INF113: Processes

Kirill Simonov

03.09.2025

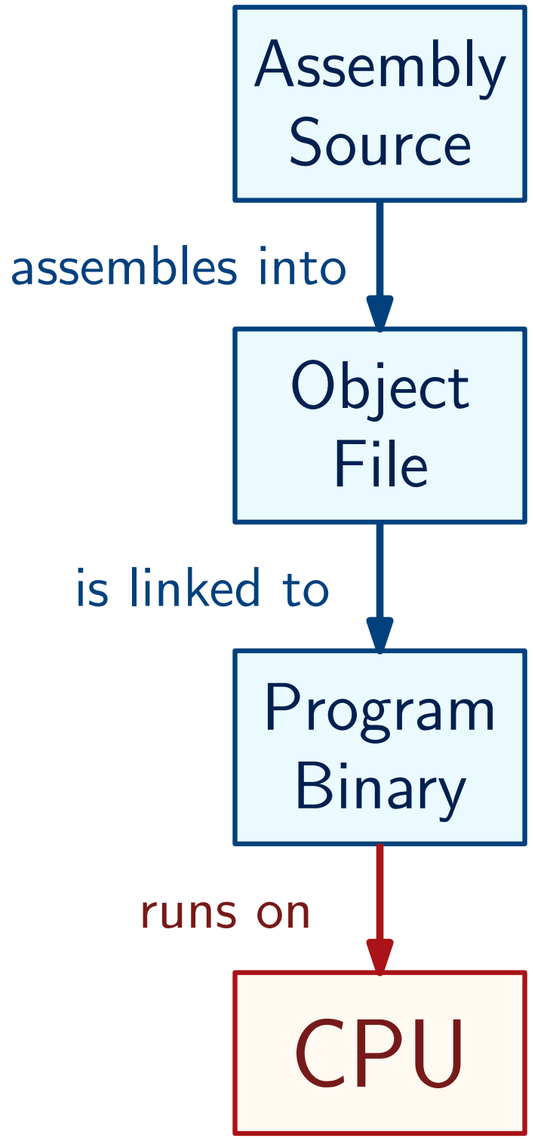
Orga

- **Group sessions finalized:**
  1 - Monday 08:15-10:00, TM51 Room A
  2 - Tuesday 10:15-12:00, RFB Grupperom 4
  4 - Thursday 10:15-12:00, **RFB Grupperom 1**
  (up to week 43, then 209M1 again)
  5 - Friday 10:15-12:00, TM51 Room C
  6 - Friday 12:15-14:00, Biologen Blokk B, K3+K4

- Lectures will continue to be recorded

- Homework: In the end of Chapter 4, `process-run.py`

From a program to a process




```
as -o program.o program.s
```

```
ld -o program program.o [...]
```

also external binaries

```
./program
```

Programs vs processes

- A **program** is a sequence of instructions

```
.code64
.section .rodata
msg:
    .ascii "Hello, world!\n"
.section .text
.global _start
_start:
    mov $1, %rax
    mov $1, %rdi
    lea msg, %rsi
    mov $14, %rdx
    syscall
    mov $60, %rax
    xor %rdi, %rdi
    syscall
```

