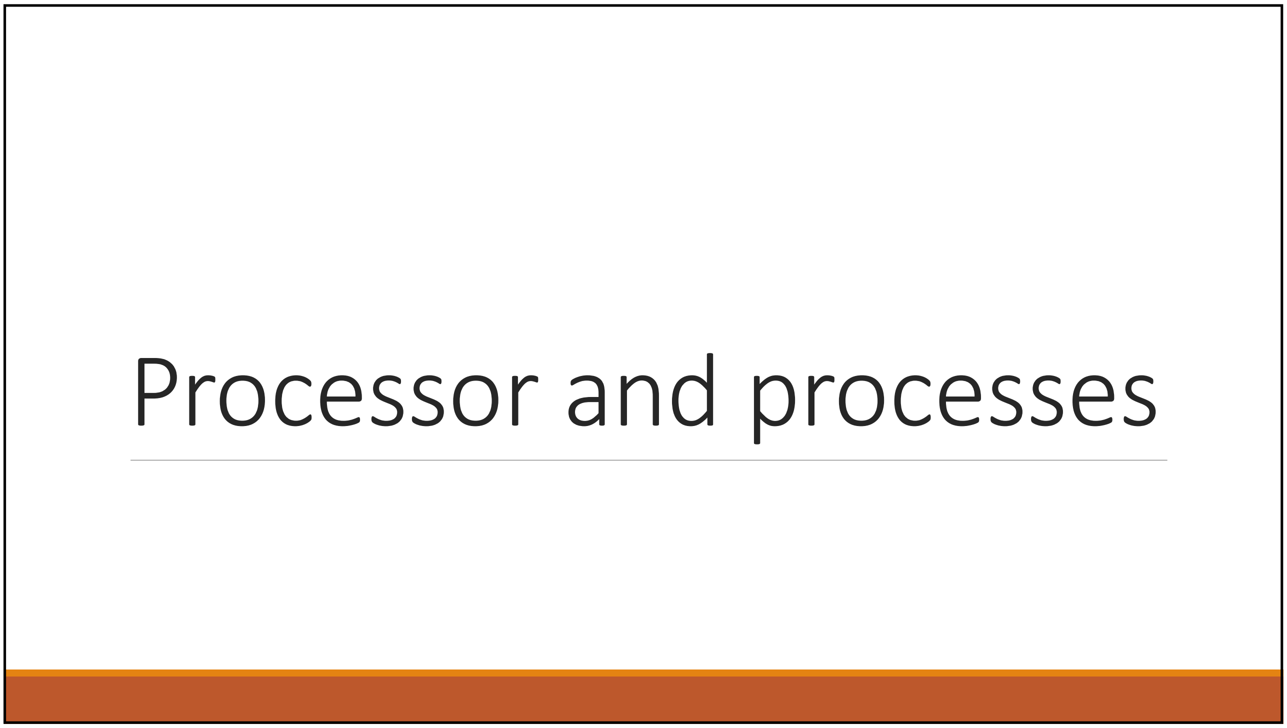
# Operating system kernel

RECAP FROM COMPUTER ARCHITECTURES AND OPERATING SYSTEMS…

1



# Processor and processes

2

## A typical architecture

- One (or several) CPU(s)
- Main memory
- A set of devices (peripherals)
- Interrupts

## The CPU

General registers

State and control registers

- Program Counter (PC o IP)
- Stack Pointer (SP)
- Program Status register (PS)

## The program status register

Condition code

Bit mask that keeps the status of the processor

CPU mode

User mode VS kernel mode

Interrupt enable bit

5

## Fetch-execution cycle

If there are pending interrupts and the interrupts are enabled

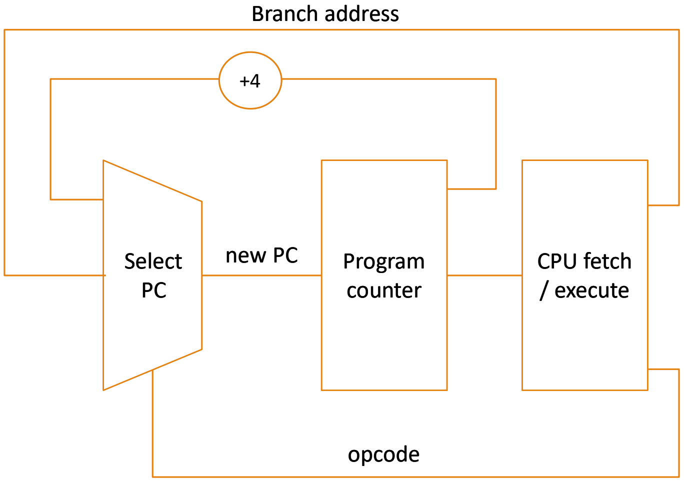
- Manages the interrupt

Else

- Loads the instruction at address PC
- Executes the instruction
- PC=PC+4 (*)

(*) assumes that the instruction occupies 4 bytes

6



Branch address

+4

Select PC

new PC

Program counter

CPU fetch / execute

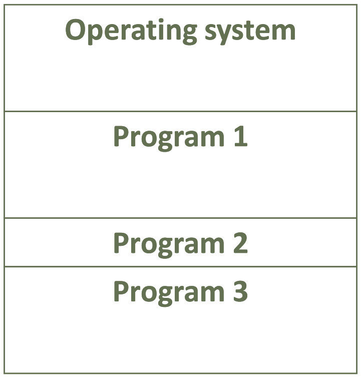
opcode

# A CPU schema

7

# Multi-programmed systems

- multi-user system: several programs loaded in memory at the same time
- Spool optimization
- Resource optimization (processor, memory, devices)



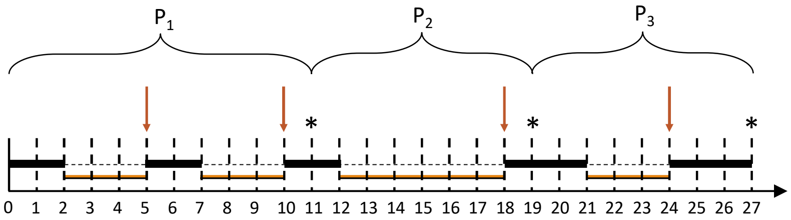
Operating system

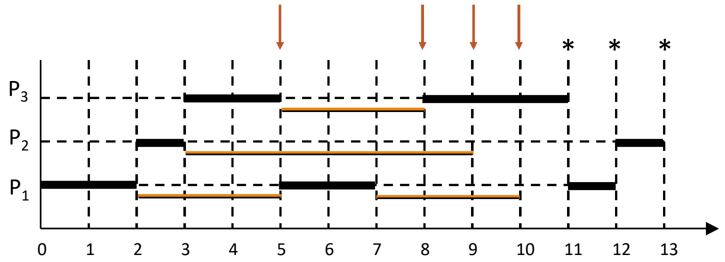
Program 1

Program 2

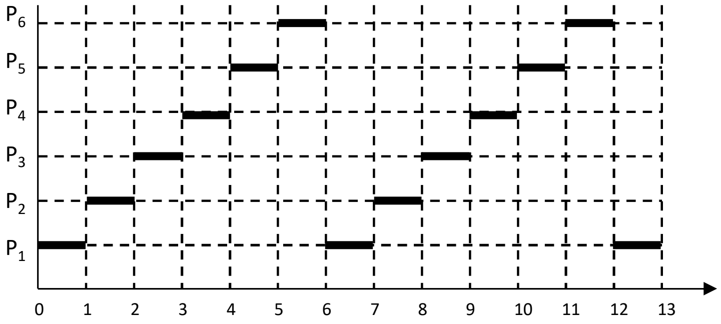
Program 3

8





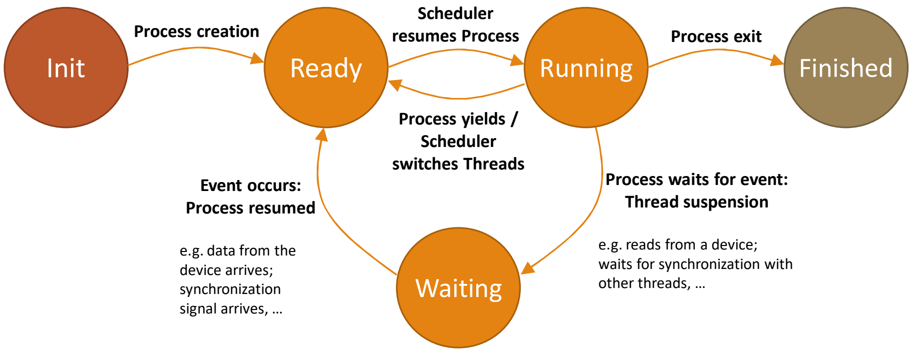
## Single task vs multi-task



## Time sharing

* One processor

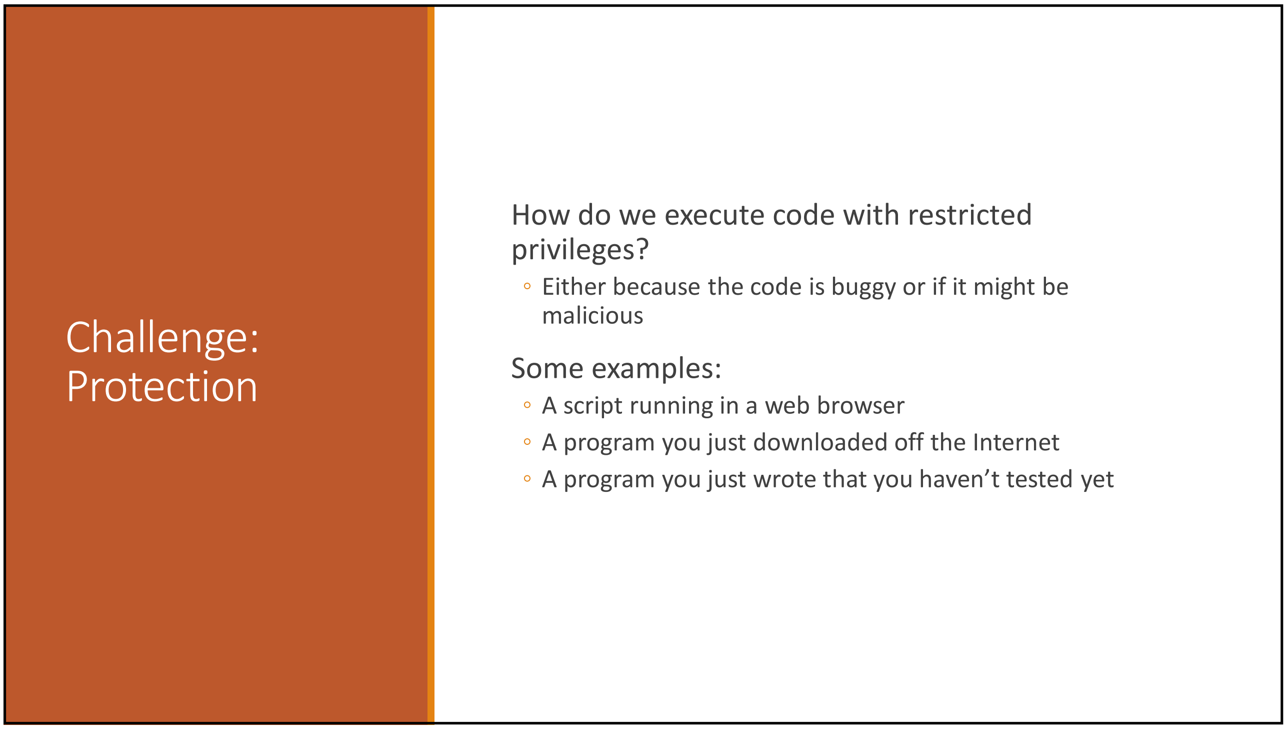
## Processes Lifecycle


Init
Process creation
Ready
Scheduler resumes Process
Running
Process exit
Finished
Process yields / Scheduler switches Threads
Event occurs: Process resumed
e.g. data from the device arrives; synchronization signal arrives, …
Waiting
Process waits for event: Thread suspension
e.g. reads from a device; waits for synchronization with other threads, …

