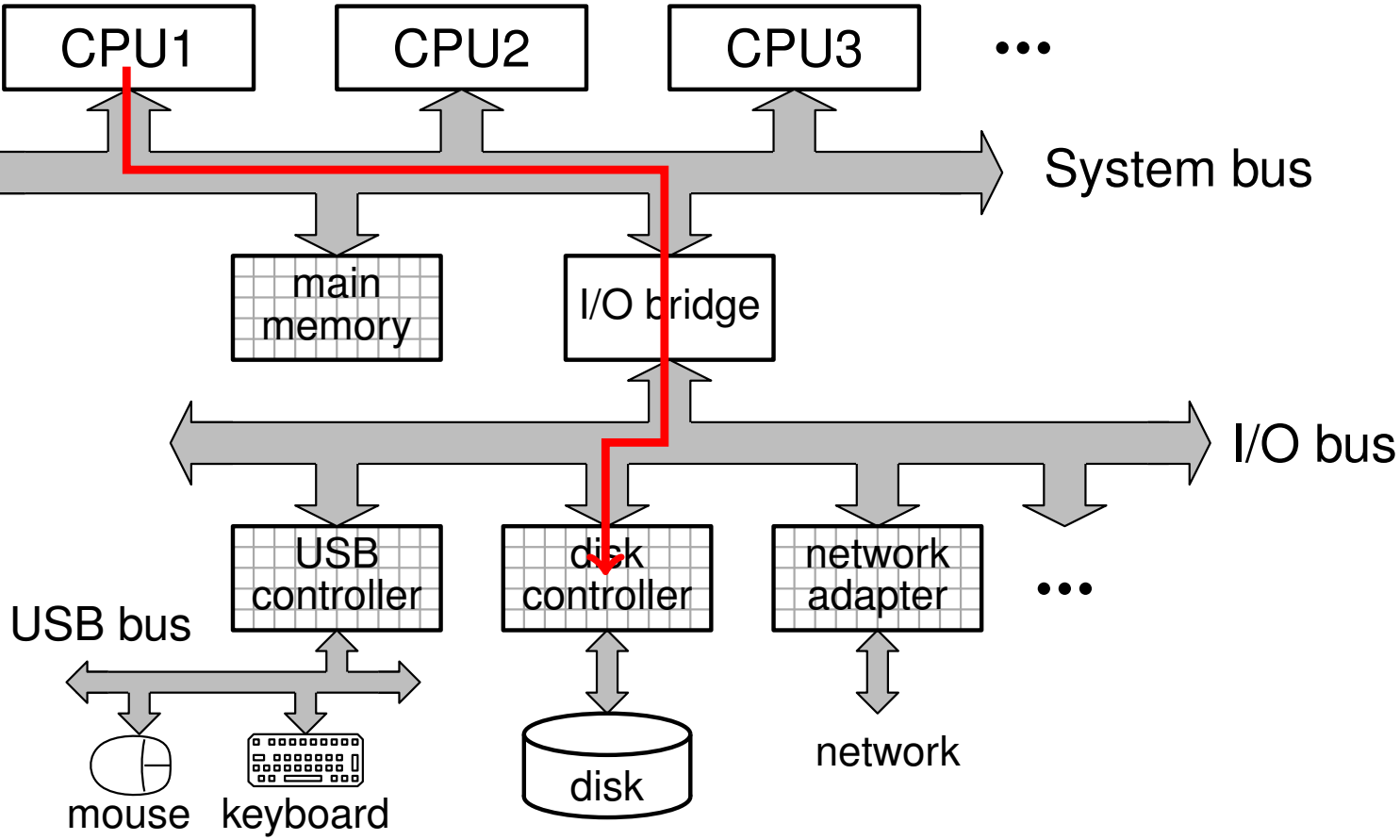
- program writes to special “memory” addresses