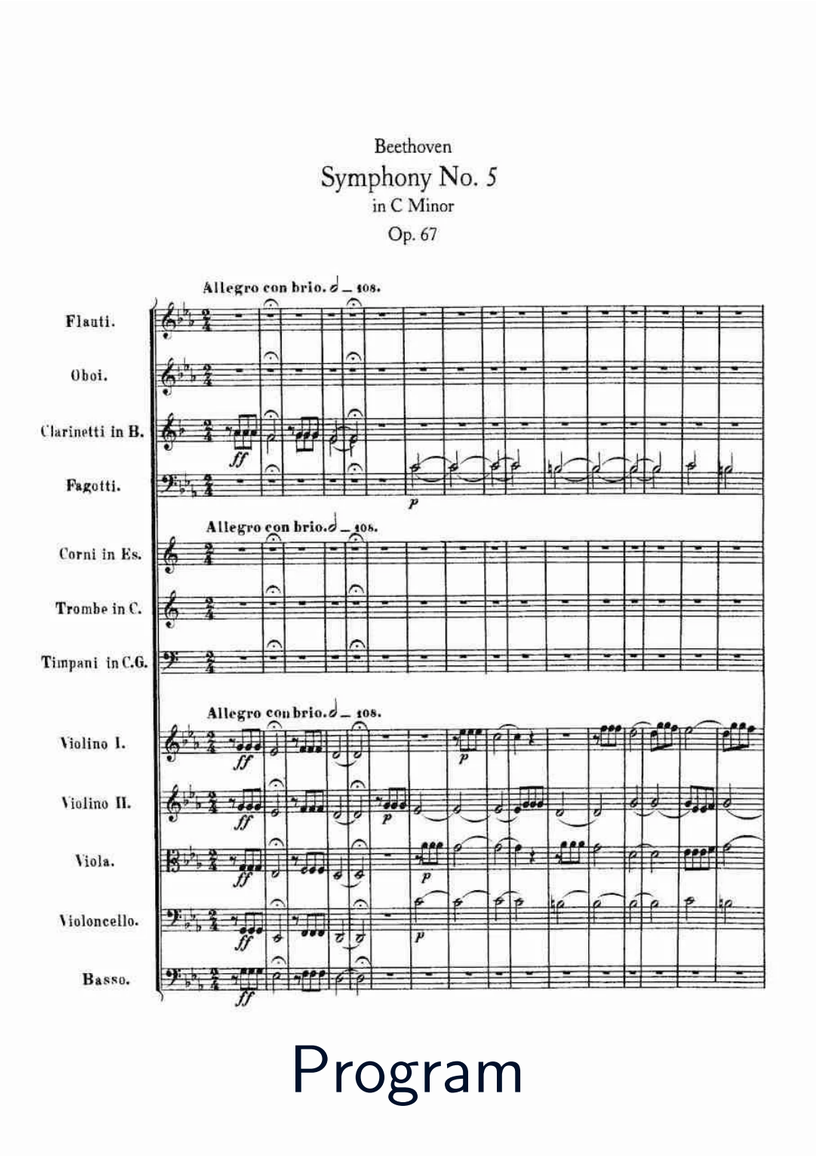
- A **process** is a *running* program

```
$ ./hello
Hello, world!
```

```
$ ps
    PID TTY          TIME CMD
 639560 pts/4    00:00:00 bash
 725811 pts/4    00:00:00 hello
 725844 pts/4    00:00:00 ps
```