11

# The Kernel Abstraction

12



## Challenge: Protection

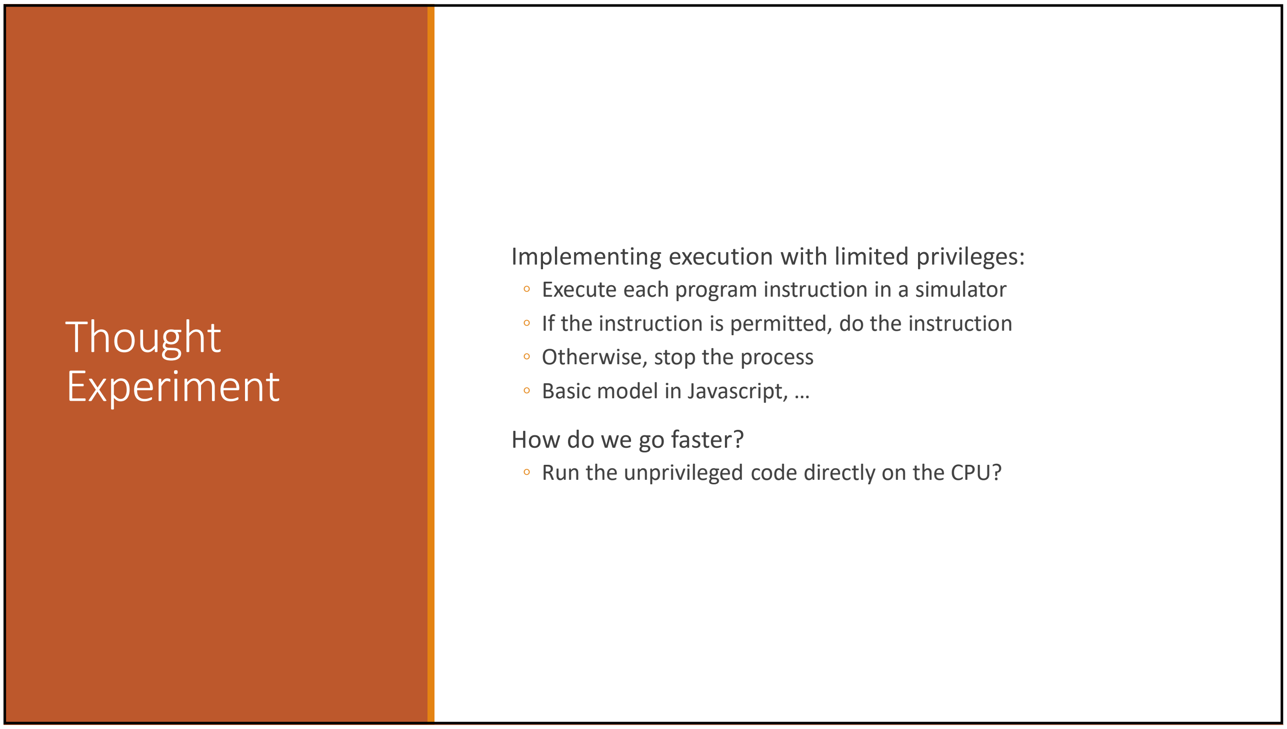
How do we execute code with restricted privileges?

- Either because the code is buggy or if it might be malicious

Some examples:

- A script running in a web browser
- A program you just downloaded off the Internet
- A program you just wrote that you haven’t tested yet

13



## Thought Experiment

Implementing execution with limited privileges:

- Execute each program instruction in a simulator
- If the instruction is permitted, do the instruction
- Otherwise, stop the process
- Basic model in Javascript, …

How do we go faster?

- Run the unprivileged code directly on the CPU?

14

## Main Points

**Process concept**

A process is an OS abstraction for executing a program with limited privileges

**Dual-mode operation: user vs. kernel**

Kernel-mode: execute with complete privileges

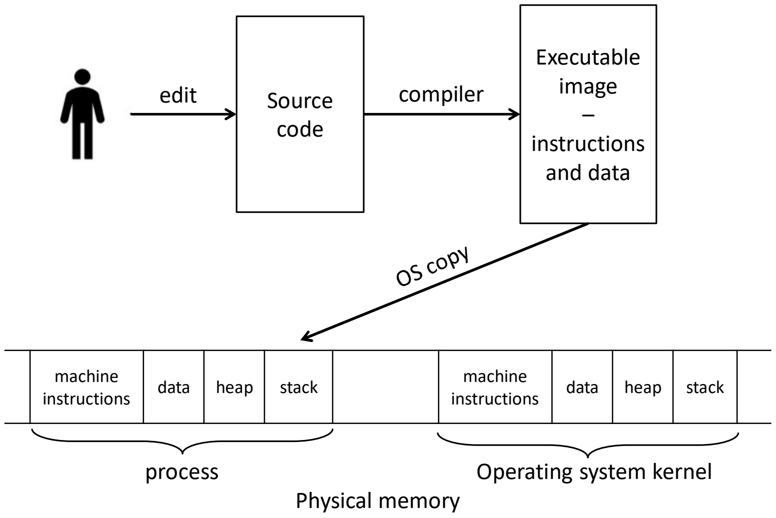
User-mode: execute with fewer privileges

**Safe control transfer**

How do we switch from one mode to the other?

15

## Process concept



edit

Source code

compiler

Executable image – instructions and data

OS copy

| machine instructions | data | heap | stack | | machine instructions | data | heap | stack |
|---|---|---|---|---|---|---|---|---|

process

Operating system kernel

Physical memory

16

## Process Concept

Process: a sequence of activities activated by a program, **running with limited rights**

- Process control block (PCB): the data structure the OS uses to keep track of a process
- Process Table: contains all PCBs
- Two elements:
  - Thread: executes a sequence of instructions within a process
    - Potentially many threads per process (for now 1:1)
    - Thread aka lightweight process
  - Address space: set of rights of a process
    - Memory that the process can access
    - Other permissions the process has (e.g., which procedure calls it can make, what files it can access)

17

## Program and Process

Program: a static sequence of instructions

Process:

- a dynamic entity: a sequence of activities described by a program
- executed on a set of CPUs with limited rights

Several processes can be activated on the same program

- The processes execute the same code
- Each process executes the program on different data and/or in different times

18

## Process Control Block

It is a data structure:

Process  name

- Can be the index of the PCB in the process table

Pointers to process threads

Assigned memory

Other assigned resources

- Files, devices, etc…

19

## Hardware Support: Dual-Mode Operation

Kernel mode

- Execution with the full privileges of the hardware
- Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet

