Processes & ECF

Science

Computer Science

CS 351: Systems Programming

Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Agenda

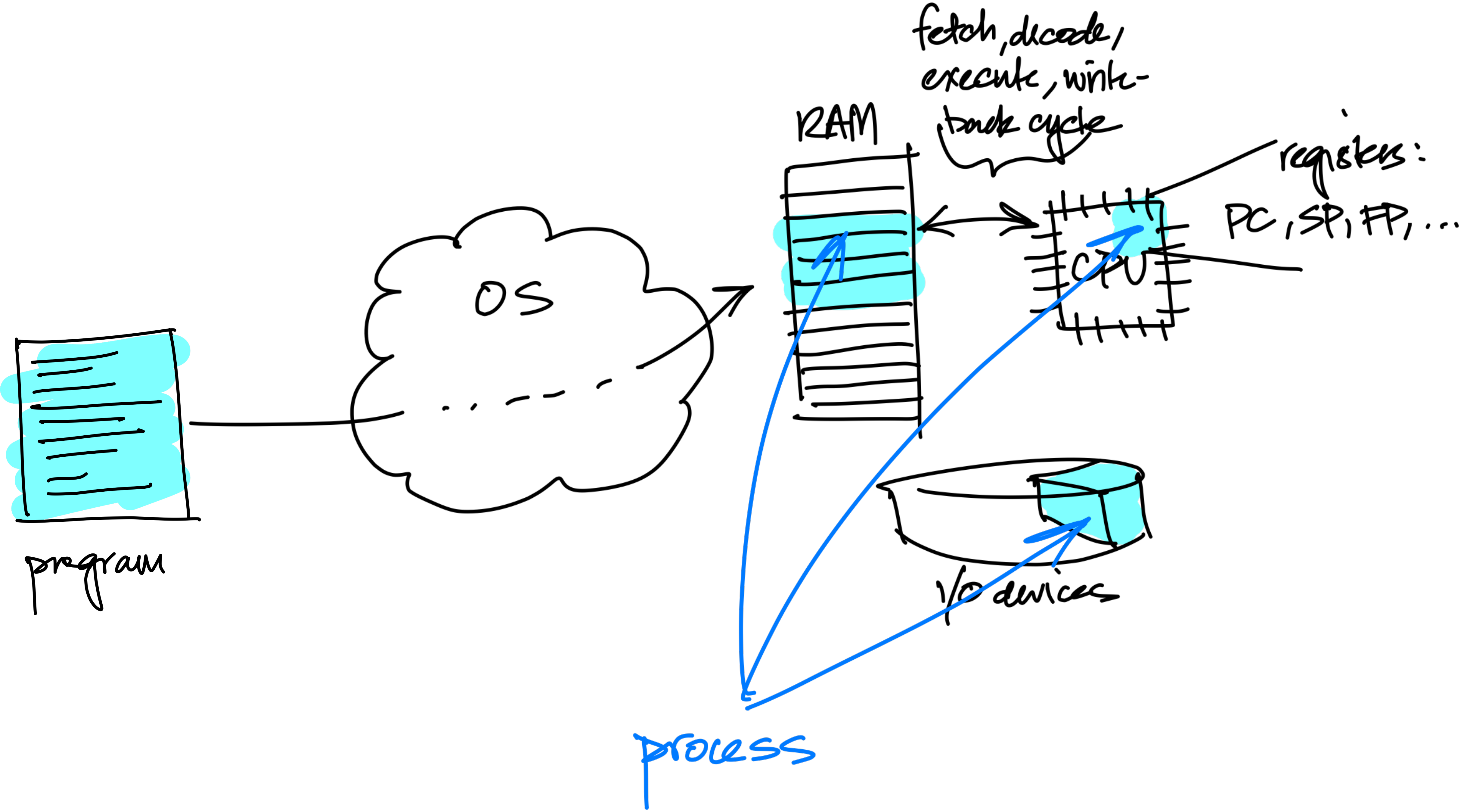
- Definition & OS responsibilities
- Exceptional control flow
    - synch vs. asynch exceptions
    - exception handling procedure

§Definition & OS responsibilities

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

programs *describe* what we want done,

processes *carry out* what we want done

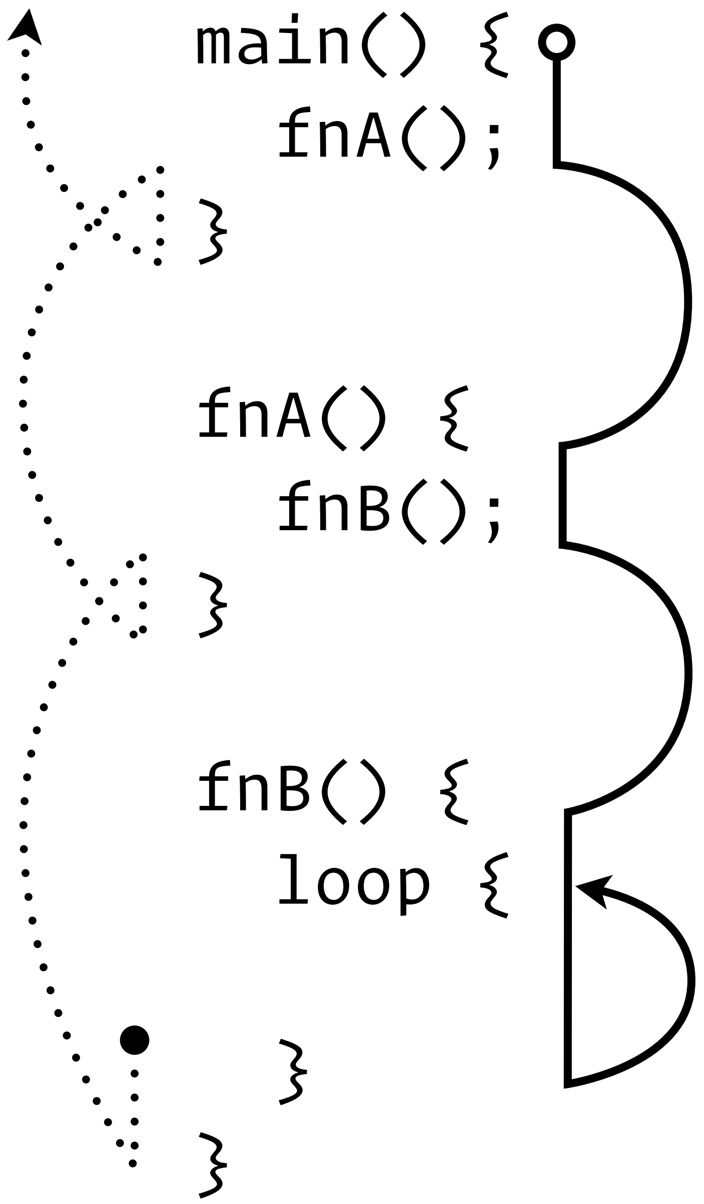

fetch, decode, execute, write-back cycle
RAM
registers:
PC, SP, FP, ...
CPU
OS
program
I/O devices
process

a process comprises …

{ code (program)

+ runtime data (global, local, dynamic)

+ PC, SP, FP & other registers }



```
main() {
  fnA();
}

fnA() {
  fnB();
}

fnB() {
  loop {

  }
}
```