in real life: combination of both
in this course: only MMIO

Memory-mapped Input/Output: illustration

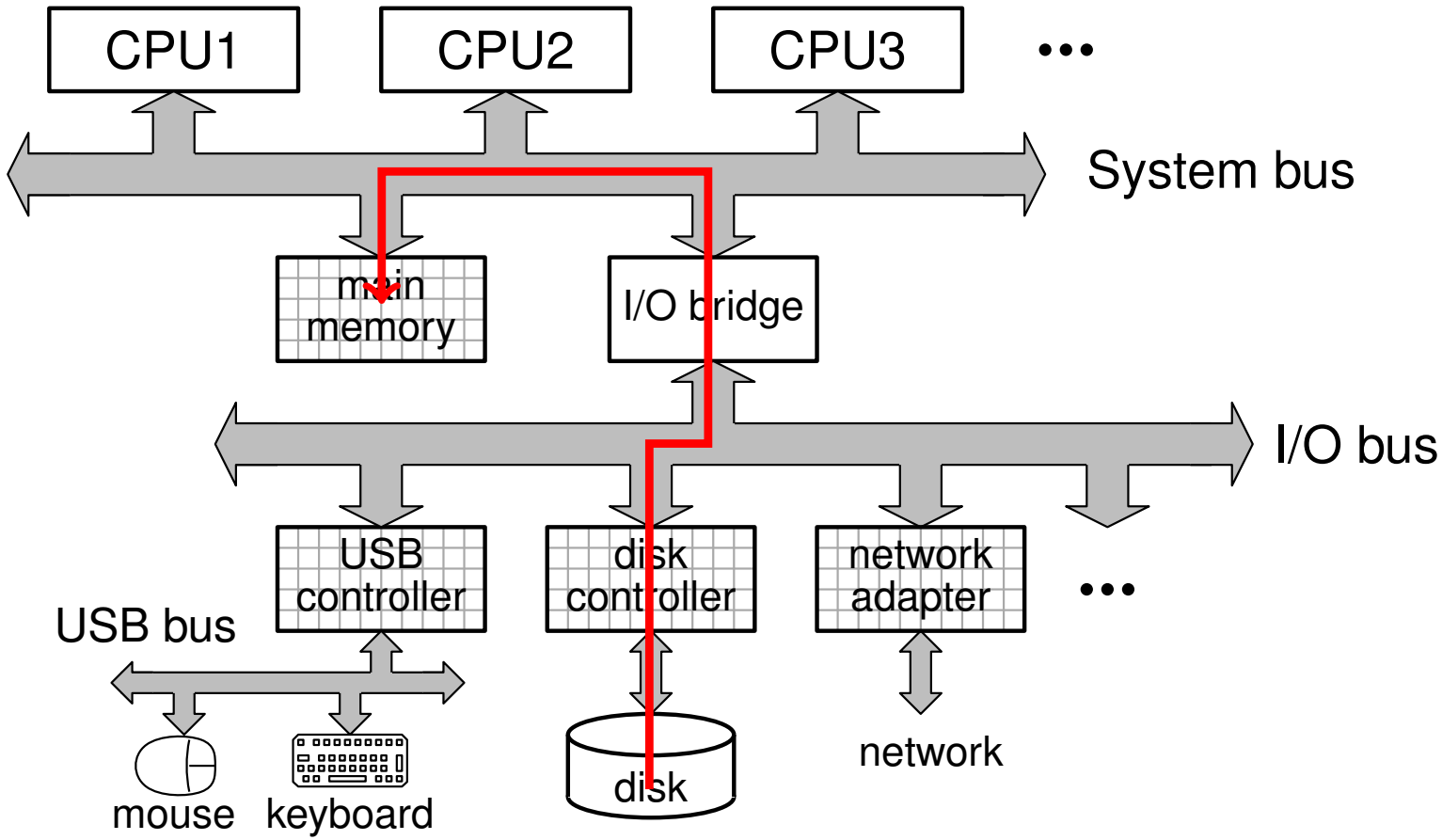


Example: reading a disk sector 1/3



To initiate a disk read operation:
CPU writes *command + block number + dest. mem. address*
to a memory address associated with the disk controller

Example: reading a disk sector 2/3



The disk controller reads the requested sector and then performs a Direct Memory Access (DMA) transfer into main memory

How hardware talks to software

Polling: frequently read the device registers from software

- complicated to write: how often is often enough ?
- inefficient at runtime: many CPU cycles wasted