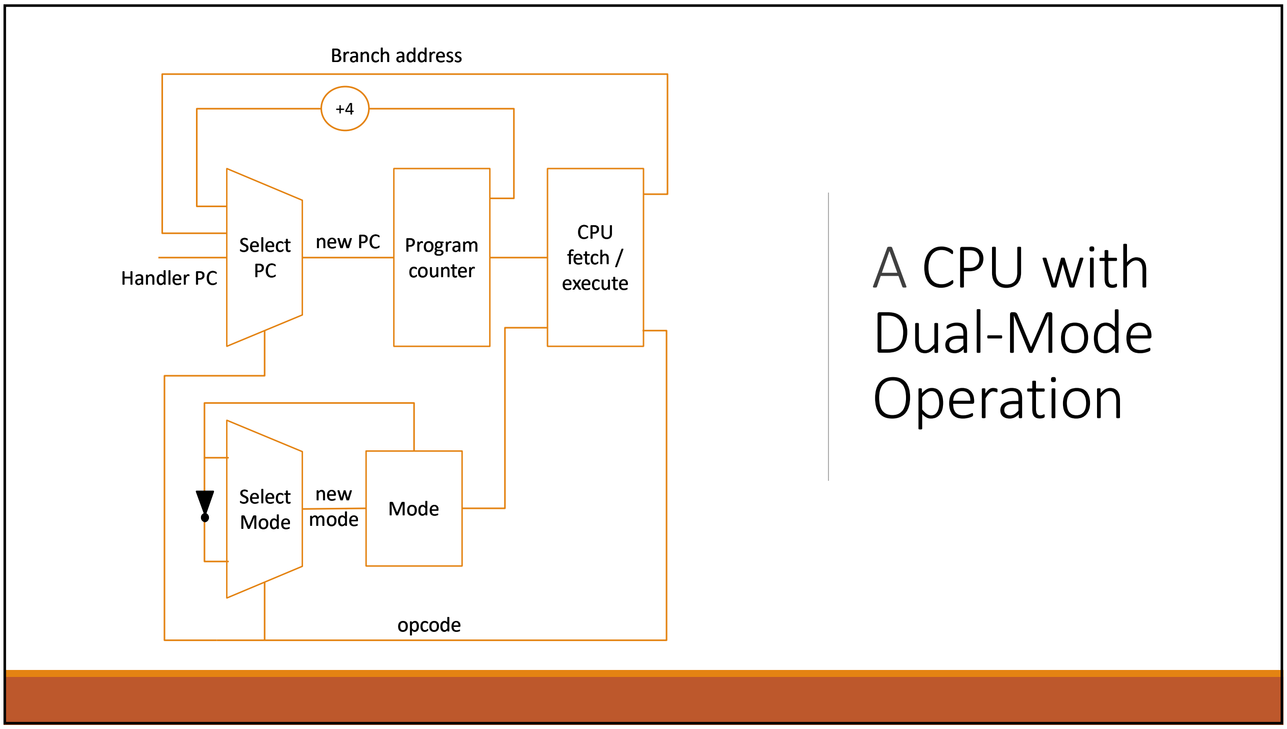
User mode

- Limited privileges
- Only those granted by the operating system kernel

On the x86, mode stored in EFLAGS register

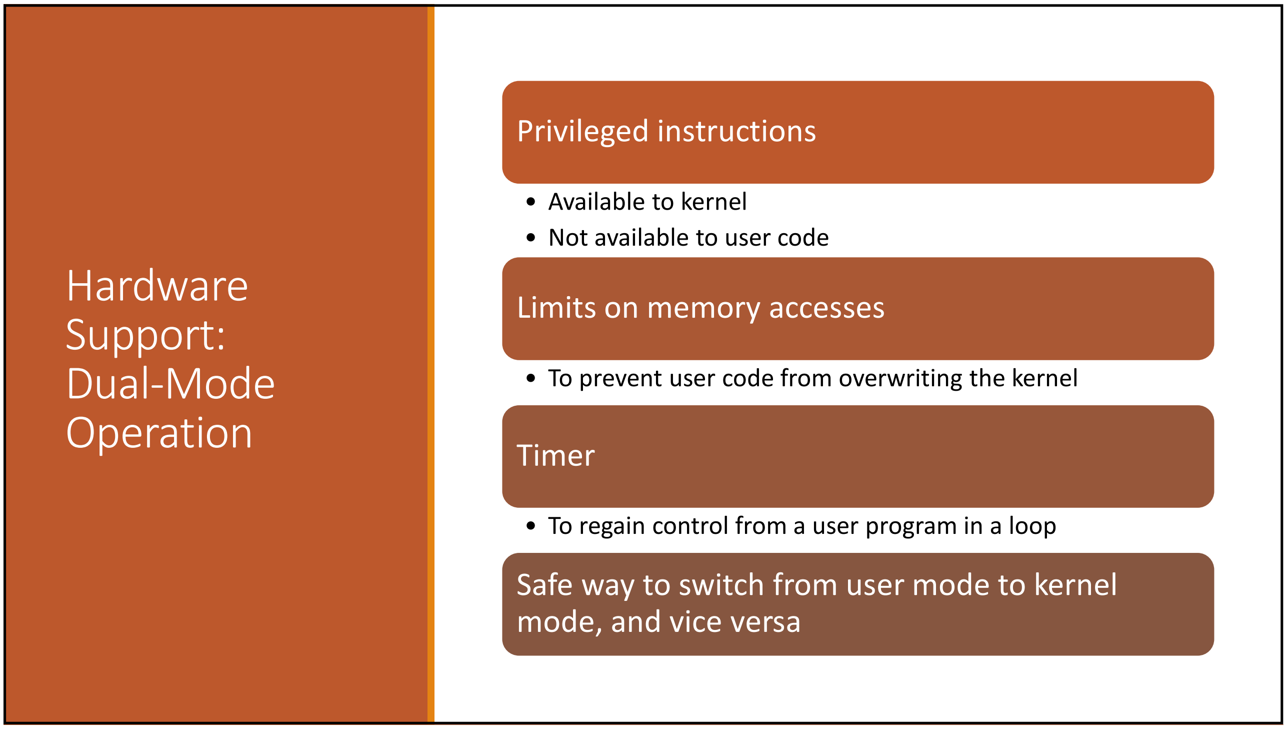
- In general in the Program Status Register

20



# A CPU with Dual-Mode Operation

21



# Hardware Support: Dual-Mode Operation

## Privileged instructions

- Available to kernel
- Not available to user code

## Limits on memory accesses

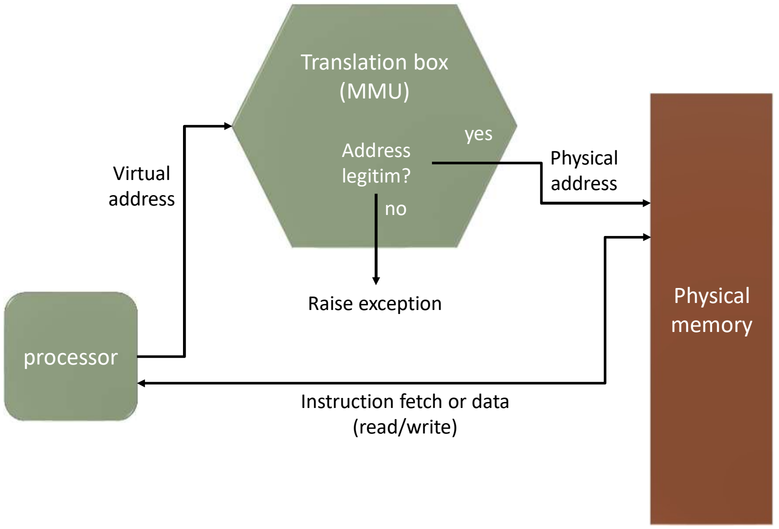
- To prevent user code from overwriting the kernel

## Timer

- To regain control from a user program in a loop

## Safe way to switch from user mode to kernel mode, and vice versa

22



# Memory protection

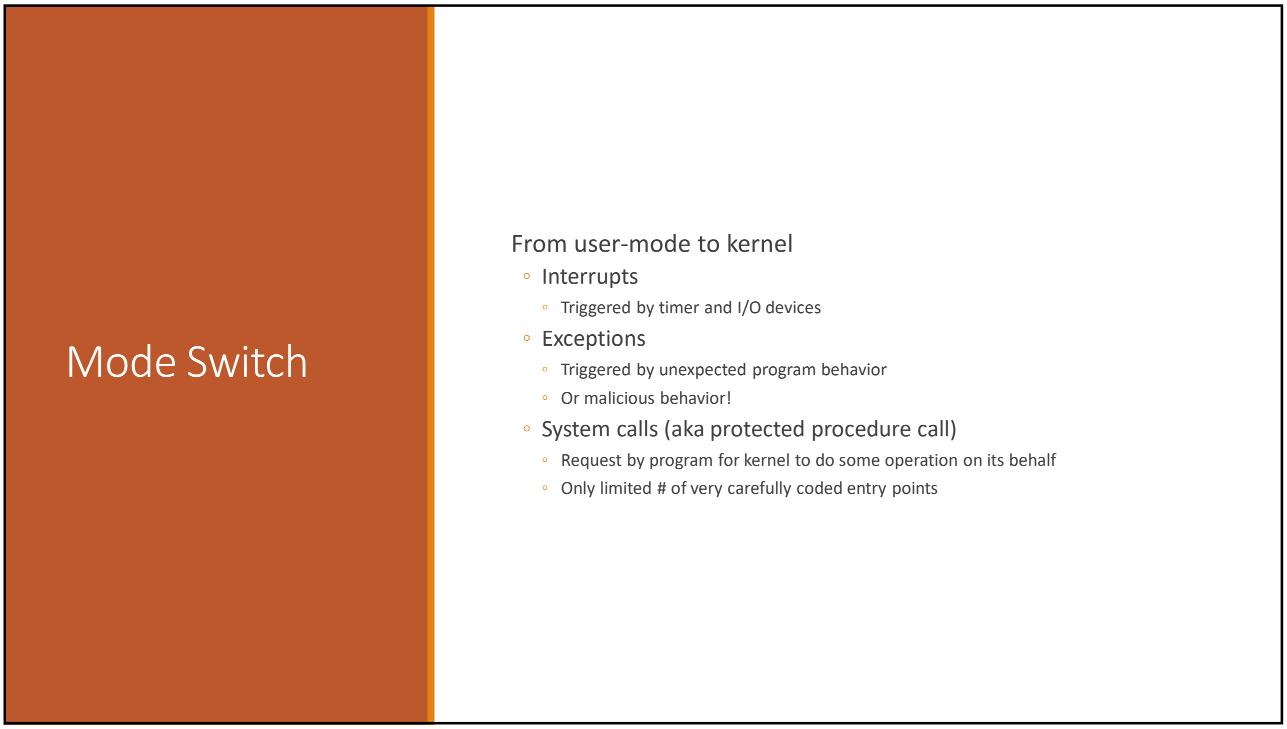
Virtual memory

- Translation done in hardware, using a table
- Table set up by operating system kernel

# Hardware Timer

Hardware device that periodically interrupts the processor

- Returns control to the kernel timer interrupt handler
- Interrupt frequency set by the kernel
  - Not by user code!
- Interrupts can be temporarily deferred
  - Not by user code!
  - Crucial for implementing mutual exclusion



## Mode Switch

From user-mode to kernel

- Interrupts
  - Triggered by timer and I/O devices
- Exceptions
  - Triggered by unexpected program behavior
  - Or malicious behavior!
- System calls (aka protected procedure call)
  - Request by program for kernel to do some operation on its behalf
  - Only limited # of very carefully coded entry points

25

## Mode Switch

From kernel-mode to user-mode

- Return from interrupt, exception, system call
  - Resume suspended execution
- New process/new thread start
  - Jump to first instruction in program/thread
- Process/thread context switch
  - Resume some other process
- User-level upcall
  - Asynchronous notification to user program

26

## How do we take interrupts safely?

Interrupt vector
- Limited number of entry points into kernel

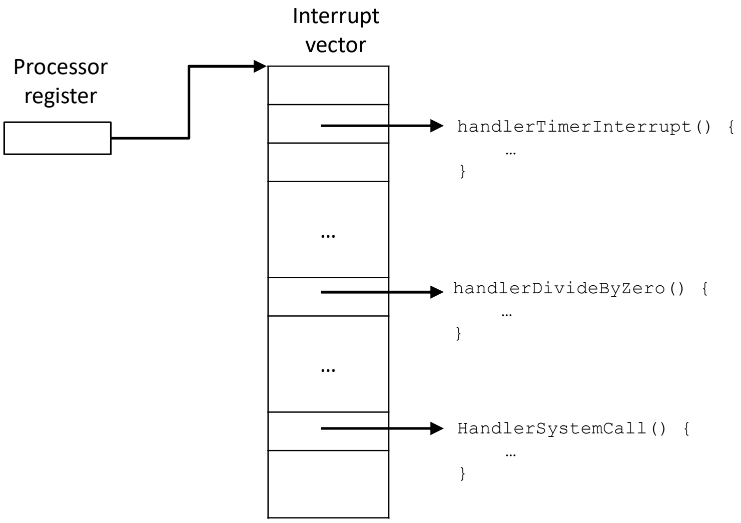
Kernel interrupt stack
- Handler works regardless of state of user code

