2/25

<u>Today</u>

- System calls
- Process
- Context Switching
- Signals

<u>Last Class</u>

→ Exceptional Control Flow.

→ Exception Classes

① Interrupts.

② System Calls (printf → write())

⟨user mode⟩ ⟨kernel mode⟩

a part of the OS

③ Faults

④ Abort

Exception Handler Table

$$\begin{tikzcd}
\text{IA32} \arrow[r] & \text{256 exception types} \arrow[d] \arrow[dr] & \\
& \text{defined by Processor} & \text{defined by OS}
\end{tikzcd}$$

#ID

0 → Divide by Zero

14 → Page Fault

128 → System Call.

$$\begin{tikzcd}
& \text{System Call} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{read} & \text{write} & \text{exit}
\end{tikzcd}$$

How to call a system call?

```
int main()                  <stdlib.h>
{
    write(1, "hello, world\n", 13);
    exit(0);
}
```

Assembly instruction to call a system call

$$\begin{tikzcd} & \text{int} \arrow[dl] & \$\,0x80 \arrow[r] & 128. \\ \text{interrupt} & & & \end{tikzcd}$$

System calls ⟶ arguments are stored in registers !

%eax → holds the system call's number (read? open ?)

%ebx, ecx, edx,
esi, edi, ebp } hold upto 6 arbitrary arguments.

Process .

↳ an instance of a program in execution . .

Each proces has a context

$$\begin{tikzcd} \text{context} \arrow[d] & \text{Process state .} \\ \text{} \end{tikzcd}$$

→ code + data
→ registers
→ CF
→ Program Counter
→ Page tables
→ - -

A process can run in

$$\begin{tikzcd} & \text{User mode} \\ \underline{\text{2 modes}} \arrow[ur] \arrow[dr] \arrow[d] & \\ \text{determined by a mode bit} & \underline{\text{Kernel}}\text{ mode.} \end{tikzcd}$$

$$\text{Initially} \longrightarrow \text{user mode.}$$

It can switch/change to kernel mode only via an exception.

$$\begin{tikzcd} & \text{execute any instruction.} \\ \text{Kernel mode} \arrow[ur] \arrow[dr] & \\ & \text{can access any memory location.} \end{tikzcd}$$

$$\boxed{\text{Remember SUP bit}}$$

Exception Control Flow.