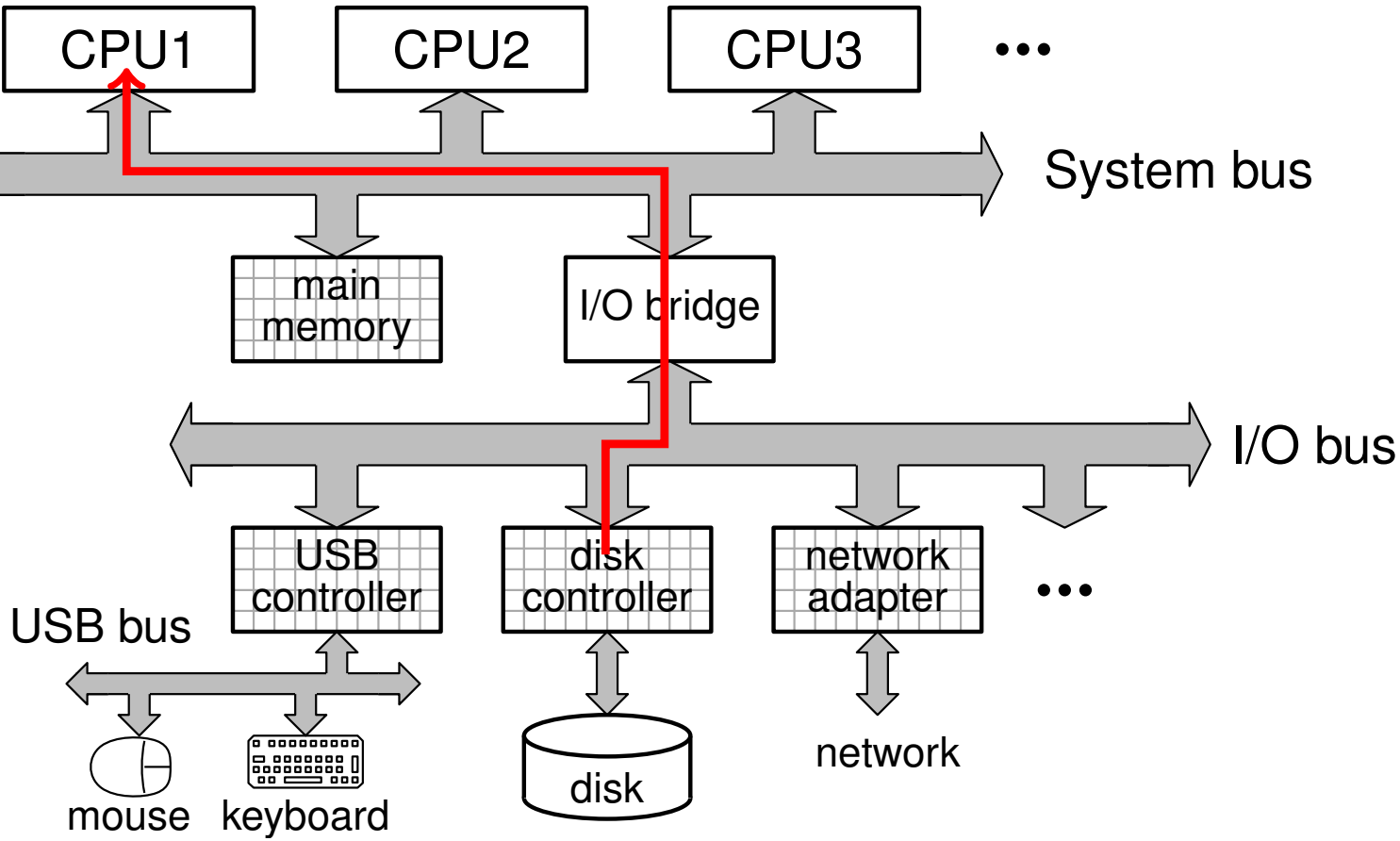
VS

Interrupts: allow outside events to preempt execution

- requires ad-hoc support in the processor...
- ... but all processors have it

in real life: avoid polling as much as possible

Example: reading a disk sector 3/3



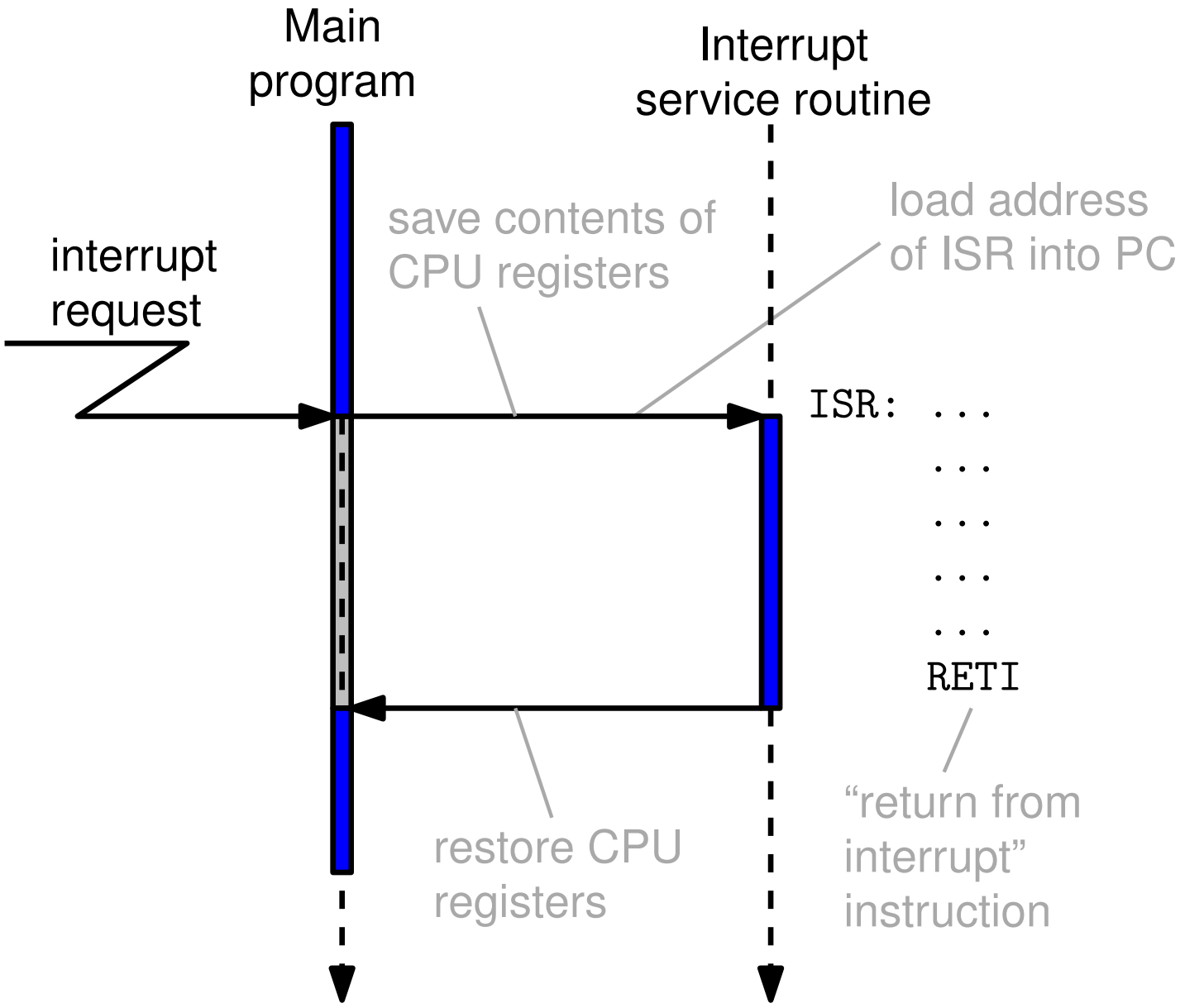
After DMA transfer is done, the disk controller notifies the CPU by sending it an Interrupt Request (IRQ)

Supporting interrupts in the processor

The Von Neumann cycle with interrupt support

```
while True do:
    fetch one machine instruction from “memory”
    decode its bits: which operation, which operands, etc
    execute the required action and store the result
    if interrupt requested then:
        save CPU state to memory: registers, PC, SR, etc
        lookup the address of the interrupt service routine
        jump there = load this address into PC
    endif
repeat
```

Interrupts: illustration



Do you speak interrupts ?