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 725811 seemann   20   0    2680   1384   1384 S   0.0   0.0   0:00.00 hello
```

Programs vs processes



Program

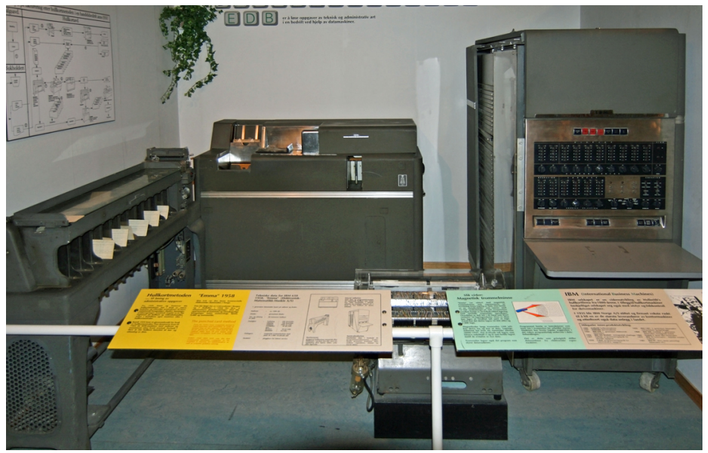
https://en.wikipedia.org/wiki/File:MITO_Orchestra_Sinfonica_RAI.jpg

Process

Processes and the OS

- The concept of a process becomes relevant within the OS

- Single-process machine: a program is just run as is



- Modern OS runs a multitude of processes
  - We can add/kill/inspect processes
  - Loading/unloading mechanism to "run at the same time"

- *Process* as the interface to deal with running programs

Caging the process

- A process should not (directly)
  - Access memory of another process
  - Write to disk
  - Communicate with devices

Challenge: Still, these functions should be there

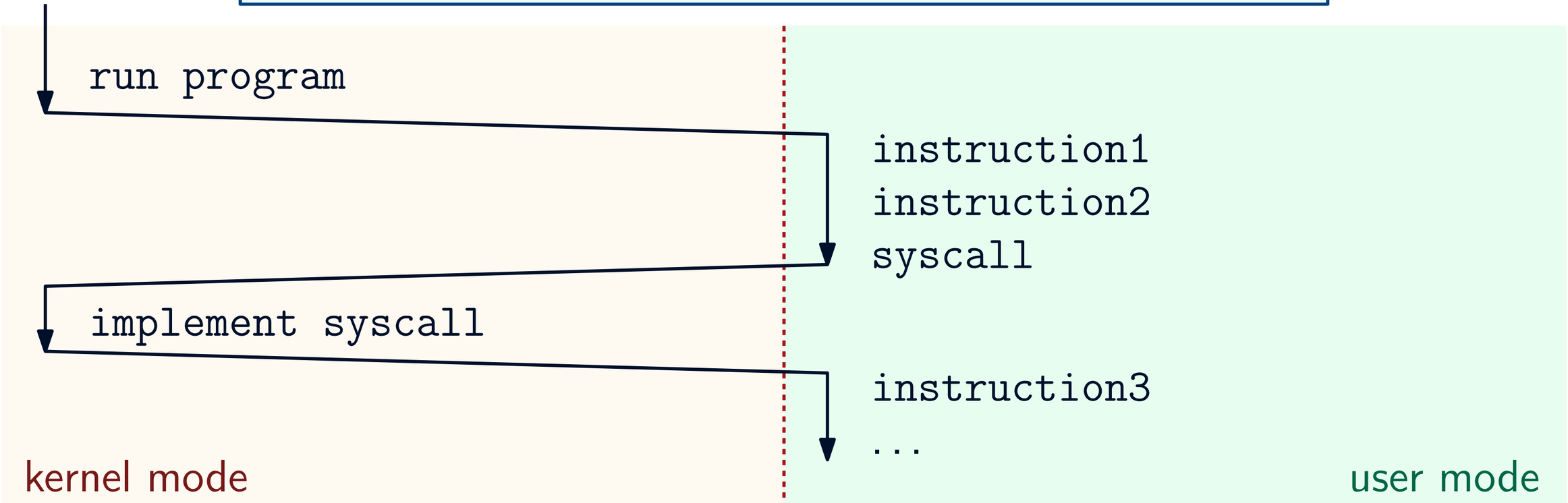
Solution: Two modes in the CPU!

- **Kernel mode**
  - CPU will run any instruction
  - Only for the OS
- **User mode**
  - CPU will only run "safe" instructions
  - OS will switch to user mode before running any process

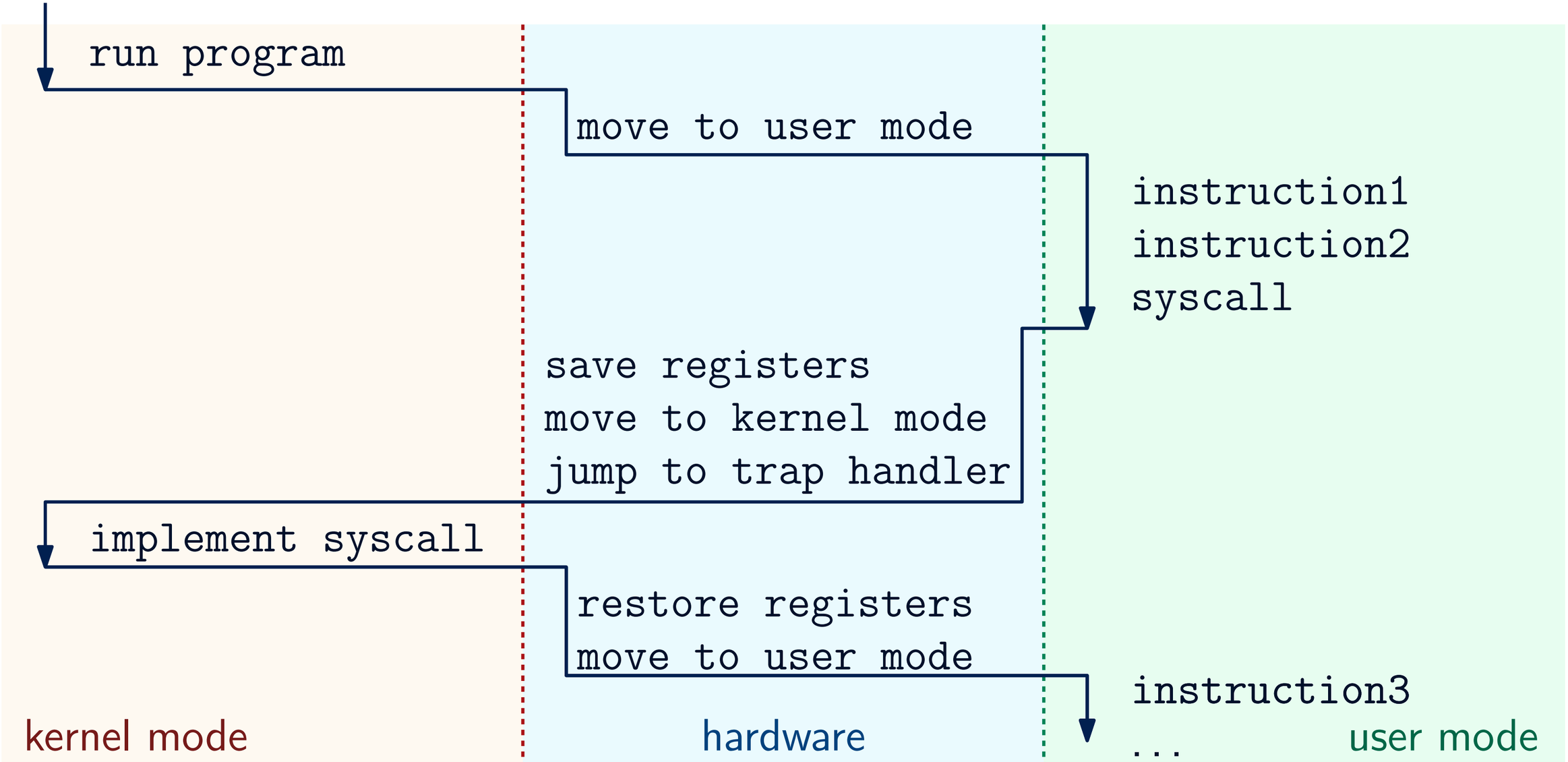
System calls

- **Challenge:** Programs still want to
  - Perform I/O
  - Dynamically allocate memory

Solution a.k.a. system call: Process asks, OS delivers



POV: Hardware



POV: Program

The program makes a `syscall`:

1. Write what exactly needs to be done
2. Pass the control back to the OS
3. OS checks that instructions left by the program are ok
4. OS completes the action and passes control back