When exception occurs

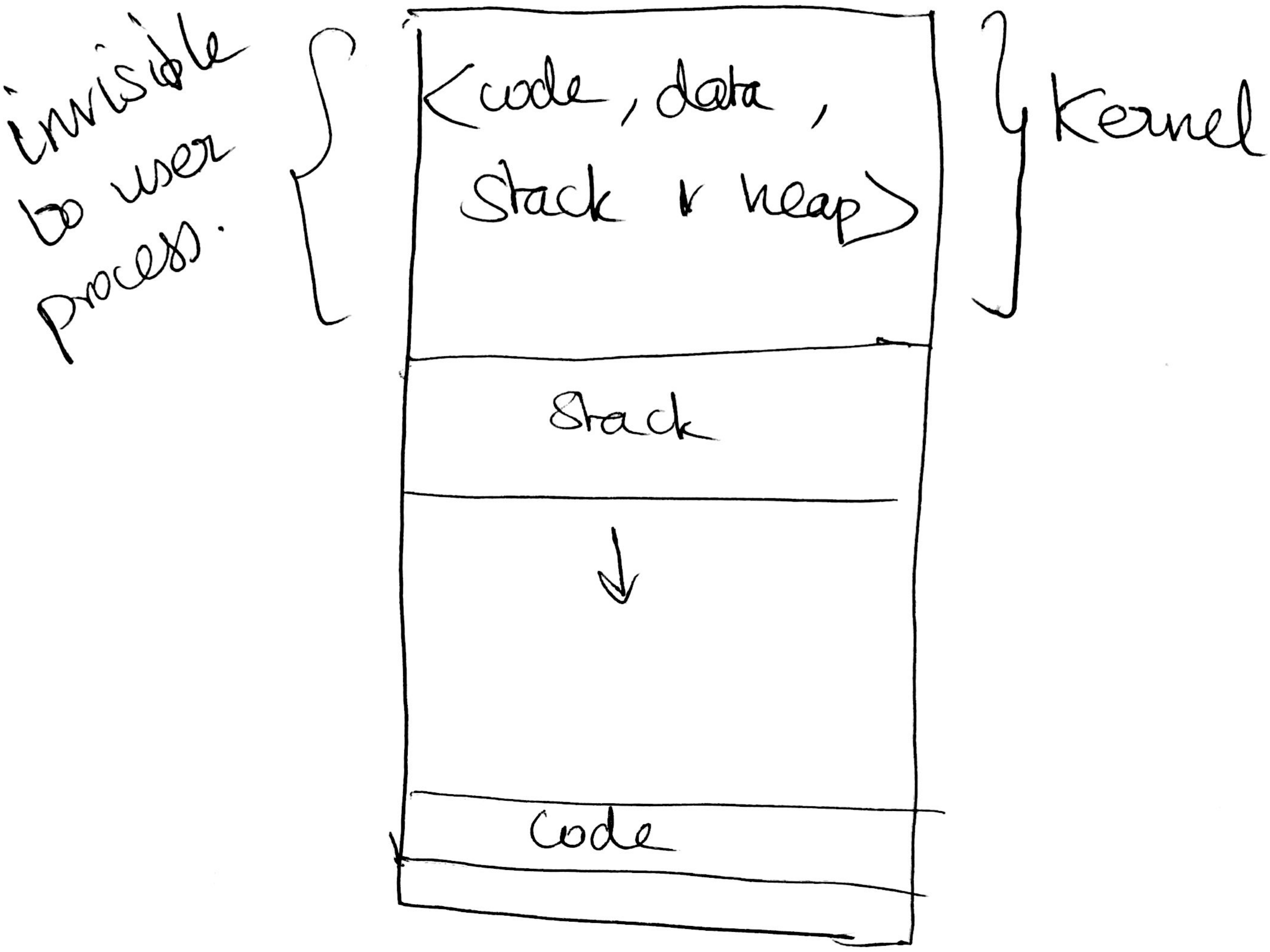
$\Big\downarrow$ control

exception handler
(the mode changes to kernel mode).

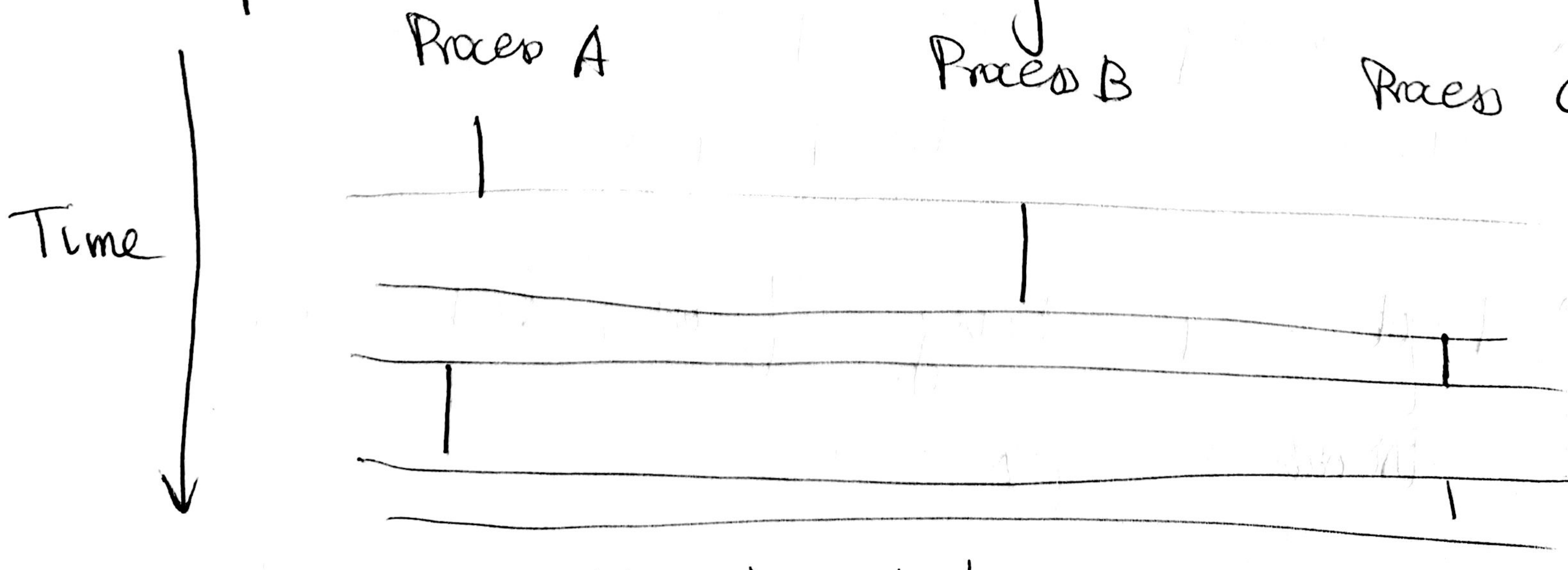
$\Big\downarrow$ runs in kernel mode & handles the exception.

returns control back to user code and mode is changed back to user mode

(still the same process !)



A system can execute more than 1 process simultaneously.



Multi-task!

Context switch ↗

<u>What happens ?</u>

1. Save the context of the current proces.
2. Restore the conext of the new process
3. Pass control to restored process

<u>Why ?</u>

1. Multi-task ! ×
2. When an exception happens.

Changing from user mode to kernel mode is not a context switch.