- IRQ = Interrupt Request
  - asynchronous «message» sent from peripheral to CPU
- ISR = Interrupt Service Routine
  - routine = program i.e. a sequence of instructions in memory
  - each ISR is located at a well-known address
  - must end with a `RETI` instruction
- concept of interrupt masking (aka disabling)
  - when interrupts masked $\Longleftrightarrow$ CPU ignores IRQs
  - any IRQ received is put on hold (not dropped)
  - implementation: boolean flag in the Status Register
- IRQs get automatically masked while running an ISR
  - driver code runs without a risk of being disturbed
  - instruction `RETI` enables interrupts again

Definition: operating system kernel

The kernel consists in all interrupt service routines (as well as all the functions called by these ISRs) and nothing more.

Interrupts come from various sources

- the System Timer
  - system tick = periodic IRQ, typically at 100Hz or 1000Hz
  - enables the OS to perceive the passing of time
  - bonus: allows the kernel to regain control of execution over untrusted application code
- input/output peripheral devices
  - keyboard, mouse, disk, network interface...
- hardware failures
  - overheating, power outage...
- exceptional software events
  - computational errors: division by zero, invalid instruction...
  - deliberate traps in the program (more on that in a minute)

Outline

1. Interface between software and hardware peripherals
2. Interface between the kernel and applications: syscalls
3. A few important UNIX syscalls

Instruction set vs Interrupts

Adding interrupt support requires new instructions:

- `RETI`: return to interrupted program (x86: `IRET`)
- `DINT`: disable (=mask) interrupts (x86: `CLI`)
- `EINT`: enable (=unmask) interrupts (x86: `STI`)

Problem

How can we protect the system against applications misusing these instructions?

example:

```
main: ...
      ...
      DINT
loop: JMP loop
      ...
```

Solution: “dual-mode operation”

all modern processors actually offer two execution modes :

supervisor mode = ring 0 = master mode = kernel mode

- unrestricted access to the real hardware
- useful to execute OS code

VS

user mode = ring 3 = slave mode

- restricted architecture: virtual machine
- some instructions are forbidden: `EINT/DINT`, `RETI`...
- useful to safely execute application (=untrusted) code

implementation: boolean flag in the Status Register

- CPU behaves differently depending on this mode bit

obviously: changing this flag is a priviledged instruction !

user mode ≠ userland

Applications execute on the "userland virtual machine"

- restricted instruction set (CPU in user mode)
  - no concept of interrupts
- restricted view on memory
  - access to some addresses denied: kernel code/data, peripheral devices, other processes' memory
  - implementation: in the «Bus interface» box (cf chap 3)

Sandboxing: one userland virtual machine per application

- virtual CPU (chap 2), virtual memory (chap 3)
- peripherals: only reachable by asking the kernel politely

Classical concept of the UNIX process

an instance of a computer program that is being executed

Operating system = illusionnist (VM) + sub-contractor (HW)

The UNIX process: remarks

Intuitively:

- one process = one program + its execution state
- execution state = value of CPU registers + memory contents

The kernel:

- shares available resources among active processes
- creates/recycles processes when necessary
  - one *Process Control Block* per process
  - PCB contains the process number aka PID

Try this on a Linux machine: `ps aux`, `top`

Dual-mode operation vs interrupts

Problem: interrupt routines must be able to access their devices

Solution: CPU changes modes when jumping to an ISR

The Von Neumann cycle with dual-mode operation

```
while True do:
    fetch, decode, execute
    if (IRQ received) and (interrupts not masked) then:
        save CPU state
        switch to supervisor mode and mask further interrupts
        lookup the ISR address
        jump to the routine = load its address into PC
    endif
repeat
```

Note: `RETI` will eventually switch us back to user mode

Dual-mode operation vs traps

Problem: how can an application invoke a kernel service ?

Bad idea: allow application code to branch into kernel code

- branch target location is arbitrary ▶ security problems
- switch to supervisor mode ▶ where ? how ?

Solution: add an instruction dedicated for this purpose

- various names: `TRAP` (68k), `INT` (x86), `SWI` (ARM)
- software interrupt = trap = exception
- works in the same way as other interrupts
  - save CPU state
  - switch to supervisor mode
  - branch to associated service routine

▶ concept of system call

System call: working principle

System call (aka syscall)

a software routine implemented in the kernel but invoked by a process through a software interrupt

userland side

- call is invoked via machine instruction `TRAP`
- agnostic to the programming language
- usually encapsulated into library function (ex: libc)