```
.code64

.section .text
.global _start

_start:
    mov $19, %rax
    mov $16, %rbx
    add %rax, %rbx

    mov %rbx, %rdi
    mov $60, %rax
    syscall
```

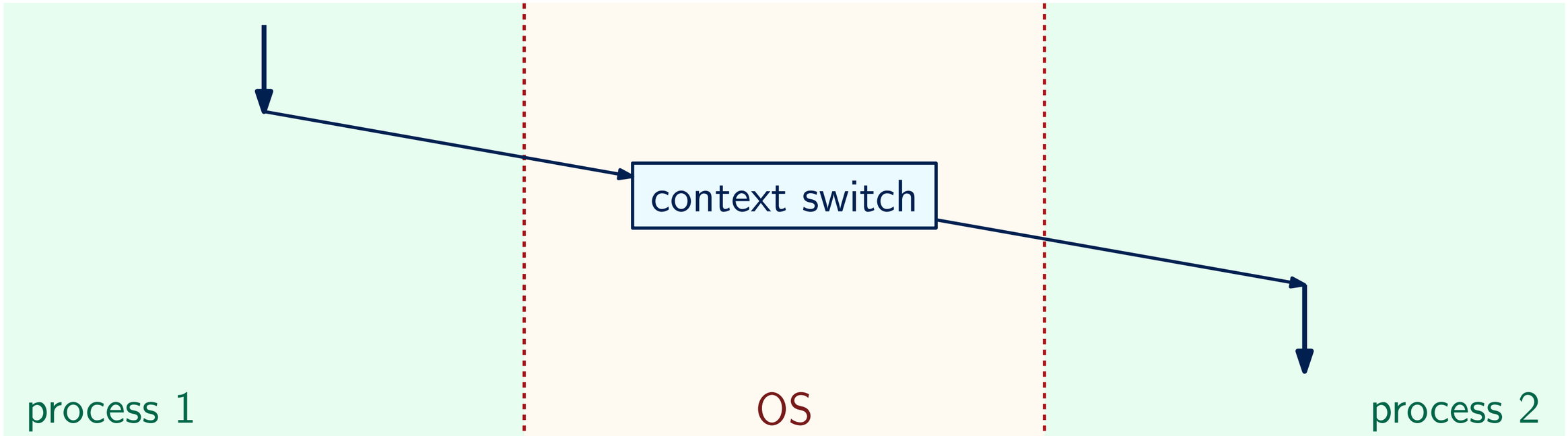
Here: the value 60 in `%rax` is interpreted as the `exit` command
OS expects to find the `exit` code in `%rdi`

Time sharing

- OS can keep multiple processes safe from each other via **user mode**

- But how to run many processes *at the same time*?
  – One CPU core goes through one sequence of instructions, one at a time

- **Solution:** Just *switch* the running process often enough so the user does not notice 😈

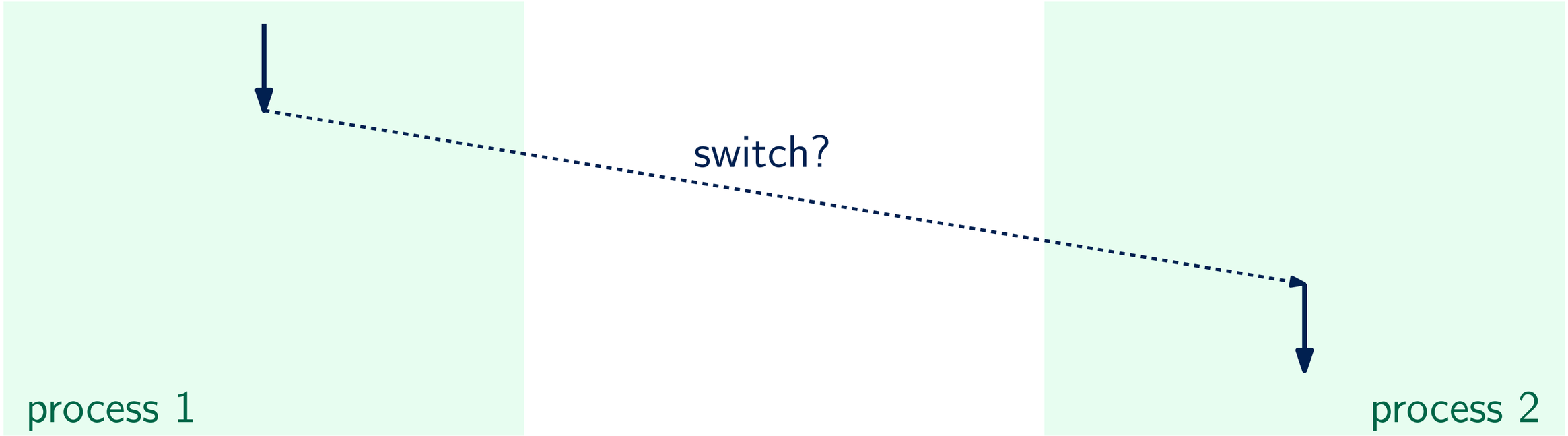


Time sharing: How-to

