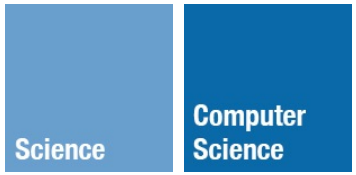
Processes & ECF



CS 351: Systems Programming
Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Agenda

- Definition & OS responsibilities
- Exceptional control flow
  - synch vs. asynch exceptions
  - exception handling procedure

§Definition & OS responsibilities

a **process** is a *program in execution*

programs *describe* what we want done,

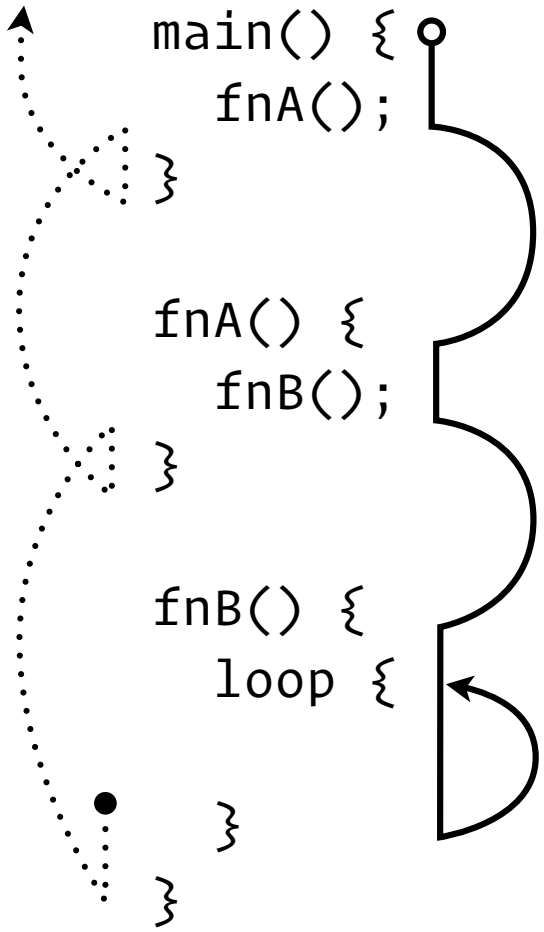
processes *carry out* what we want done

a process comprises ...

{ code (program)

+ runtime data (global, local, dynamic)

+ PC, SP, FP & other registers }



essential to program execution is *predictable, logical control flow*

which requires that nothing disrupt the program mid-execution

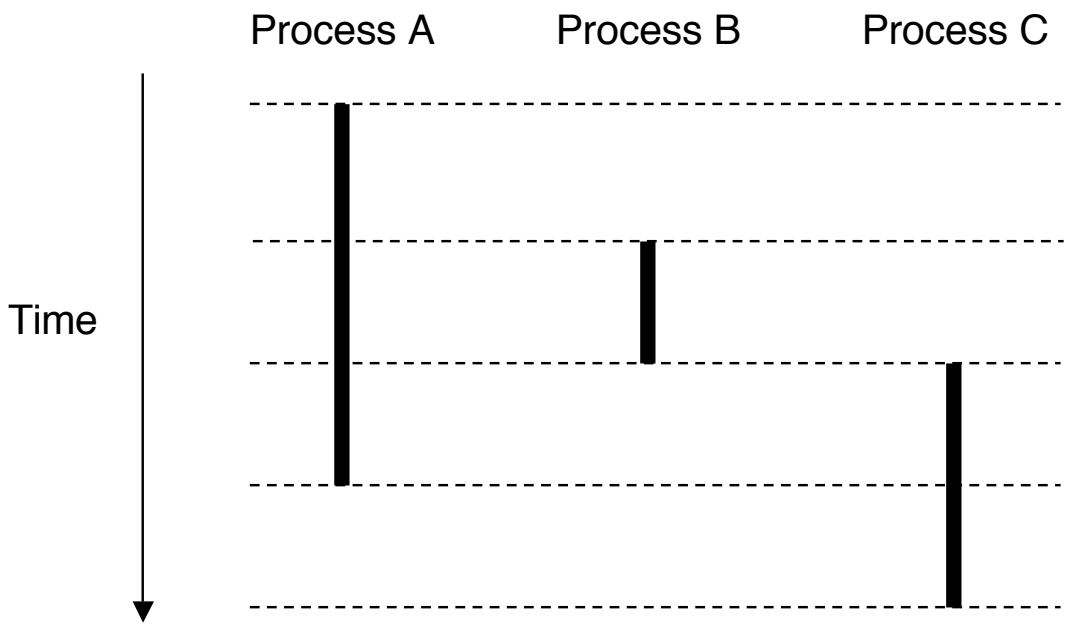
easiest way to guarantee this is for a process to "own" the CPU for its entire duration

... downsides?

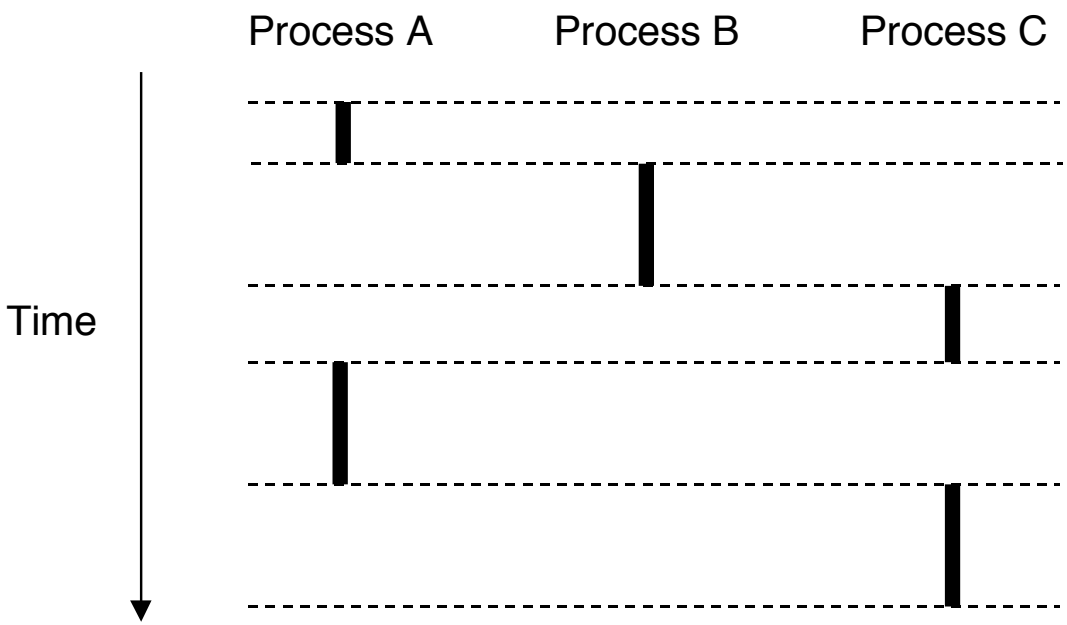
1. No multitasking!
2. A malicious (or badly written) program can "take over" the CPU forever
3. An idle process (e.g., waiting for input) will underutilize the CPU

