3150 - Operating Systems

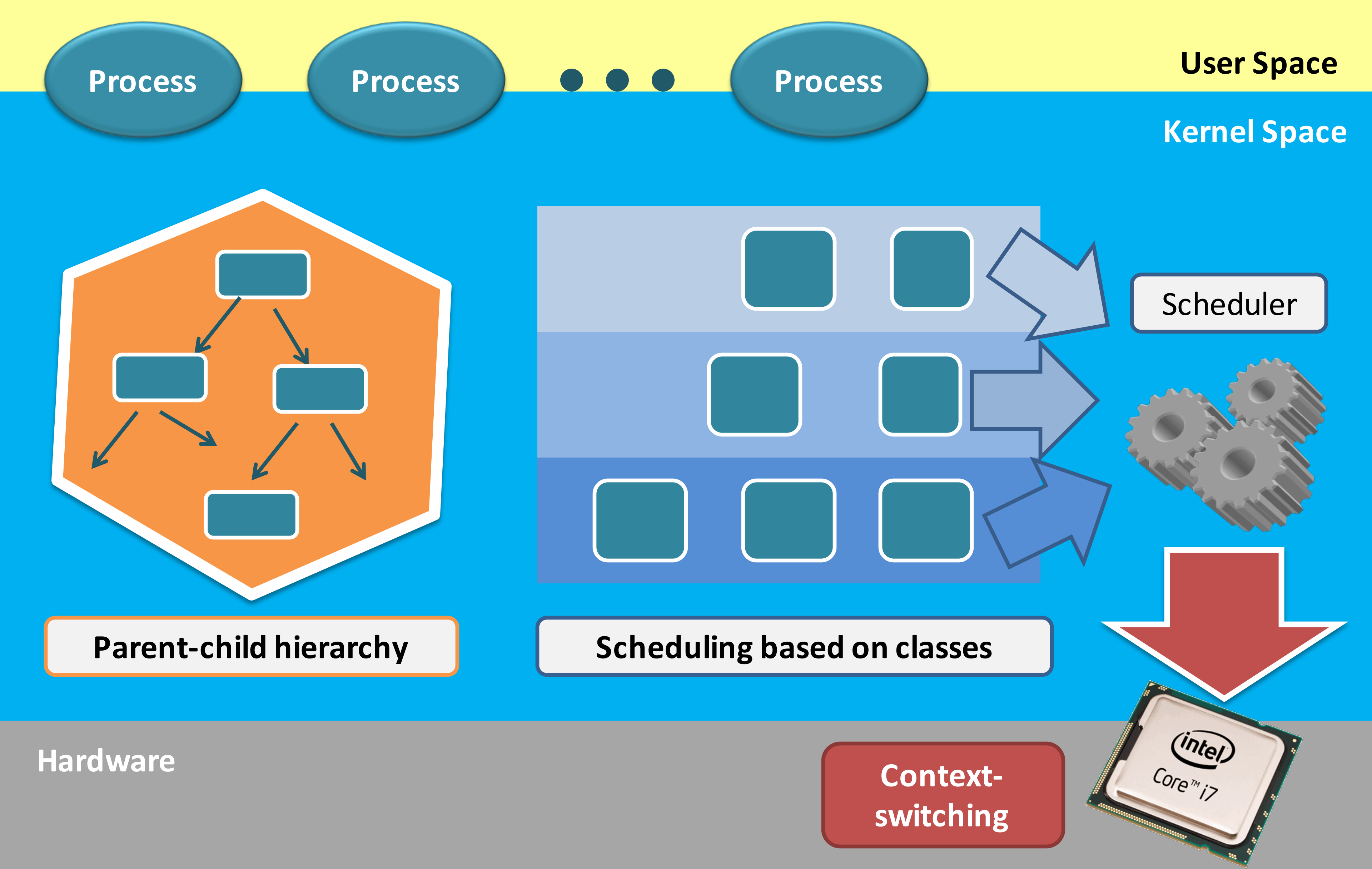
Dr. WONG Tsz Yeung

P. 55-58, 61-75, 71-79 are animations.
Don't print them.

Chapter 2, part 4 – Process Organization and Scheduling

-things becoming complicated when there are more than one player on the same field…

Outline


Process
Process
Process
User Space
Kernel Space
Scheduler
Parent-child hierarchy
Scheduling based on classes
Hardware
Context-switching
intel
Core™ i7

Topics

- The first process and process organization in Linux;

Booting up a computer

EXTRA

- At a high-level view…

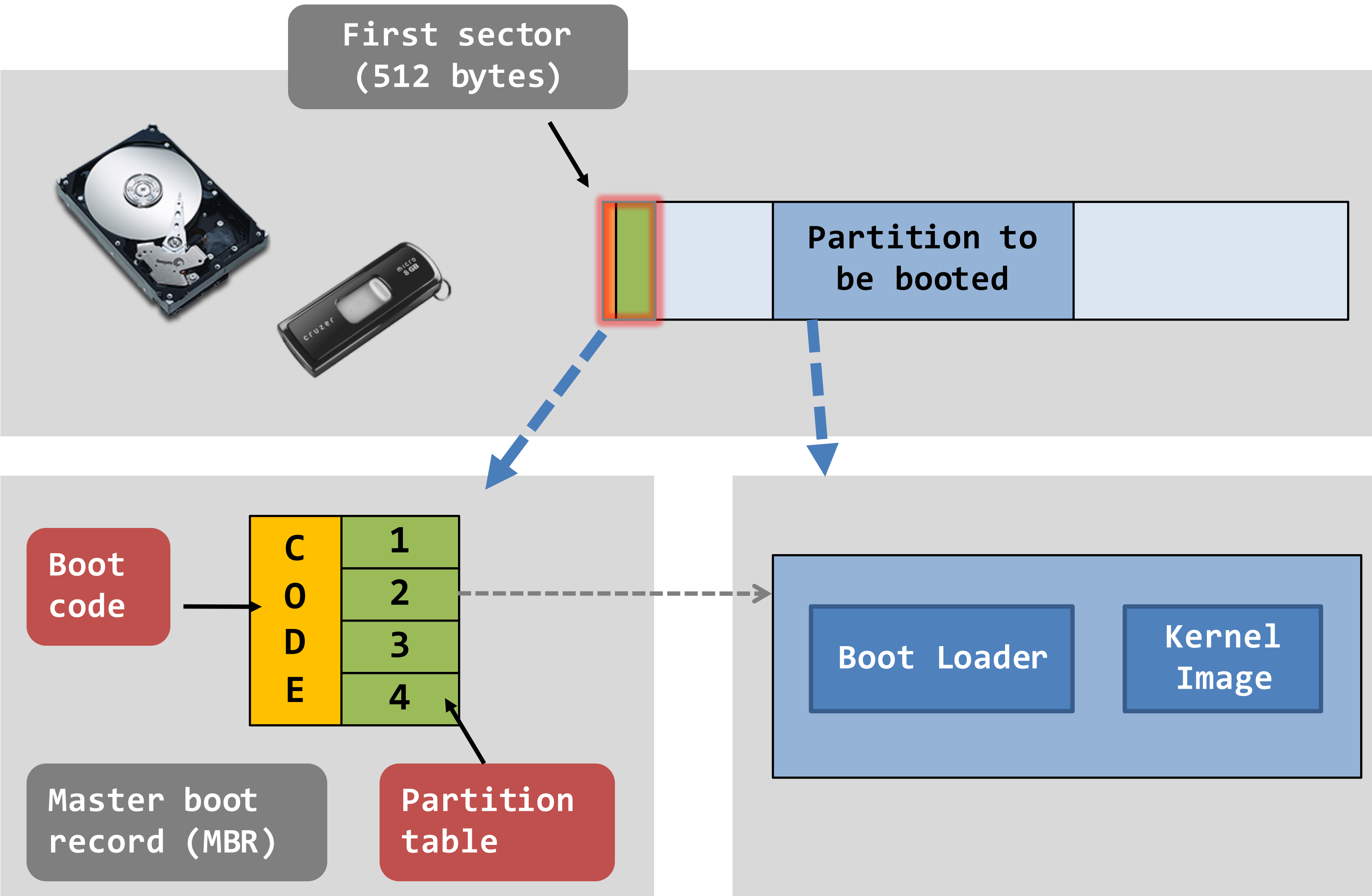
Step 3. The boot code decides which OS to be booted. It also knows how each OS can be booted.

Step 1. BIOS locates the device that the computer will boot from.

Step 2. The booting device, usually a storage device, contains **boot code**. Then, the boot code is executed.

Booting device

EXTRA



MBR & Boot loader

EXTRA

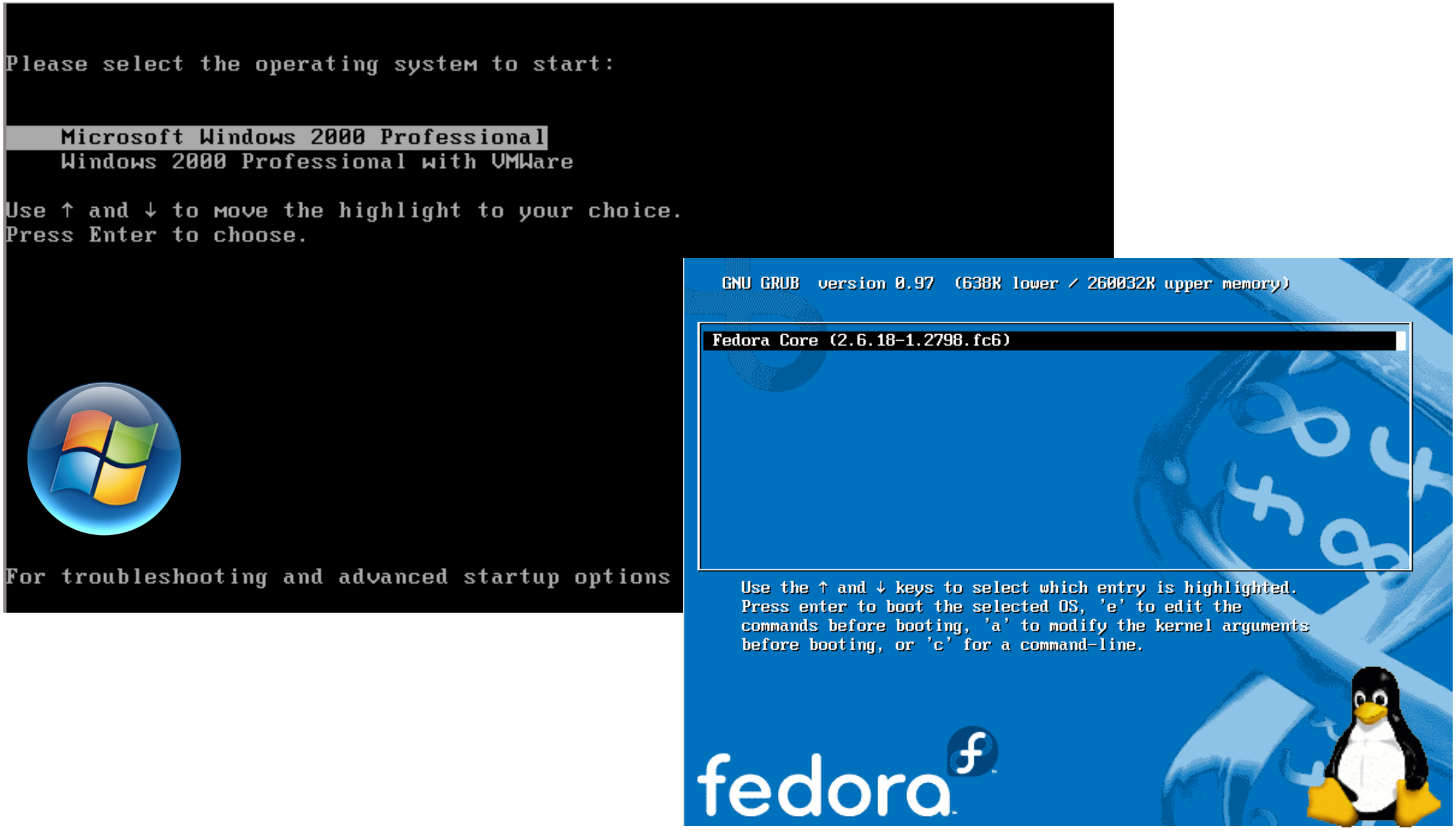
- **Master boot record** (MBR) stores two things:
  - Boot code; and
  - Partition table.

- The job of the boot program is to execute a boot loader in the *bootable partition*.
  - Linux: **`GRUB - GRand Unified Bootloader`**;
  - Windows: **`C:\boot.ini`**.

- The job of the boot loader is to locate and boot the **kernel image**.

MBR & Boot loader

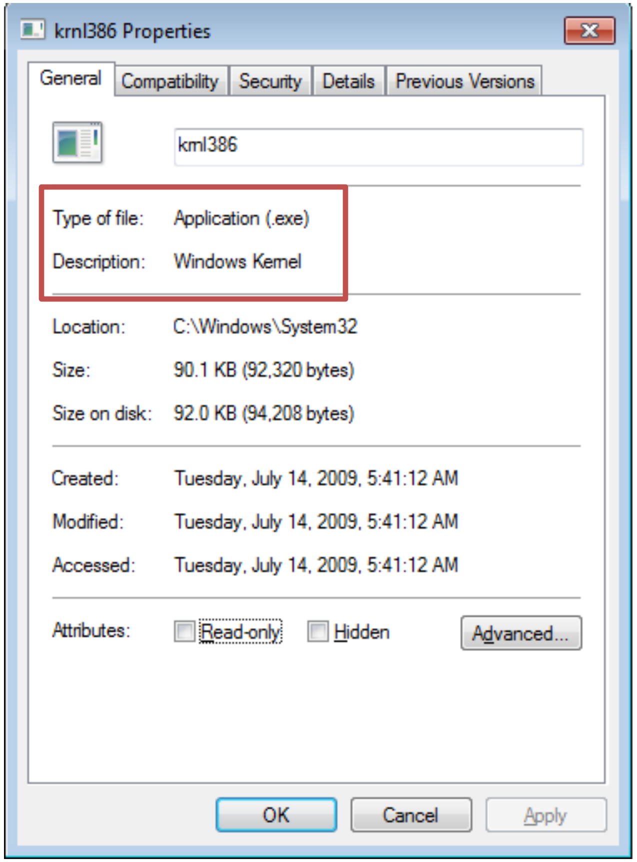
EXTRA



MBR & Boot loader

EXTRA

- Kernel is a piece of software, and is *just a file*! We call it the kernel image.
  - Linux: **/boot/vmlinuz***;
  - Win XP: **C:\windows\system32\ntoskrnl.exe**;
  - Win 7: **C:\windows\system32\krnl386.exe**.



- Kernel initialization
  - When the kernel image is found, the kernel starts.
  - It initializes all kernel subsystems.
    - E.g., initialize memory layout, initialize drivers, etc.

The first process

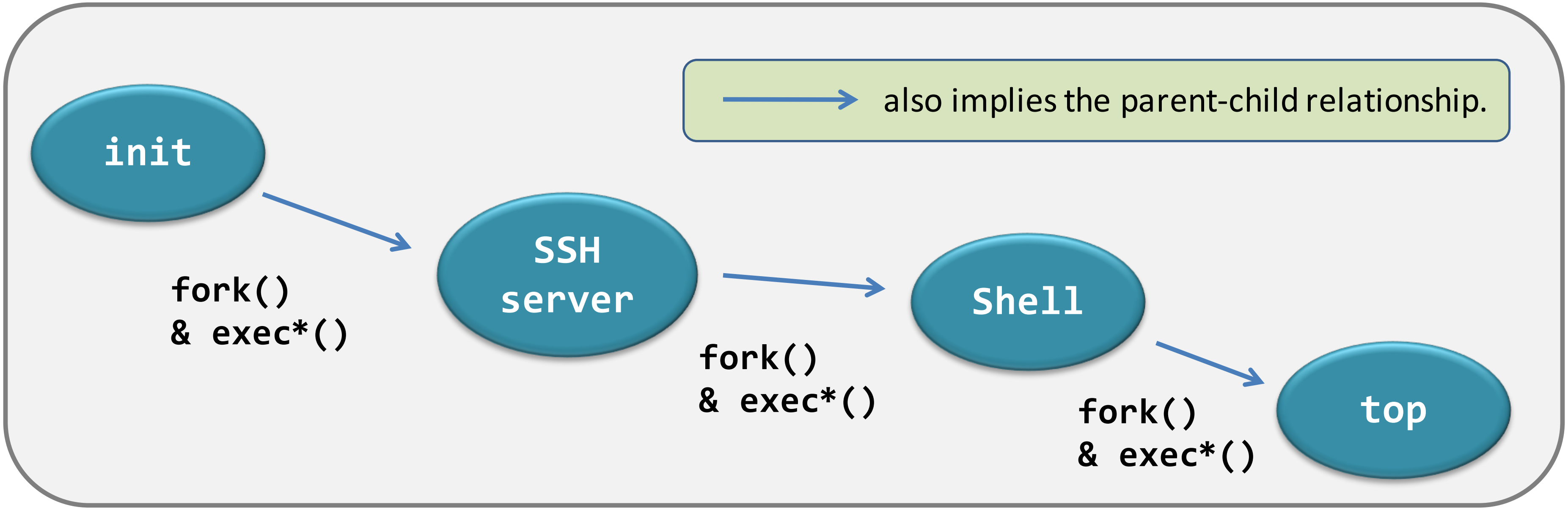
- We now focus on the process-related events.
  - The kernel, while it is booting up, creates the first process – **`init`**.

- The "**`init`**" process:
  - has **PID = 1**, and
  - is running the program code "**`/sbin/init`**".

- Its first task is to **create more processes**...
  - Using **`fork()`** and **`exec*()`**.