Policy: Scheduling

will see next week

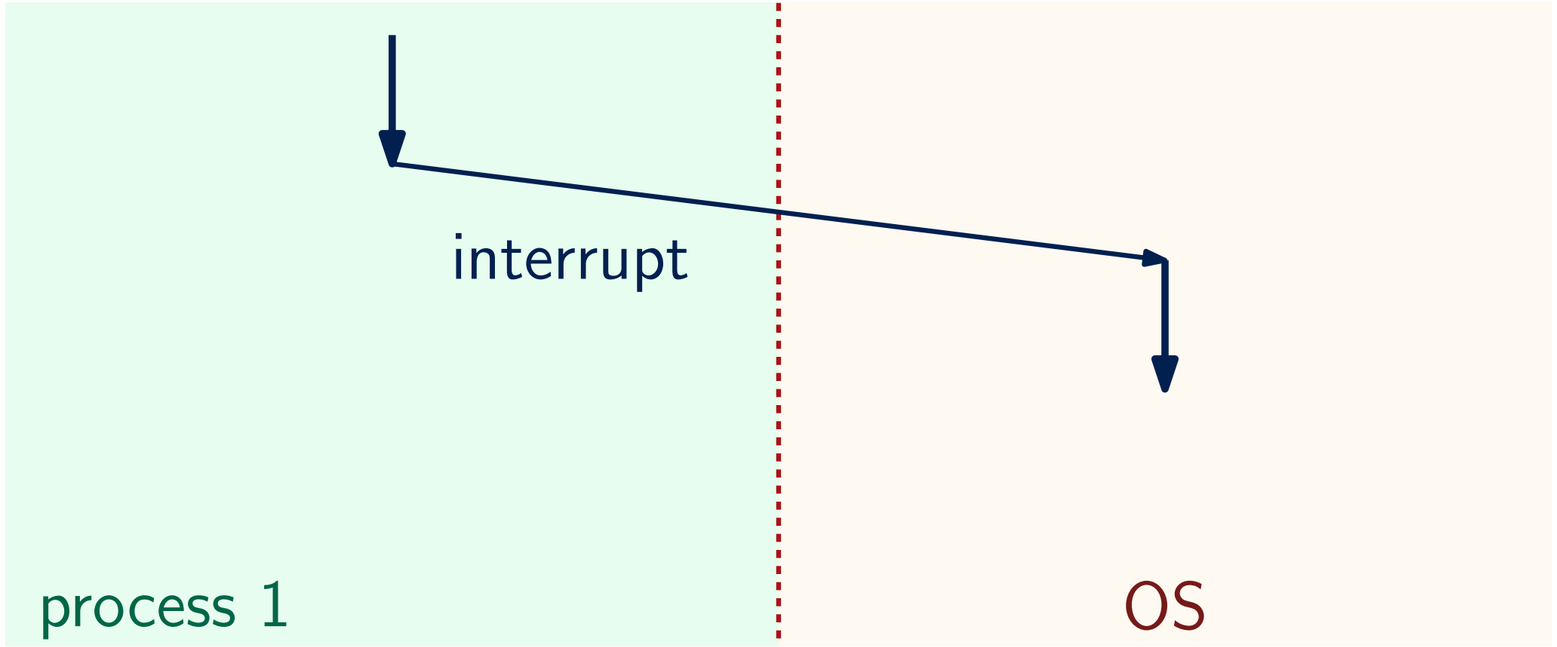
- I.e., which process gets to run?
- Challenge: achieve perfect interactivity
- While leaving enough compute for time-intensive programs
- And not spending too much resources on switching



Time sharing: How-to

Mechanism: Interrupt

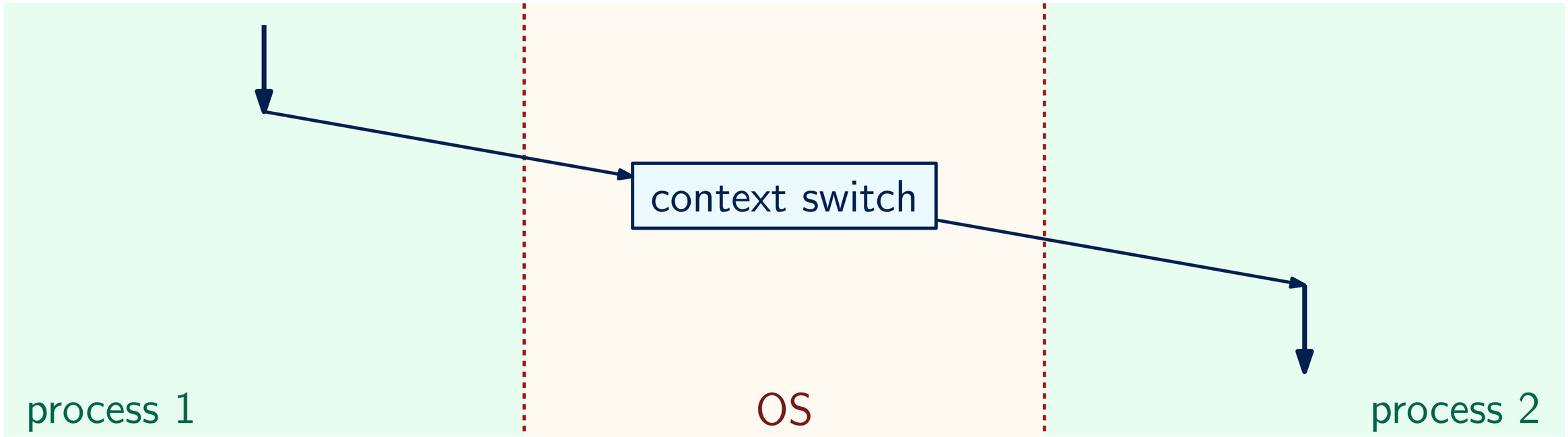
- I.e., how to gain control to perform the switch
- A process should not be able to monopolize the CPU
- Yet the OS is on hold while it is run



Time sharing: How-to

Mechanism: Context switch

- I.e., how to actually perform the switch
- OS should save a complete snapshot of the process
- To seamlessly restore it later