Interrupt masking
- Handler is non-blocking

Atomic transfer of control
- Single instruction to change:
  - Program counter
  - Stack pointer
  - Memory protection
  - Kernel/user mode

Transparent restartable execution
- User program does not know interrupt occurred

27



```
handlerTimerInterrupt() {
      …
}

handlerDivideByZero() {
      …
}

HandlerSystemCall() {
      …
}
```

## Interrupt vector

Table set up by OS kernel; pointers to code to run on different events

28

# Interrupt Stack

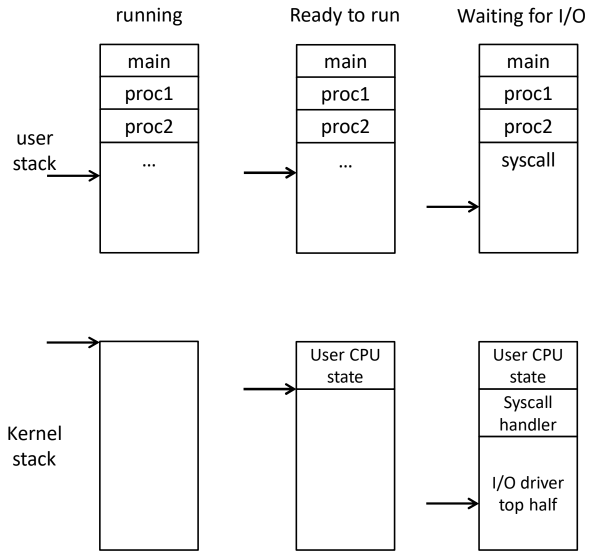
Per-processor, located in kernel (not user) memory

- Usually a thread has both: kernel and user stack

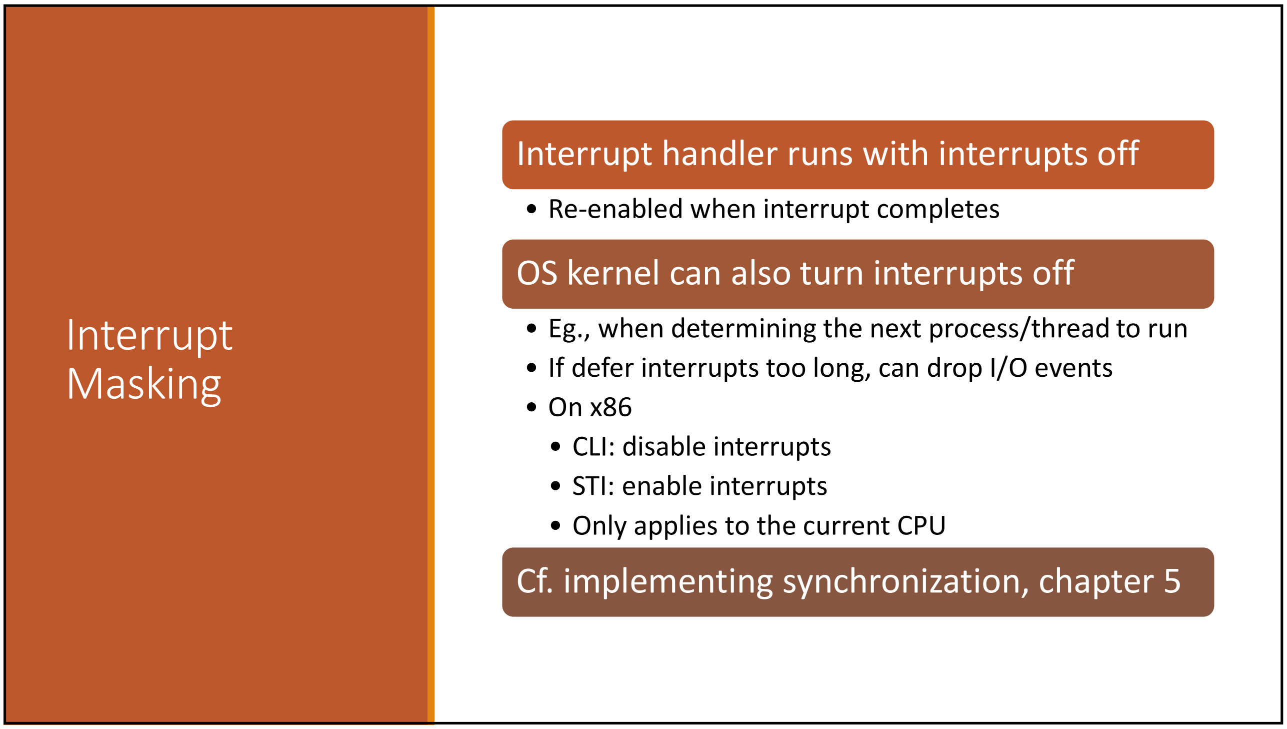
Why can't interrupt handler run on the stack of the interrupted user process?

29

# Interrupt stack



30



## Interrupt Masking

**Interrupt handler runs with interrupts off**

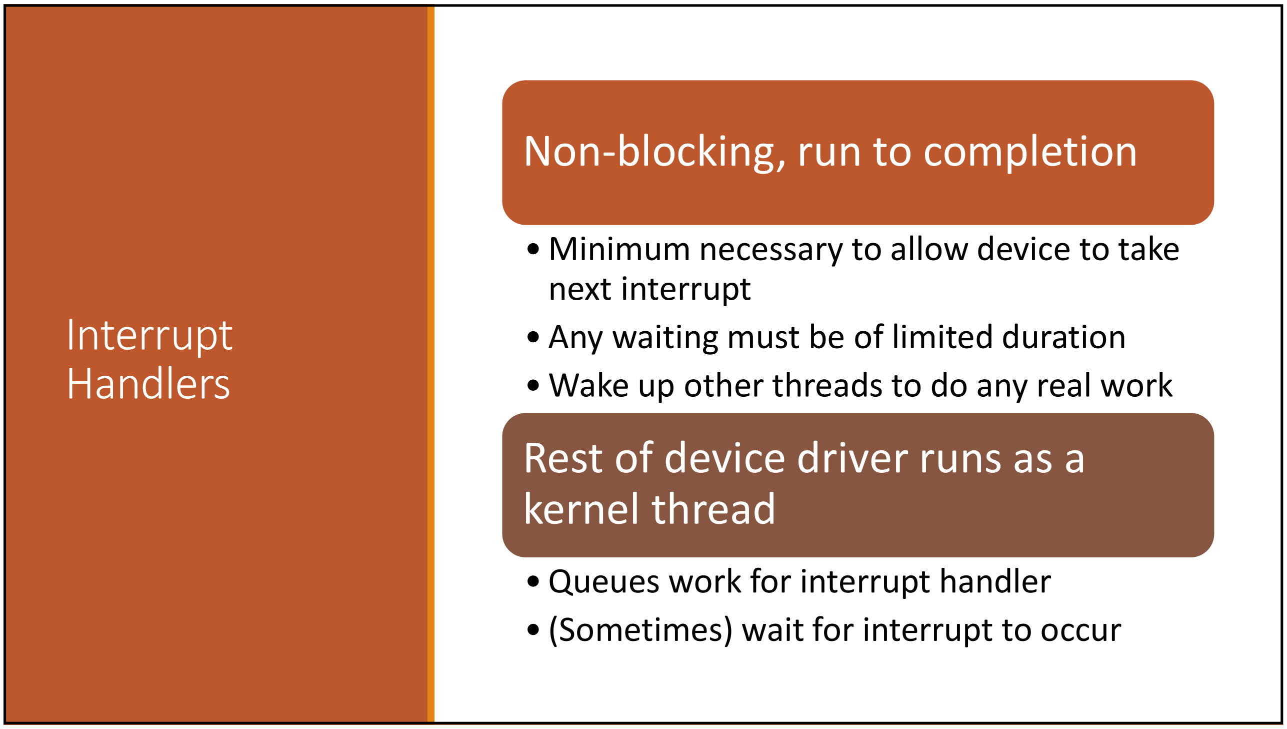
- Re-enabled when interrupt completes

**OS kernel can also turn interrupts off**

- Eg., when determining the next process/thread to run
- If defer interrupts too long, can drop I/O events
- On x86
  - CLI: disable interrupts
  - STI: enable interrupts
  - Only applies to the current CPU

**Cf. implementing synchronization, chapter 5**

31



## Interrupt Handlers

**Non-blocking, run to completion**

- Minimum necessary to allow device to take next interrupt
- Any waiting must be of limited duration
- Wake up other threads to do any real work

**Rest of device driver runs as a kernel thread**

- Queues work for interrupt handler
- (Sometimes) wait for interrupt to occur

32

## Atomic Mode Transfer

On interrupt (x86):

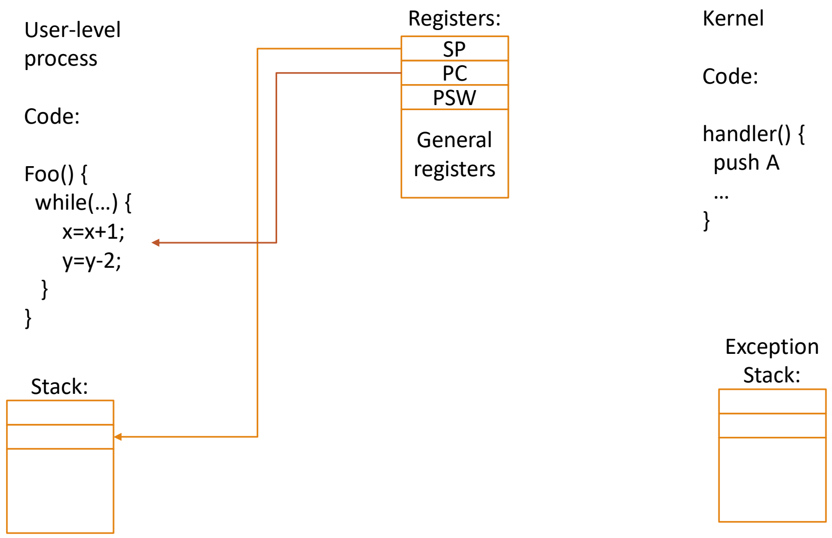
- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes)
- Switch to kernel mode
- Vector through interrupt table
- Switch to handler PC & kernel PSW

In Hardware

- Interrupt handler saves registers it might clobber

33

## Before



User-level process

Code:

```
Foo() {
  while(…) {
      x=x+1;
      y=y-2;
  }
}
```

Stack:

Registers:

SP

PC

PSW

General registers

Kernel

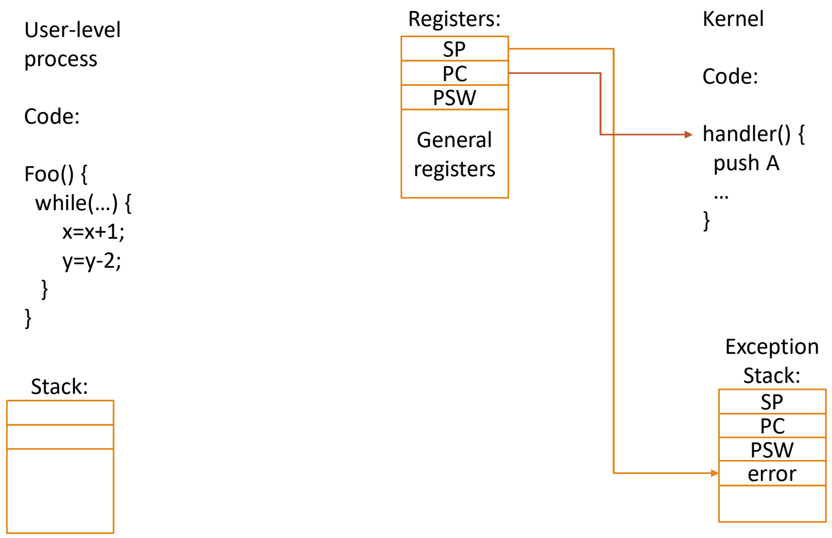
Code:

```
handler() {
  push A
  …
}
```

Exception Stack:

34

# During



User-level process

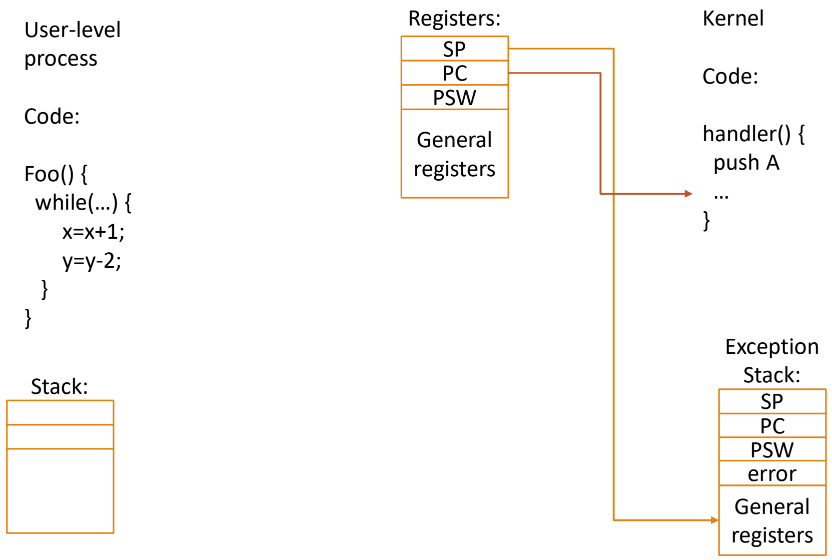
Code:

```
Foo() {
  while(…) {
      x=x+1;
      y=y-2;
   }
}
```

Stack:

35

# After



User-level process

Code:

```
Foo() {
  while(…) {
      x=x+1;
      y=y-2;
   }
}
```

Stack:

36

## At end of handler

Handler restores saved registers

Atomically return to interrupted process/ thread (IRET instruction)

- Restore program counter
- Restore program stack
- Restore processor status word/condition codes
- Switch to user mode
- Enable interrupts

37

## Interrupt management a simple example

Initial state: interrupt ‘500’ occurs when executing instruction A000

Registri nella CPU

| | |
|---|---|
| PC | A000 |
| PS | PSW P |
| SP | FFFF |
| R1 | AAAA |
| R2 | BBBB |
| … | |

Memoria

programma P

| | |
|---|---|
| …. | … |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| … | … |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| … | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

| | |
|---|---|
| … | … |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | … |

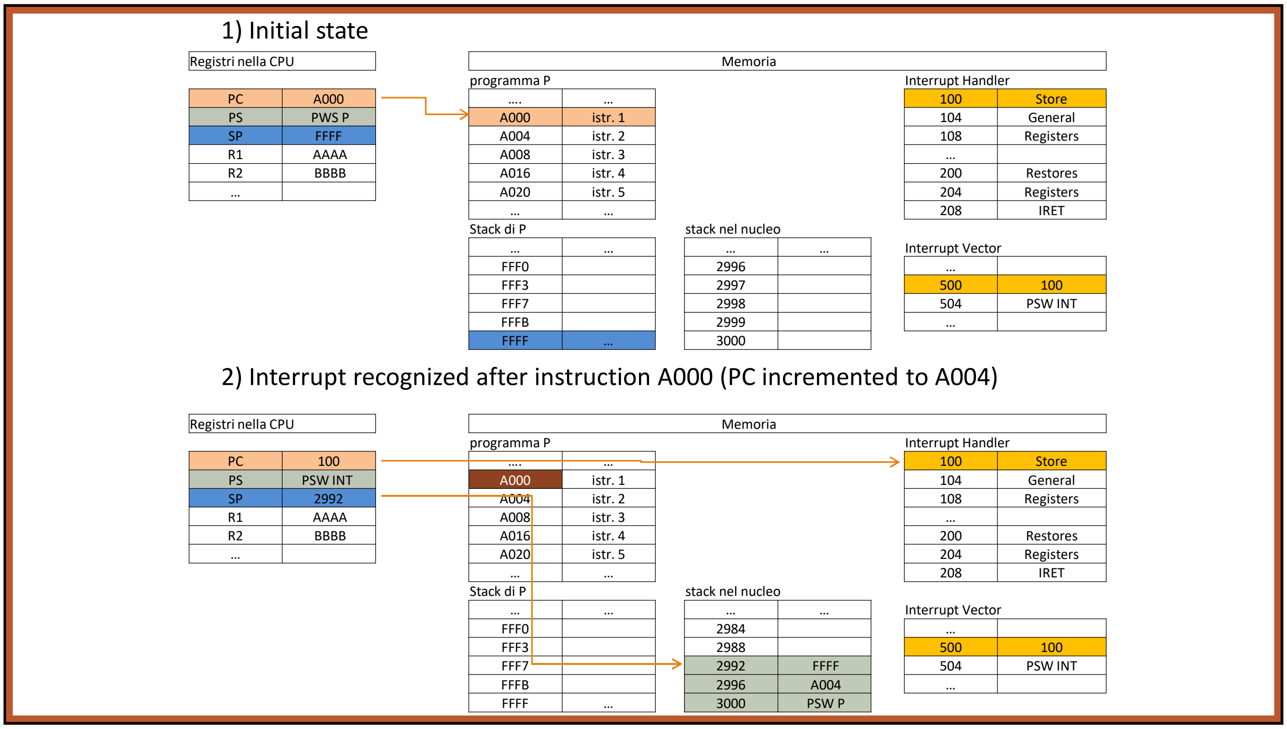
stack nel nucleo

| | |
|---|---|
| … | … |
| 2996 | |
| 2997 | |
| 2998 | |
| 2999 | |
| 3000 | |

Interrupt Vector

| | |
|---|---|
| … | |
| 500 | 100 |
| 504 | PSW INT |
| … | |

38



### 1) Initial state

Registri nella CPU

| | |
|---|---|
| PC | A000 |
| PS | PWS P |
| SP | FFFF |
| R1 | AAAA |
| R2 | BBBB |
| … | |

Memoria

programma P

| | |
|---|---|
| .... | … |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| … | … |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| … | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

| | |
|---|---|
| … | … |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | … |

stack nel nucleo

| | |
|---|---|
| … | … |
| 2996 | |
| 2997 | |
| 2998 | |
| 2999 | |
| 3000 | |

Interrupt Vector

| | |
|---|---|
| … | |
| 500 | 100 |
| 504 | PSW INT |
| … | |

### 2) Interrupt recognized after instruction A000 (PC incremented to A004)

Registri nella CPU

| | |
|---|---|
| PC | 100 |
| PS | PSW INT |
| SP | 2992 |
| R1 | AAAA |
| R2 | BBBB |
| … | |

Memoria

programma P

| | |
|---|---|
| .... | … |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| … | … |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| … | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

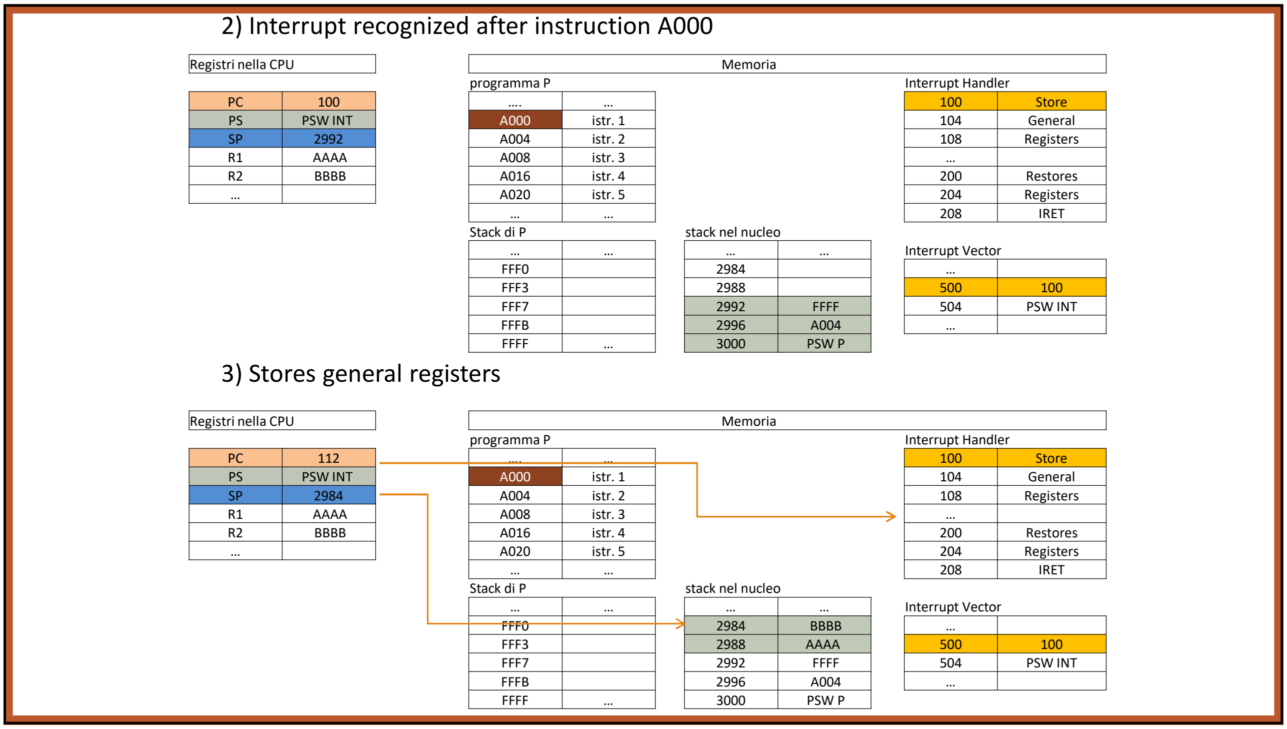
| | |
|---|---|
| … | … |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | … |

stack nel nucleo

| | |
|---|---|
| … | … |
| 2984 | |
| 2988 | |
| 2992 | FFFF |
| 2996 | A004 |
| 3000 | PSW P |

Interrupt Vector

| | |
|---|---|
| … | |
| 500 | 100 |
| 504 | PSW INT |
| … | |



### 2) Interrupt recognized after instruction A000

Registri nella CPU

| | |
|---|---|
| PC | 100 |
| PS | PSW INT |
| SP | 2992 |
| R1 | AAAA |
| R2 | BBBB |
| … | |

Memoria

programma P

| | |
|---|---|
| .... | … |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| … | … |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| … | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

| | |
|---|---|
| … | … |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | … |

stack nel nucleo

| | |
|---|---|
| … | … |
| 2984 | |
| 2988 | |
| 2992 | FFFF |
| 2996 | A004 |
| 3000 | PSW P |

Interrupt Vector

| | |
|---|---|
| … | |
| 500 | 100 |
| 504 | PSW INT |
| … | |

### 3) Stores general registers

Registri nella CPU

| | |
|---|---|
| PC | 112 |
| PS | PSW INT |
| SP | 2984 |
| R1 | AAAA |
| R2 | BBBB |
| … | |

Memoria

programma P

| | |
|---|---|
| .... | … |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| … | … |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| … | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

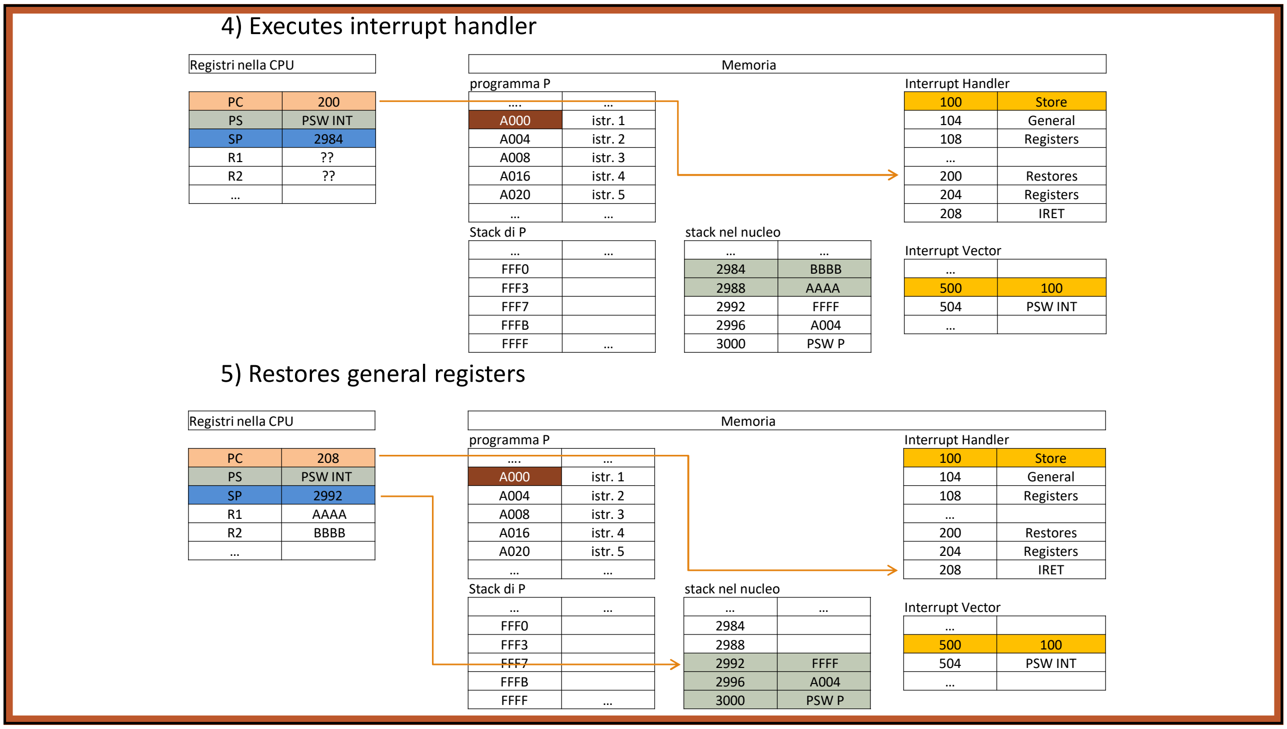
| | |
|---|---|
| … | … |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | … |

stack nel nucleo

| | |
|---|---|
| … | … |
| 2984 | BBBB |
| 2988 | AAAA |
| 2992 | FFFF |
| 2996 | A004 |
| 3000 | PSW P |

Interrupt Vector

| | |
|---|---|
| … | |
| 500 | 100 |
| 504 | PSW INT |
| … | |



## 4) Executes interrupt handler

Registri nella CPU

| | |
|---|---|
| PC | 200 |
| PS | PSW INT |
| SP | 2984 |
| R1 | ?? |
| R2 | ?? |
| ... | |

Memoria

programma P

| | |
|---|---|
| .... | ... |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| ... | ... |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| ... | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

| | |
|---|---|
| ... | ... |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | ... |

stack nel nucleo

| | |
|---|---|
| ... | ... |
| 2984 | BBBB |
| 2988 | AAAA |
| 2992 | FFFF |
| 2996 | A004 |
| 3000 | PSW P |

Interrupt Vector

| | |
|---|---|
| ... | |
| 500 | 100 |
| 504 | PSW INT |
| ... | |

## 5) Restores general registers

Registri nella CPU

| | |
|---|---|
| PC | 208 |
| PS | PSW INT |
| SP | 2992 |
| R1 | AAAA |
| R2 | BBBB |
| ... | |

Memoria

programma P

| | |
|---|---|
| .... | ... |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| ... | ... |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| ... | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

| | |
|---|---|
| ... | ... |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | ... |

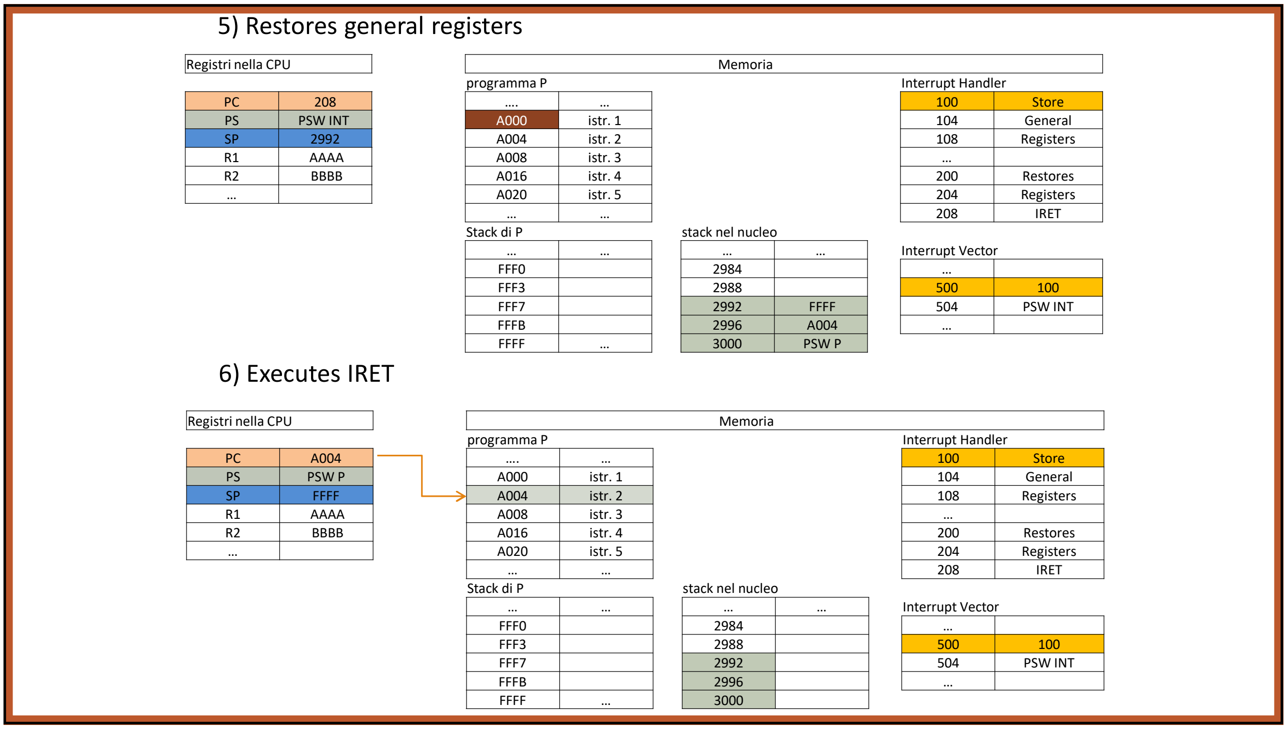
stack nel nucleo

| | |
|---|---|
| ... | ... |
| 2984 | |
| 2988 | |
| 2992 | FFFF |
| 2996 | A004 |
| 3000 | PSW P |

Interrupt Vector

| | |
|---|---|
| ... | |
| 500 | 100 |
| 504 | PSW INT |
| ... | |

41



## 5) Restores general registers

Registri nella CPU

| | |
|---|---|
| PC | 208 |
| PS | PSW INT |
| SP | 2992 |
| R1 | AAAA |
| R2 | BBBB |
| ... | |

Memoria

programma P

| | |
|---|---|
| .... | ... |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| ... | ... |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| ... | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

| | |
|---|---|
| ... | ... |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | ... |

stack nel nucleo

| | |
|---|---|
| ... | ... |
| 2984 | |
| 2988 | |
| 2992 | FFFF |
| 2996 | A004 |
| 3000 | PSW P |

Interrupt Vector

| | |
|---|---|
| ... | |
| 500 | 100 |
| 504 | PSW INT |
| ... | |

## 6) Executes IRET

Registri nella CPU

| | |
|---|---|
| PC | A004 |
| PS | PSW P |
| SP | FFFF |
| R1 | AAAA |
| R2 | BBBB |
| ... | |

Memoria

programma P

| | |
|---|---|
| .... | ... |
| A000 | istr. 1 |
| A004 | istr. 2 |
| A008 | istr. 3 |
| A016 | istr. 4 |
| A020 | istr. 5 |
| ... | ... |