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

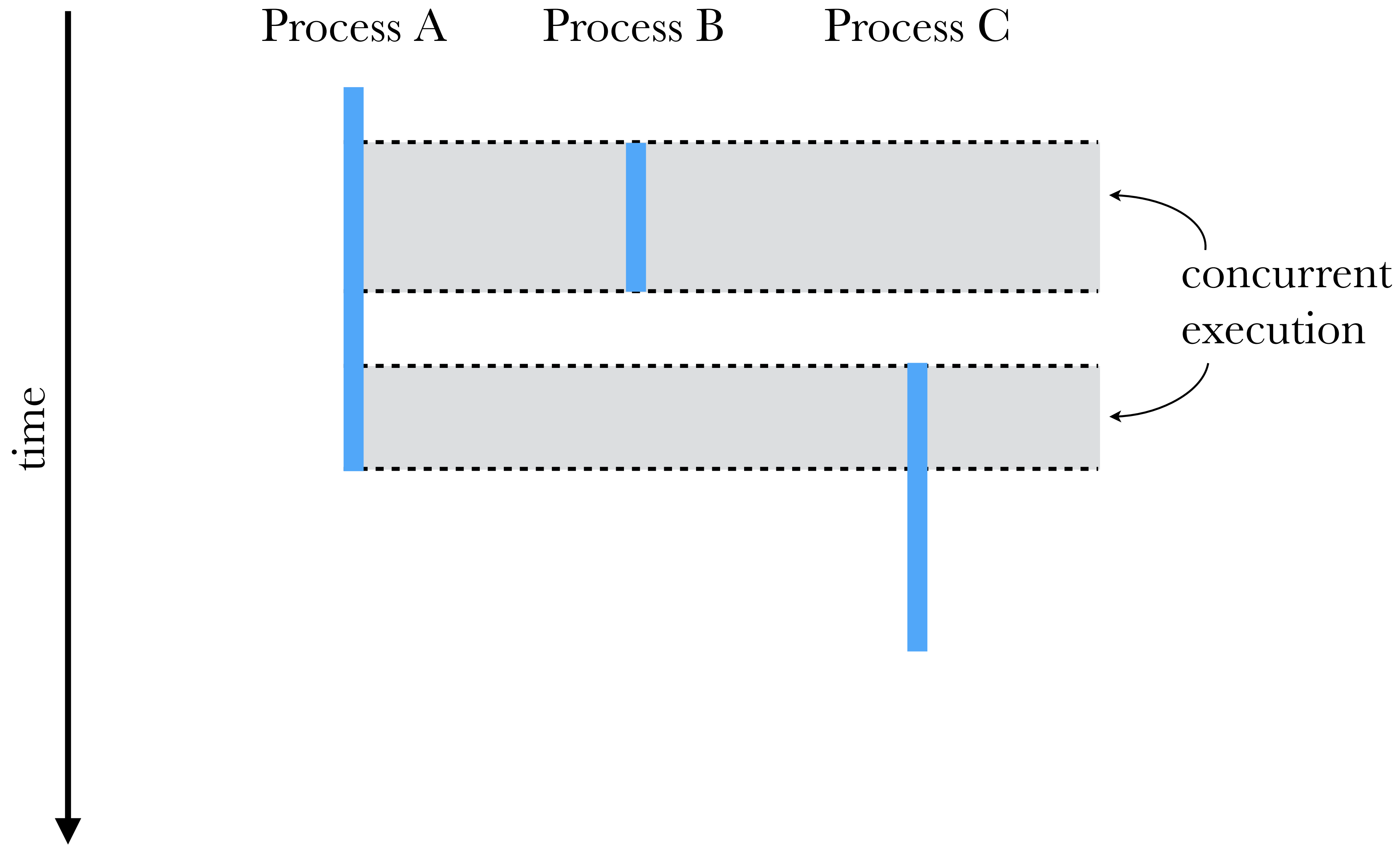
which requires that nothing disrupt the program mid-execution easiest way to guarantee this is for a process to “own” the CPU for its entire duration (i.e., no-one else allowed to run)

... downsides?

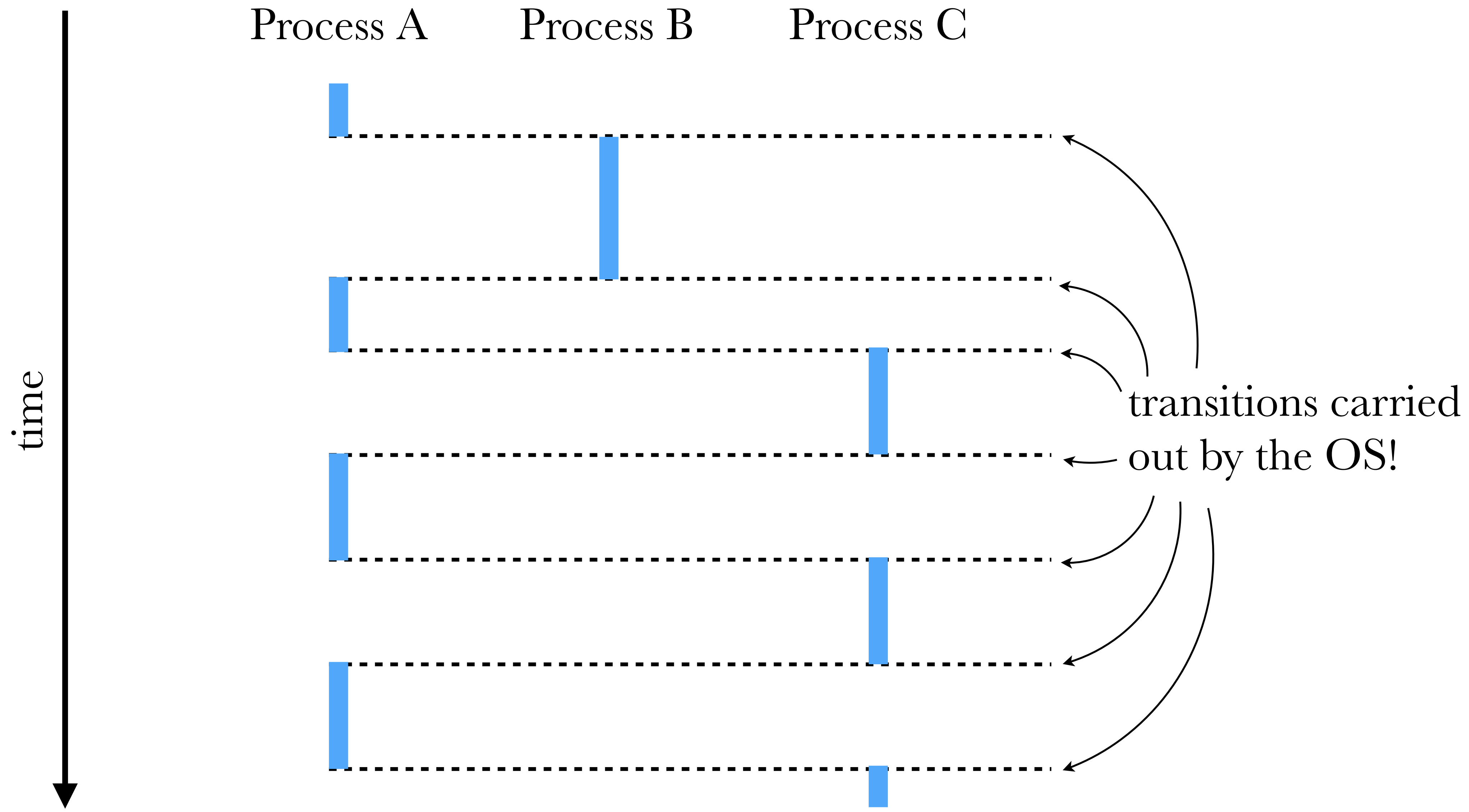
1. No multitasking!
2. A malicious (or badly written) program can "take over" the CPU forever
3. An idle process (e.g., waiting for input) will underutilize the CPU

the operating system simulates a *seamless logical control flow* for each active process

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



Logical control flow



Physical flow (1 CPU)