the operating system presents each process with a simulated, *seamless logical control flow*

many of which can be taking place *concurrently* on one or more CPUs



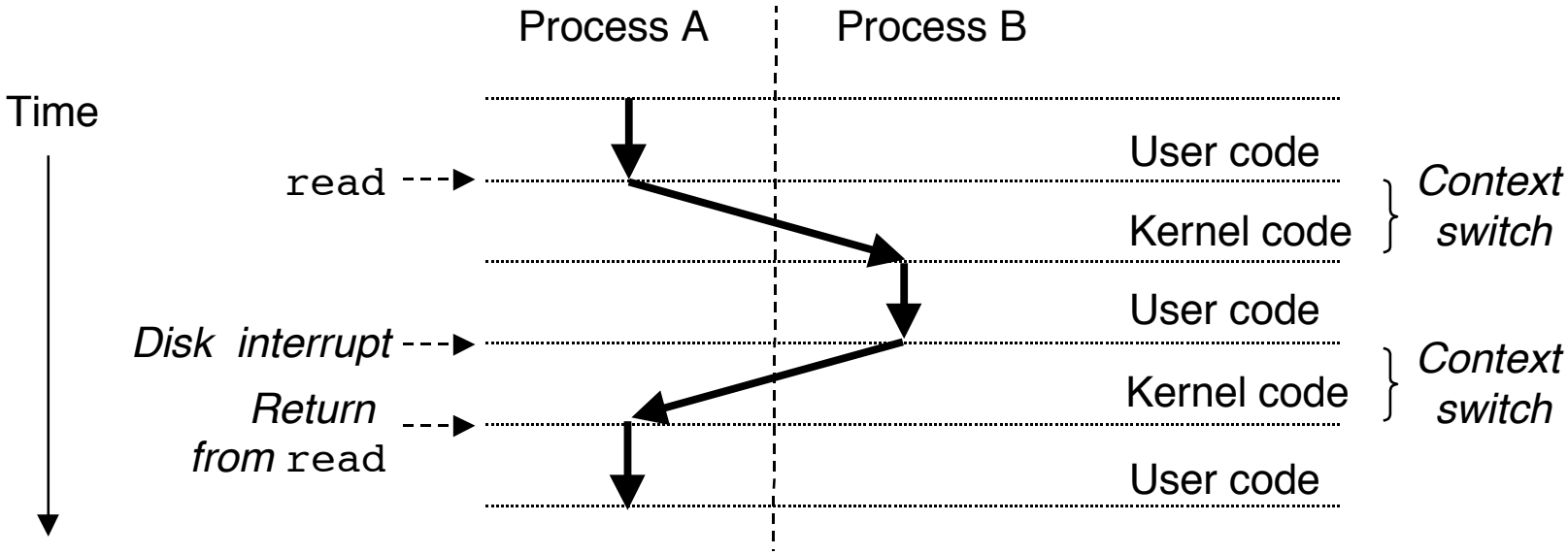
Logical control flow



Physical flow (1 CPU)

to do this, we need (1) a hardware mechanism to periodically interrupt the current process to load the OS, (2) an OS procedure that decides which processes to run, in what order, and (3) a routine for seamlessly transitioning between processes

(1) is the *periodic clock interrupt;*
(2) is the OS *scheduler;*
(3) is the *context switch*



Need new diagram that shows context switches triggered by the clock interrupt.

Context switches

to implement scheduling and carry out context switches, the OS must maintain a wealth of *per-process metadata*

a process comprises ...

{ code (program)

+ runtime data (global, local, dynamic)

+ PC, SP, FP & other registers

+ "process control block" (*OS metadata*) }

a process comprises ...

{ code (program)

+ runtime data (global, local, dynamic)

+ PC, SP, FP & other registers

+ *e.g., PID, mem/CPU usage, pending syscalls* }

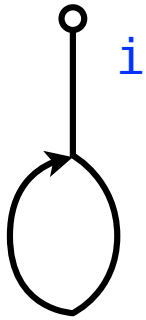
context switches are *external* to a process's *logical* control flow (dictated by user program)

— part of *exceptional* control flow

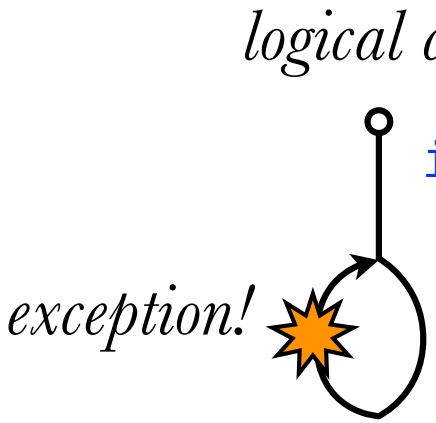
§Exceptional Control Flow

```
int main() {
    while (1)
    {
        printf("hello world!\n");
    }
    return 0;
}
```