Interrupt Handler

| | |
|---|---|
| 100 | Store |
| 104 | General |
| 108 | Registers |
| ... | |
| 200 | Restores |
| 204 | Registers |
| 208 | IRET |

Stack di P

| | |
|---|---|
| ... | ... |
| FFF0 | |
| FFF3 | |
| FFF7 | |
| FFFB | |
| FFFF | ... |

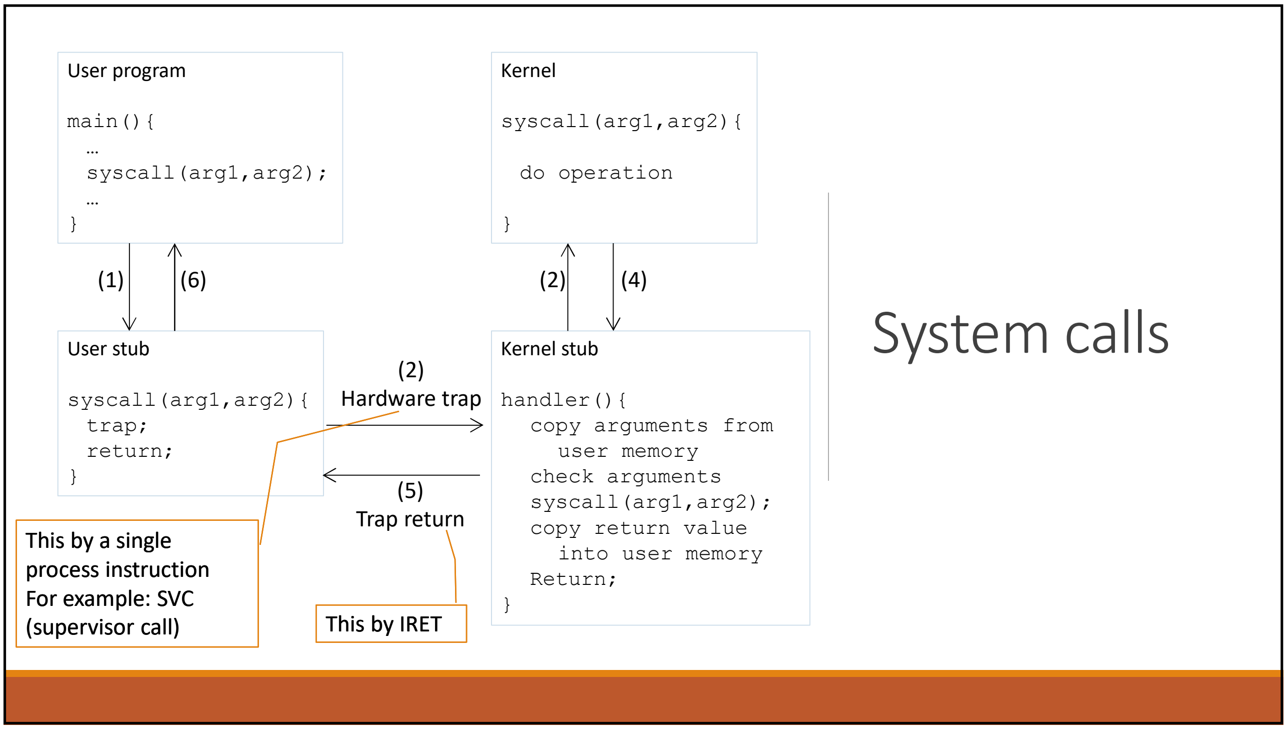
stack nel nucleo

| | |
|---|---|
| ... | ... |
| 2984 | |
| 2988 | |
| 2992 | |
| 2996 | |
| 3000 | |

Interrupt Vector

| | |
|---|---|
| ... | |
| 500 | 100 |
| 504 | PSW INT |
| ... | |

42



## System calls

User program

```
main(){
  …
  syscall(arg1,arg2);
  …
}
```

Kernel

```
syscall(arg1,arg2){

  do operation

}
```

User stub

```
syscall(arg1,arg2){
  trap;
  return;
}
```

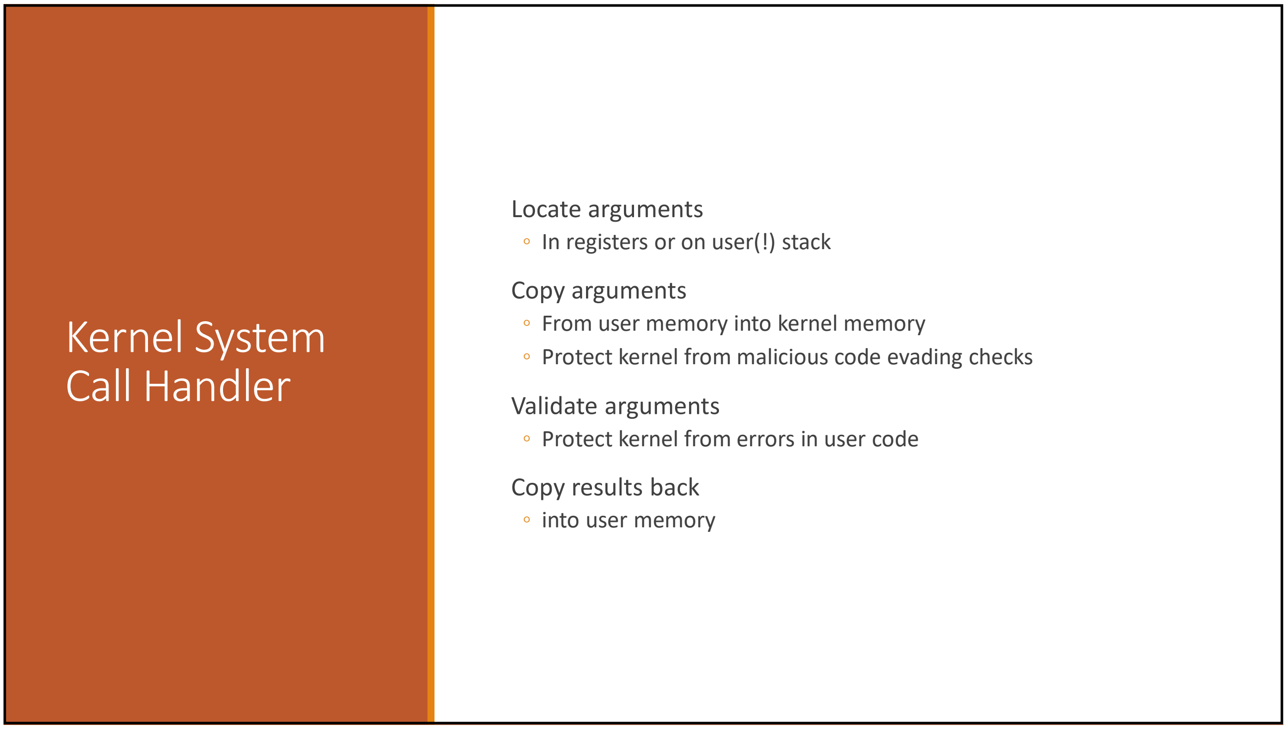
Kernel stub

```
handler(){
   copy arguments from
      user memory
   check arguments
   syscall(arg1,arg2);
   copy return value
      into user memory
   Return;
}
```

This by a single process instruction For example: SVC (supervisor call)

This by IRET

43



## Kernel System Call Handler

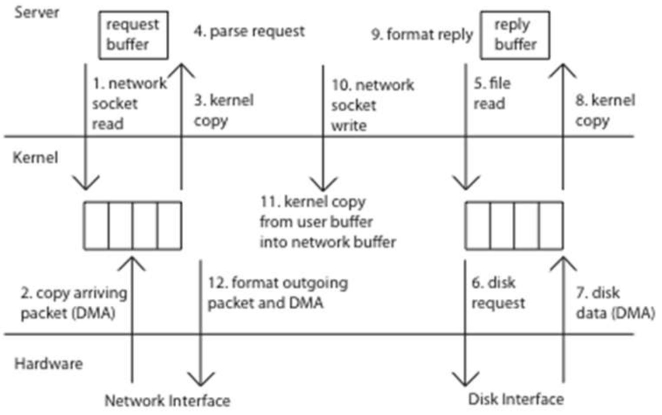
Locate arguments
- In registers or on user(!) stack

Copy arguments
- From user memory into kernel memory
- Protect kernel from malicious code evading checks

Validate arguments
- Protect kernel from errors in user code

Copy results back
- into user memory

44



## Web server example

45

## Upcall: User-level interrupt

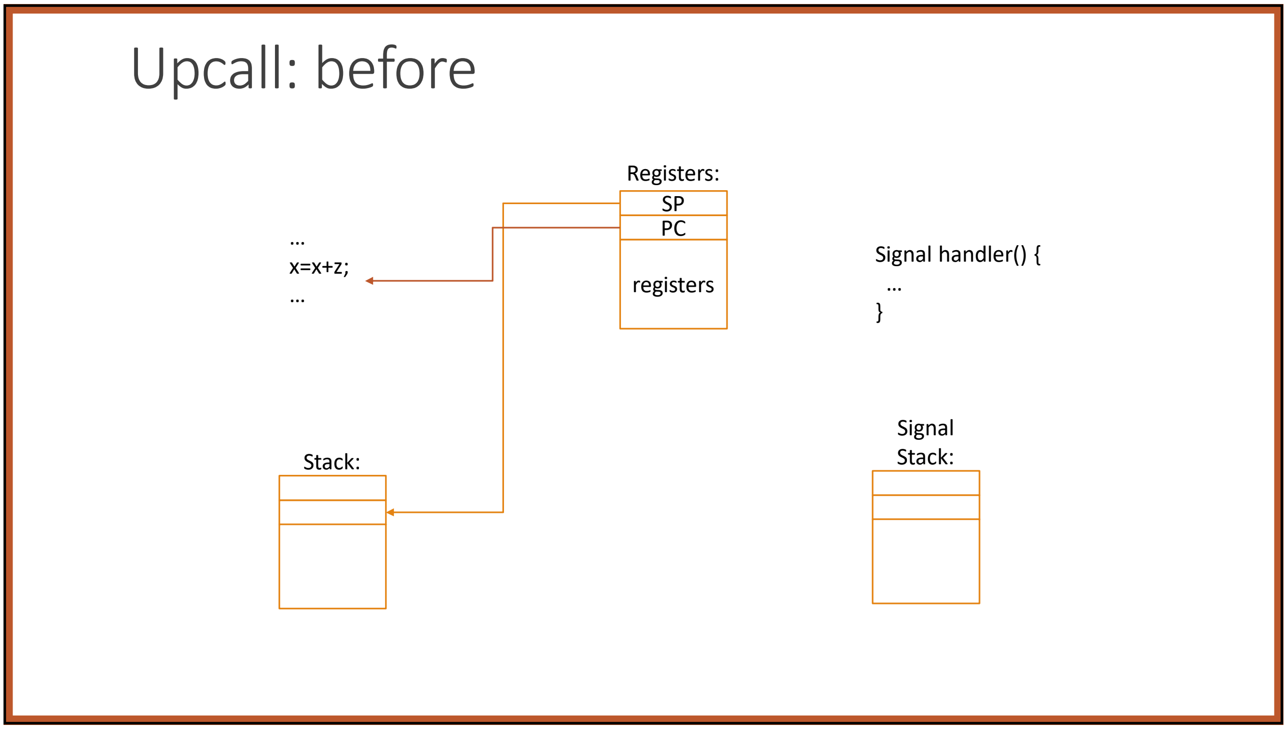
Also known as UNIX signals

- Notify user process of event that needs to be handled right away
  - Time-slice for user-level thread manager
  - Interrupt delivery for VM player (see later)