- While using only a handful of resources

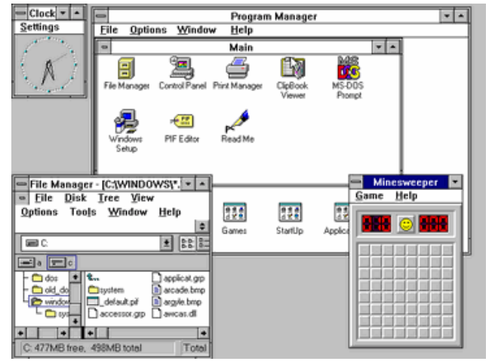


Interrupts

- OS needs to be in control to even consider switching
- Recall: OS gains control upon a system call
- But what if the process makes no system calls? i.e., endless loop or a long computation

Solution 1: Cooperative approach

- Assume that any reasonable program makes a system call soon enough
- Could even be a special "yield" syscall that expects no action
- Examples: Windows 3.1, classic MacOS, modern embedded systems
- However, too optimistic...

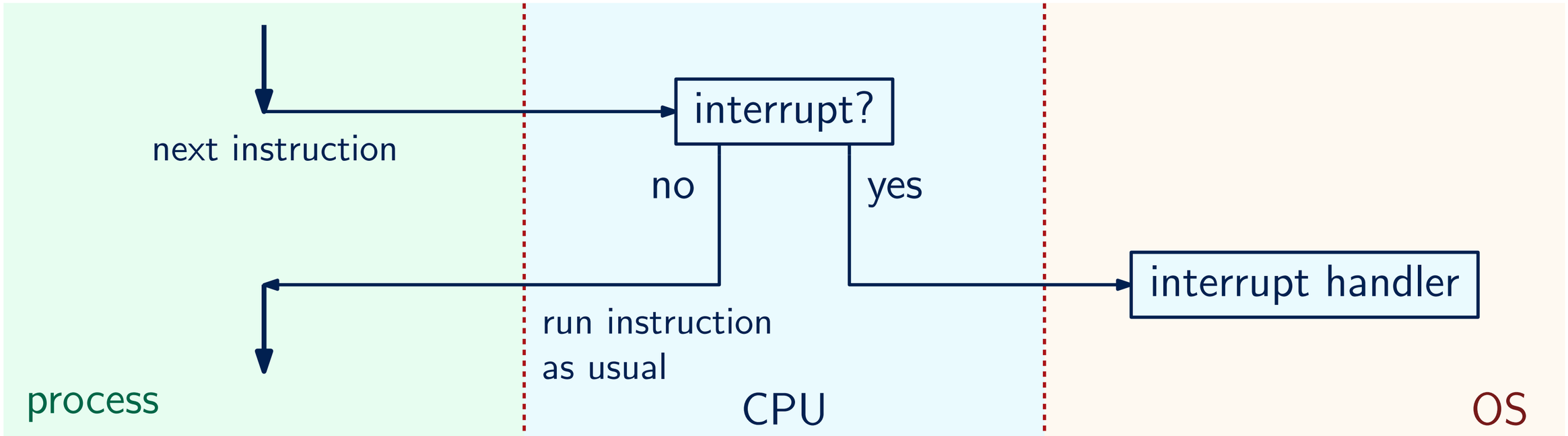


Interrupts

Solution 2: Non-cooperative approach

used everywhere now