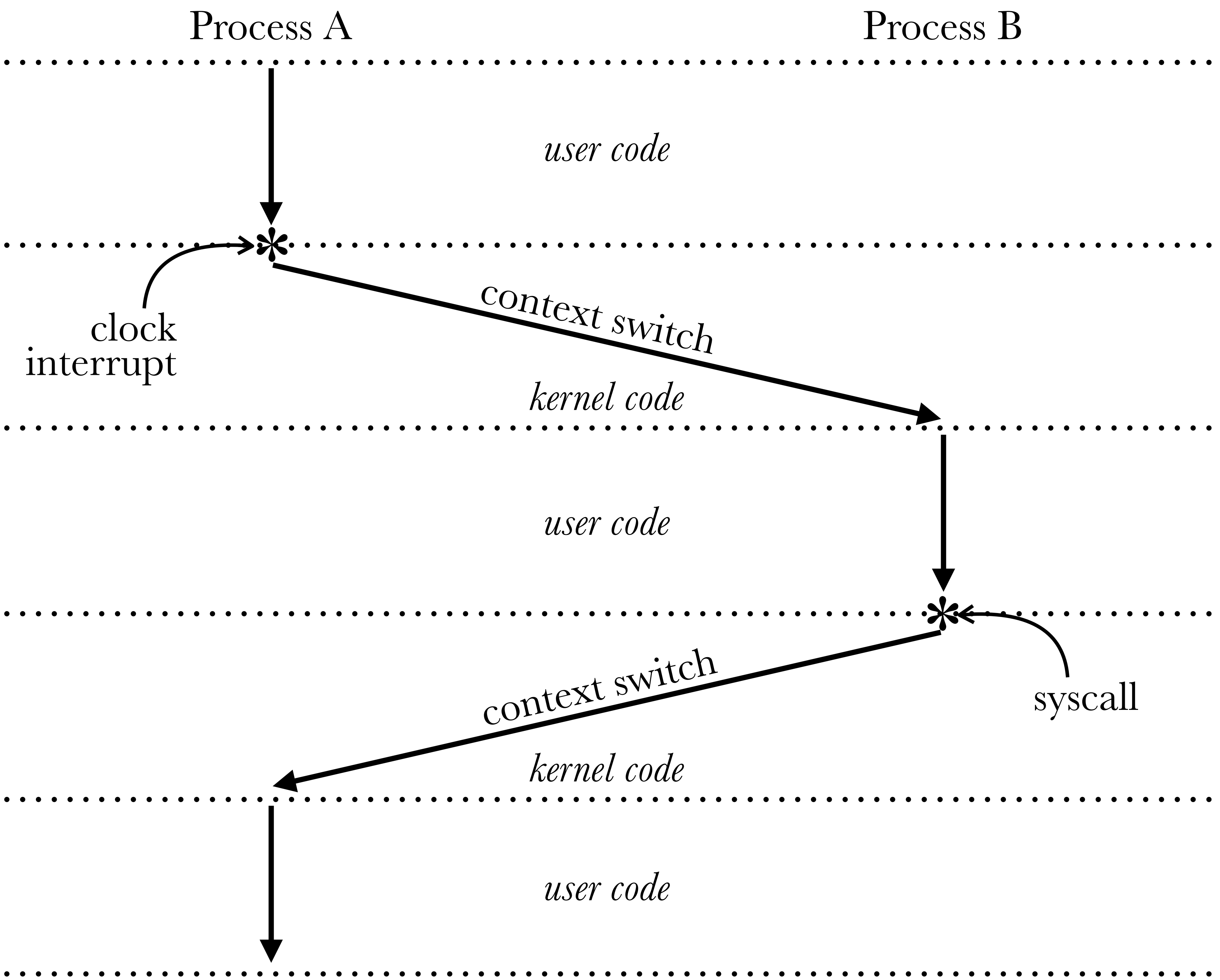
to implement this, we need

1. a mechanism to *periodically interrupt* the current process to run the OS
2. an OS module that *schedules* processes
3. a routine to help seamlessly *switch* between processes seamlessly

(1) is the *periodic clock interrupt;*

(2) is the OS *scheduler;*

(3) is the *context switch*


Process A
Process B
user code
clock interrupt
context switch
kernel code
user code
syscall
context switch
kernel code
user code

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

+ OS metadata, aka *process control block* } a process comprises ...

{ code (program)

+ runtime data (global, local, dynamic)

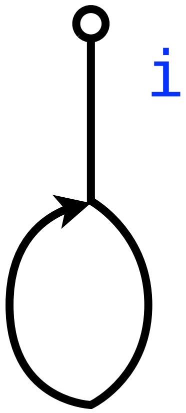
+ PC, SP, FP & other registers

+ *e.g., PID, mem/CPU usage, pending syscalls* } actions that take place outside a process's logical control flow (e.g., context switches), but may still affect its behavior are part of the process's ***exceptional* control flow**

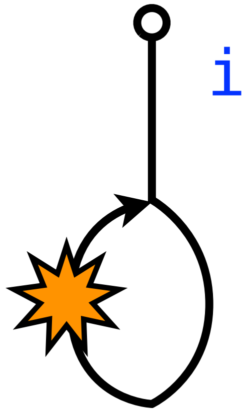
§ Exceptional Control Flow

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

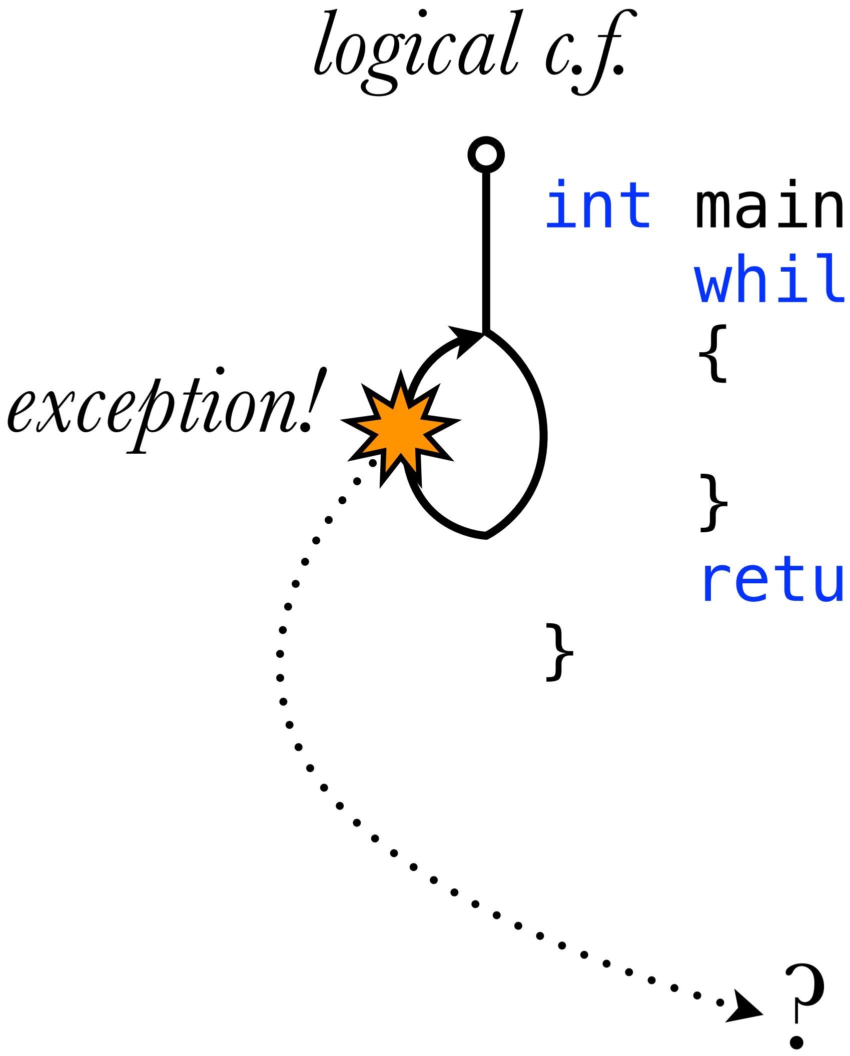
logical c.f.