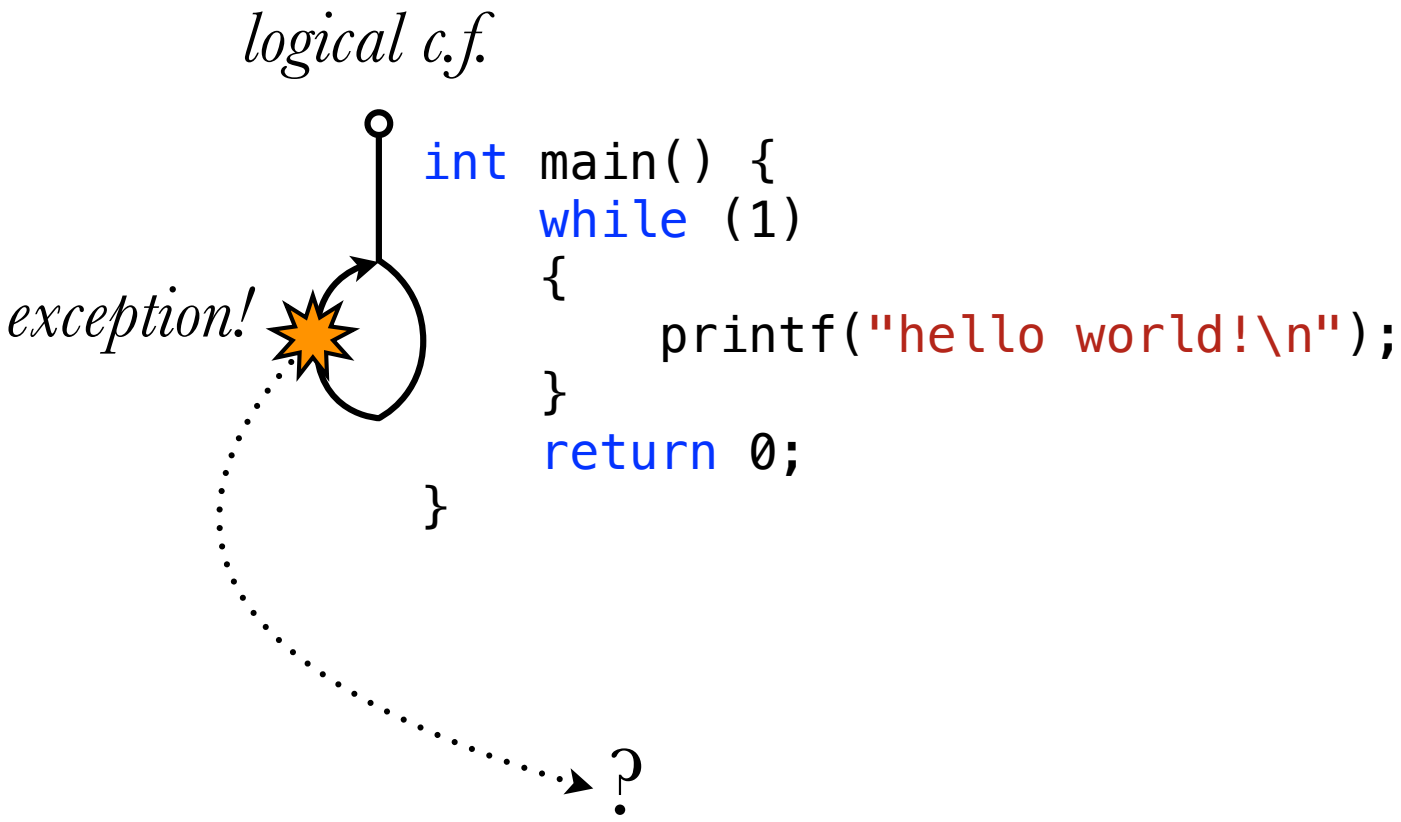
logical c.f.



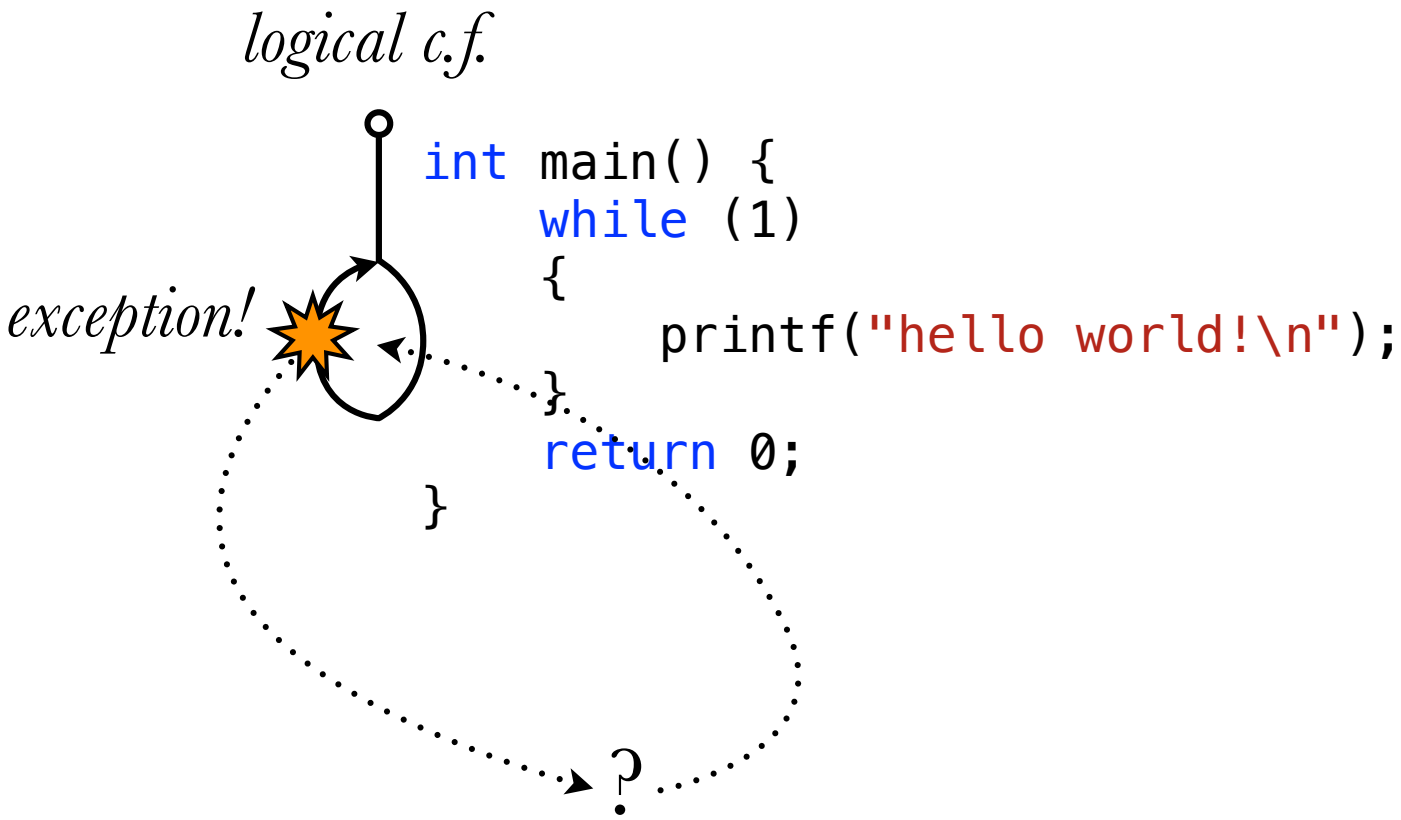
```
int main() {
    while (1)
    {
        printf("hello world!\n");
    }
    return 0;
}
```




```c
int main() {
    while (1)
    {
        printf("hello world!\n");
    }
    return 0;
}
```




```c
int main() {
    while (1)
    {
        printf("hello world!\n");
    }
    return 0;
}
```


logical c.f.
exception!
int main() {
while (1)
{
printf("hello world!\n");
}
return 0;
}
?