Direct analogue of kernel interrupts

- Signal handlers – fixed entry points
- Separate signal stack
- Automatic save/restore registers – transparent resume
- Signal masking: signals disabled while in signal handler

46



# Upcall: before

Registers:
SP
PC
registers

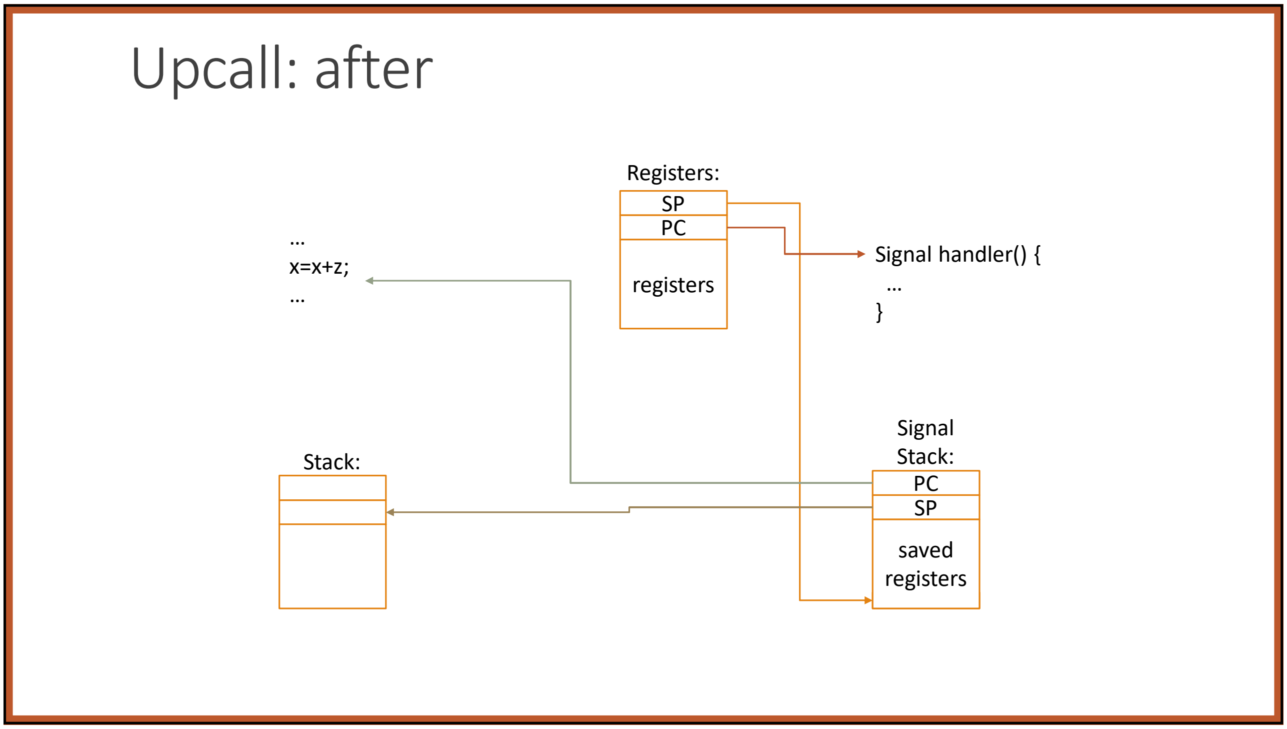
...
x=x+z;
...

Signal handler() {
...
}

Stack:

Signal
Stack:

47



# Upcall: after

Registers:
SP
PC
registers

...
x=x+z;
...

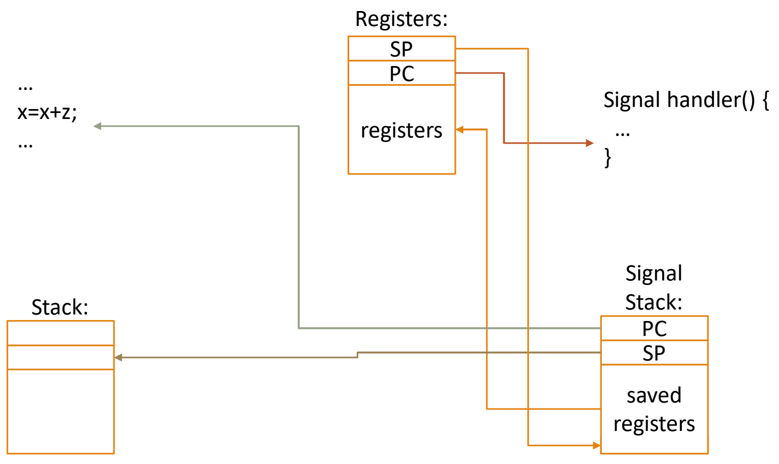
Signal handler() {
...
}

Stack:

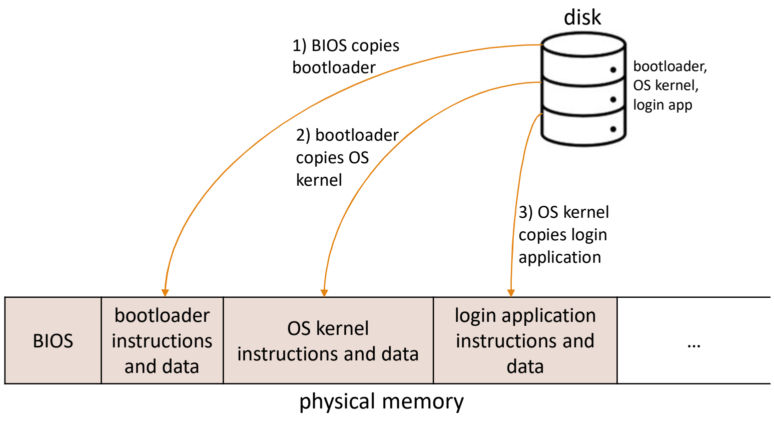
Signal
Stack:
PC
SP
saved
registers

48

# Upcall: end of handler



49



disk

bootloader, OS kernel, login app

1) BIOS copies bootloader

2) bootloader copies OS kernel

3) OS kernel copies login application

| BIOS | bootloader instructions and data | OS kernel instructions and data | login application instructions and data | … |
|---|---|---|---|---|

physical memory

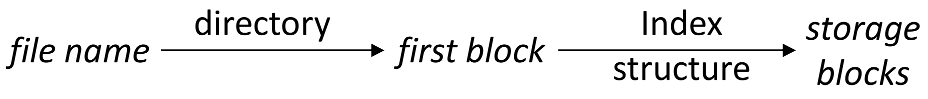
# Booting

50

# Notes on FAT file system

51

## Named Data in a FAT File System

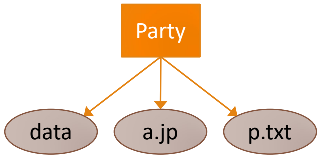


*file name* → directory → *first block* → Index structure → *storage blocks*

52

# Directories

- Each file (and directory) belongs to a directory;
- Each directory is a **data structure** that links file names to file attributes
  - Example of attributes: file size, address on disk, access rights, time of last access, time of creation, …



A possible implementation (**FAT file systems**):

- The directory is a table, it associates each file name to its file descriptor (which includes all attributes of the file)

| name | descriptor |
|---|---|
| data | Attribute: type, first data block, etc. |
| a.jp | Attribute: type, first data block, etc. |
| p.txt | Attribute: type, first data block, etc. |

Implementation of directory Party

53

# FAT file system

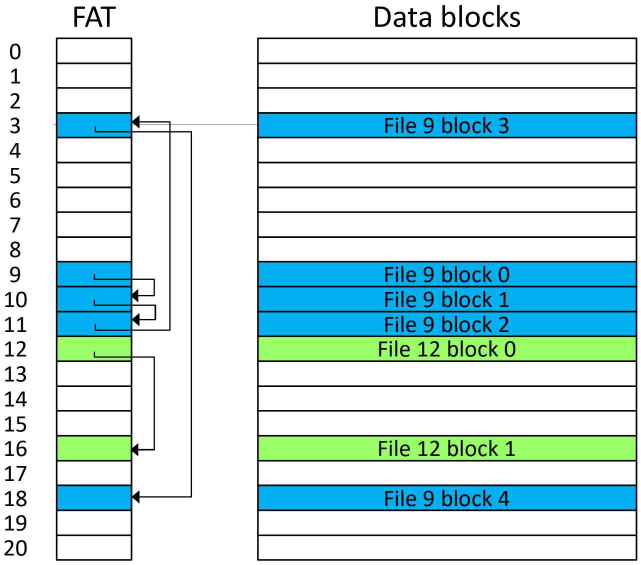
File Allocation Table (FAT) :

Linked list index structure

- Simple, easy to implement
- Still widely used (e.g., thumb drives)

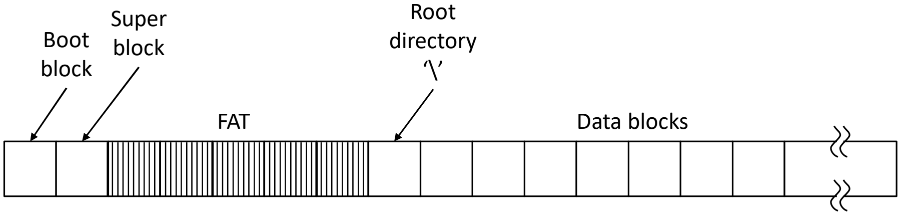
File table:

- Linear map of all blocks on disk
- Each file a linked list of blocks



54

# Physical disk organization with FAT



55