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



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

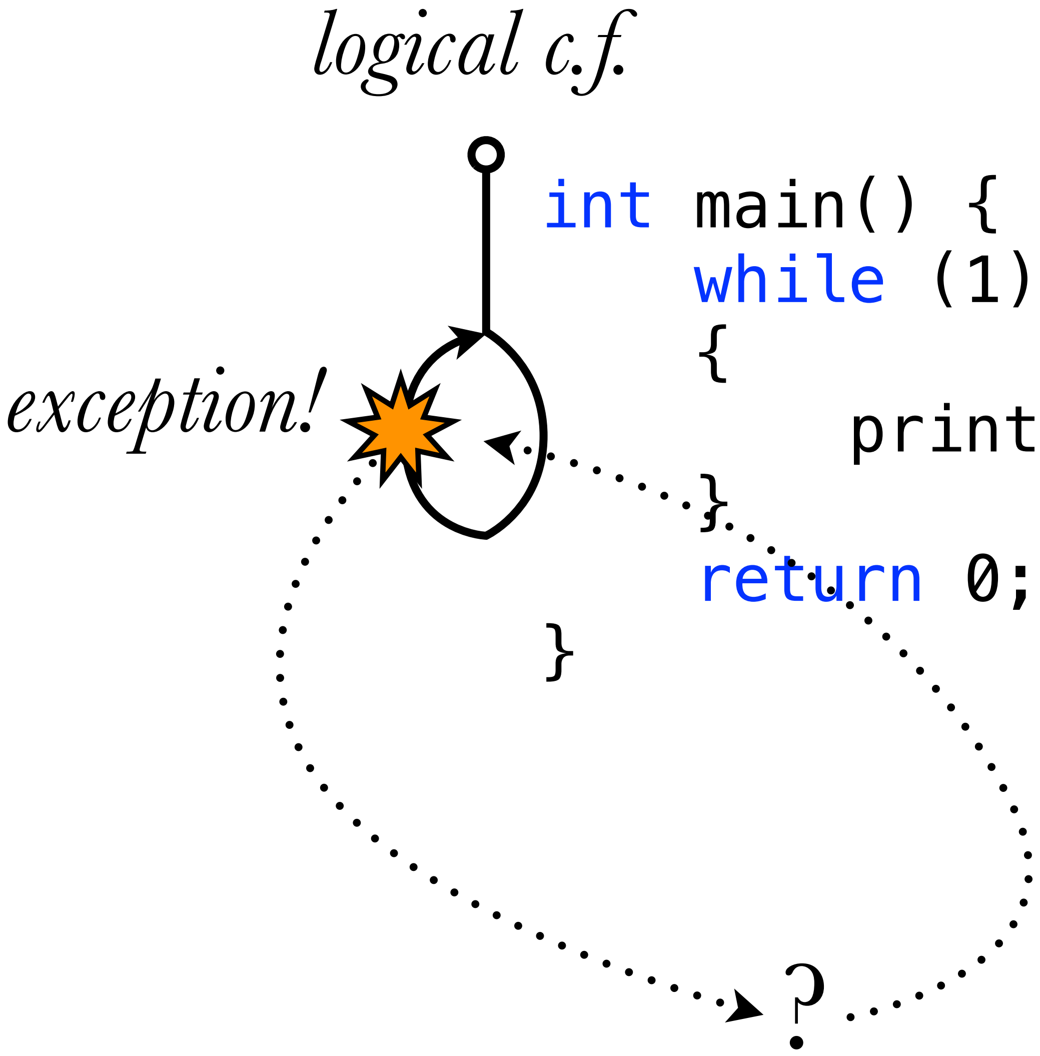


logical c.f.

exception!

?

```
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 (i.e., the one actively running on the CPU)

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

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

```
movl len, %edx
movl str, %ecx
movl $1,  %ebx
movl $4,  %eax # syscall num
int  $0x80     # trap instr
...
```

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

i.e., looks like a function call!

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

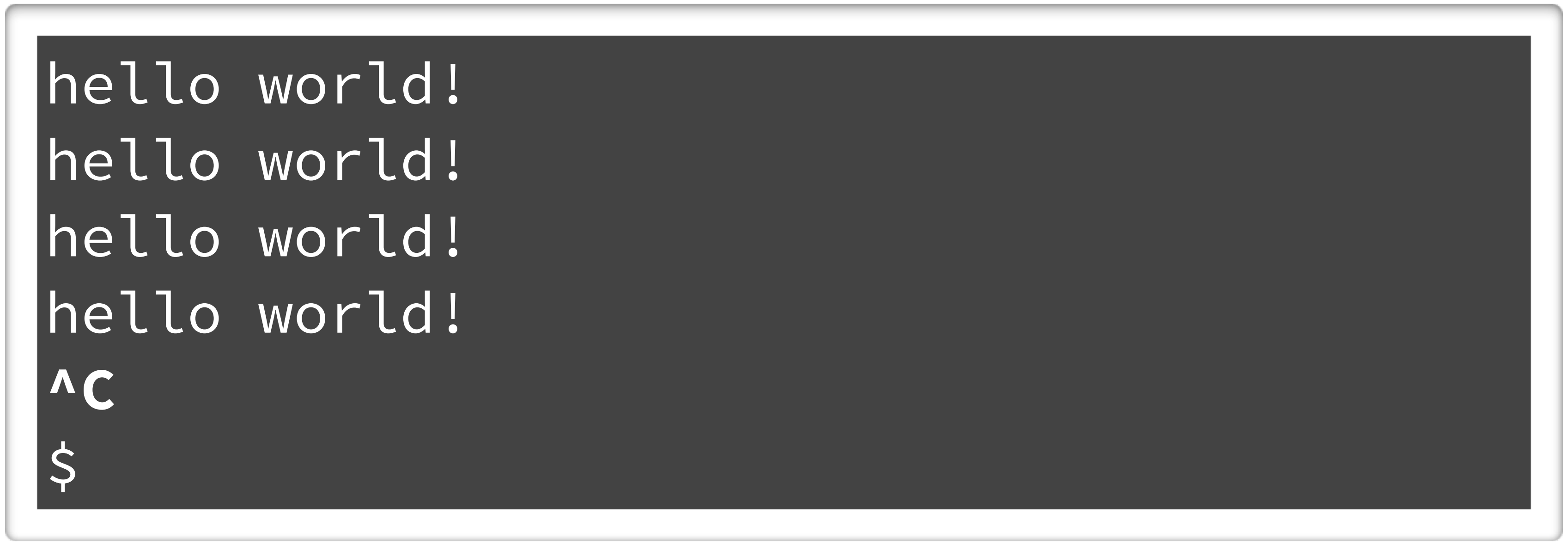
aborts are *unintentional* and *irrecoverable*

i.e., abort = program/OS termination

e.g., memory ECC error

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

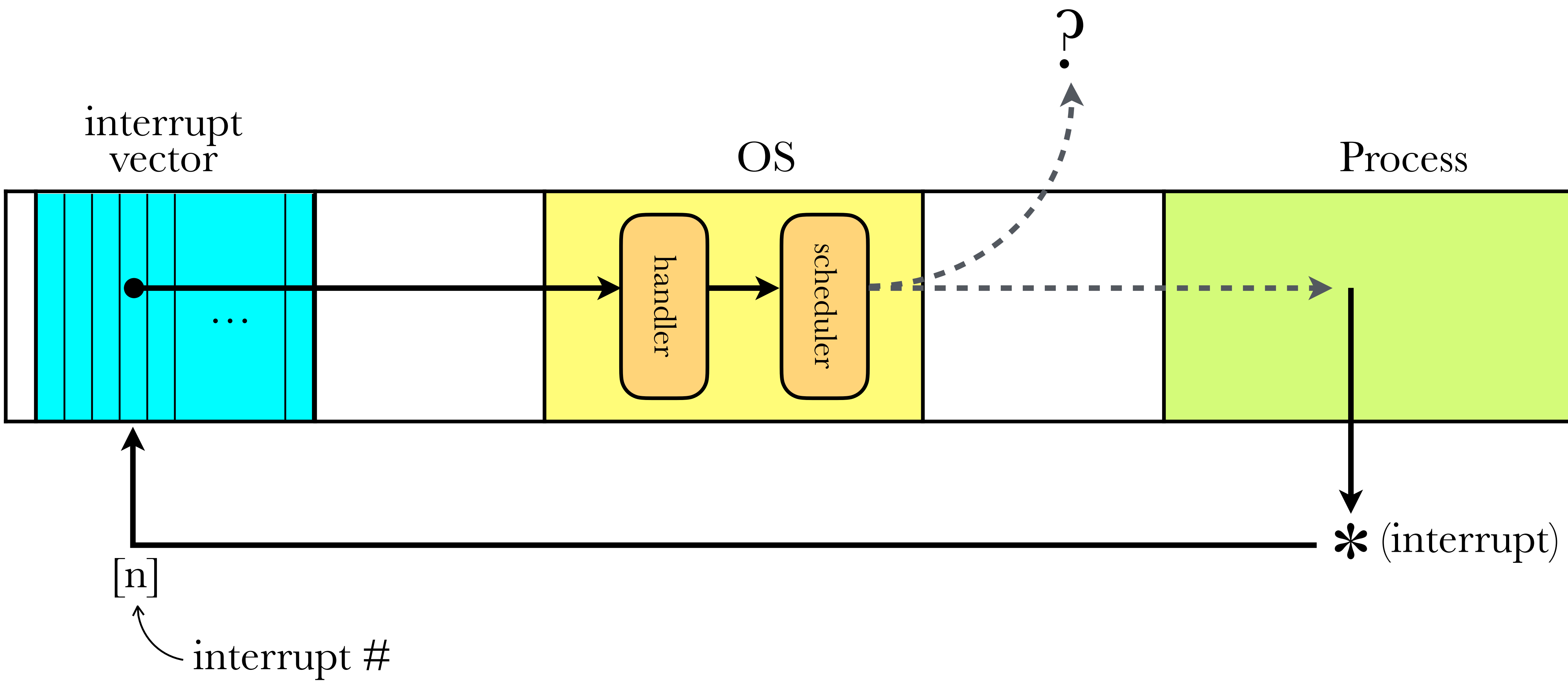
```c
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```