(Some architectures may do this)

<u>Context switch</u> because of a system call.

<u>Example</u> → read()

① Process wants to read something from the disk. → read()

② Change the process from user mode to kernel mode.

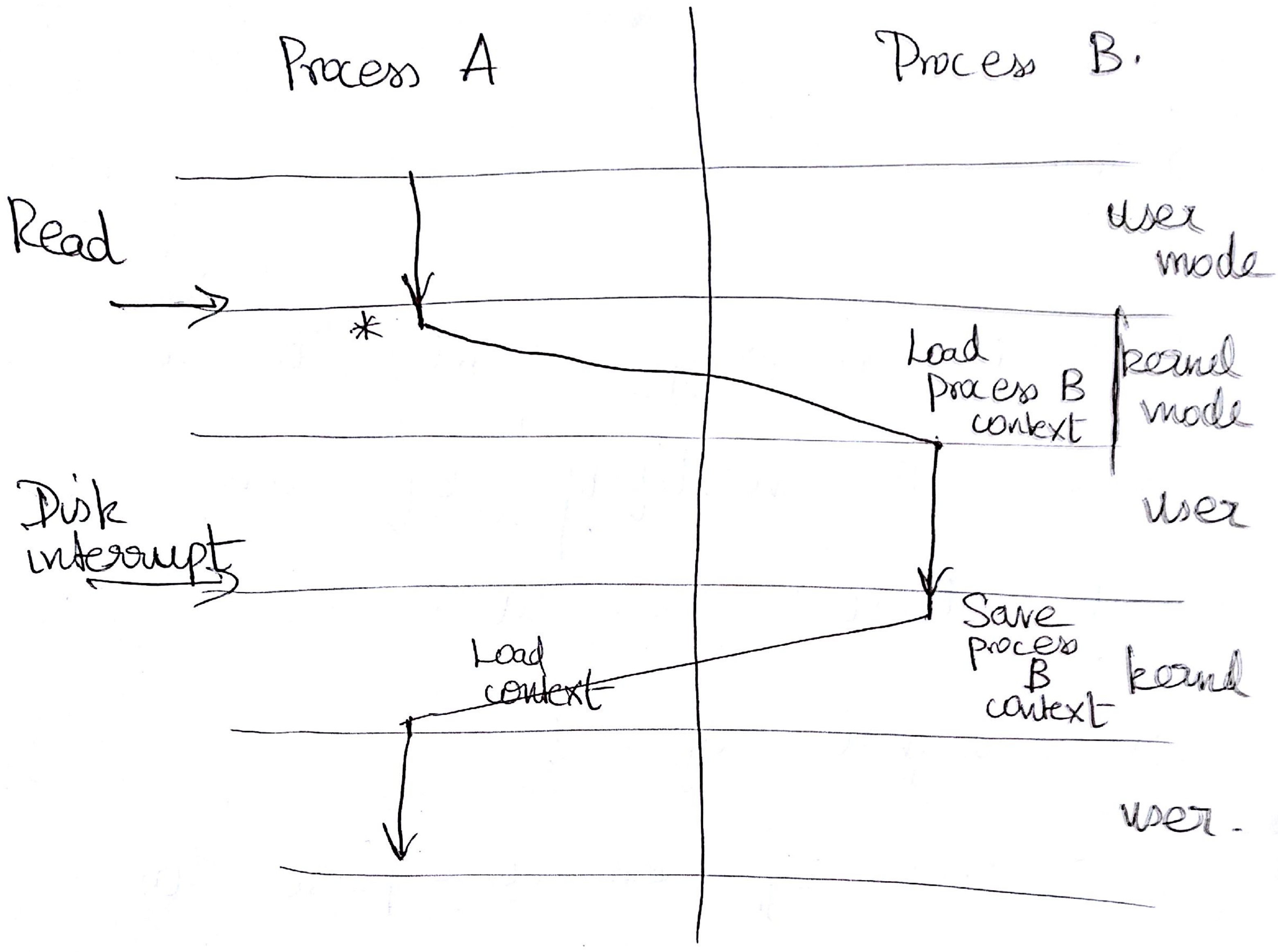
③ But <u>Reading from disk takes time</u>

1 CPU cycle	$\longrightarrow$	1 s.
L1 cache	$\longrightarrow$	3 s.
1 SSD	$\longrightarrow$	5 days.
1 Rotational Hard Disk	$\longrightarrow$	10 months

· We dont want to wait

should not

④ It will switch to some other process !



* mode switches to kernel.
Requests DMA and arranges the disk to send an interrupt once it's done //

Signal

↳ a message sent to a process to notify it of an event that occured.

30 signal types !

↳ each type corresponds to some event.

Eg.

Divide by zero → SIGFPE

Illegal memory reference → SIGSEGV