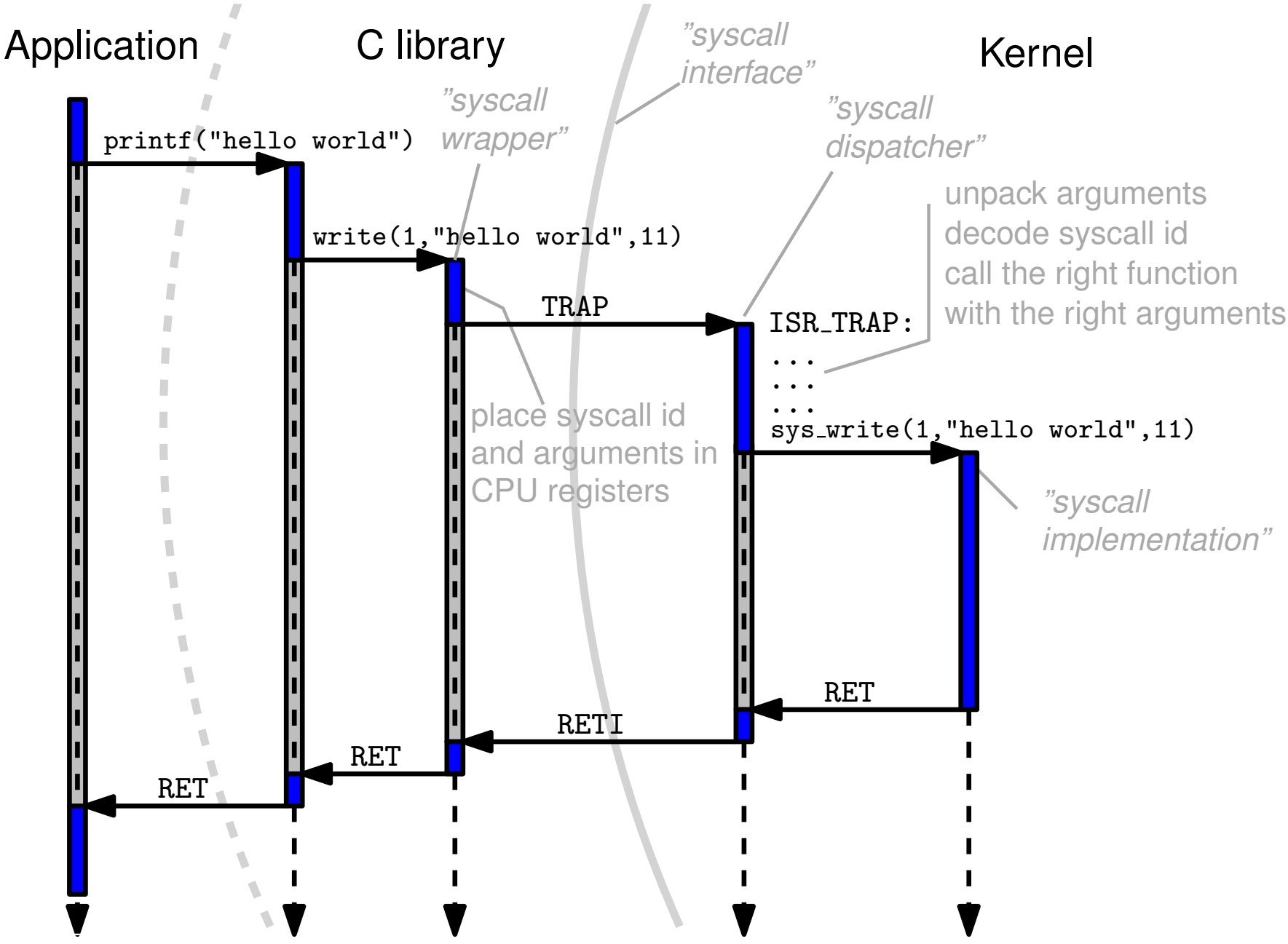
kernel side

- all syscalls go through the ISR associated with `TRAP`
- which itself calls the right function in the kernel,
- and finally returns execution to the application with `RETI`

Examples

- `read()`, `write()`, `fork()`, `getpid()`, `gettimeofday()`...
- Linux has hundreds of different syscalls

System call: illustration



Observe system calls in action: `ltrace` and `strace`

Try this on a Linux machine:

hello.c

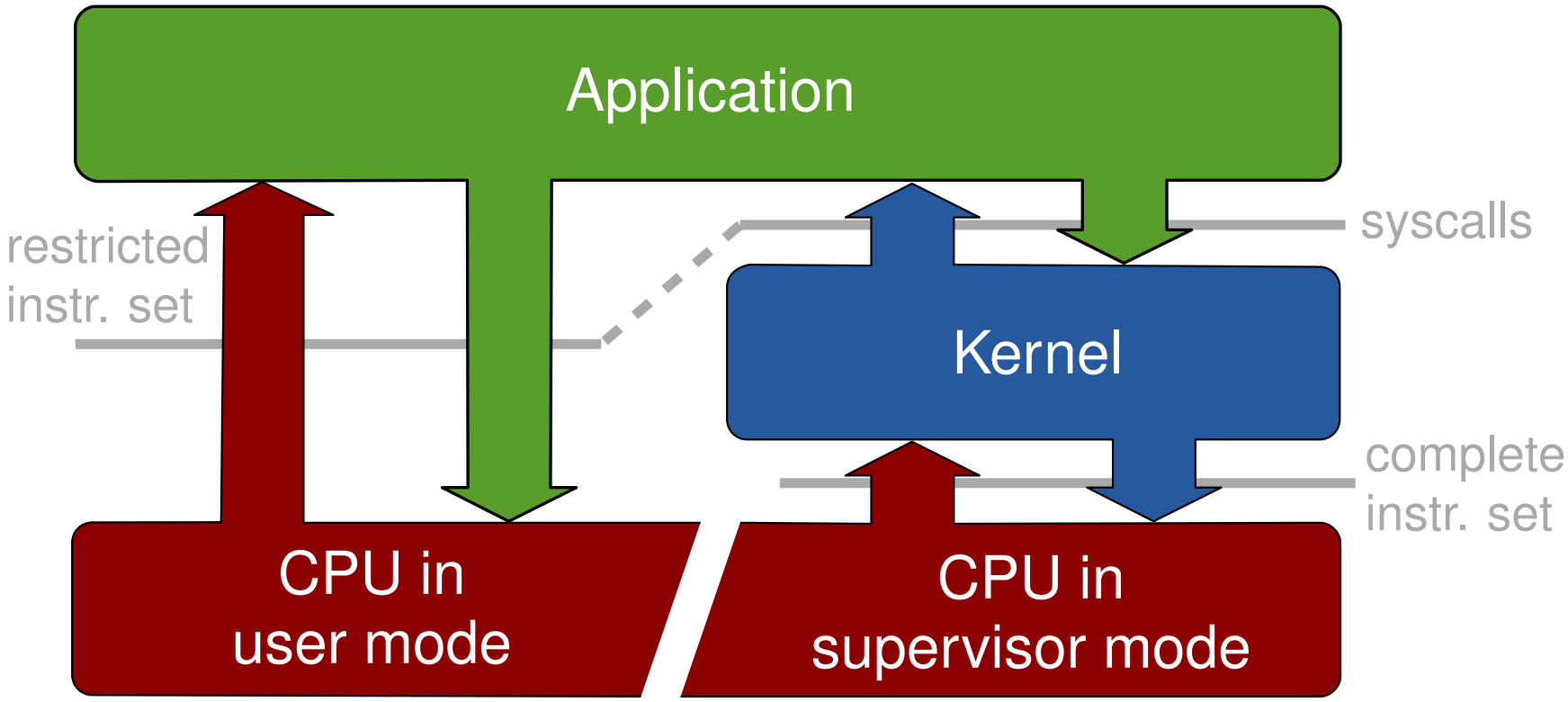
```
#include <stdio.h>

void main()
{
    printf("hello linux world !\n");
}
```

- `gcc -o ./hello ./hello.c`
- `ltrace ./hello`
- `strace ./hello`

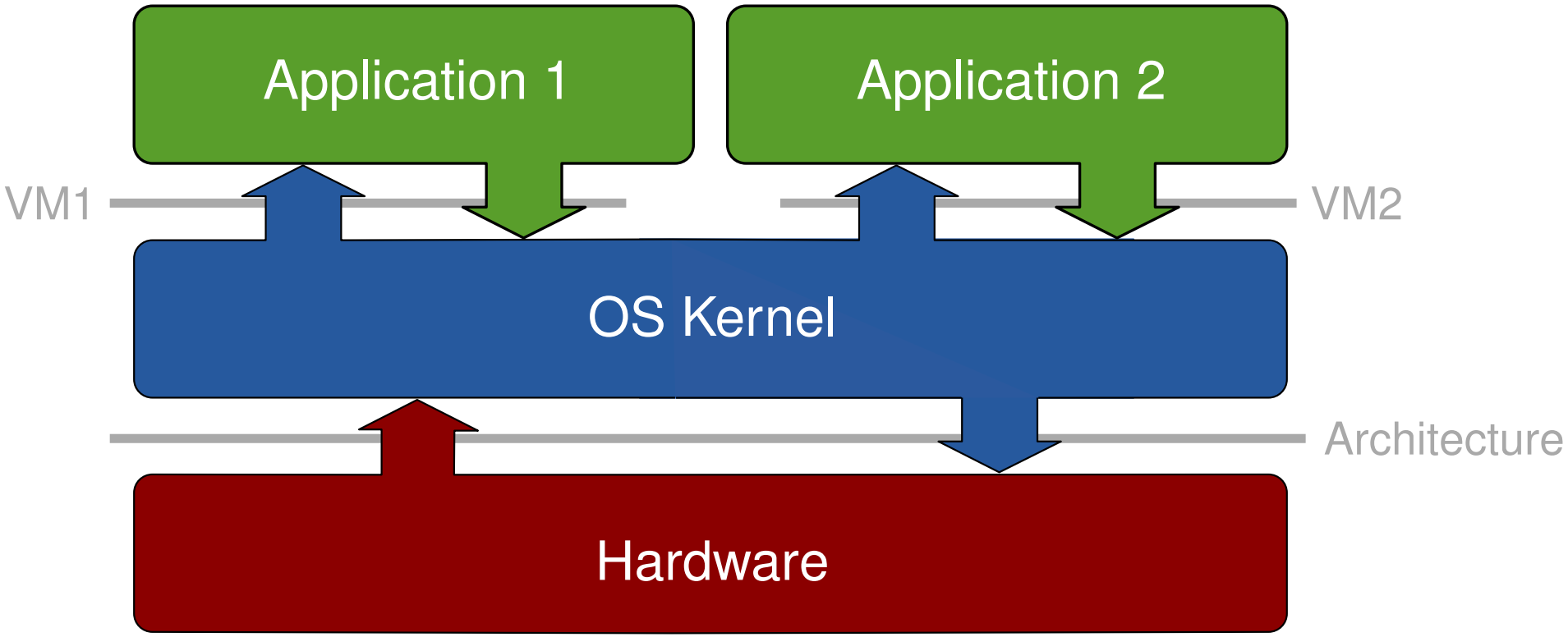
also: `man ltrace` and `man strace`

Userland VM: summary



- application code executed directly by the CPU but in user mode
- to invoke a kernel method: must use the system call interface