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

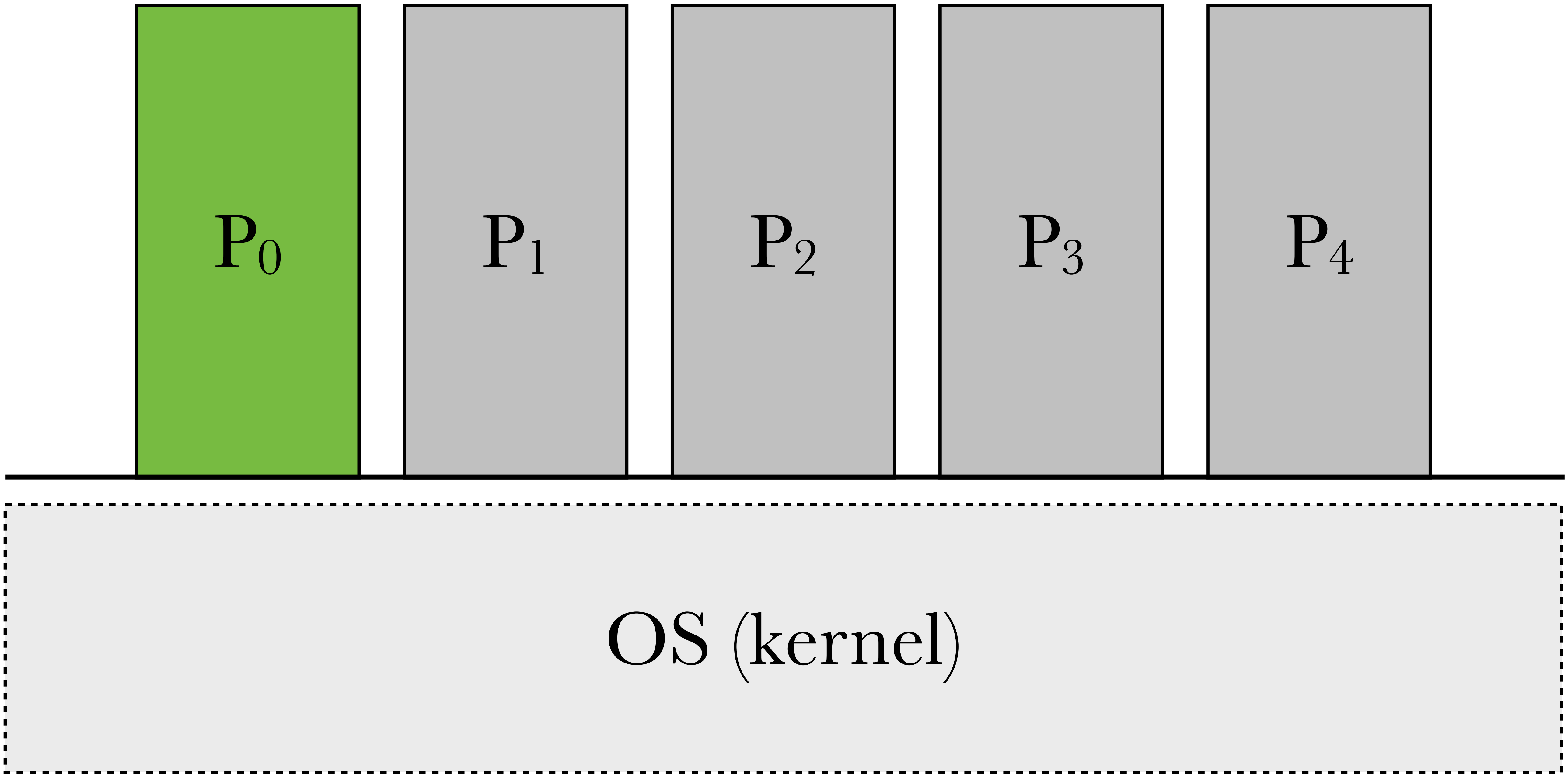
hardware initiated asynchronous exceptions are known as *interrupts* e.g., ctrl-C, ctrl-alt-del, power switch interrupts are associated with specific processor (hardware) pins

- checked after every CPU cycle
- associated with handler functions via the "interrupt vector"


interrupt vector
OS
Process
handler
scheduler
...
?
✻ (interrupt)
[n]
interrupt #