Two classes of exceptions:

I. synchronous

II. asynchronous

I. synchronous exceptions are caused by the *currently executing* instruction

3 subclasses of synchronous exceptions:

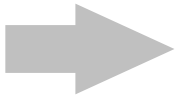
1. traps
2. faults
3. aborts

1. traps

traps are *intentionally* triggered by a process

e.g., to invoke a system call

```
char *str = "hello world";
int len = strlen(str);
write(1, str, len);
```



```
mov   edx, len
mov   ecx, str
mov   ebx, 1
mov   eax, 4   ; syscall #4
int   0x80     ; trap to OS
```

return from trap (if it happens) resumes execution at the next logical instruction

2. faults

faults are usually *unintentional*, and may be recoverable or irrecoverable

e.g., segmentation fault, protection fault, page fault, div-by-zero

often, return from fault will result in *retrying* the faulting instruction

— esp. if the handler "fixes" the problem

3. aborts

aborts are *unintentional* and *irrecoverable*

i.e., abort = program/OS termination

e.g., memory ECC error

II. asynchronous exceptions are caused by events *external to* the current instruction

```c
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```

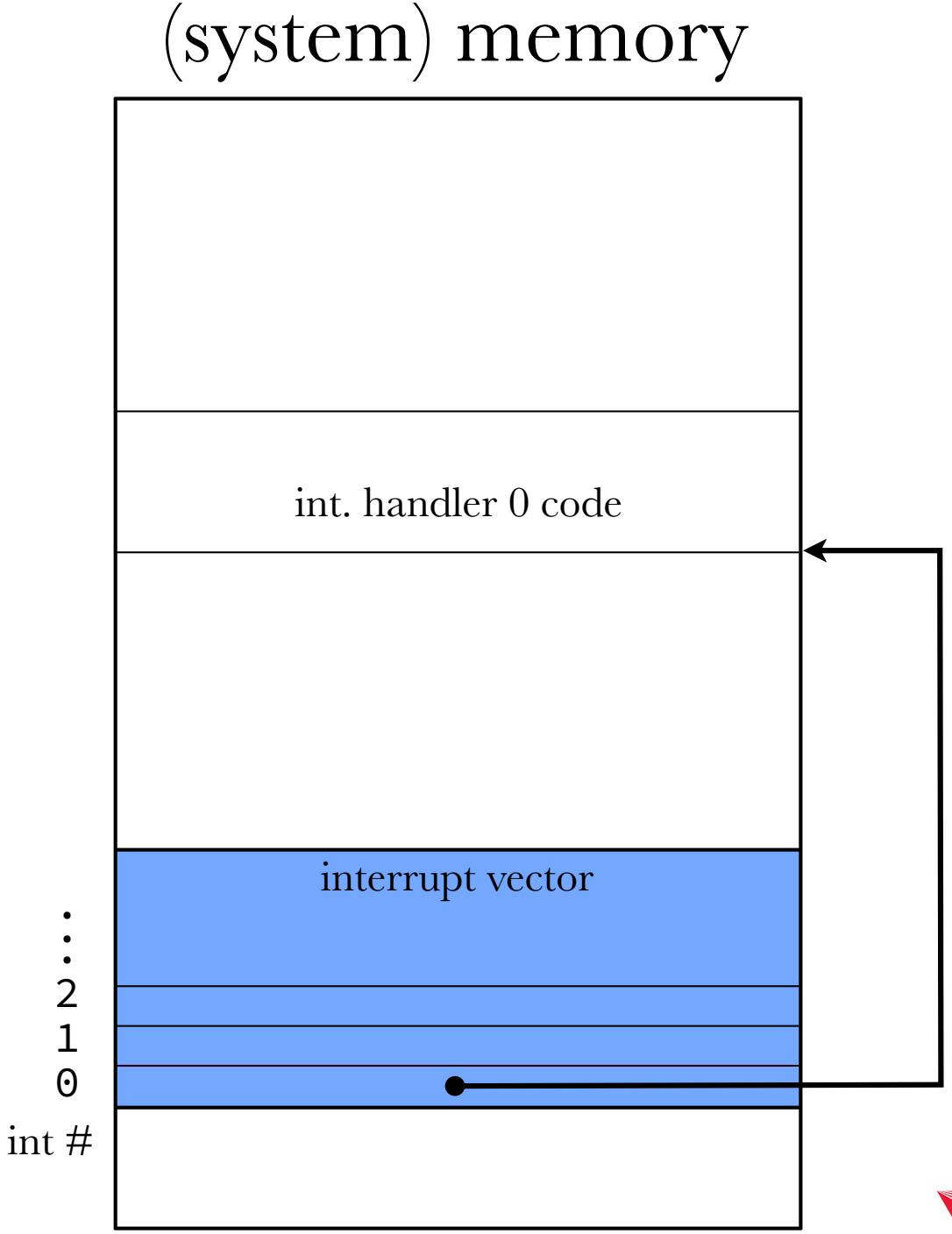
```
hello world!
hello world!
hello world!
hello world!
^C
$
```

hardware initiated asynchronous exceptions are known as *interrupts*

e.g., ctrl-C, ctrl-alt-del, power switch

interrupts are associated with specific processor (hardware) pins

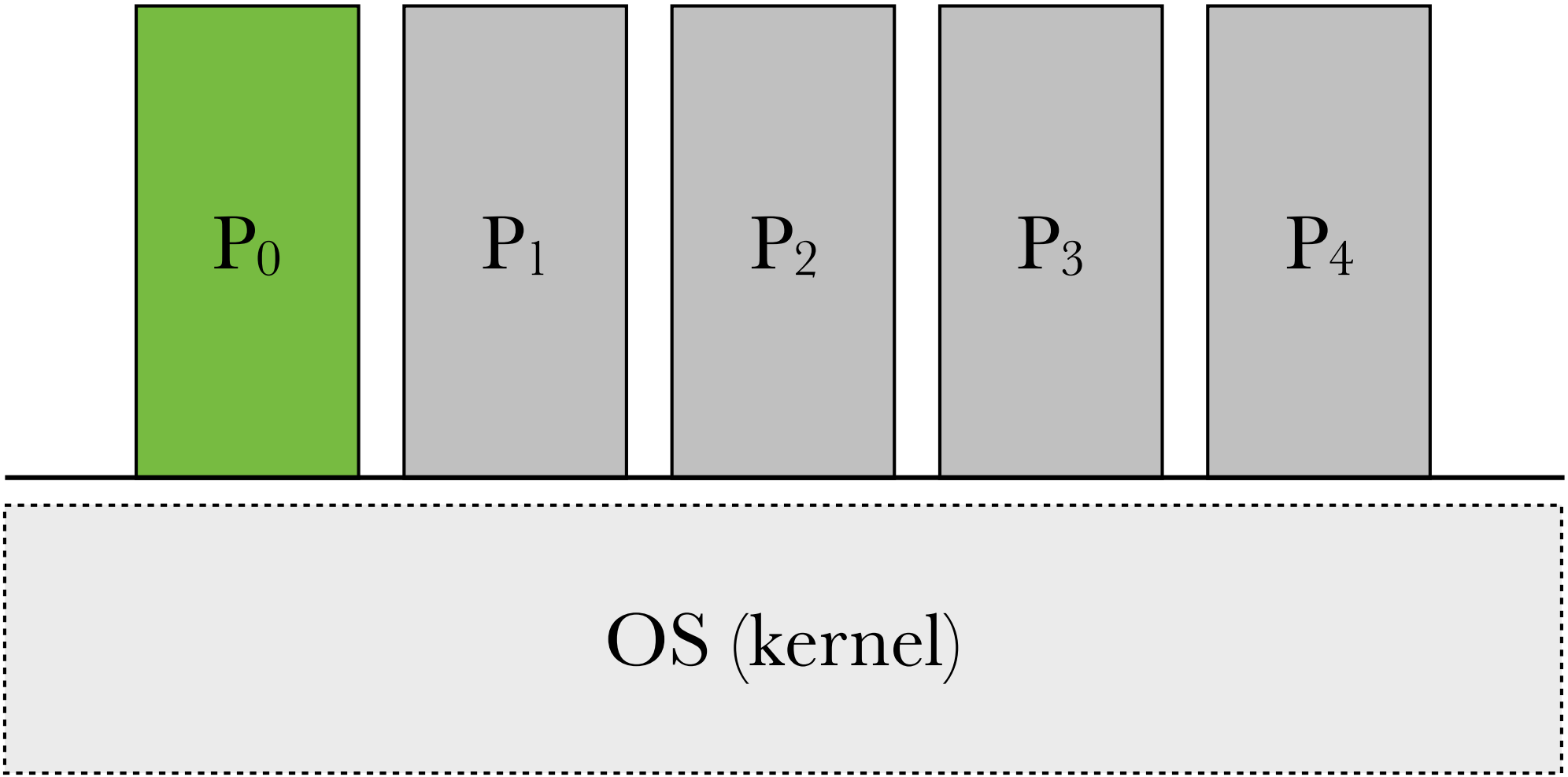
- checked after every CPU cycle
- associated with interrupt handlers