Operating system placement



- each application runs as a userland process
- the kernel mediates all accesses to peripherals

Outline

1. Interface between software and hardware peripherals
2. Interface between the kernel and applications: syscalls
3. A few important UNIX syscalls

System calls exist on all operating systems

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shmget()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

source: Silberschatz. *Operating Systems Concepts Essentials (2011)*. p 59

A C function with a syscall inside: gettimeofday()

gsalagnac@plouf: man gettimeofday — 80×24 — ⌘5

```
GETTIMEOFDAY(2)            BSD System Calls Manual            GETTIMEOFDAY(2)

NAME
     gettimeofday, settimeofday -- get/set date and time

SYNOPSIS
     #include <sys/time.h>

     int
     gettimeofday(struct timeval *restrict tp, void *restrict tzp);

     int
     settimeofday(const struct timeval *tp, const struct timezone *tzp);

DESCRIPTION
     The system's notion of the current Greenwich time and the current time
     zone is obtained with the gettimeofday() call, and set with the
     settimeofday() call.  The time is expressed in seconds and microseconds
     since midnight (0 hour), January 1, 1970.  The resolution of the system
     clock is hardware dependent, and the time may be updated continuously or
     in ``ticks.''  If tp is NULL and tzp is non-NULL, gettimeofday() will
     populate the timezone struct in tzp.  If tp is non-NULL and tzp is NULL,
:
```

System calls: Remarks

Syscalls usually shown as C functions (because history)

But even in C, system call $\neq$ library call

- details depends on your kernel
- Windows vs Linux vs Mac OS X vs FreeBSD vs ...

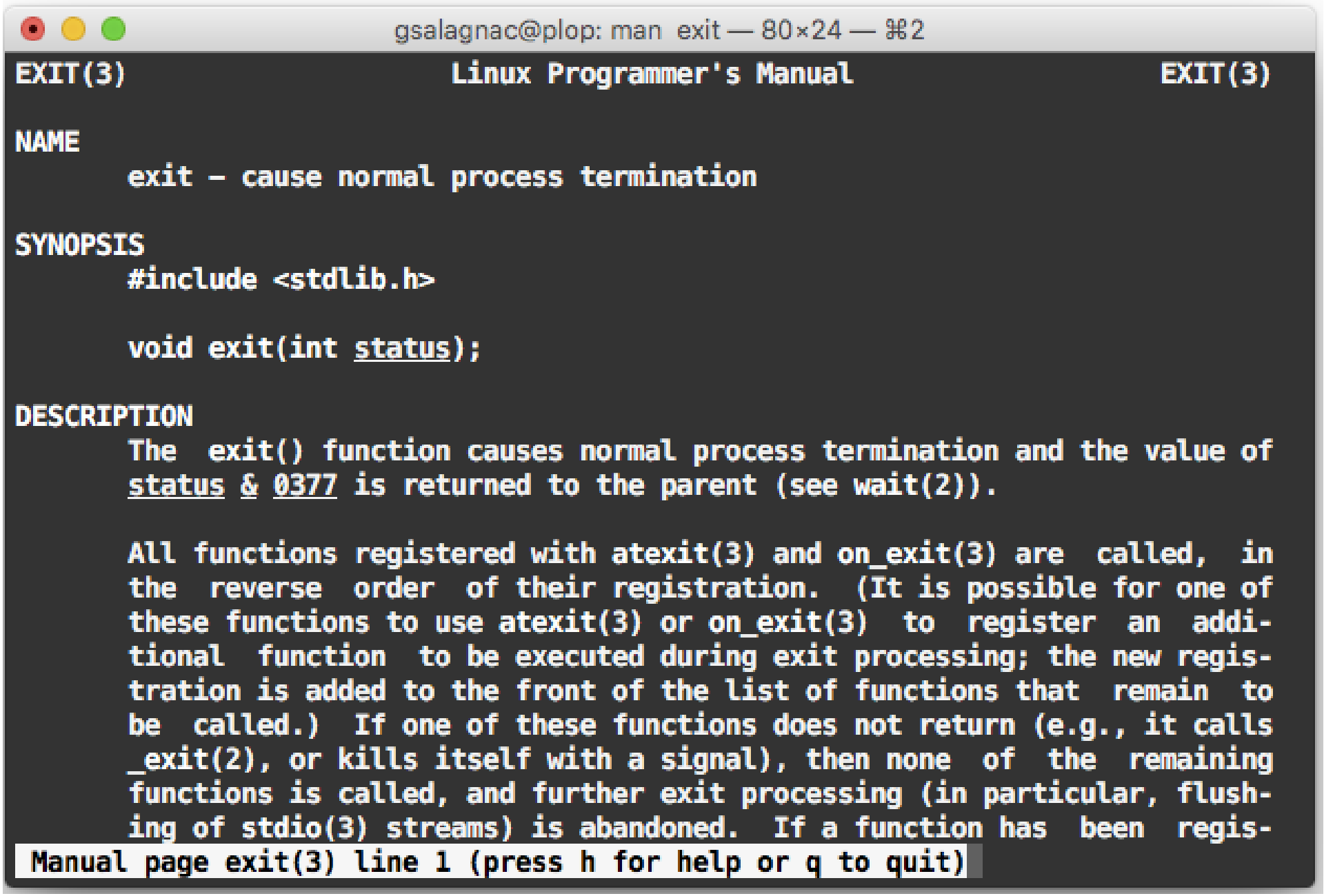
RTFM: `man gettimeofday` = `man 2 gettimeofday`