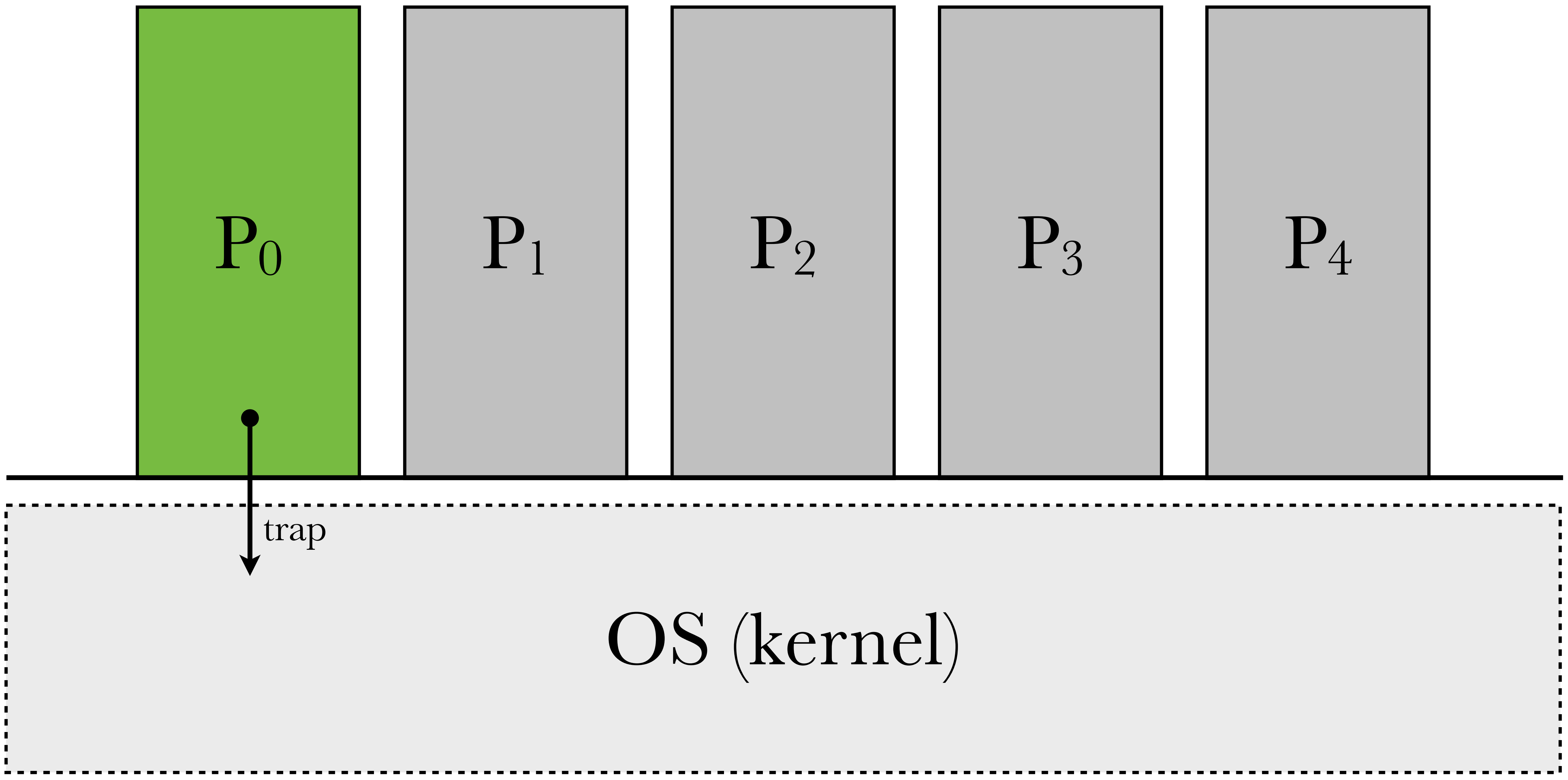
interrupt procedure (typical)

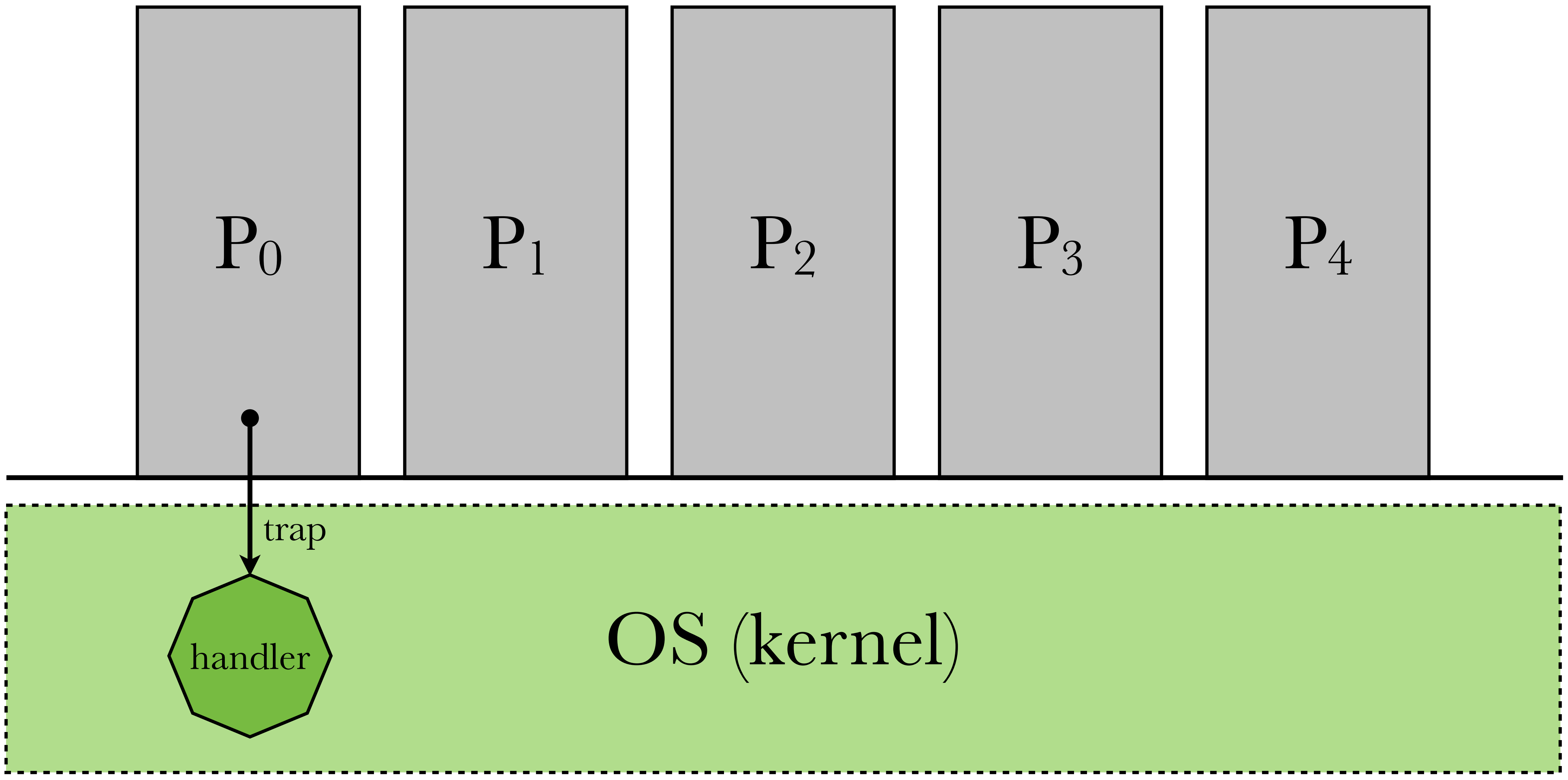
- save context (for outgoing process)
- load OS
- run handler & scheduler
- load context (for incoming process)
- return