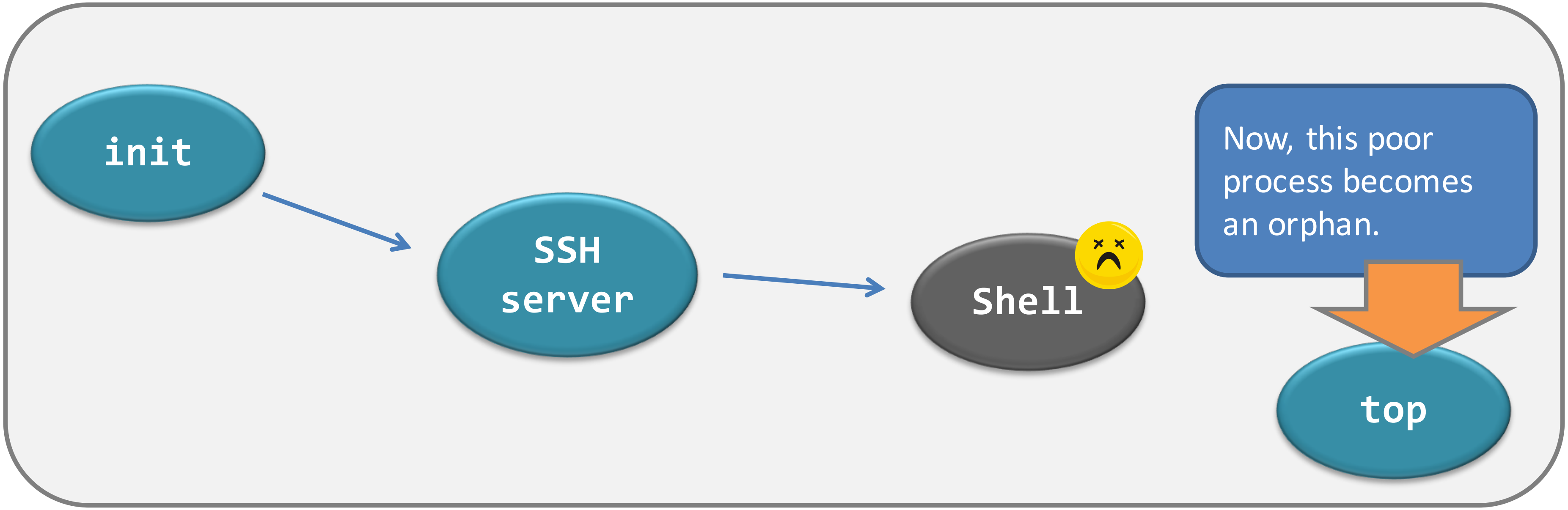
Process blossoming

- You can view the tree with the command:
  - "**pstree**"; or
  - "**pstree -A**" for ASCII-character-only display.



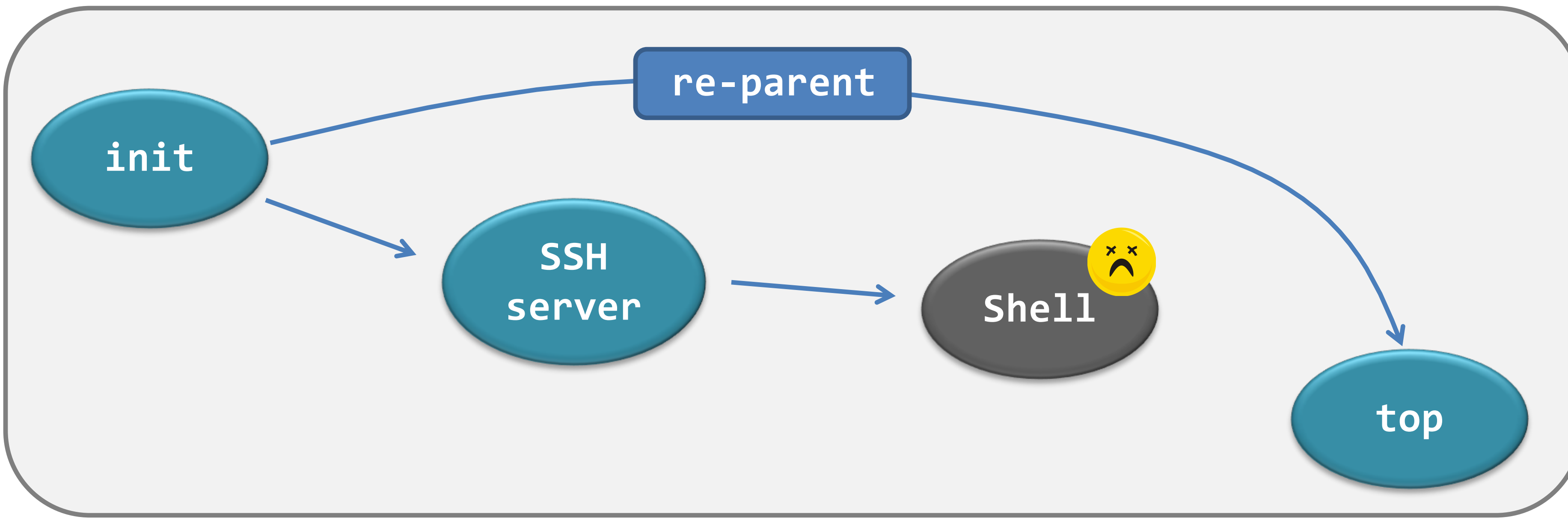
Process blossoming…with orphans?

- However, termination can happen, at any time and in any place…
  - This is no good because an orphan turns the hierarchy from a **tree** into a **forest**!
  - Plus, no one would know the termination of the orphan.



Process blossoming…with re-parent!

- In Linux, we have the **re-parent operation**.
  - The "**init**" process will become the step-mother of all orphans.
    - Well…Windows maintains a *forest-like process hierarchy*.



Re-parent example

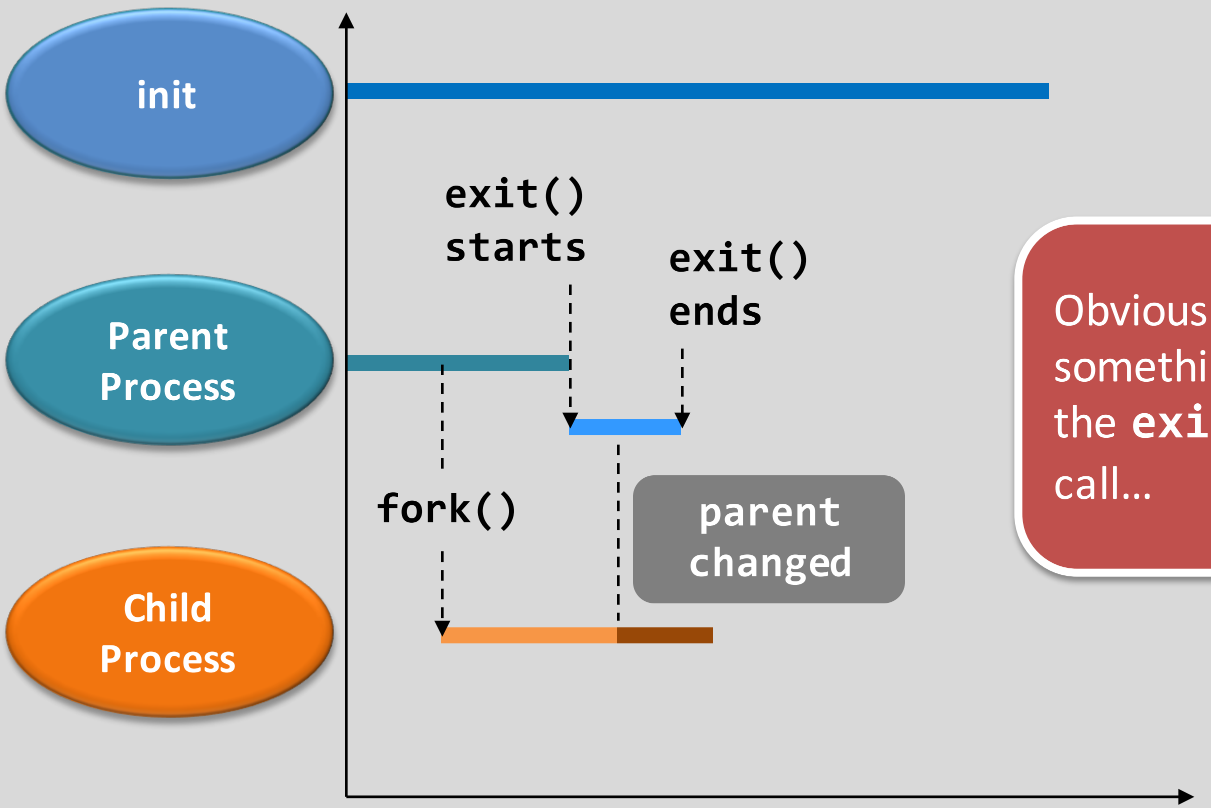
```c
int main(void) {
    int i;
    if(fork() == 0) {
        for(i = 0; i < 5; i++) {
            printf("(%d) parent's PID = %d\n",
                   getpid(), getppid() );
            sleep(1);
        }
    }
    else
        sleep(1);
    printf("(%d) bye.\n", getpid());
}
```

```
$ ./reparent_example
(1235) parent's PID = 1234
(1235) parent's PID = 1234
(1234) bye.
$ (1235) parent’s PID = 1
(1235) parent’s PID = 1
(1235) parent’s PID = 1
(1235) bye.
$ _
```

getppid() is the system call that returns the parent’s PID of the calling process.

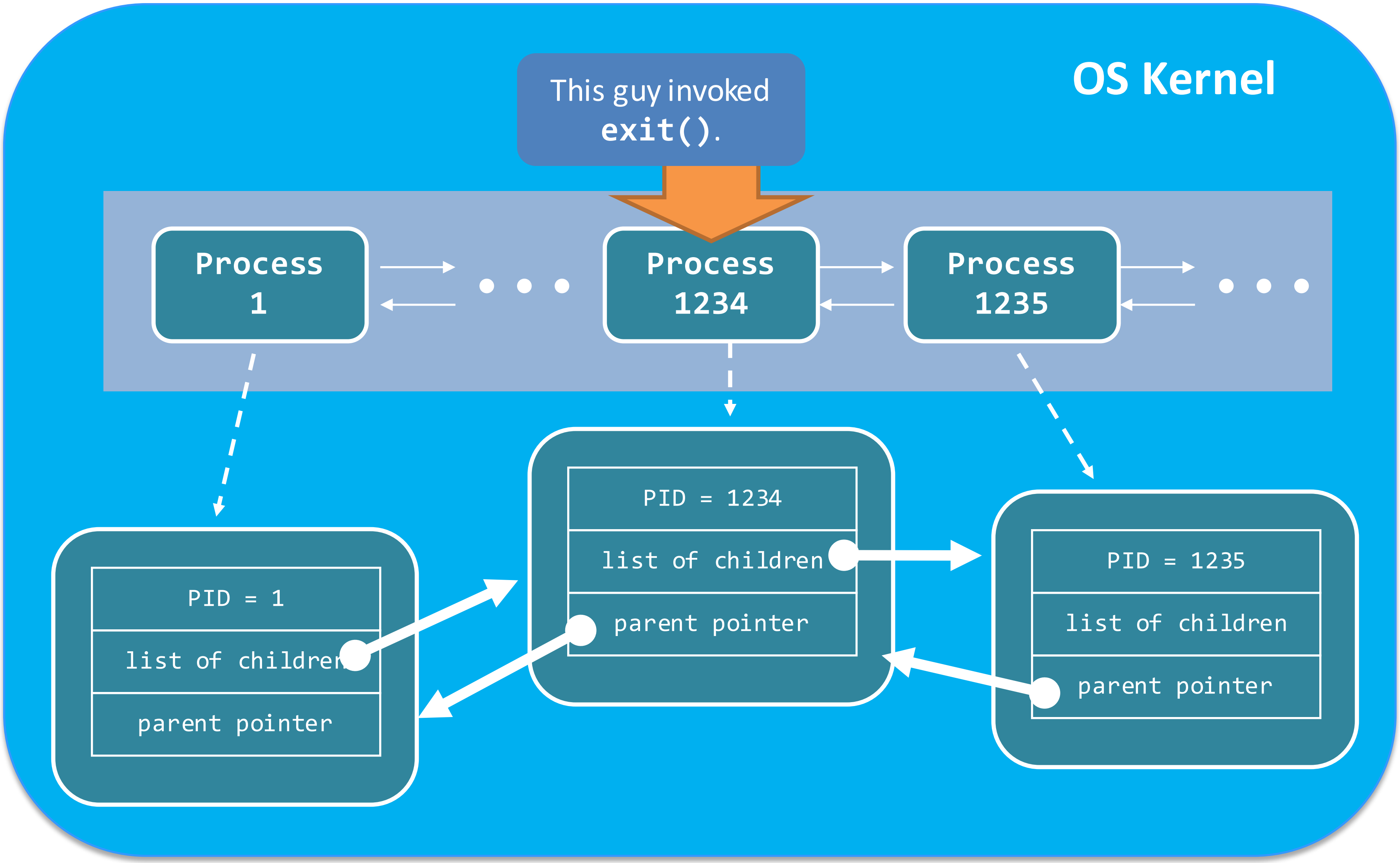
```
[examples@3150] cat reparent_example.c
```

What had happened during re-parent?

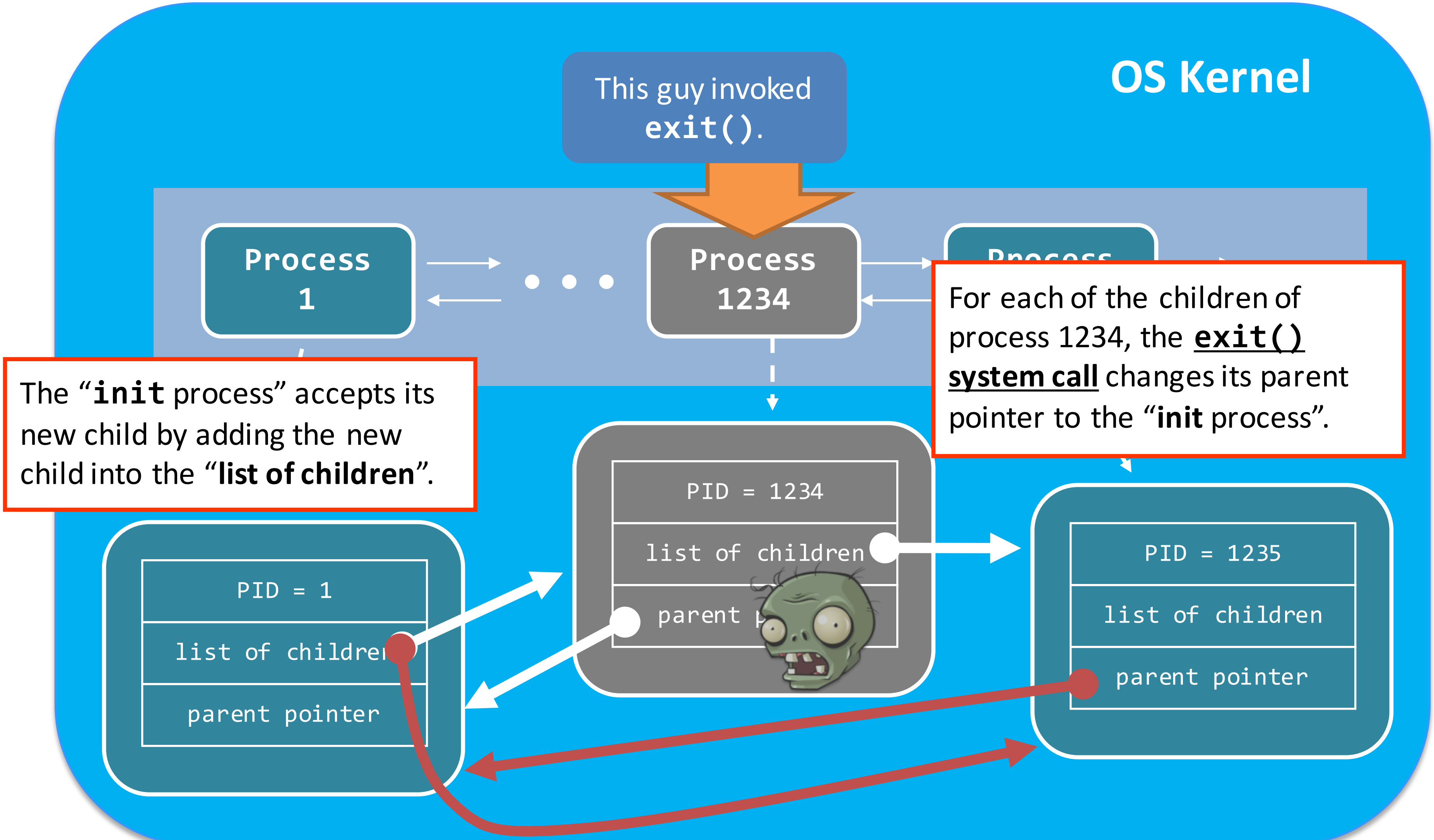


Obviously, there is something to do with the **`exit()`** system call…

What had happened during re-parent?

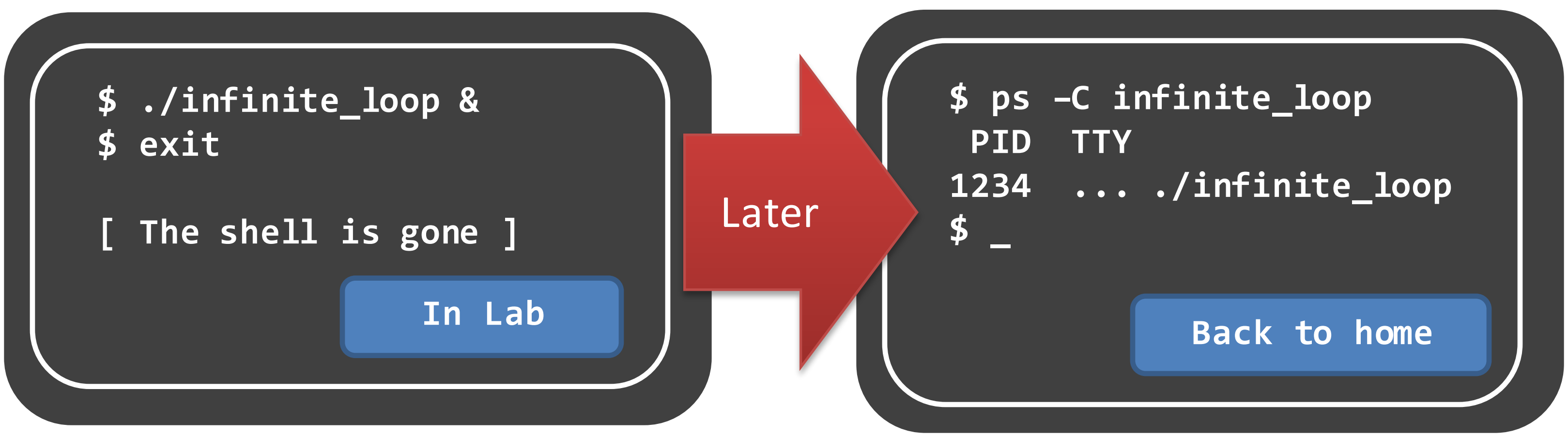


What had happened during re-parent?



A short summary

- **Observation 1**
  - The re-parent operation allows processes running **without the need of a parent terminal**.
  - Thus, the **background jobs** survive even though the hosting terminal is closed.



A short summary

- **Observation 1**
  - The re-parent operation allows processes running **without the need of a parent terminal**.
  - Thus, the **background jobs** survive even though the hosting terminal is closed.

- **Observation 2**
  - The processes in Linux is always organized as a tree.
  - Because of the re-parent operation, there is always **only one process tree**.