Every UNIX has several sections for manual pages:

1. shell commands: `ls`, `cd`, `cat`...
2. system calls: `getpid()`, `open()`, `read()`...
3. C library functions: `printf()`, `malloc()`, `sqrt()`...
4. special files (`/dev/...`) and device drivers

... etc

Terminating the current process: exit()



Creating a new process: fork()

gsalagnac@plop: man 2 fork — 80×24 — ⌘3

```
FORK(2)                    Linux Programmer's Manual                    FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a new process by duplicating the calling process.  The
       new process is referred to as the child process.  The  calling  process
       is referred to as the parent process.

       The child process and the parent process run in separate memory spaces.
       At the time of fork() both memory spaces have the same content.  Memory
       writes,  file  mappings (mmap(2)), and unmappings (munmap(2)) performed
       by one of the processes do not affect the other.

       The child process is an exact duplicate of the  parent  process  except
       for the following points:

 Manual page fork(2) line 1 (press h for help or q to quit)
```

The fork() syscall: remarks

This is the only way to create a UNIX process

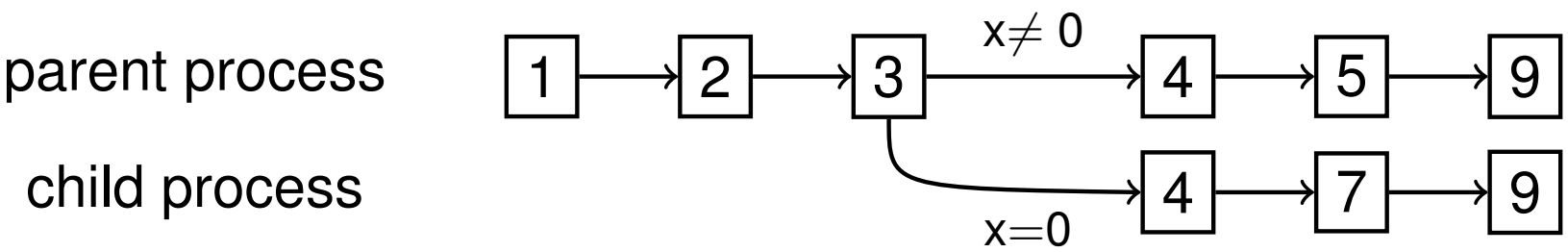
- `fork()` duplicates the calling process
- both processes then execute concurrently
- each process has a its distinct memory space

"call once, return twice" paradigm

- in the new process (aka child) `fork()` returns 0
- in the original process, `fork()` return the PID of the child

The fork() system call: illustration

```
1 // only one process
2 int y = 5 ;
3 int x = fork();
4 if ( x != 0 ) {
5   // parent only
6 } else {
7   // child only
8 }
9 // both processes
```



Changing programs in the same process: exec()

gsalagnac@plouf: man execl — 80×24 — ⌘2

```
EXEC(3)                   BSD Library Functions Manual                   EXEC(3)

NAME
     execl, execle, execlp, execv, execvp, execvP -- execute a file

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <unistd.h>

     extern char **environ;

     int
     execl(const char *path, const char *arg0, ... /*, (char *)0 */);

DESCRIPTION
     The exec family of functions replaces the current process image with a
     new process image.  The functions described in this manual page are
     front-ends for the function execve(2).  (See the manual page for
     execve(2) for detailed information about the replacement of the current
     process.)
:
```

Wait until a child process terminates: wait()

gsalagnac@plouf: man 2 wait — 80×24 — ⌘2

```
WAIT(2)                    BSD System Calls Manual                    WAIT(2)

NAME
     wait, wait4, waitpid -- wait for process termination

SYNOPSIS
     #include <sys/wait.h>

     pid_t
     wait(int *stat_loc);

     pid_t
     wait4(pid_t pid, int *stat_loc, int options, struct rusage *rusage);

     pid_t
     waitpid(pid_t pid, int *stat_loc, int options);

DESCRIPTION
     The wait() function suspends execution of its calling process until
     stat_loc information is available for a terminated child process, or a
     signal is received.  On return from a successful wait() call, the
     stat_loc area contains termination information about the process that
     exited as defined below.
:
```

My first command-line shell

```
char command[...];
char  params[...];
main()
{
    while(true)
    {
        print_prompt();
        read_command(&command, &params);
        pid=fork();

        if (pid != 0) {
           wait(&status);
        } else {
           exec(command, params);
        }
    }
}
```

Outline

1. Interface between software and hardware peripherals
2. Interface between the kernel and applications: syscalls
3. A few important UNIX syscalls

Summary

Computer architecture

- the Von Neumann cycle + interrupts
- dual-mode operation: supervisor mode vs user mode

Kernel

- consists in all Interrupt Service Routines
  - including the *syscall dispatcher* and the system timer ISR
- and all functions called by these ISRs

Userland

- a “virtual machine” for applications to run on
- simplified, restricted view of the underlying architecture

System call

- interface for applications to invoke kernel methods
- TRAP instruction encapsulated in library function

OS = kernel + libraries + utility programs