- Needs additional hardware support
- *Interrupt timer:* CPU periodically stops the process regardless, and calls OS
- Special address points to *interrupt handler*, initialized at boot



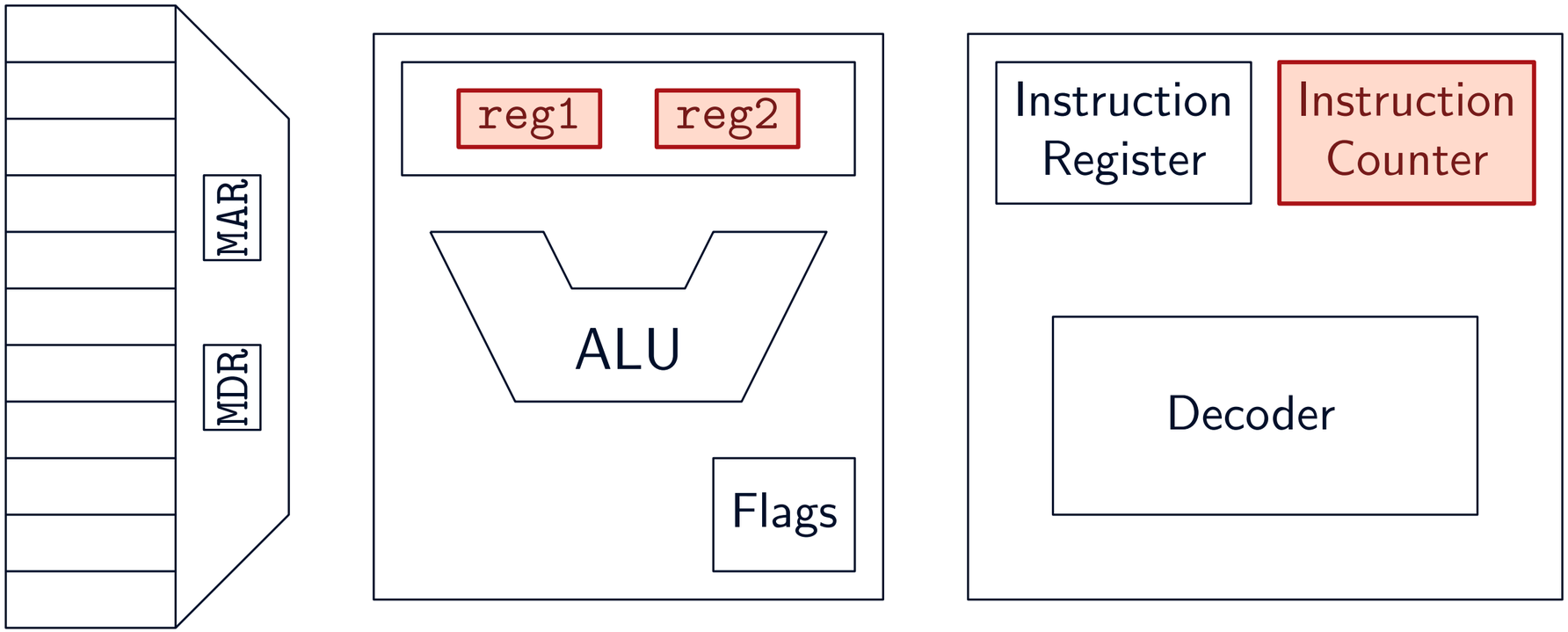
Context switching

Main task: Stop the process, so that it can be restored later

- What do we need to describe the *state* of the process?
- Content of the registers describes the state of the CPU

could safely save them

- OS also stores its own internal information about processes



Storing processes

- OS keeps a list of all processes
- Example: process data in xv6

xv6—a simplified Unix-like system

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };
```