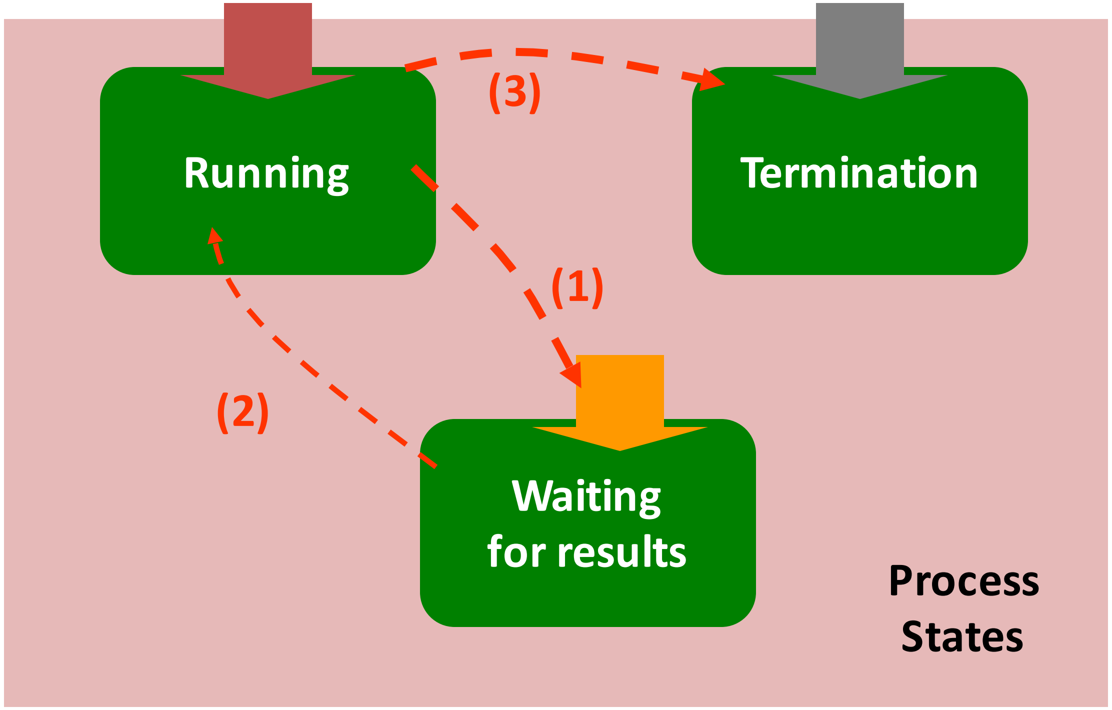
Topics

- The first process and process organization in Linux;
- **Process lifecycle;**

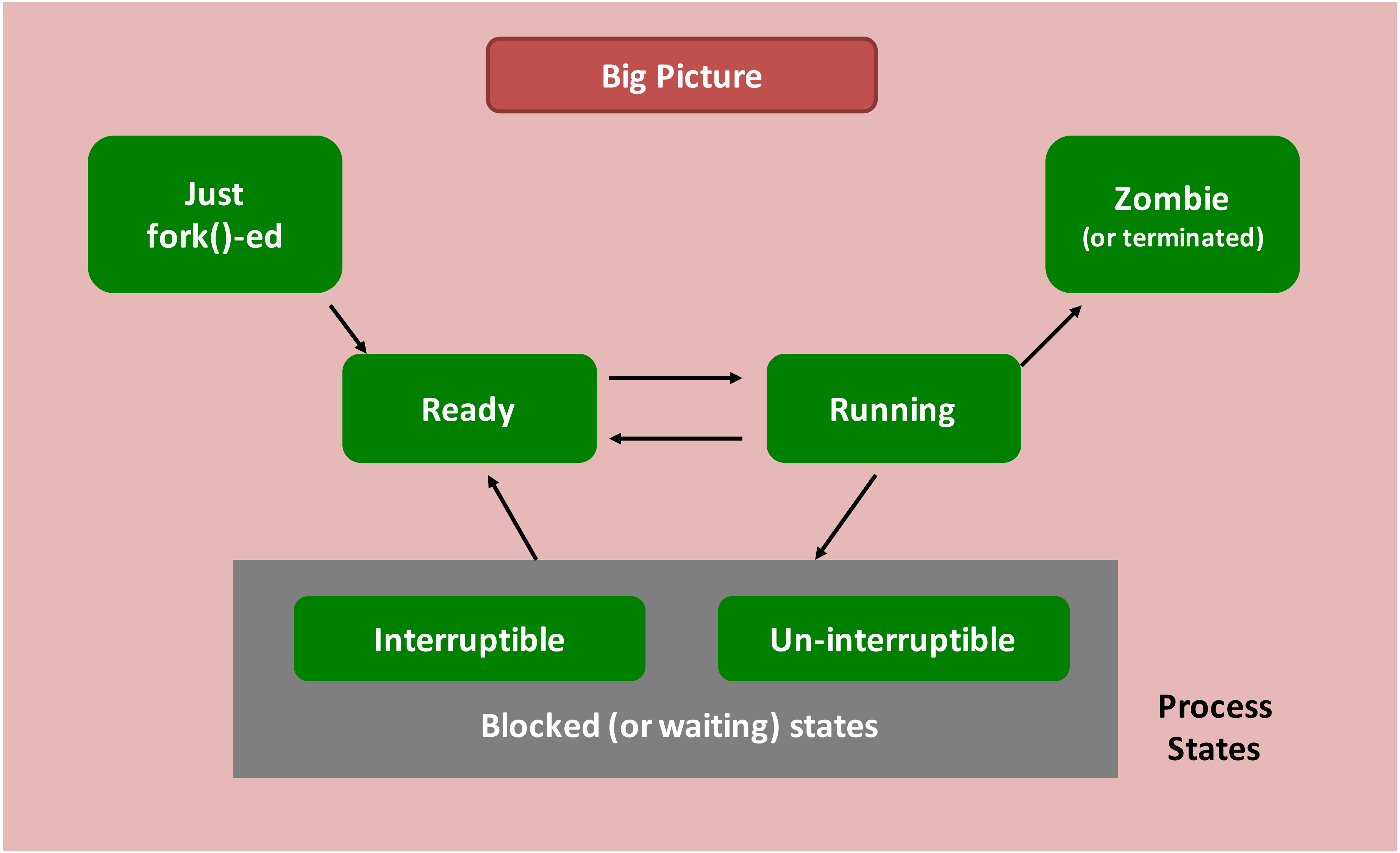
Programmer's point of view…

- This is how a fresh programmer looks at a process' life cycle.

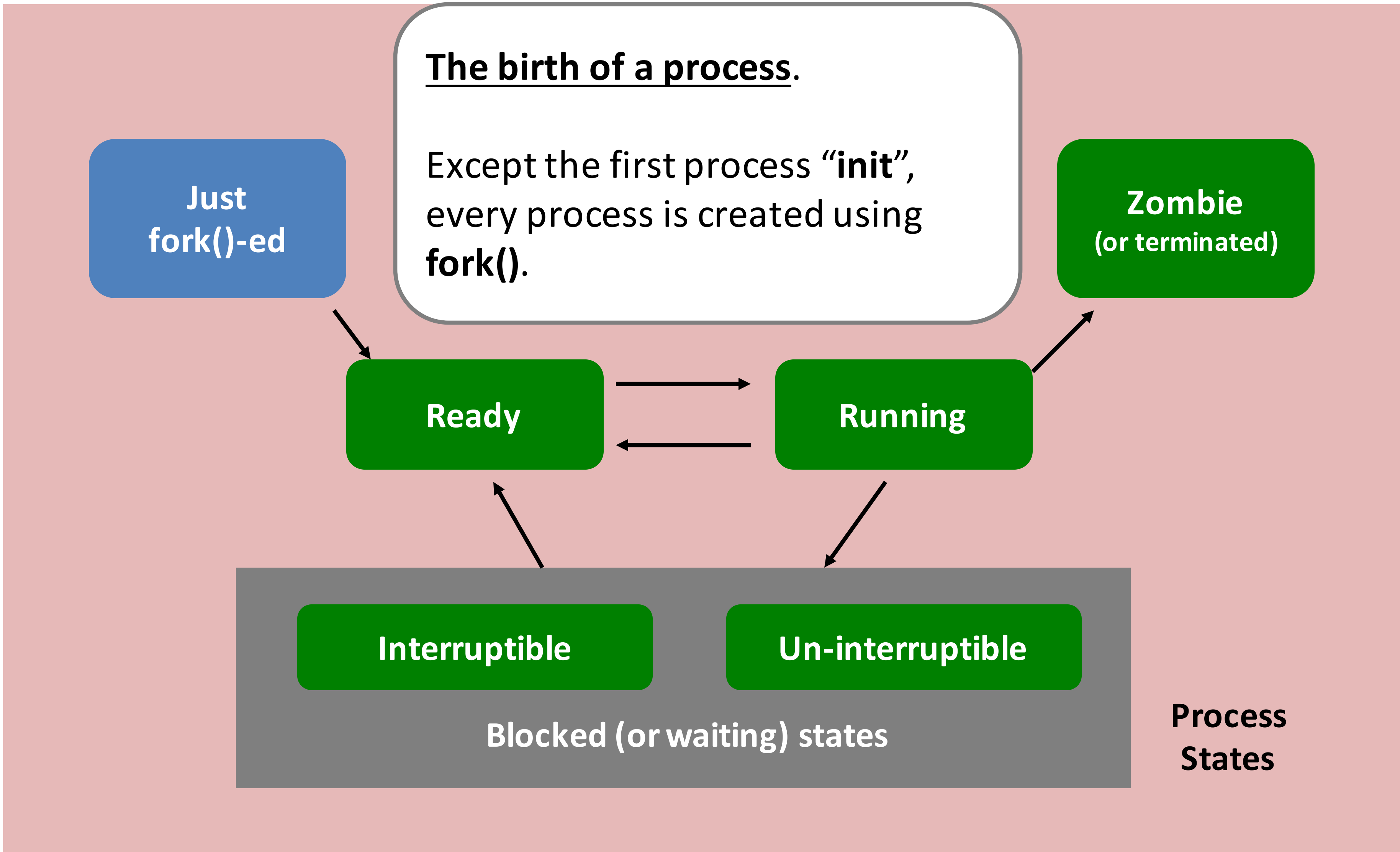
```
int main(void) {
    int x = 1;
    getchar();
    return x;
}
```



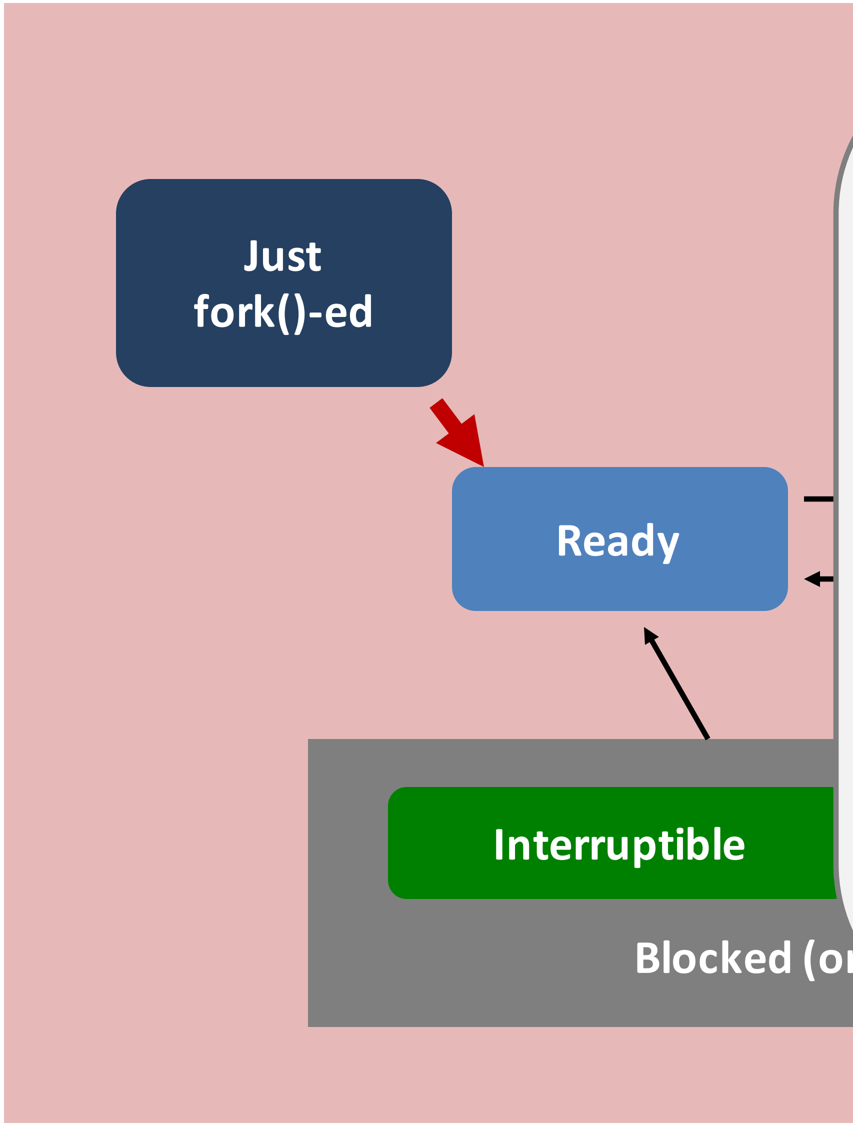
Kernel’s point of view…



Kernel’s point of view…



Kernel’s point of view…



<u>The process is ready</u>.
It means it is ***ready to run* but is not running**.

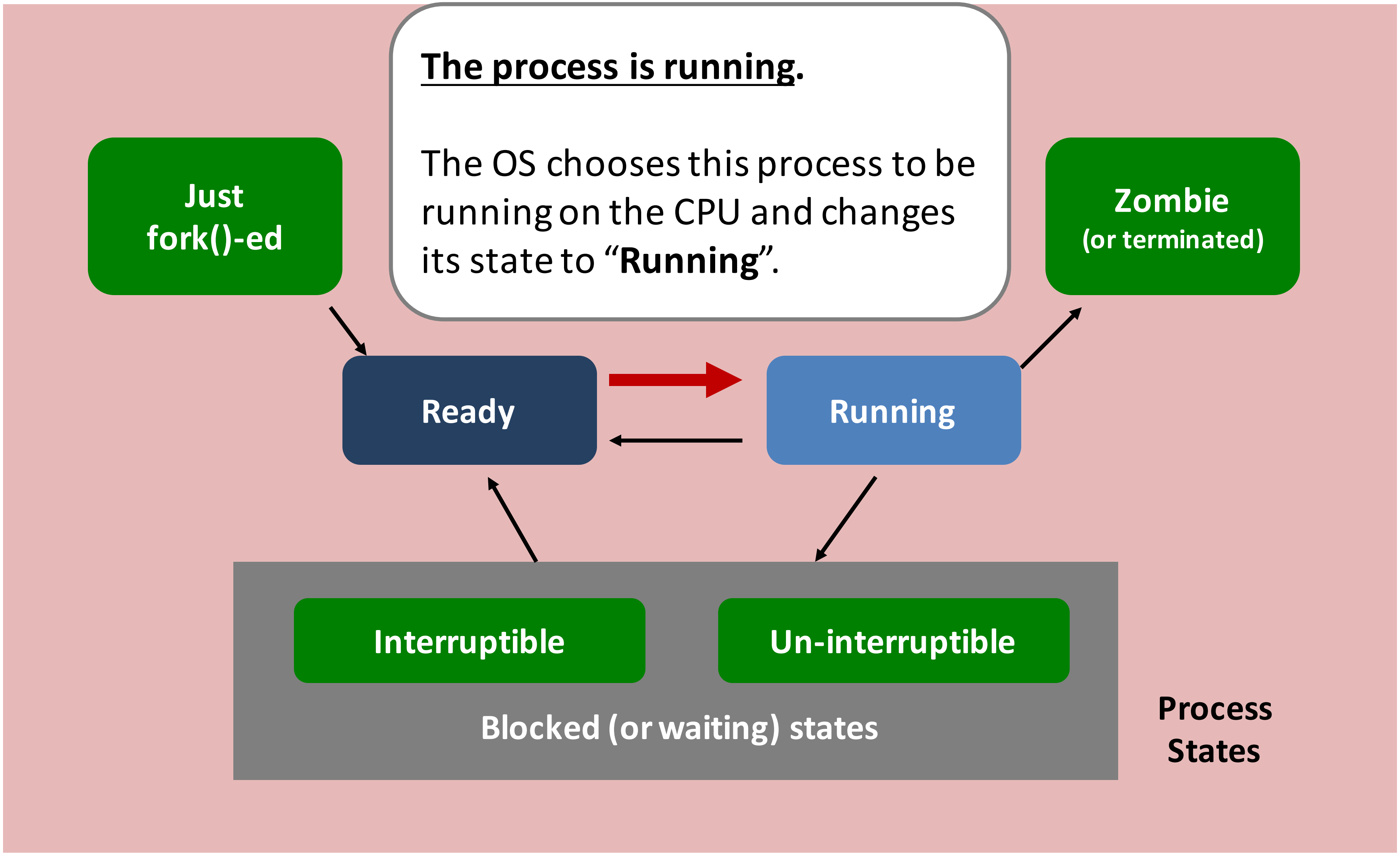
A process may become “**<u>ready</u>**” after…

- it is just created by **`fork()`**;
- it has been running on the CPU for some time and the OS chooses another process to run;
- returning from blocked states.