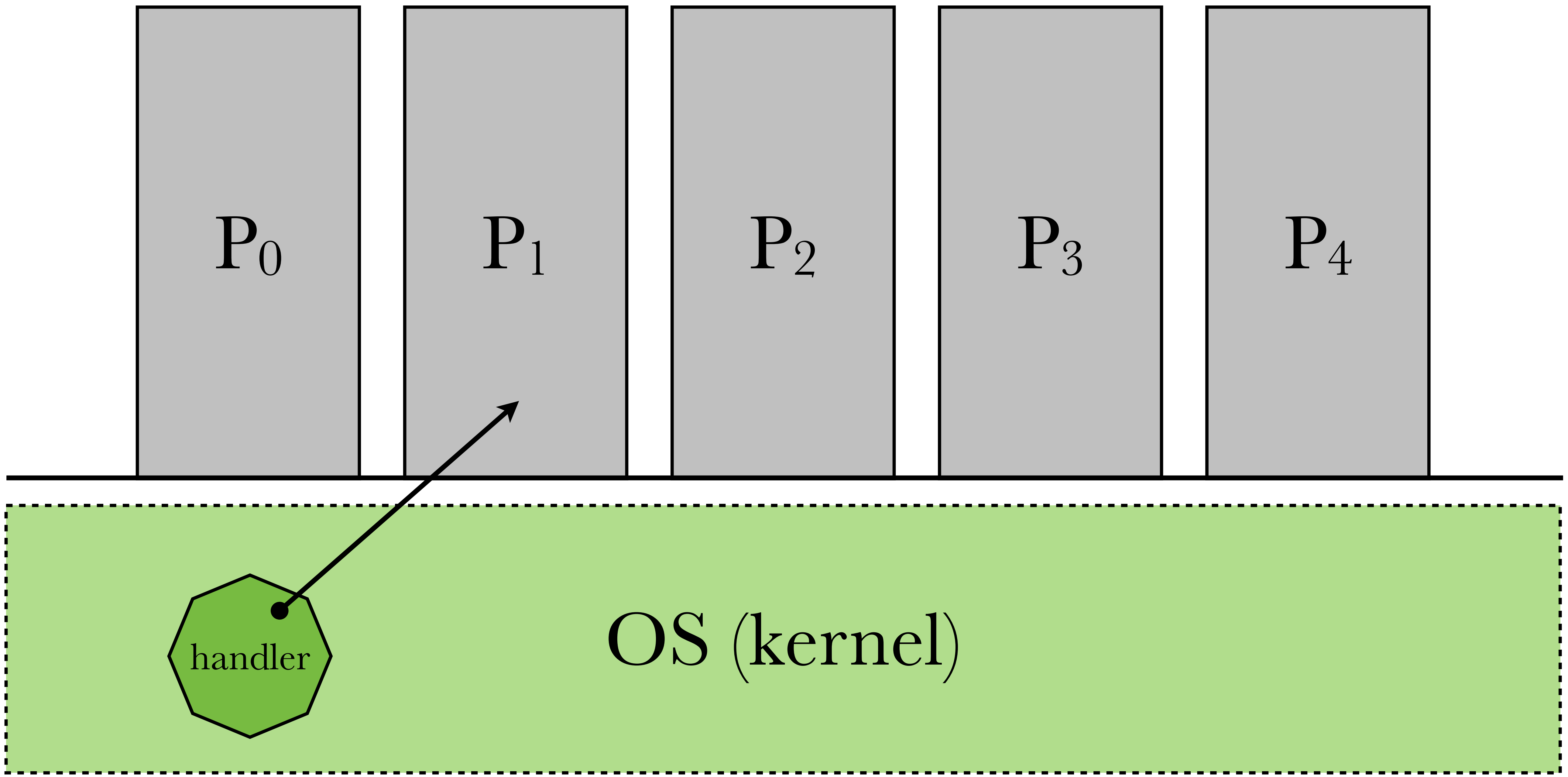


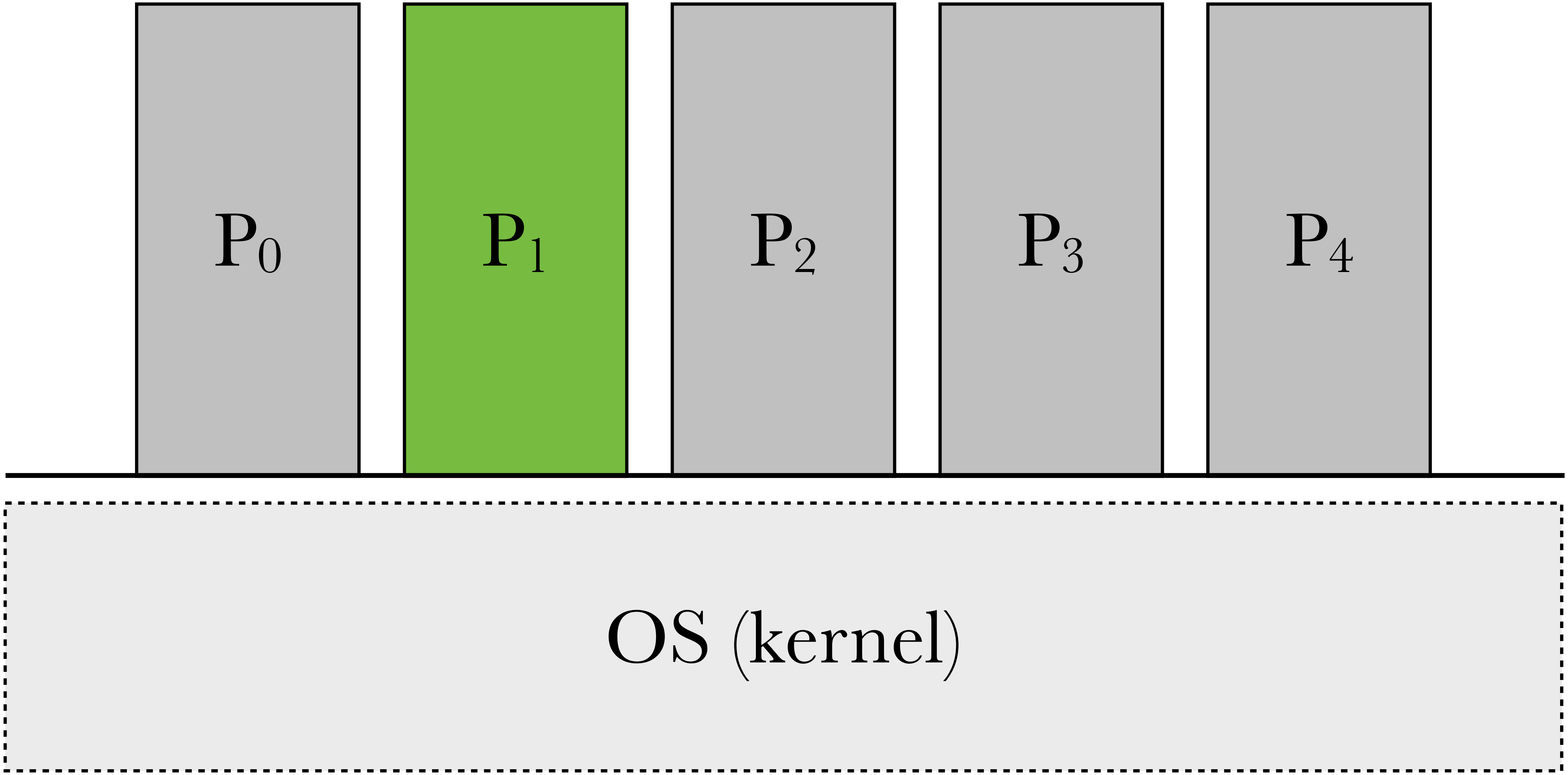
P_0 P_1 P_2 P_3 P_4

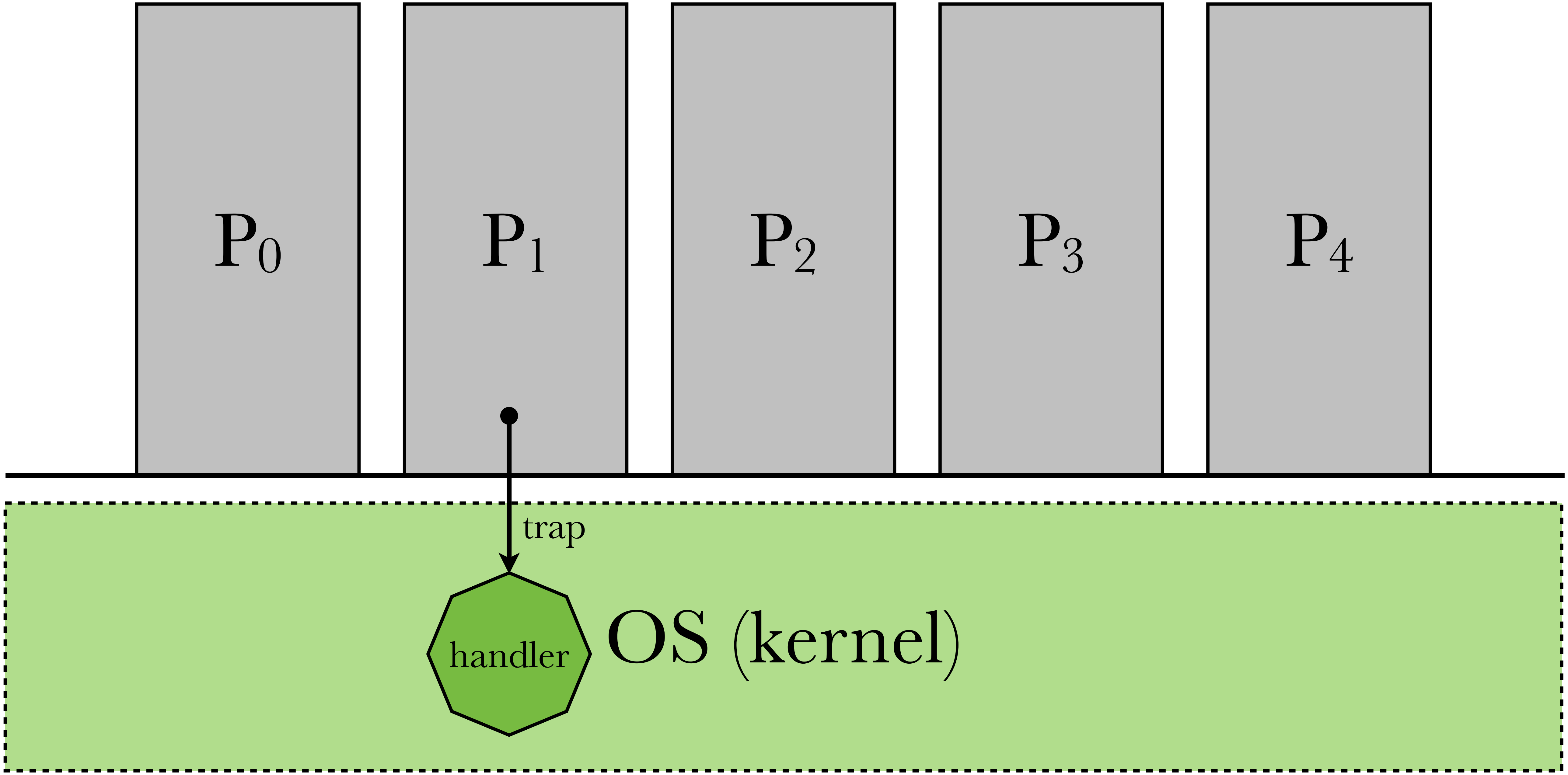
OS (kernel)

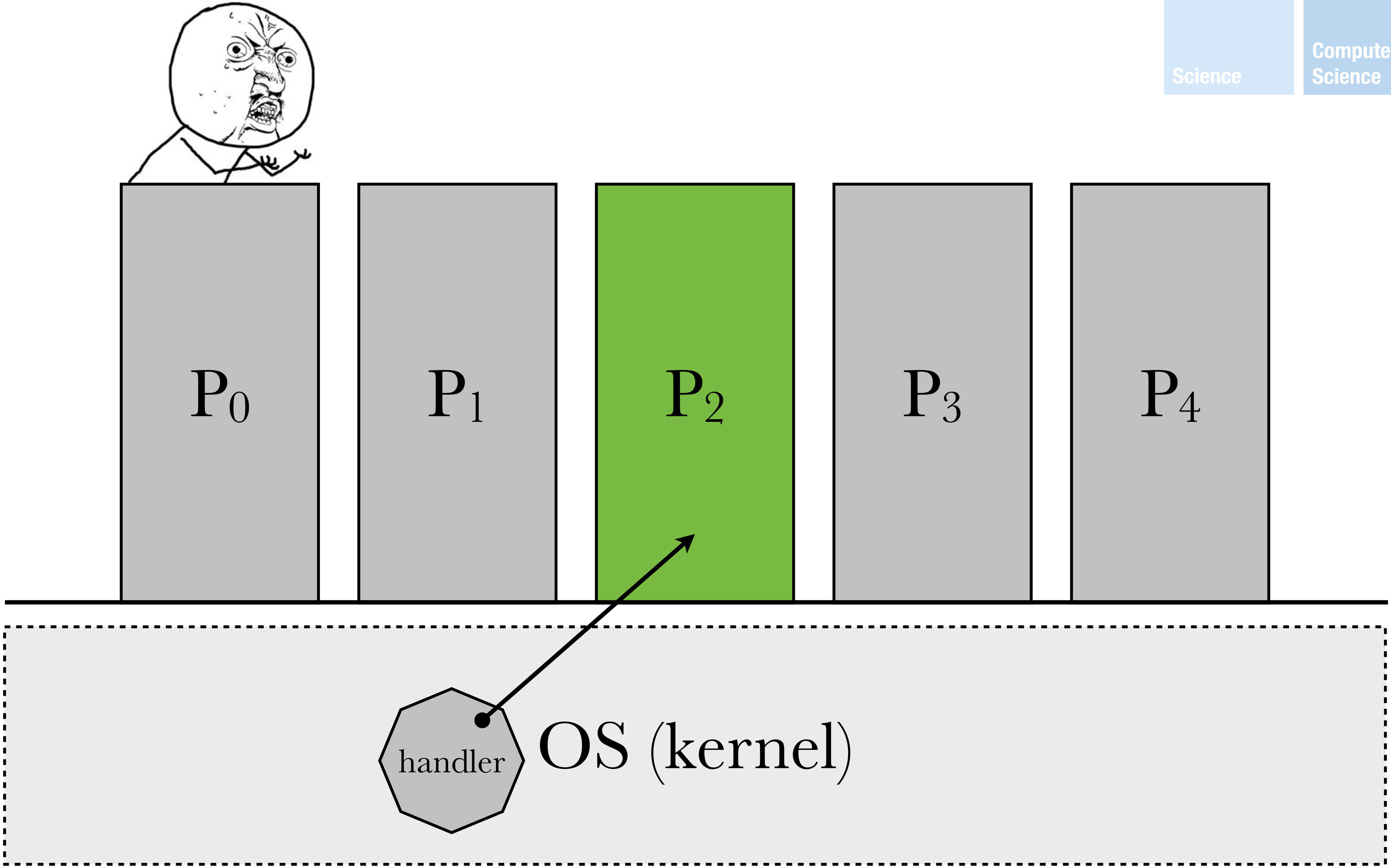

P_0
P_1
P_2
P_3
P_4
trap
OS (kernel)


P_0
P_1
P_2
P_3
P_4
trap
handler
OS (kernel)


P_0
P_1
P_2
P_3
P_4
handler
OS (kernel)


P_0
P_1
P_2
P_3
P_4
OS (kernel)


P_0
P_1
P_2
P_3
P_4
trap
handler
OS (kernel)


P_0
P_1
P_2
P_3
P_4
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 and as efficiently as possible