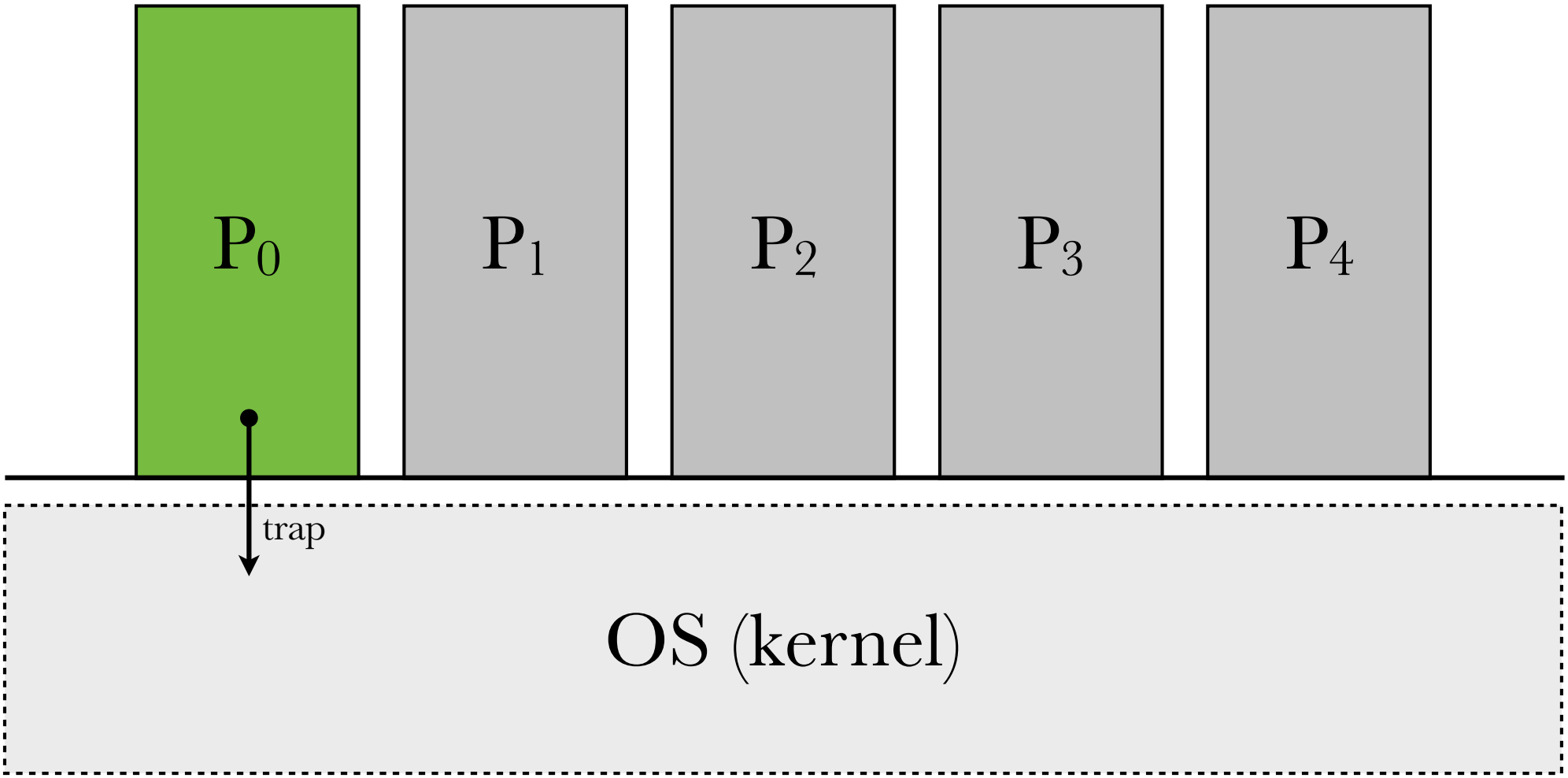

(system) memory
int. handler 0 code
interrupt vector
:
2
1
0
int #

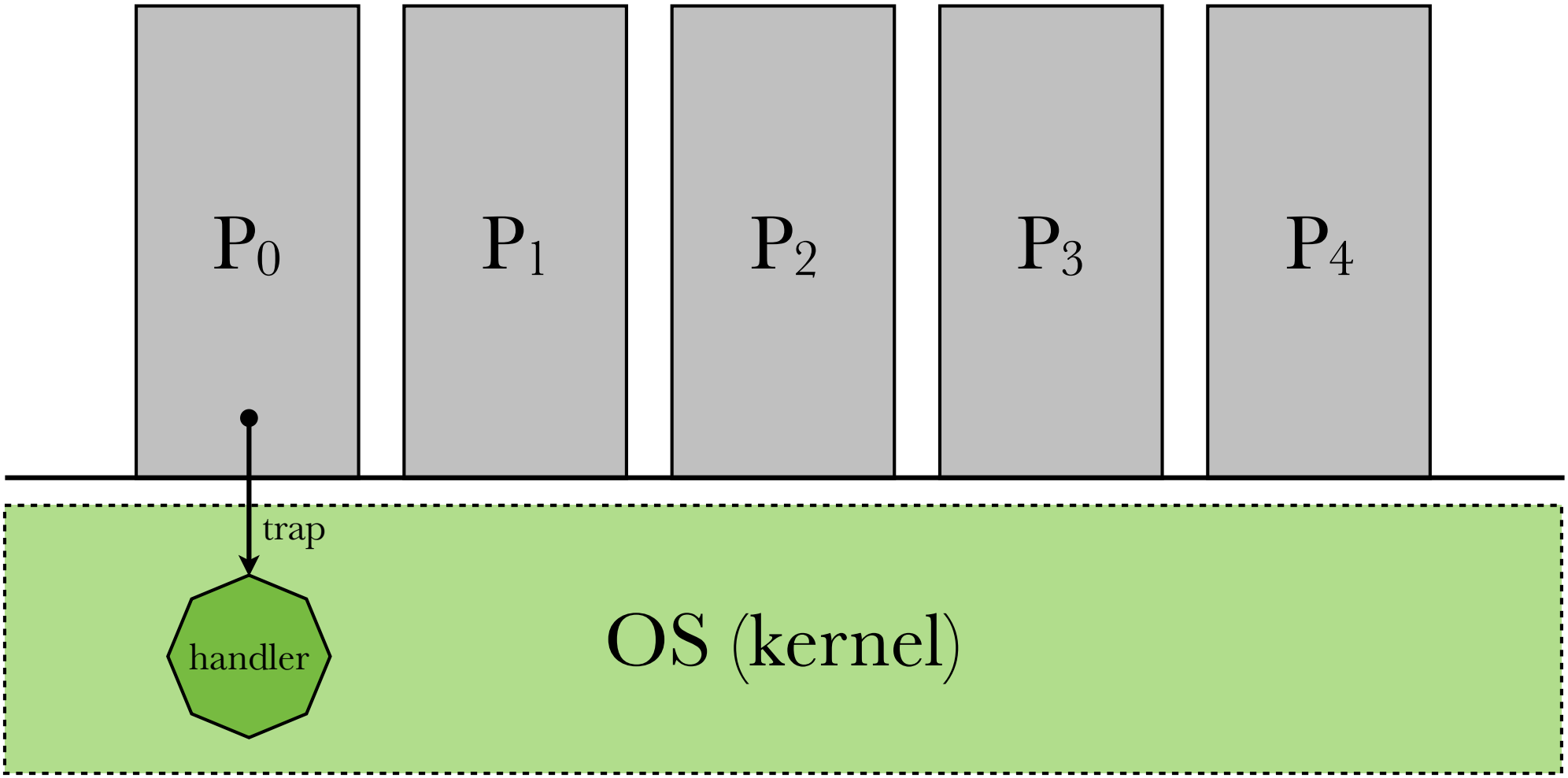
interrupt procedure (typical)

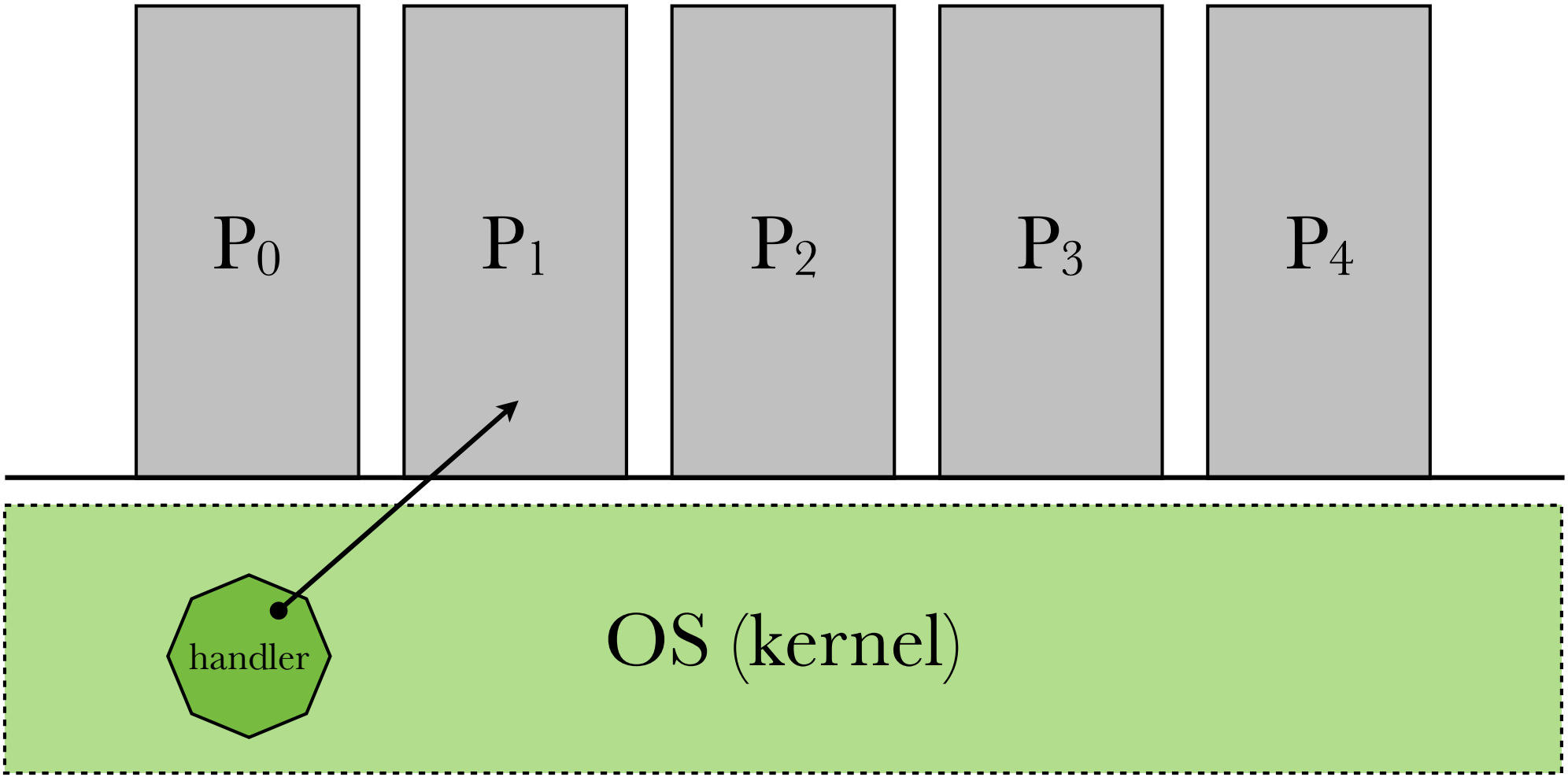
- save context (e.g., user process)
- load OS context
- execute handler
- load context (for …?)
- return

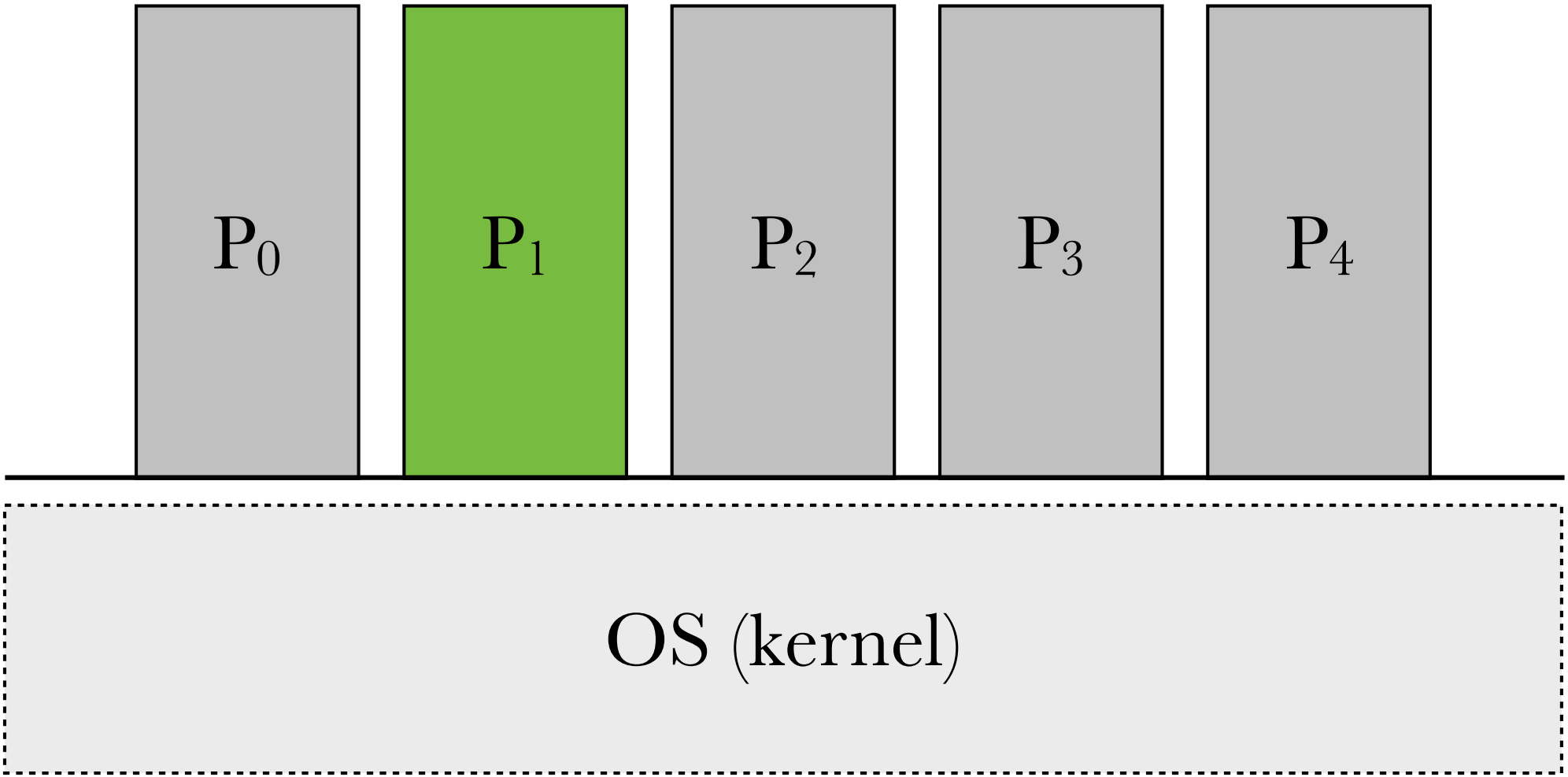
important: after switching context to the OS (for exception handling), there is *no guarantee if/when* a process will be switched back in!