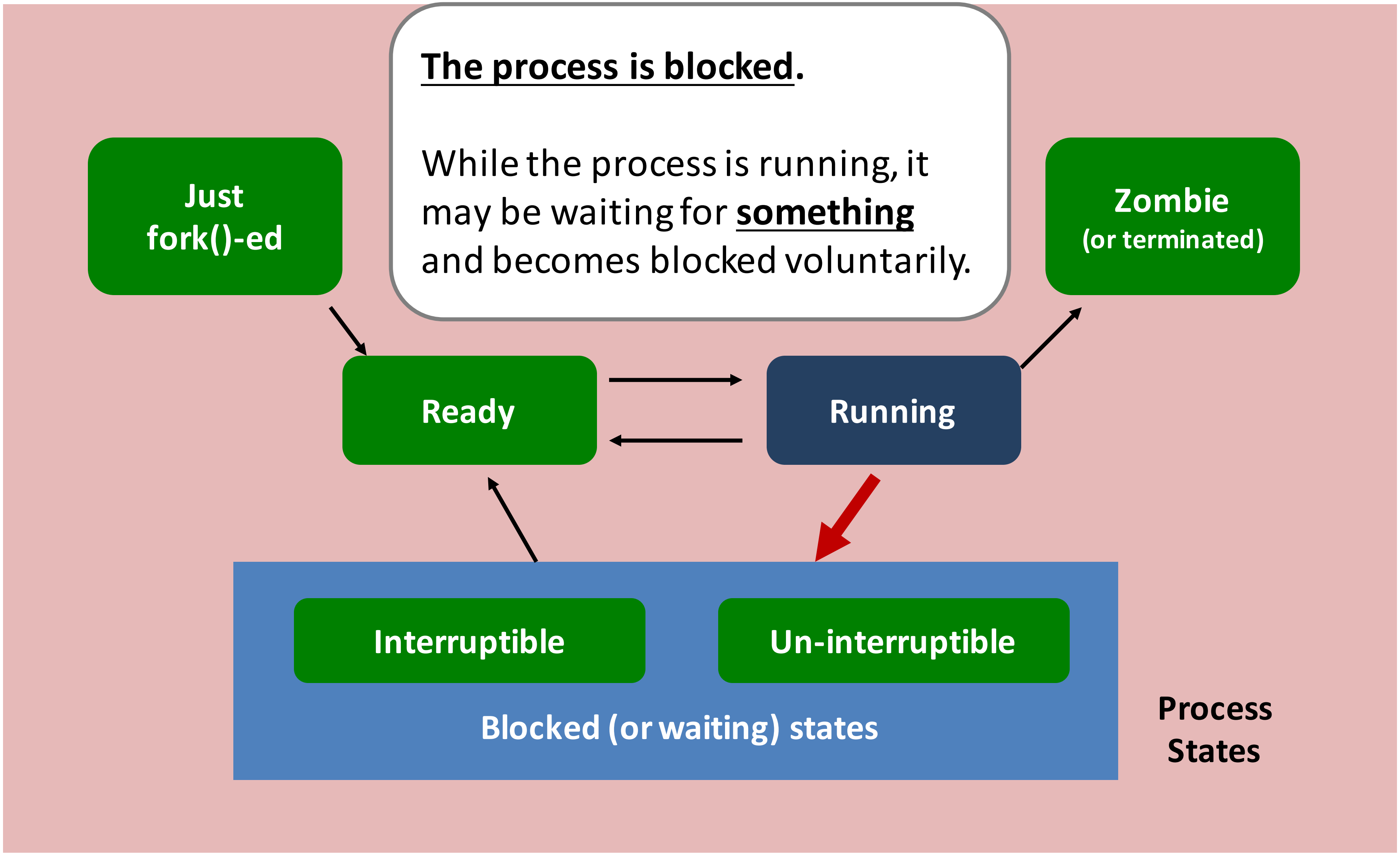
Kernel's point of view…

<u>The process is running</u>.

The OS chooses this process to be running on the CPU and changes its state to "**Running**".



Kernel’s point of view…

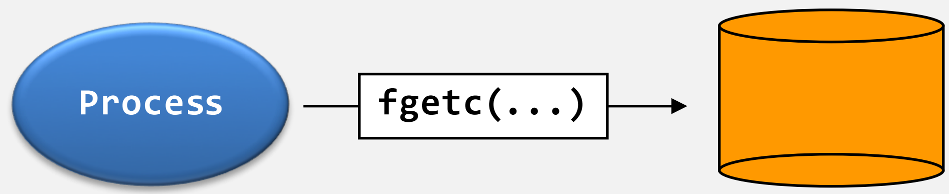


Kernel's point of view…

Example. **Reading a file**.

Sometimes, the process has to wait for the response from the device and, therefore, it is **blocked**.

Nevertheless, this blocking state is **interruptible**. E.g., "**Ctrl + C**" can gets the process out of the waiting state (but goes to termination state instead).



Interruptible

Un-interruptible

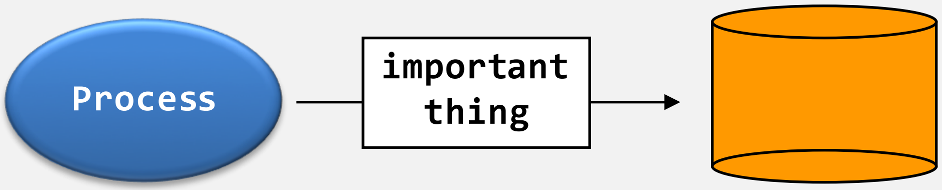
Blocked (or waiting) states

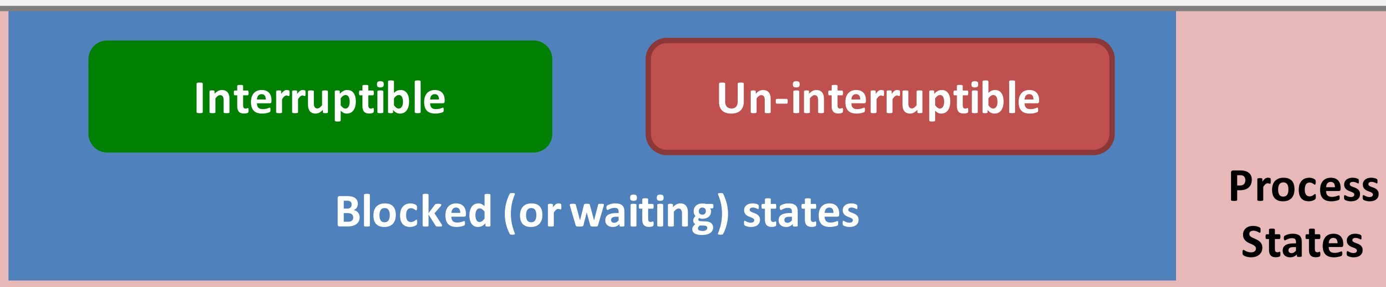
Process States

Kernel’s point of view…

Sometimes, a process needs to wait for a resource but it doesn’t want to be disturbed while it is waiting. In other words, **the process wants that resource very much**. Then, the process status is set to the **un-interruptible** status.

You can’t find too many examples up to our current knowledge, unless … you dig through the *kernel codes*!

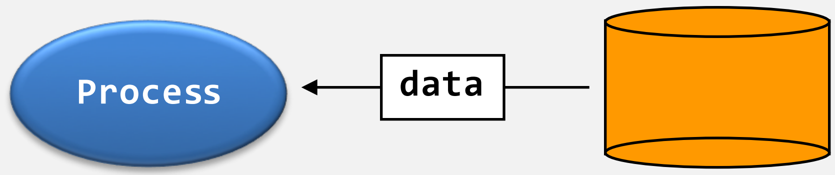


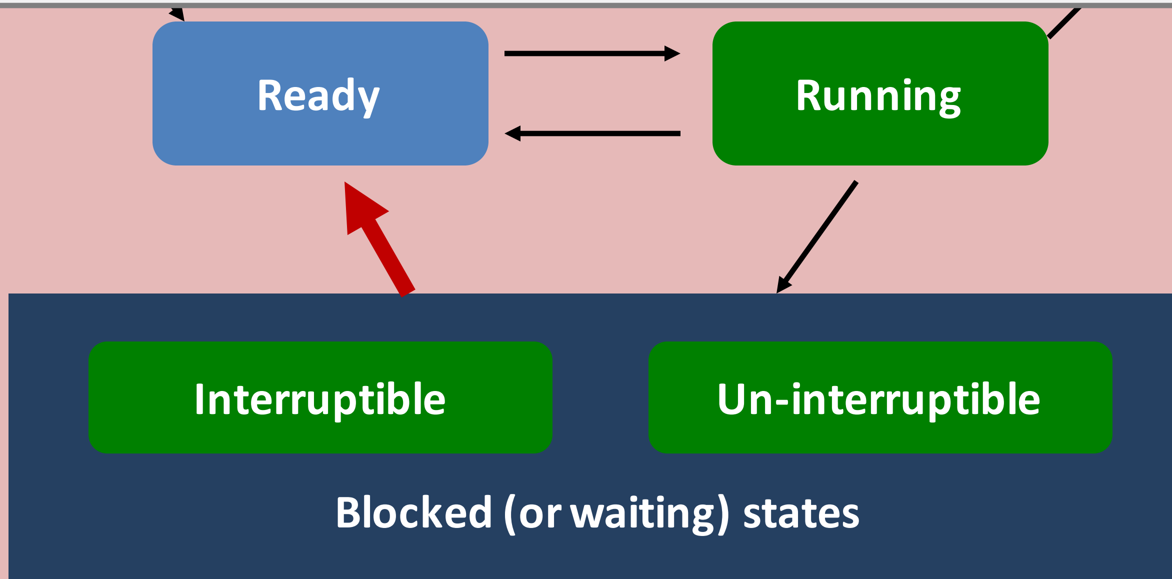


Kernel’s point of view…

<u>Return back to ready.</u>

When response arrives, the status of the process changes back to **Ready**. from any one of the blocked states.





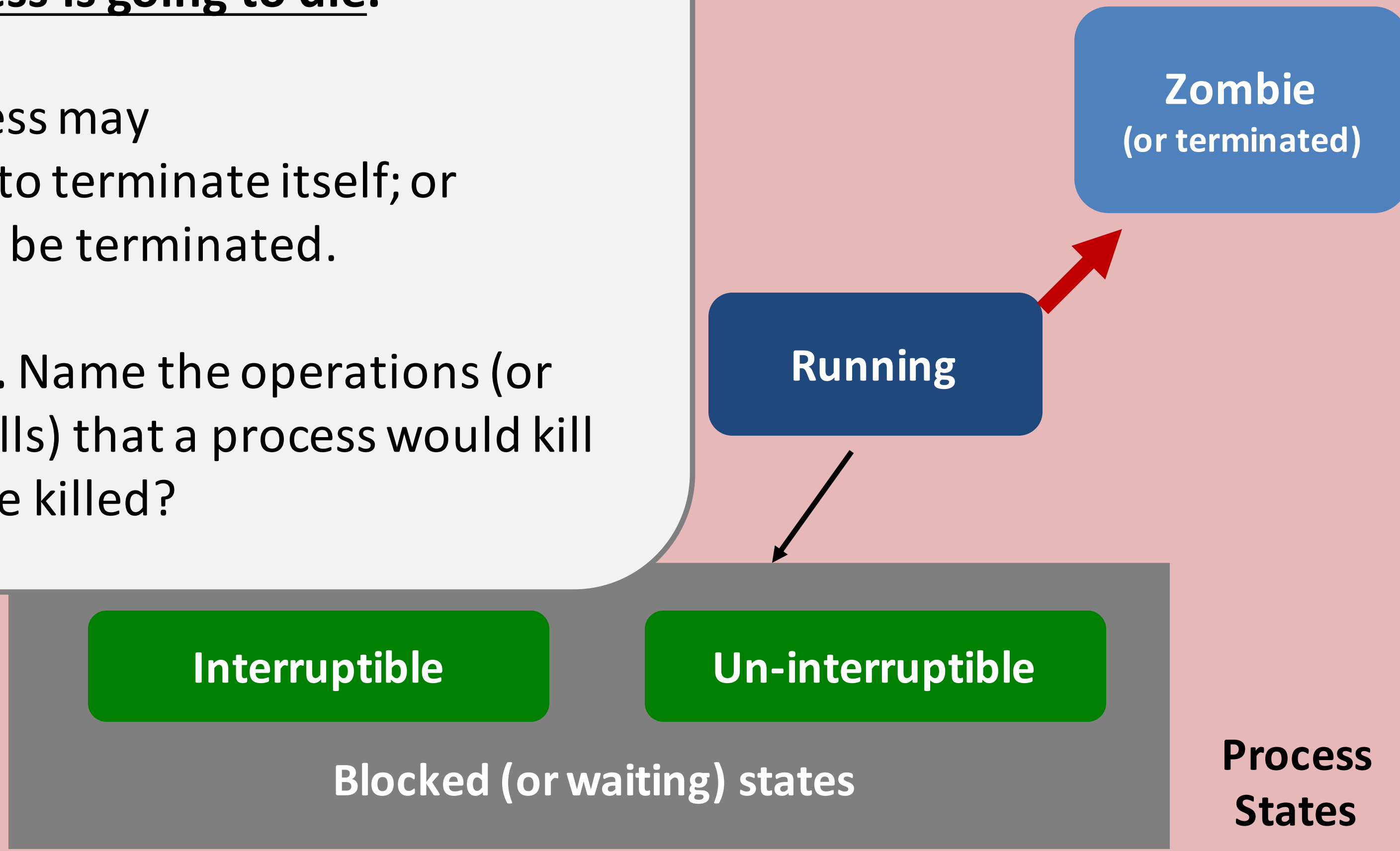
Process States

Kernel’s point of view…

<u>The process is going to die</u>.

The process may
- choose to terminate itself; or
- force to be terminated.

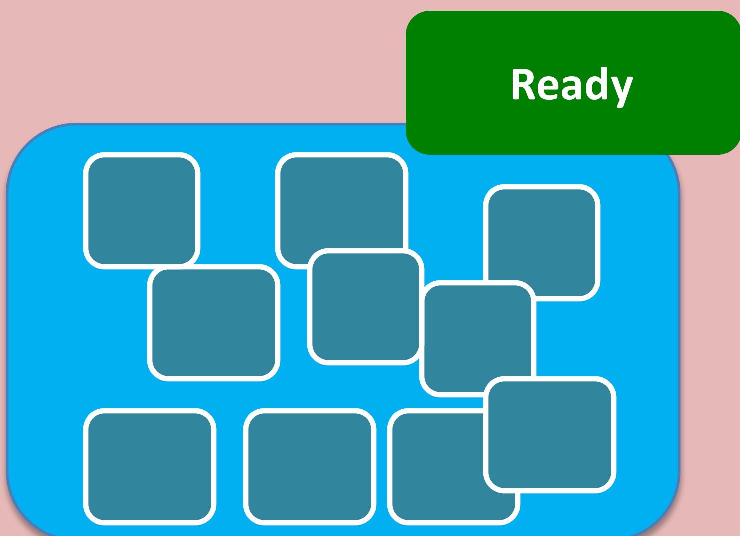
<u>Question</u>. Name the operations (or system calls) that a process would kill itself or be killed?



Process States

Kernel’s point of view…

So, what is process scheduling?



Scheduling is about how to make **all the ready processes** become “**Running**”.

Plus, there are other goals to be satisfied.

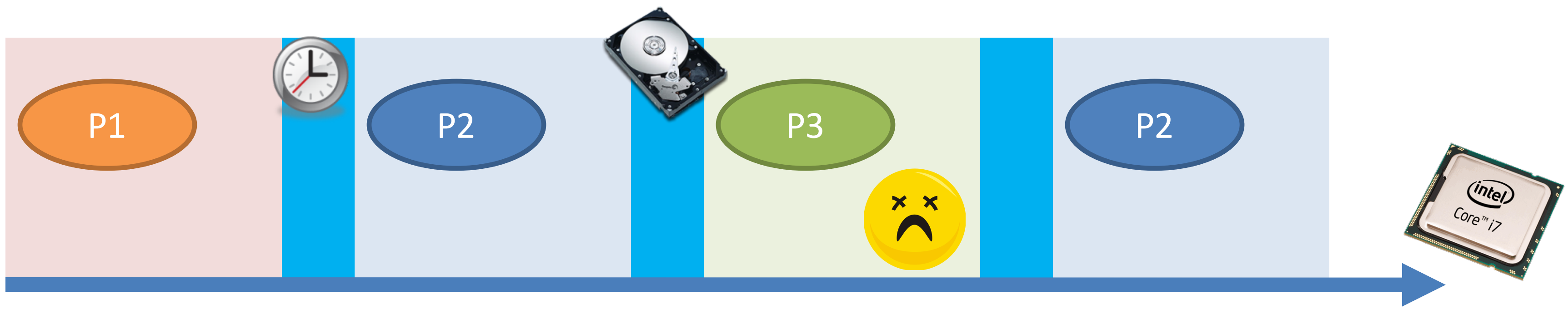
Topics

- The first process and process organization in Linux;
- Process lifecycle;
- **Context switching;**

What is context switching?

VERY IMPORTANT

- Before we can jump into the process scheduling topic, we have to understand what "**context switching**" is.



Scheduling is the procedure that decides which process to run next.

Context switching is the actual switching procedure, from one process to another.

 Timer interrupt.

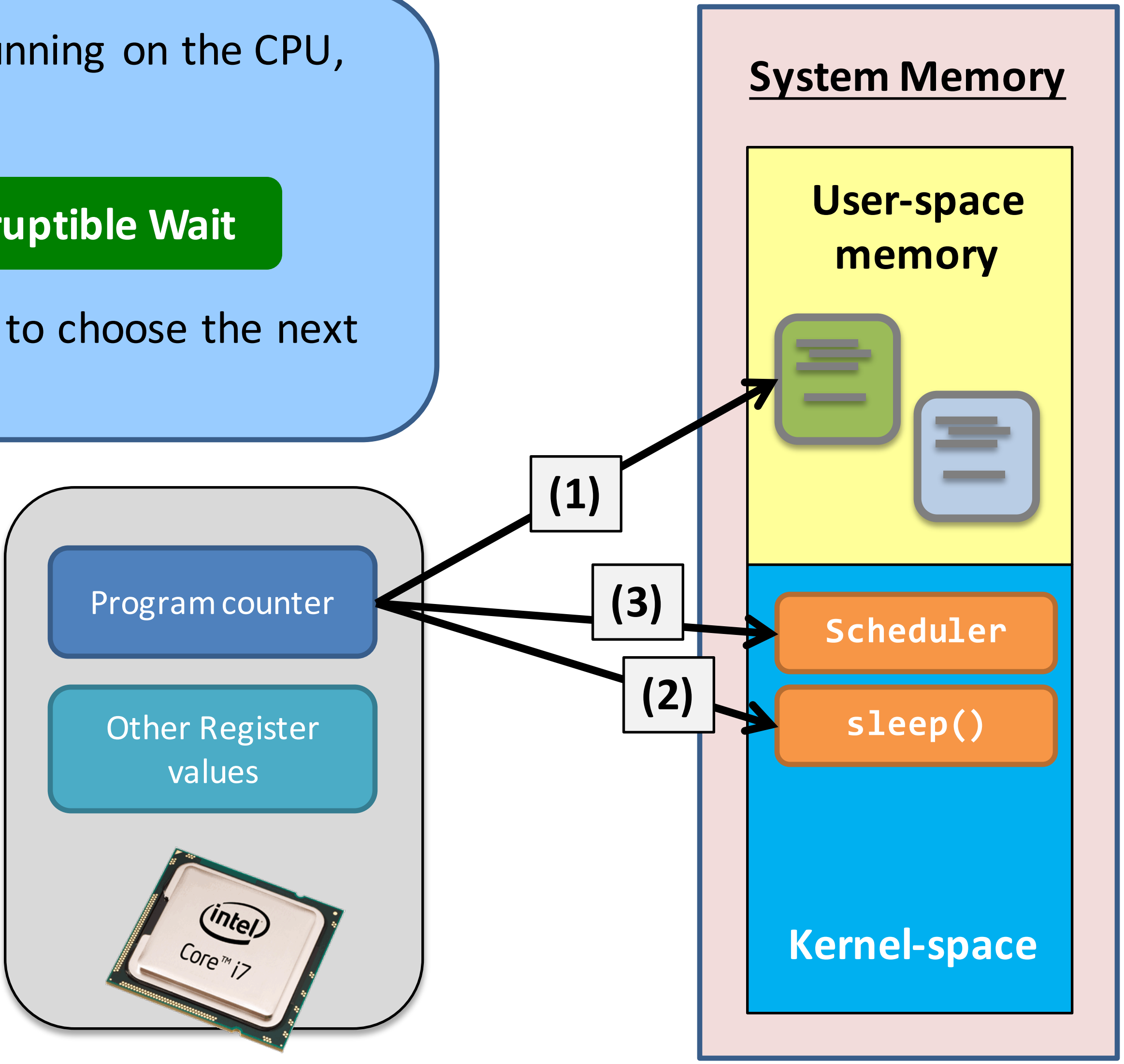
 Hardware interrupt.