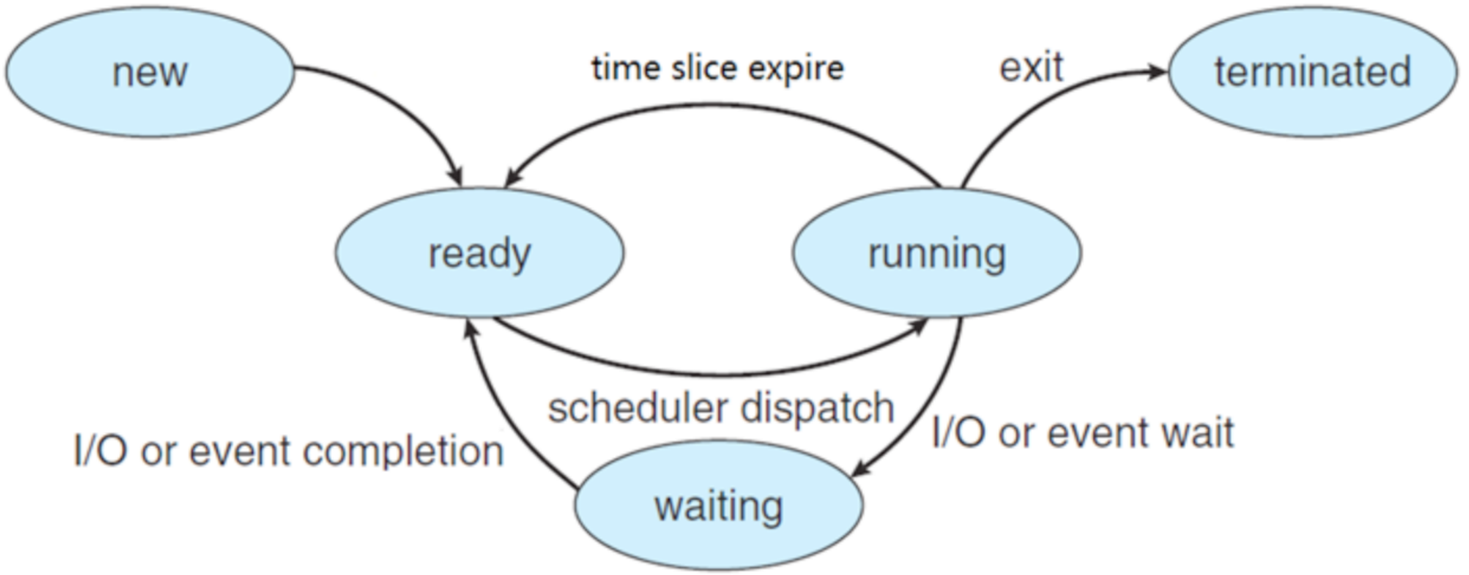
Storing processes

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem; // Start of process memory
  uint sz; // Size of process memory
  char *kstack; // Bottom of kernel stack for this process
  enum proc_state state; // Process state
  int pid; // Process ID
  struct proc *parent; // Parent process
  void *chan; // If !zero, sleeping on chan
  int killed; // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd; // Current directory
  struct context context; // Switch here to run process
  struct trapframe *tf; // Trap frame for the current interrupt
};
```

Process states

- Process actions:
  - create
  - destroy
  - stop/resume
  - status
  - wait for finish

  available to the OS and via syscall



Common process states in Linux:

```
D: uninterruptible sleep
R: running or runnable
S: interruptible sleep (waiting for an
event to complete)
T: stopped by job control
Z: ‘‘zombie’’ (terminated but not reaped)
```

Process tools

- `ps` gives the list of processes

```
ps aux #list all processes
```

- `top` and `htop` provide interactive monitoring
- `/proc`—a virtual filesystem to work with processes

```
cat /proc/cpuinfo
cat /proc/639560/cmdline
```

- with `kill` we can send signals to process with a known `pid`
  - `killall`—a wrapper to address processes by name
  - `pkill`—a wrapper to address processes by an expression

```
kill -SIGTERM 639560
```