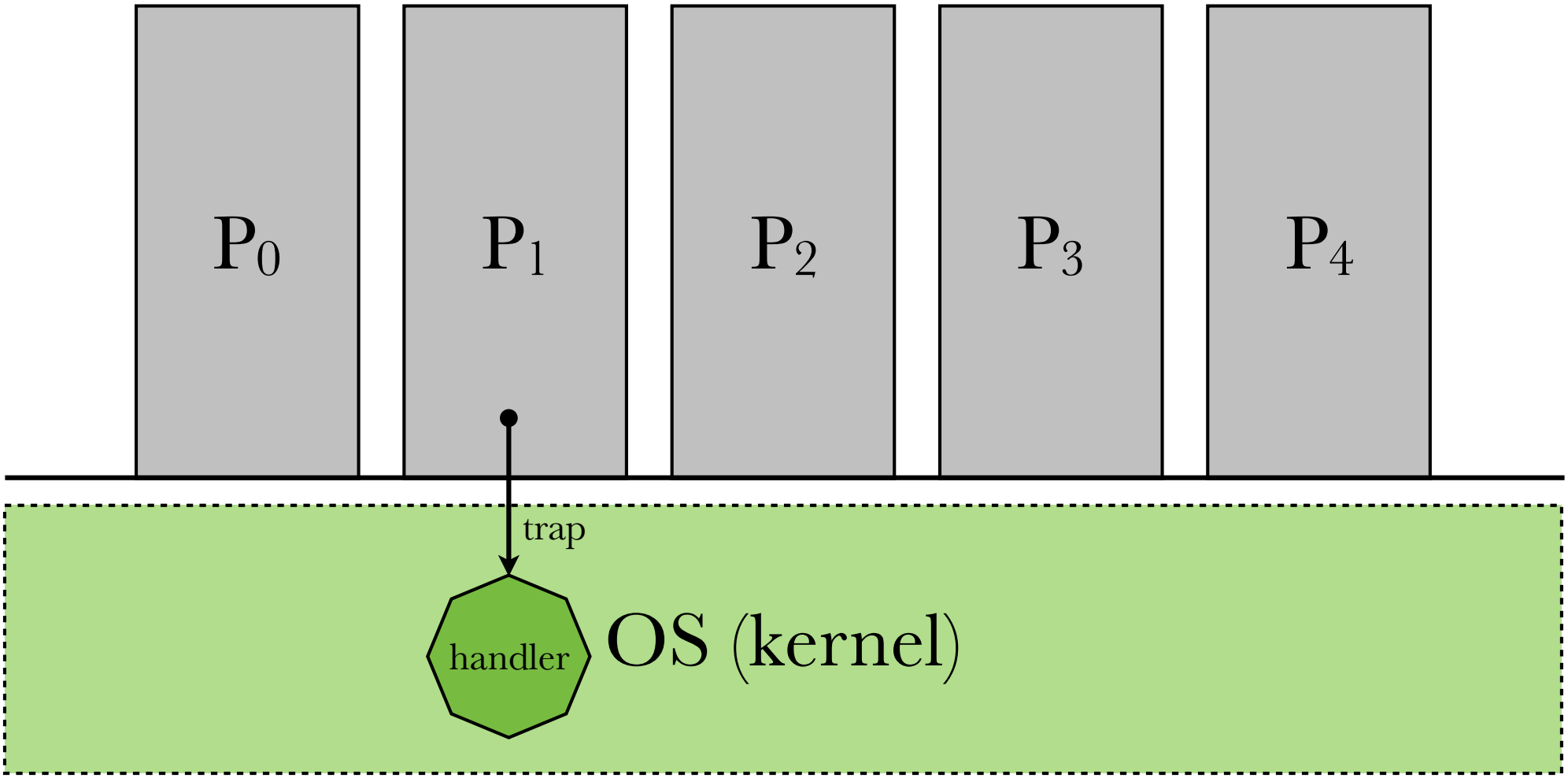

P0
P1
P2
P3
P4
OS (kernel)

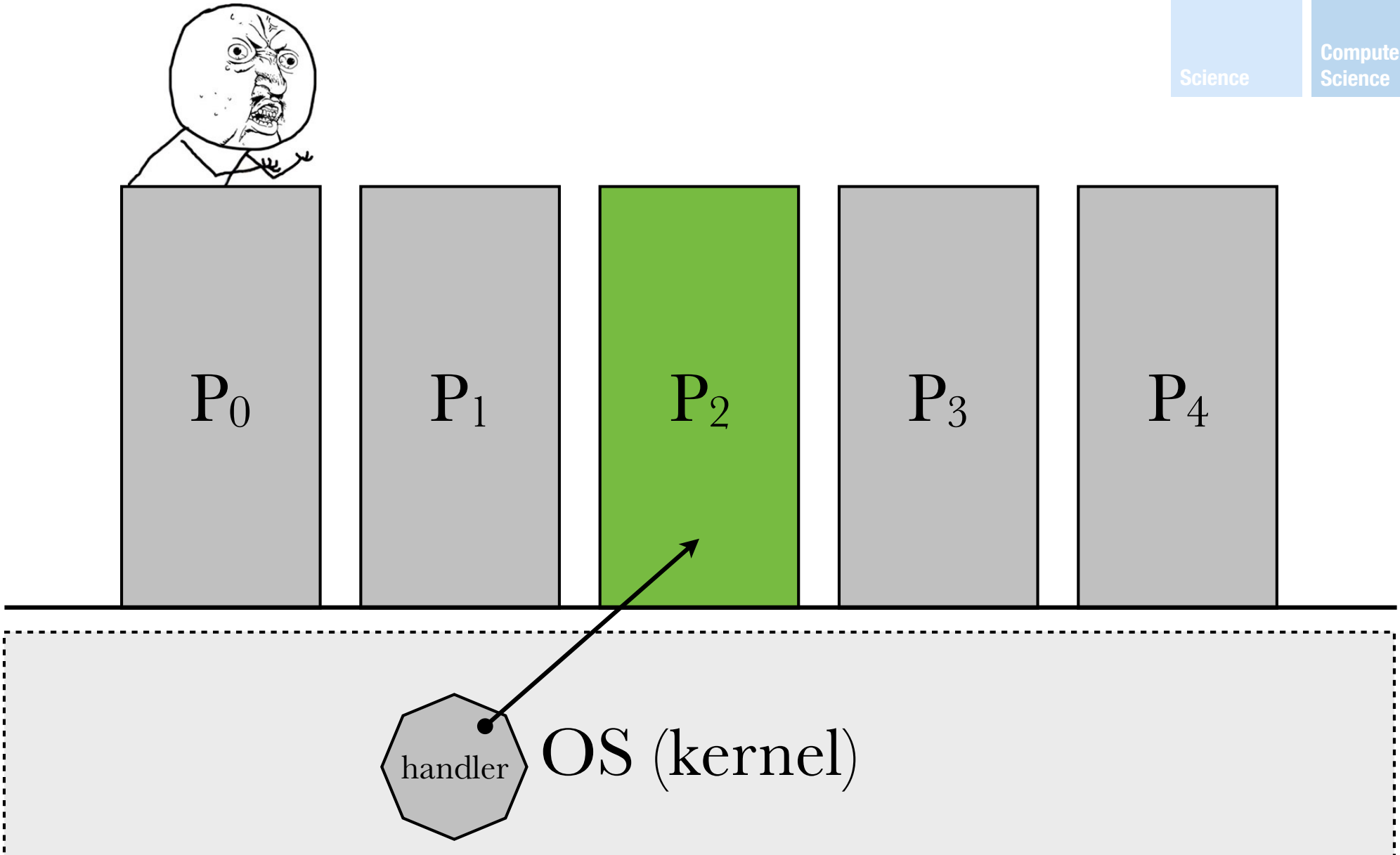

P0
P1
P2
P3
P4
trap
OS (kernel)


P0
P1
P2
P3
P4
trap
handler
OS (kernel)


P0
P1
P2
P3
P4
handler
OS (kernel)


P_0
P_1
P_2
P_3
P_4
OS (kernel)


P0
P1
P2
P3
P4
trap
handler
OS (kernel)


P0
P1
P2
P3
P4
handler
OS (kernel)

switching context to the kernel is potentially *very expensive*

— but the only way to invoke system calls and access I/O

moral (to be reinforced ad nauseum):

use system calls (traps) sparingly!