Switching from one process to another.

Suppose this process gives up running on the CPU, e.g., calling **`sleep()`**. Then:

Running ➔ **Interruptible Wait**

Now, it is time for the scheduler to choose the next process to run.



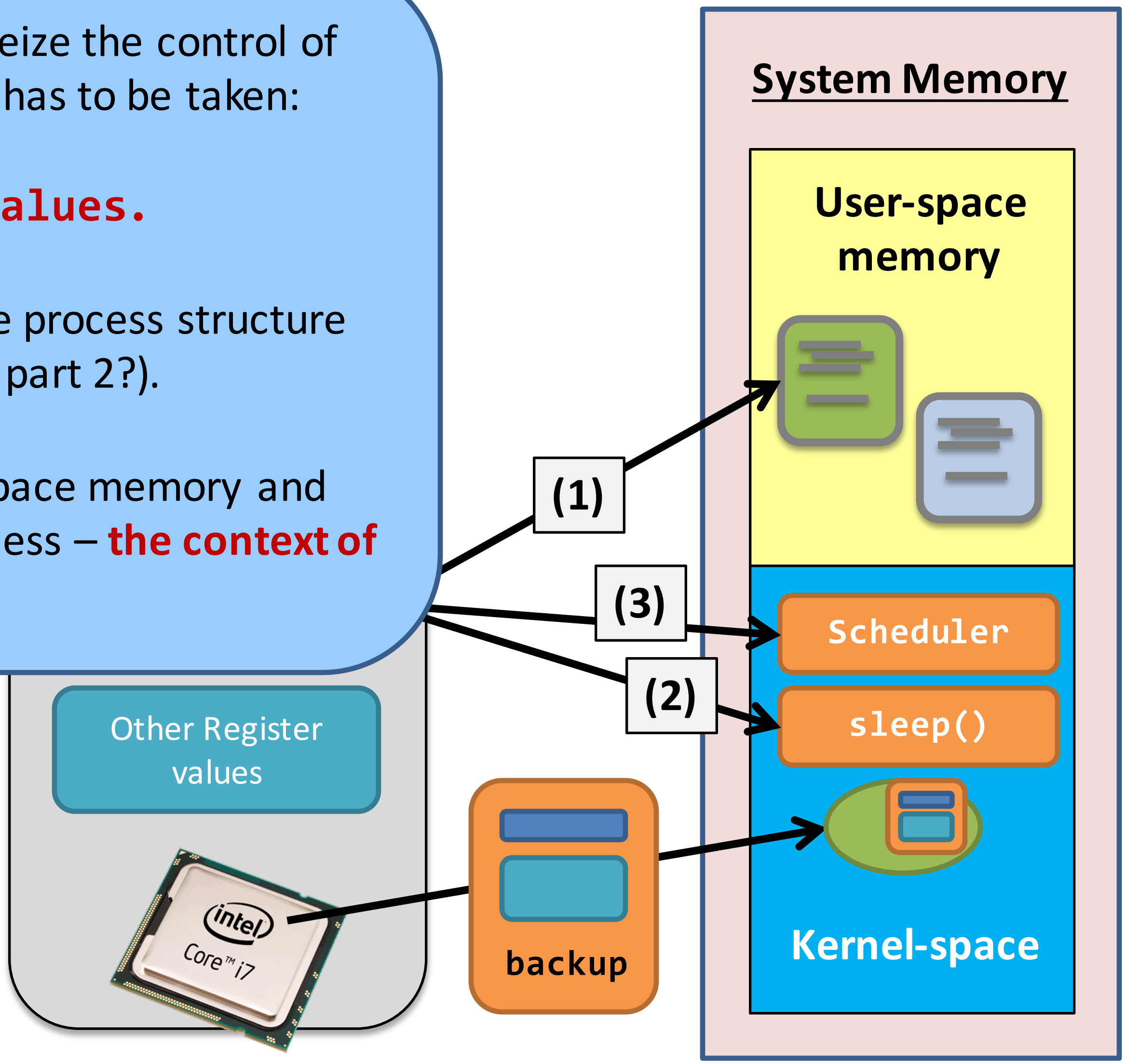
Switching from one process to another.

But, before the scheduler can seize the control of the CPU, a very important step has to be taken:

`Backup all registers' values.`

The backup will be stored in the process structure (remember, "**task structure**" in part 2?).

We call the union of the user-space memory and the registers' values of the process – **the context of a process**.



Analogy

When you want to switch to another game on your game console, ***you must save all your progress before switching***.

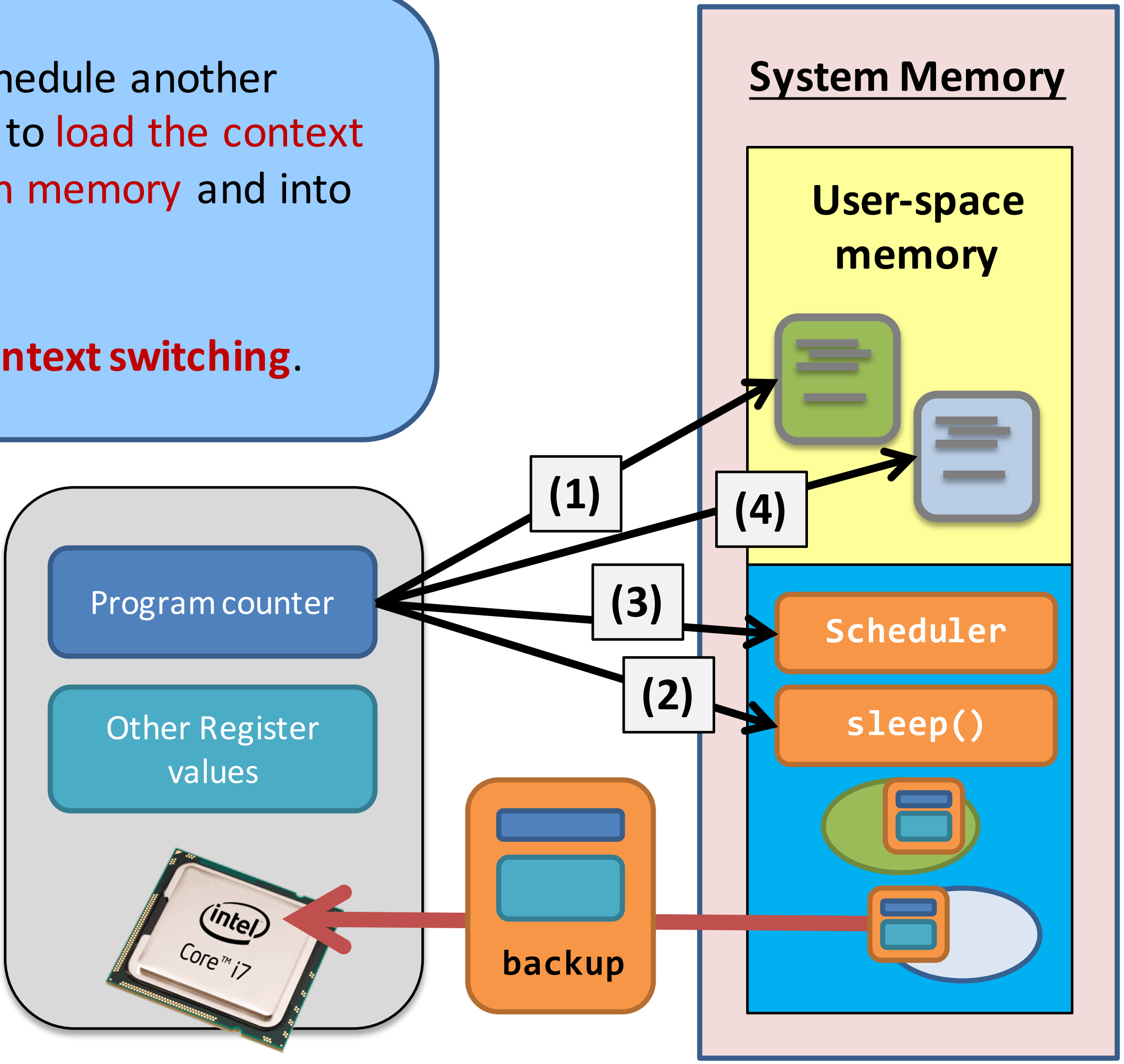
Switching from one process to another.

Say, the scheduler decides to schedule another process. Then, the schedule has to load the context of the new process into the main memory and into the CPU.

We call the entire operation: context switching.

Analogy

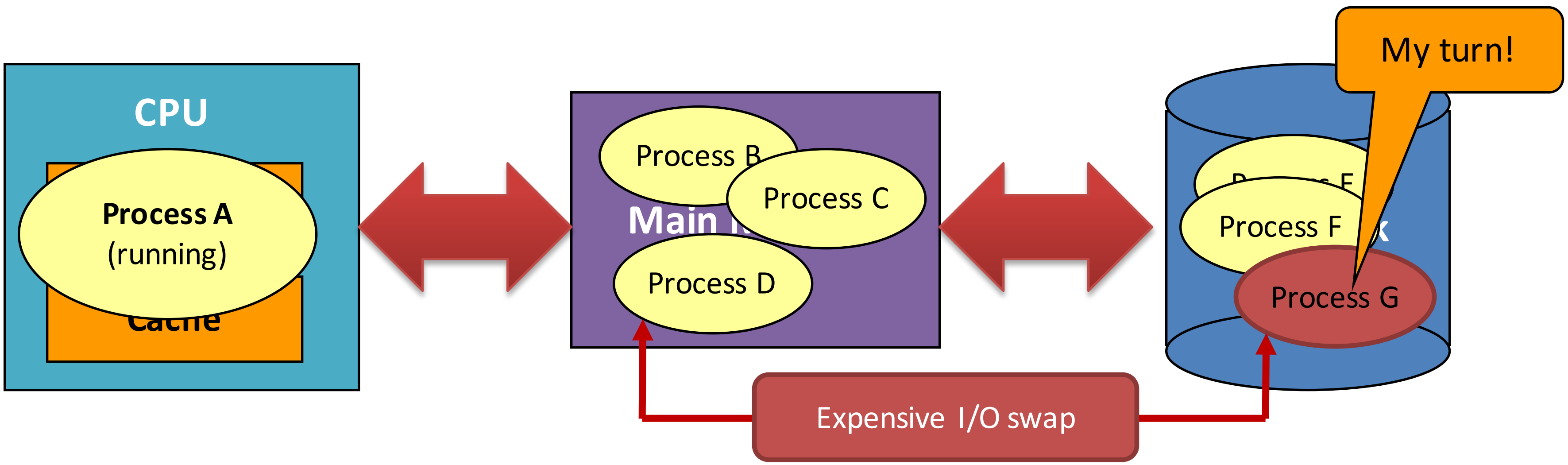
Back to the game console example, after you have changed to a new game, you load the game progress you previously saved.



Context switching has a price to pay…

- However, context switching may be expensive…
  - The target process may be currently <u>stored in the hard disk</u>.

- So, **minimizing the number of context switching** may help boosting system performance.



Topics

- The first process and process organization in Linux;
- Process lifecycle;
- Context switching;
- **Process scheduling.**
    - **some basics.**

What is process scheduling?

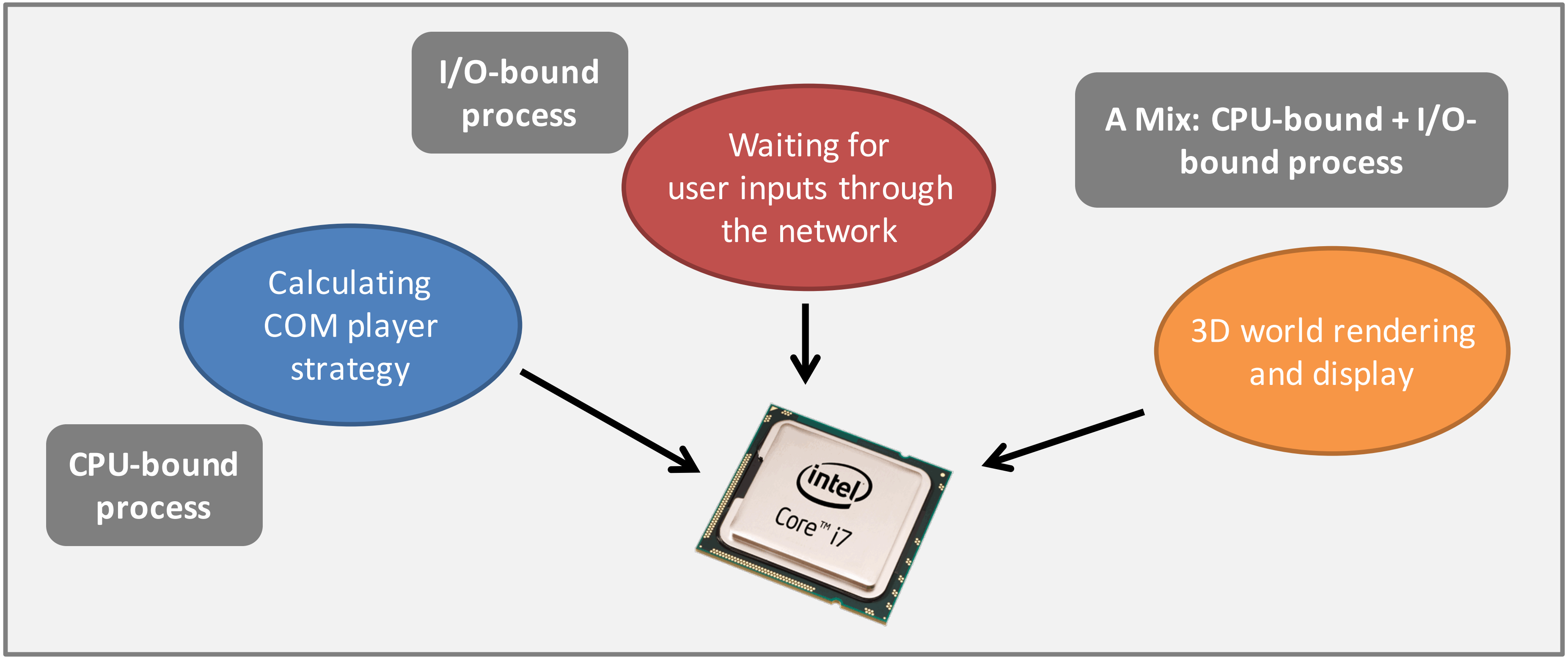
- Scheduling is an important topic in the research of the operating system.
  - Related theoretical topics are covered in CSCI5420.

- Scheduling is required because the number of computing resource – the CPU – is **limited**.

CPU-bound Process	I/O-bound process
Spends most of its running time on the CPU, i.e., **user-time > sys-time**	Spends most of its running time on I/O, i.e., **sys-time > user-time**
Examples - CSCI2100 assignments, AI programs.	**Examples** - **/bin/ls**, networking programs.

What is process scheduling?

- A 3D online game analogy.
  - *For your information*: under most cases, those tasks implemented as **threads** instead of processes.



Process scheduling properties

- Usually, process scheduling is triggered when the following cases happen:

A new process is created.	When "**fork()**" is invoked and returns successfully. Then, whether the parent or the child is scheduled is up to the scheduler's decision.
An existing process is terminated.	**The CPU is freed**. The scheduler should choose another process to run.
A process waits for I/O.	**The CPU is freed**. The scheduler should choose another process to run.
A process finishes waiting for I/O.	The interrupt handling routine **makes a scheduling request**, if necessary.

See? Those four events are all happening **inside kernel**.

It is very rare that a user-level program makes an explicit scheduling request.

Do you like to know more: "man **`sched_yield()`**"

Classes of process scheduling

- Non-preemptive scheduling.

What is it?	When a process is chosen by the scheduler, the process would never leave the scheduler until… -the process voluntarily waits for I/O, or -the process voluntarily releases the CPU, e.g., **`exit()`**.
What is the catch?	If the process is *<u>purely CPU-bound</u>*, it will seize the CPU from the time it is chosen until it terminates.
Where can I find it?	Nowhere…but it could be found back in the mainframe computers in 1960s.
Pros	Good for systems that emphasize **the time in finishing tasks**. - Because the task is running without others' interruption.
Cons	Bad for nowadays systems in which **user experience** and **multi-tasking** are the primary goals.

Classes of process scheduling

- Preemptive scheduling.

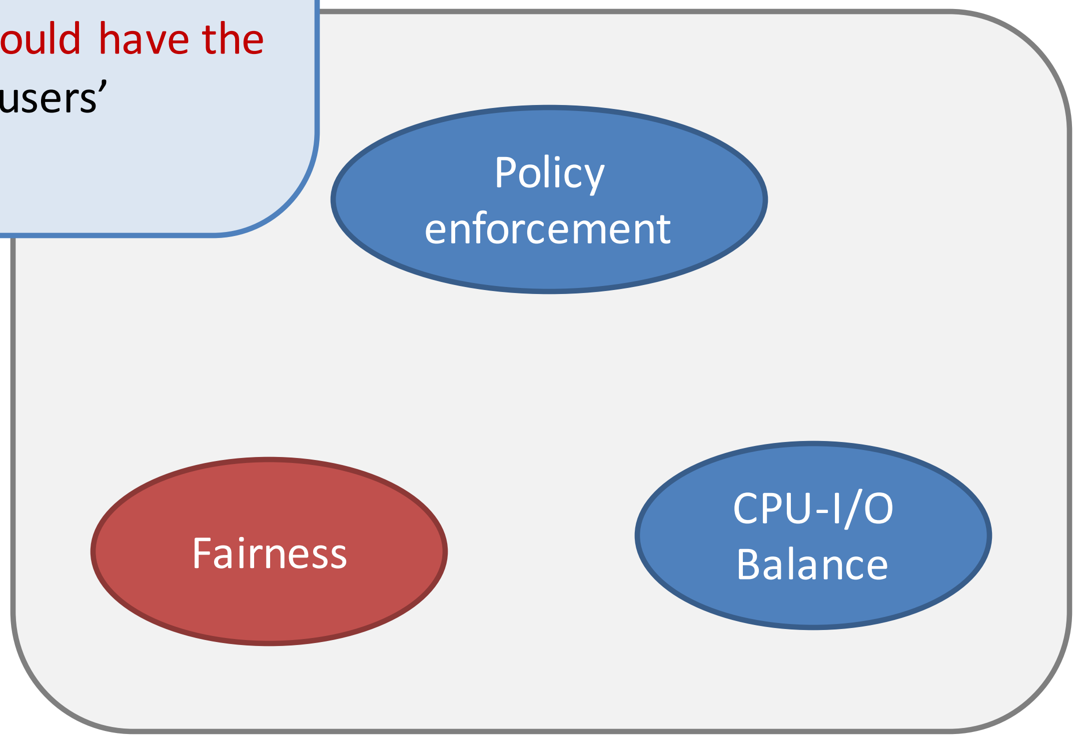
What is it?	When a process is chosen by the scheduler, the process would never leave the scheduler until... -the process voluntarily waits for I/O, or -the process voluntarily releases the CPU, e.g., **`exit()`**. -**particular kinds of interrupts and events are detected**.
What is the catch?	If that particular event is the ***periodic clock interrupt***, then you can have a **time-sharing system**.
Where can I find it?	Everywhere! This is the design of nowadays systems.
Pros	Good for systems that emphasize **interactive-ness**. - Because every task will receive attentions from the CPU.
Cons	Bad for systems that emphasize the time in finishing tasks.

Scheduler’s common goals

Fairness.

There should be no bias among the processes. Each process should have a (more or less) fair share of the CPU.

E.g., the administrator’s processes should have the same amount of CPU share than the users’ processes.

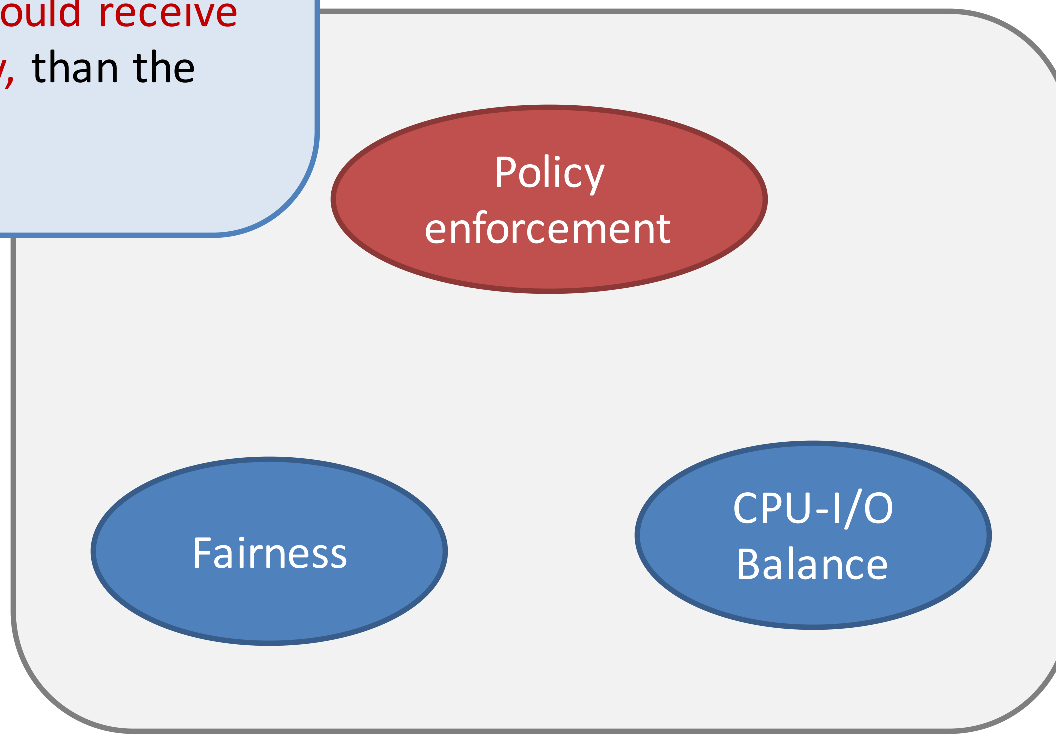


Scheduler’s common goals

Policy enforcement.

Stated policies must be carried out, without any exceptions.

E.g., the administrator’s processes should receive more CPU share, i.e., a higher priority, than the users’ processes.

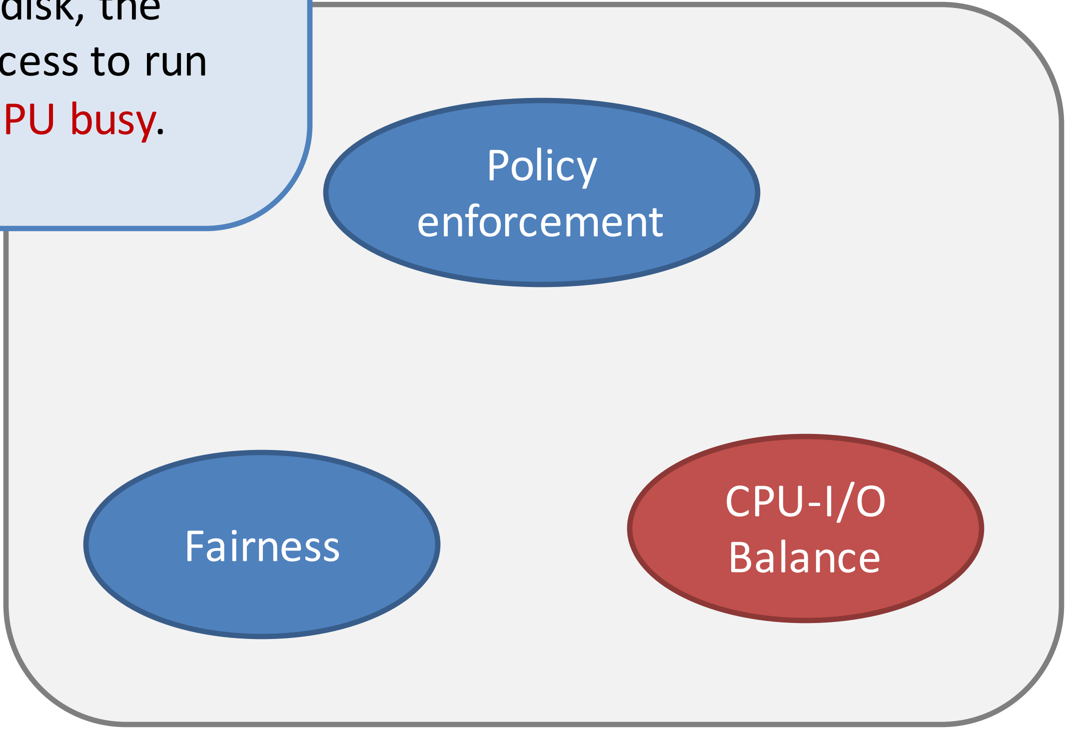


Scheduler’s common goals

Balance.

It is a good practice to keep all parts of the system busy.

E.g., When a process is accessing the disk, the scheduler should choose another process to run so as to make both the disk and the CPU busy.

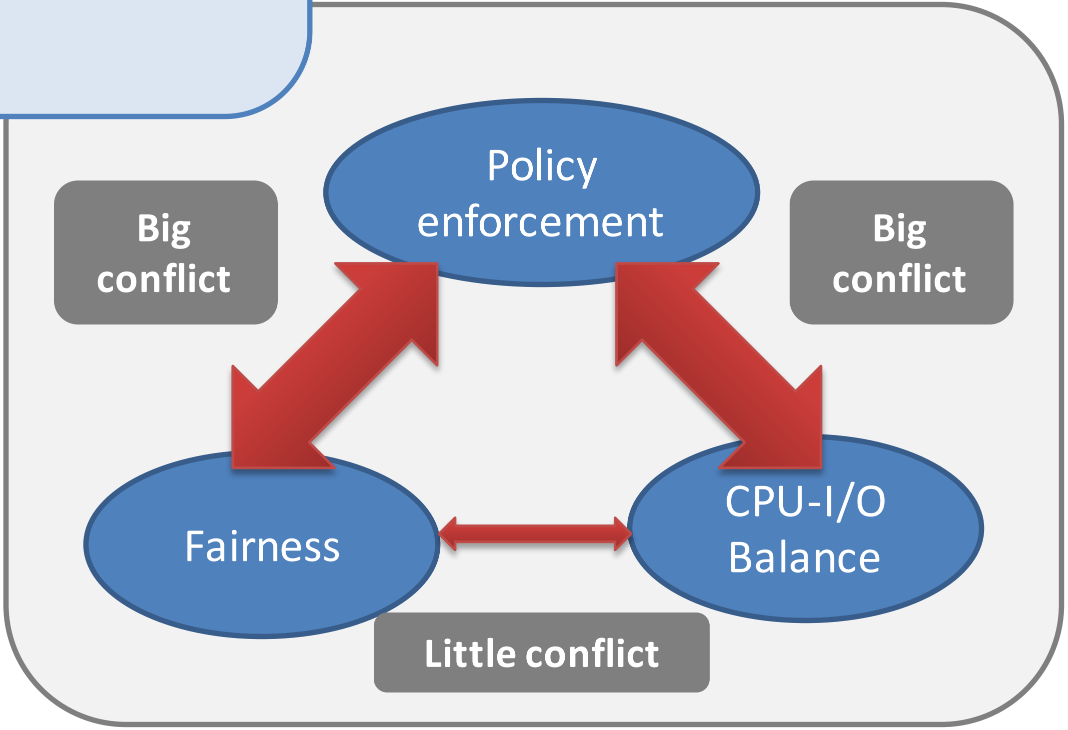


Scheduler’s common goals

The Conflicting Criteria.

The three goals, by nature, have a conflict of interest.

By the end of this part, you will see how Linux scheduler deal with this matter.



Topics

- The first process and process organization in Linux;
- Process lifecycle;
- Context switching;
- **Process scheduling.**
  - some basics.
  - **different algorithms.**

Scheduling algorithms

- **Inputs to the algorithms.**

A set of tasks: **P1** **P2** **P3** **P4**

For each task…: **Arrival Time** **CPU requirement**

It is interesting to note that this is a non-sense!

How can we know the requirement of each task?

Online VS Offline

An **offline scheduling algorithm** assumes that you know all the processes submitted to the system before hand. But, an **online scheduling algorithm** does not have such an assumption.

Yet, every real scheduler has to work in an "**online scenario**". So, we have to think in an "online" way…

Scheduling algorithms

- **Outputs of the algorithms.**

Scheduling order

Number of context switching

Individual & average turnaround time

Individual & average waiting time

Turnaround time	The time between the arrival of the task and the termination of the task.
Waiting time	The total time that a task (1) is not yet terminated, but (2) does not receive any attentions from the CPU.

Different algorithms

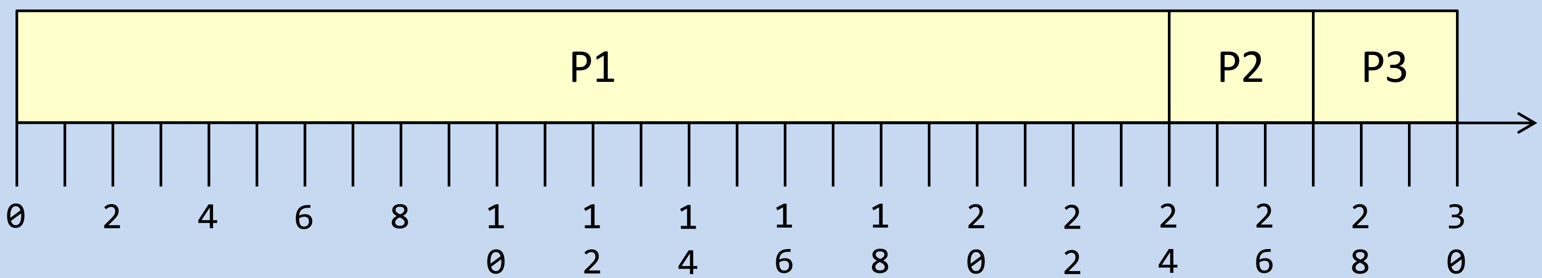
Algorithms	Preemptive?	Target System
First-come, first-serve or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern (Skipped in lecture)
Priority scheduling with multiple queues.	The real implementation!	

First-come, first-served scheduling

- Example 1.

Output

Gantt Chart



Waiting time: P1 = 0; P2 = 23; P3 = 25;

Average waiting time = (0+23+25)/3 = 16;

Turnaround time: P1 = 24; P2 = 26; P3 = 28;

Average turnaround time = (24+26+28)/3 = 26;

Task	Arrival Time	CPU Req.
P1	0	24
P2	1	3
P3	2	3

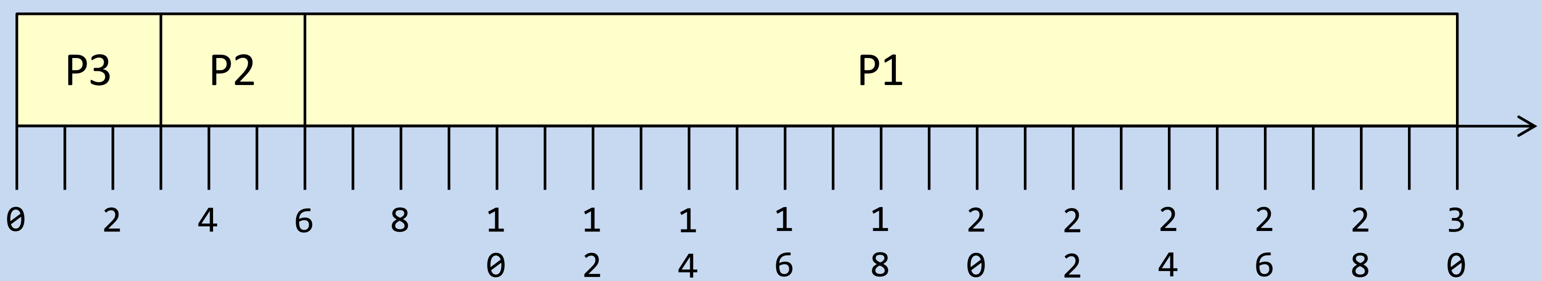
Input

First-come, first-served scheduling

- Example 2.

Output

Gantt Chart



Waiting time: P1 = 4; P2 = 2; P3 = 0;

Average waiting time = (4+2+0)/3 = 2;
(which is 16 in the previous case)

Turnaround time: P1 = 28; P2 = 5; P3 = 3;

Average turnaround time = (28+5+3)/3 = 12;
(which is 26 in the previous case)

Task	Arrival Time	CPU Req.
P3	0	3
P2	1	3
P1	2	24

Input order changed

First-come, first-served scheduling

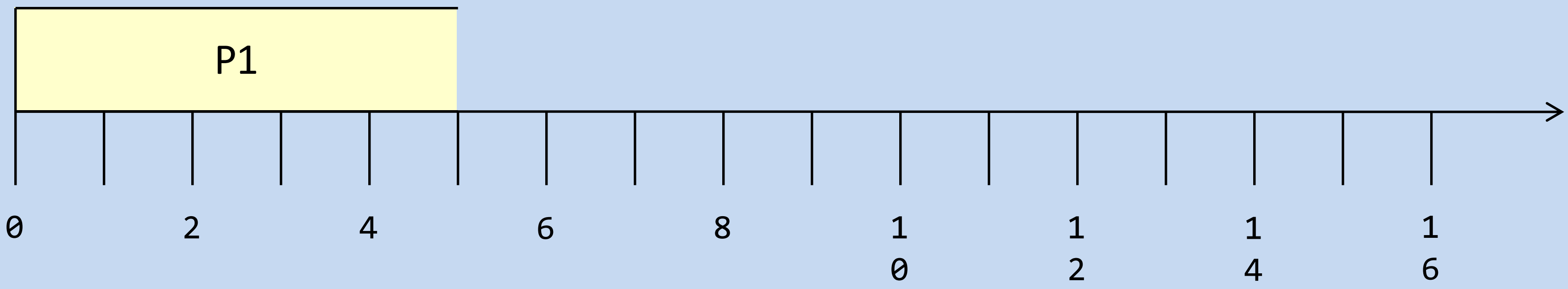
- A short summary:
  - FIFO scheduling is sensitive to the input.
  - Think about the scenario:
    - Someone is standing before you in the queue in KFC, and
    - you find that he/she is ordering the **bucket chicken meal** (P1 in example 1)!!!!
    - So, two people (P2 and P3) are unhappy while only P1 is happy.
  - Can we do something about this?

Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-serve or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern (Skipped in lecture)
Priority scheduling with multiple queues.	The real implementation!	

Non-preemptive SJF

Animation; don't print



Time = 5

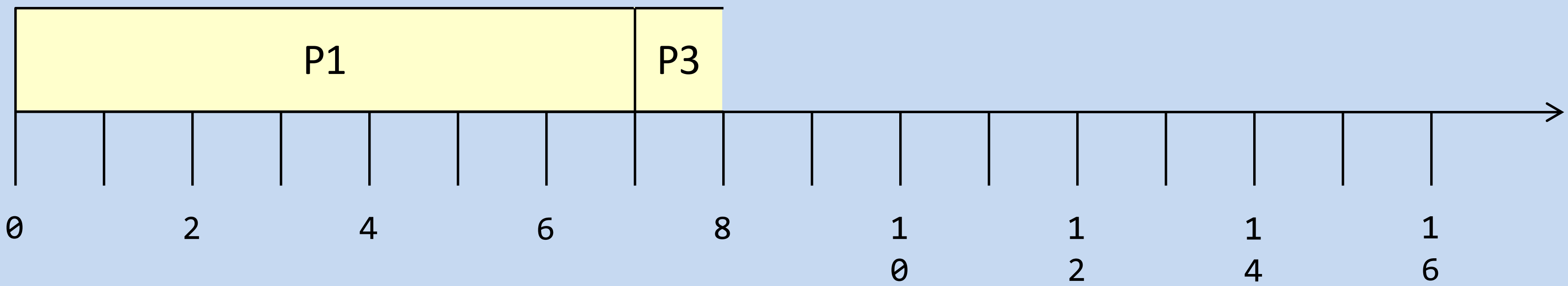
Set of processes

P1 P2 P3 P4

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

Animation; don't print



Time = 7

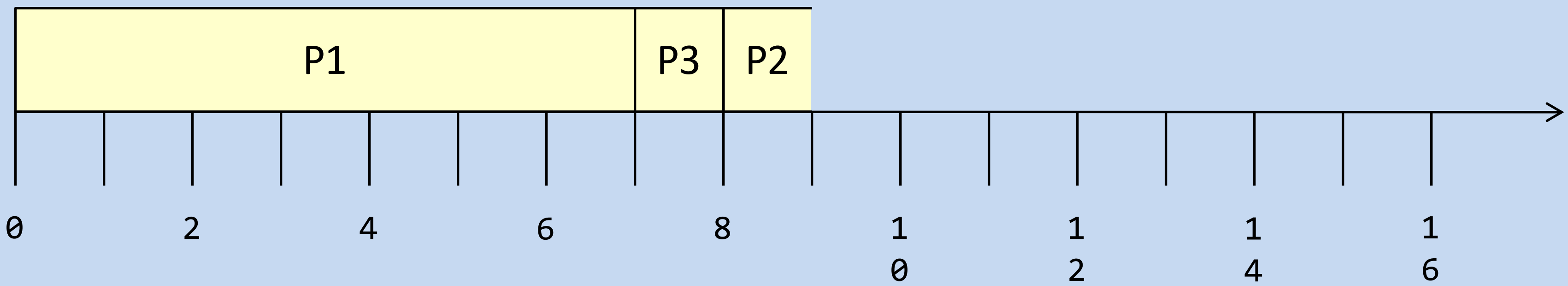
Set of processes

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

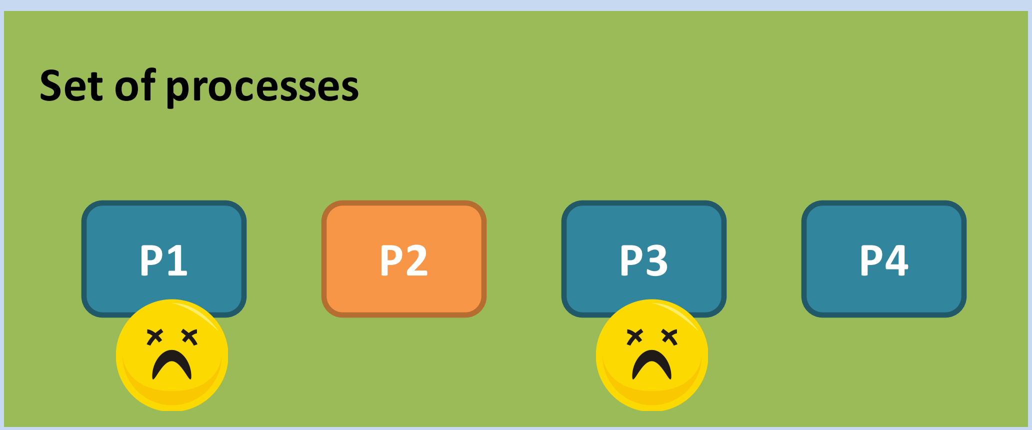
Non-preemptive SJF

Animation; don't print

In this example, we use **FIFO** to break the tie.



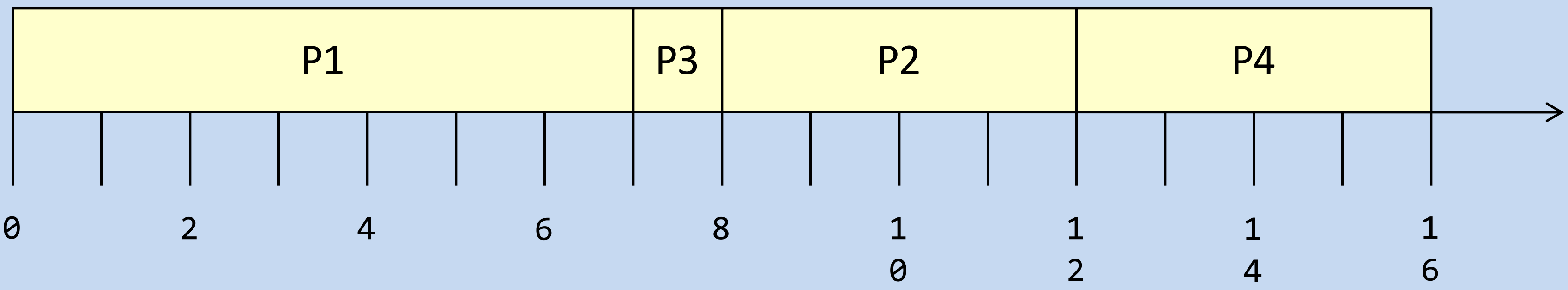
Time = 8



Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

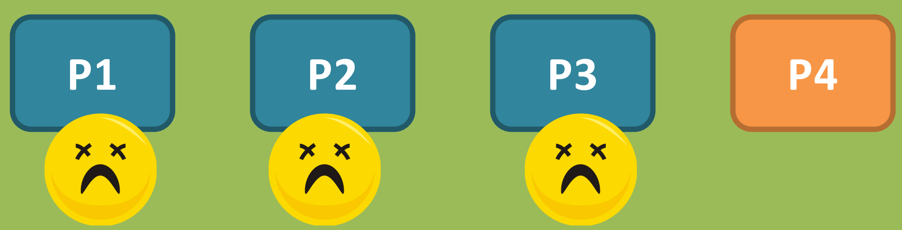
Non-preemptive SJF

Animation; don't print



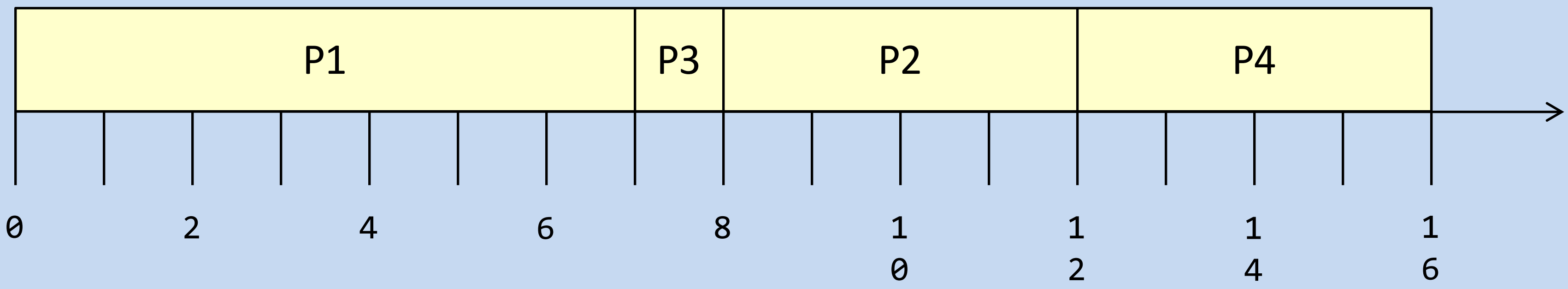
Time = 16

Set of processes



Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

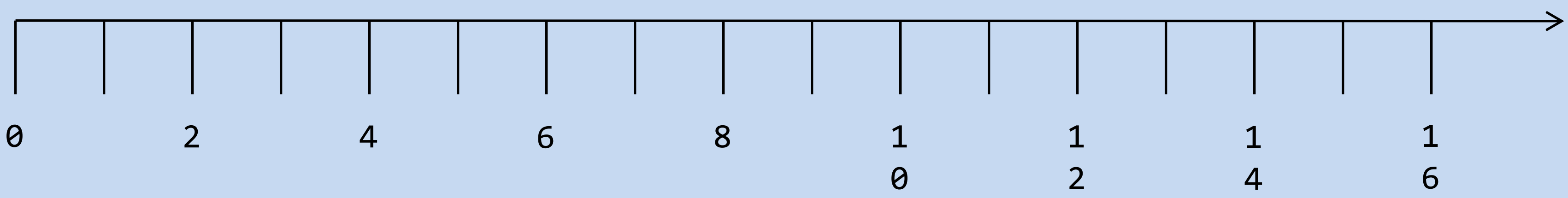



```
Waiting time:
   P1 = 0; P2 = 6; P3 = 3; P4 = 7;
   Average = (0 + 6 + 3 + 7) / 4 = 4.
```

```
Turnaround time:
   P1 = 7; P2 = 10; P3 = 4; P4 = 11;
   Average = (7 + 10 + 4 + 11) / 4 = 8.
```

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Preemptive SJF



<u>Rules for preemptive scheduling</u>
(for this example only)

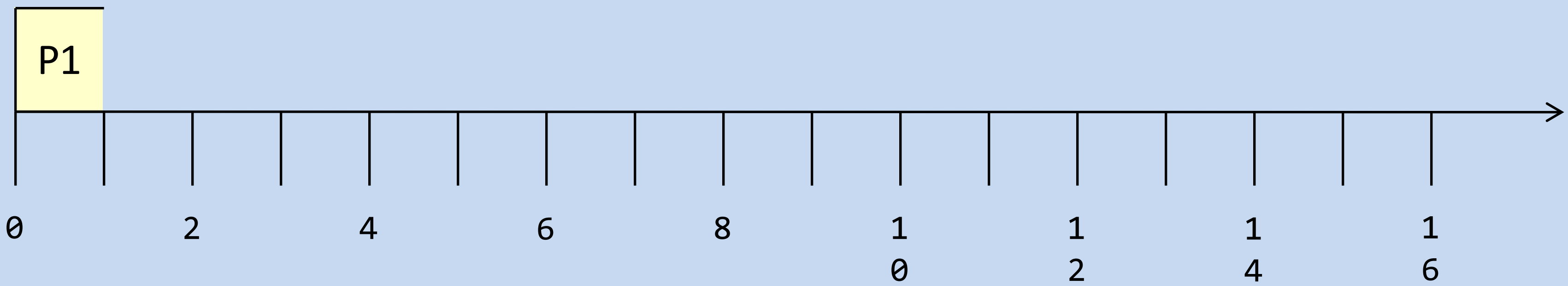
-Preemption happens when a new process arrives at the system.

-Then, the scheduler steps in and selects the next task based on **their remaining CPU requirements**.

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Preemptive SJF

Animation; don't print



Time = 0

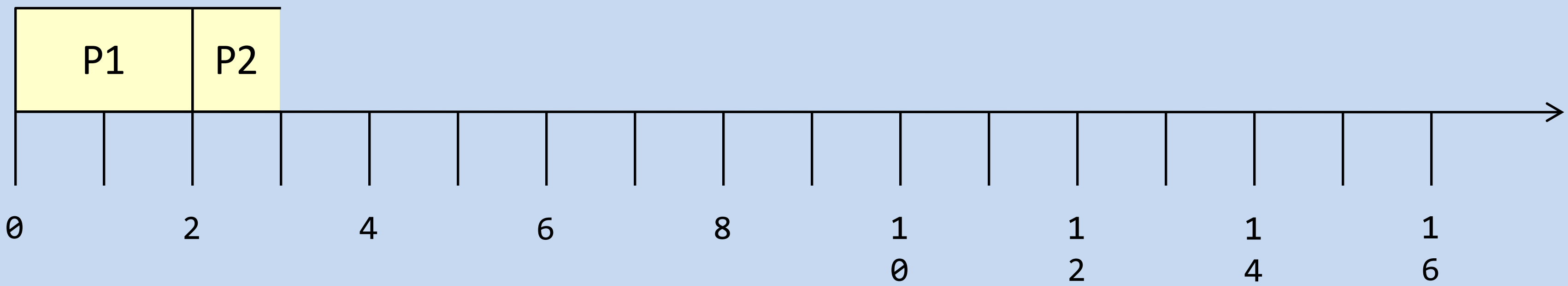
Set of processes

P1

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Preemptive SJF

Animation; don't print



Time = 2

P2 is selected!

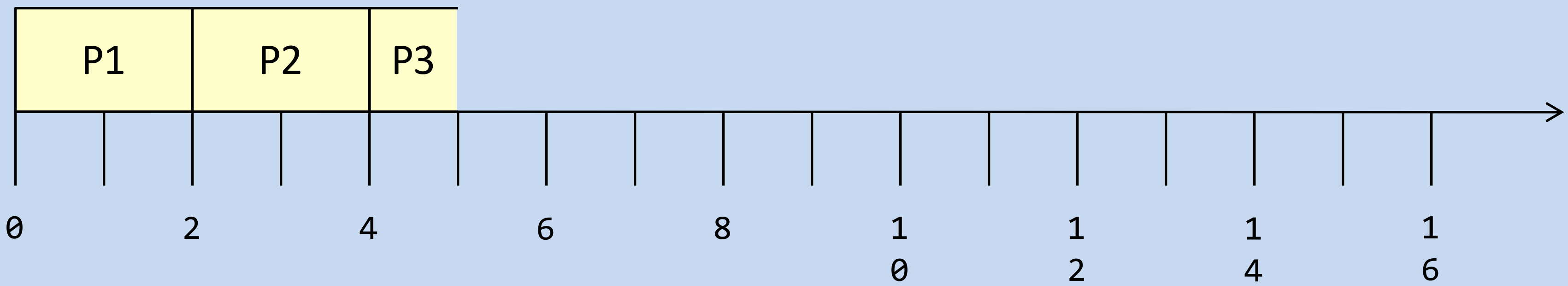
Set of processes

P1 **P2**

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	4
P3	4	1	1
P4	5	4	4

Preemptive SJF

Animation; don't print



Time = 4

P3 is selected!

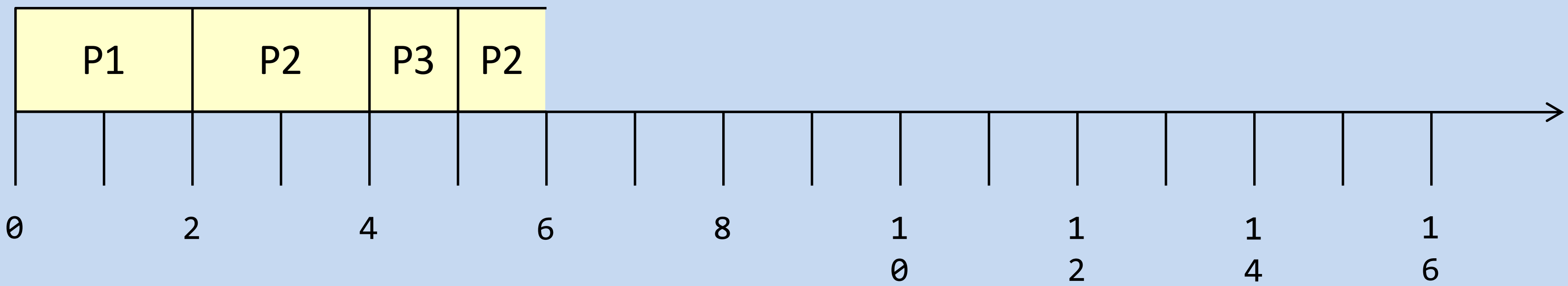
Set of processes

P1 **P2** **P3**

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	1
P4	5	4	4

Preemptive SJF

Animation; don't print



Time = 5

P2 is selected!

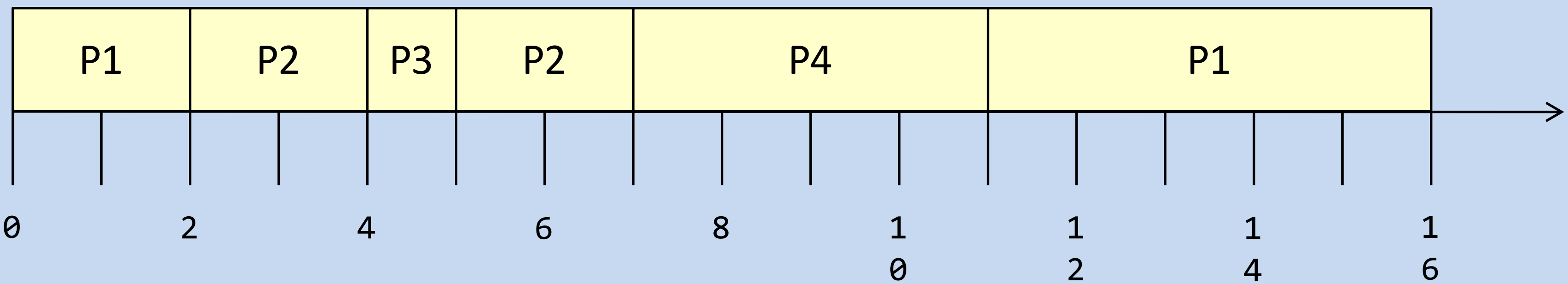
Set of processes

P1 **P2** **P3** **P4**

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	4

Preemptive SJF

Animation; don't print

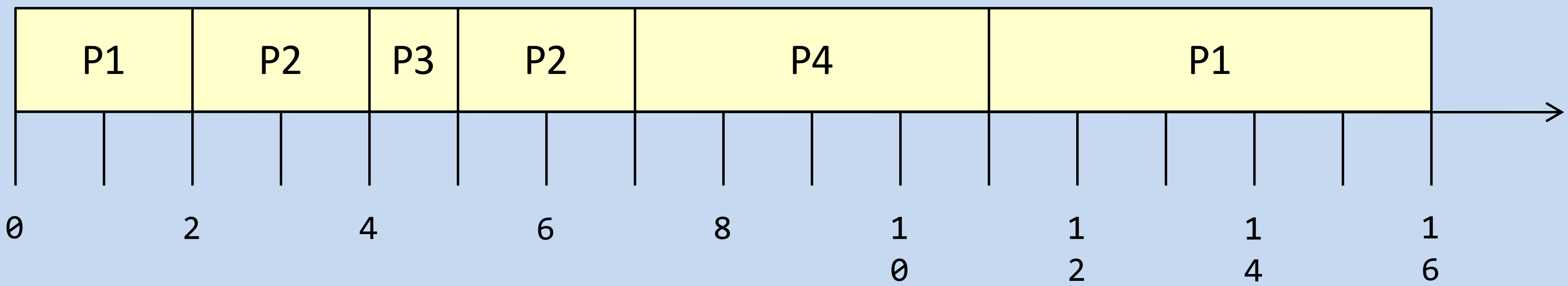


Time = 16

Set of processes

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

Preemptive SJF



Waiting time:

P1 = 9; P2 = 1; P3 = 0; P4 = 2;

Average = (9 + 1 + 0 + 2) / 4 = 3.

Turnaround time:

P1 = 16; P2 = 5; P3 = 1; P4 = 6;

Average = (16 + 5 + 1 + 6) / 4 = 7.

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

SJF: Preemptive or not?

	Non-preemptive SJF	Preemptive SJF
Average waiting time	4	**3 (smallest)**
Average turnaround time	8	**7 (smallest)**
# of context switching	**3 (smallest)**	5

The waiting time and the turnaround time decrease at the expense of the **increased number of context switching**.

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-serve or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern (Skipped in lecture)
Priority scheduling with multiple queues.	The real implementation!	

Round-robin

- Round-Robin (RR) scheduling is preemptive.
  - Every process is given a **quantum**, or the amount of time allowed to execute.
  - When the quantum of a process is **<u>used up</u>** (i.e., 0), the process releases the CPU and **this is the preemption**.
  - Then, the scheduler steps in and it chooses **the next process which has a non-zero quantum** to run.
  - If all processes in the system has used up the quantum, it will be re-charged to its initial value.
  - Processes are therefore running one-by-one, like a circular queue.

Round-robin

Rules for Round-Robin
(for this example only)

-The quantum of every process is fixed and is 2 units.

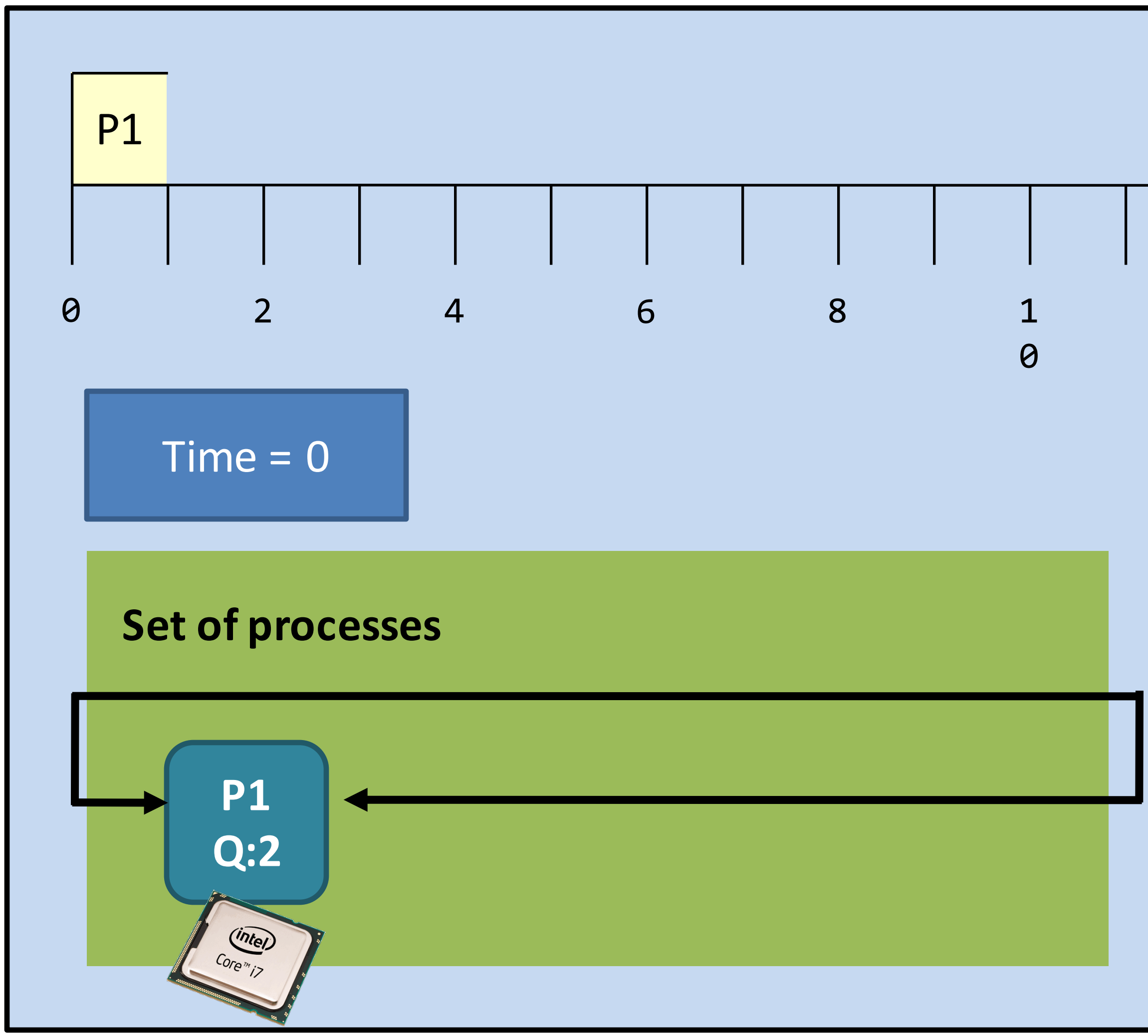
-The process queue is sorted according the processes' arrival time, in an ascending order.
(This rule allows us to break tie.)

-After recharge, the scheduler will choose the next process which follows the previously-executed process in the queue.
(This rule guarantees fairness.)

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Round-robin

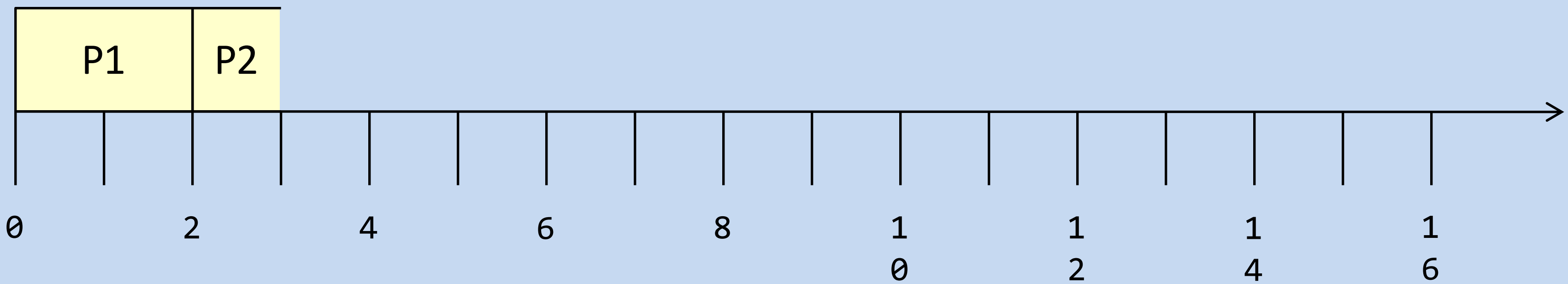
Animation; don't print



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Round-robin

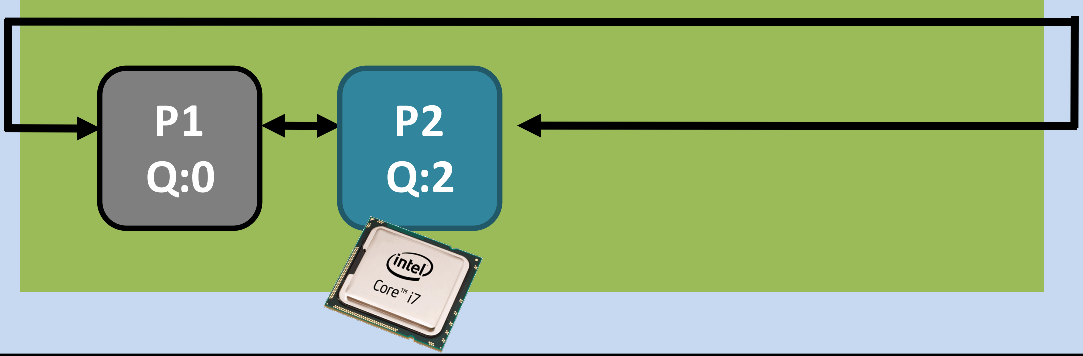
Animation; don't print



Time = 2

P1's quantum is 0;
P2 is selected!

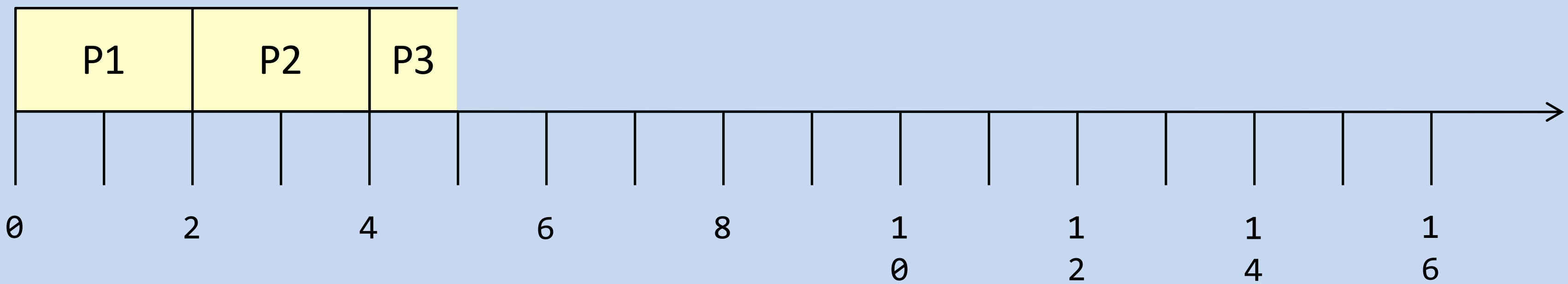
Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	4
P3	4	1	1
P4	5	4	4

Round-robin

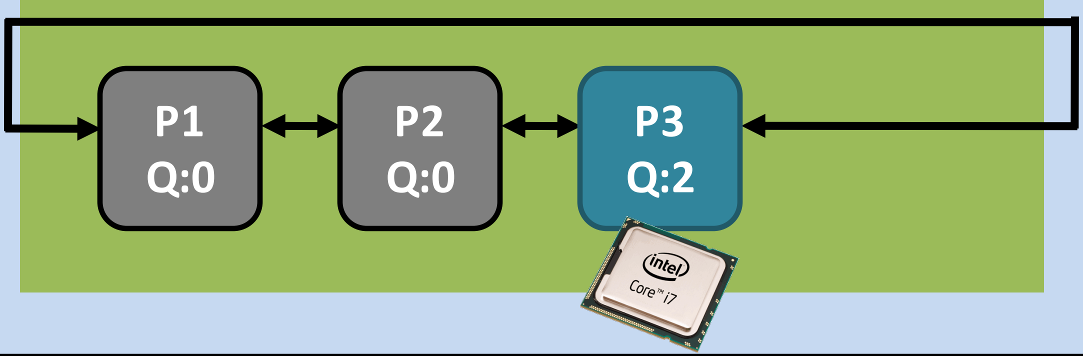
Animation; don't print



Time = 4

P1's & P2's quanta are 0;
P3 is selected!

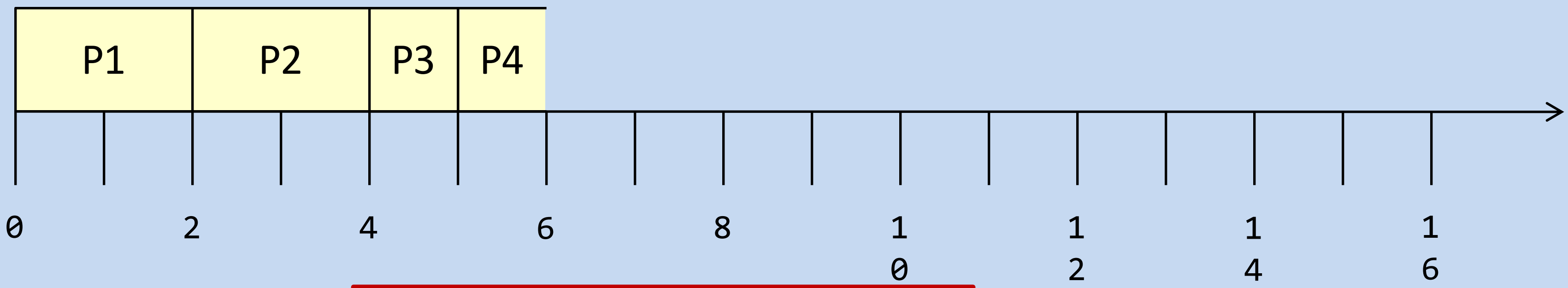
Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	1
P4	5	4	4

Round-robin

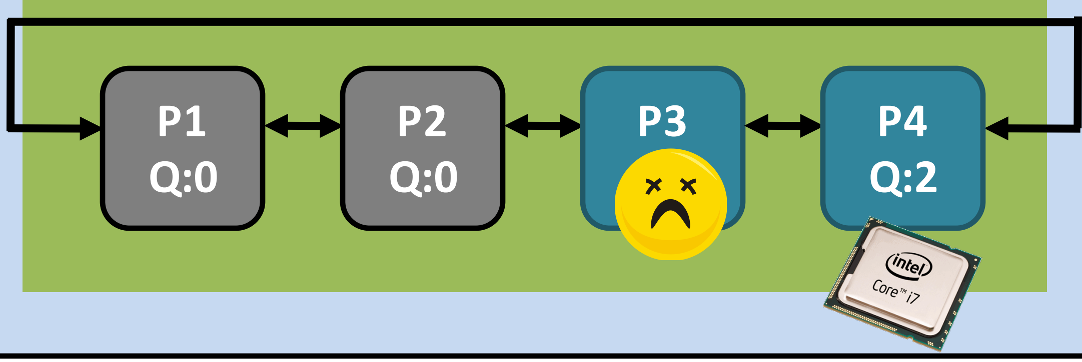
Animation; don't print



Time = 5

P1's & P2's quanta are 0;
P4 is selected!

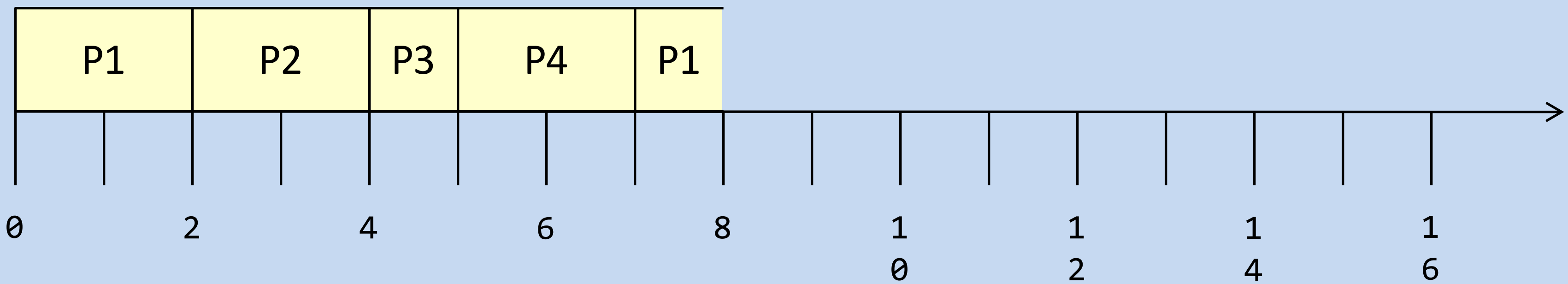
Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	2

Round-robin

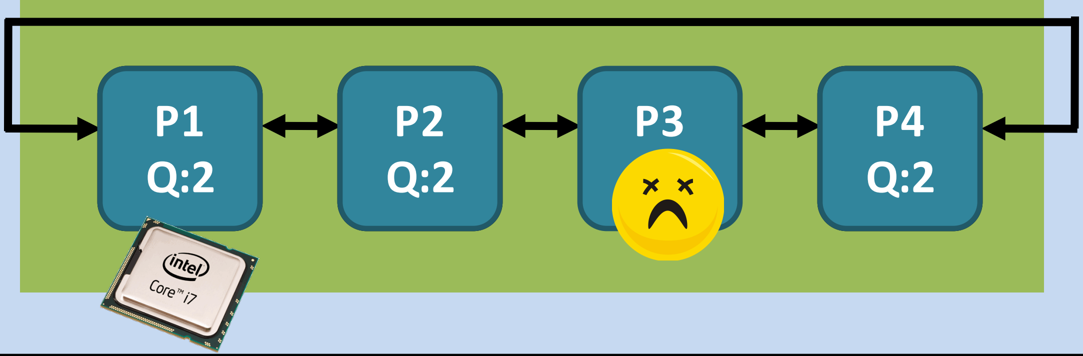
Animation; don't print



Time = 7

Now, recharge is needed.
P1 is selected.

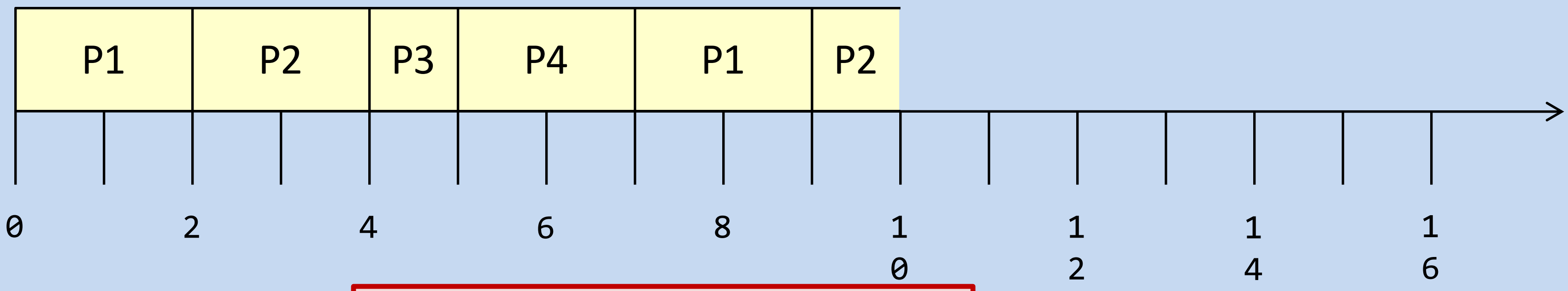
Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	2

Round-robin

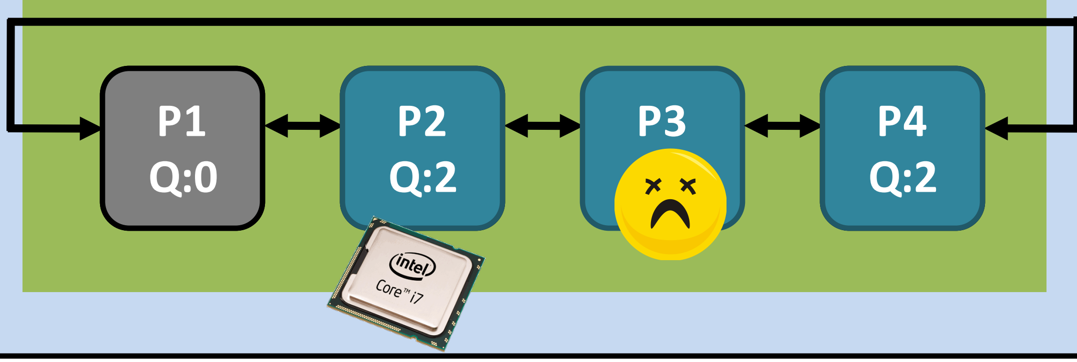
Animation; don't print



Time = 9

P1's quantum is 0;
P2 is selected!

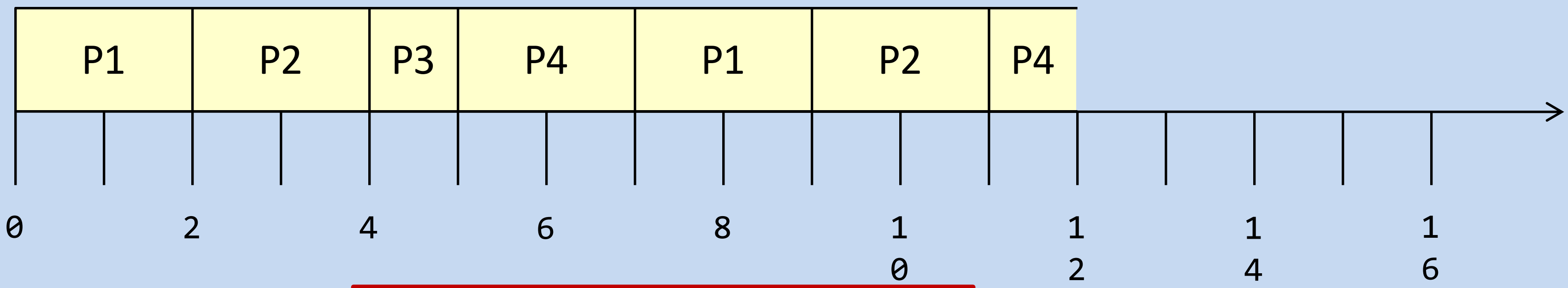
Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	2
P3	4	1	0
P4	5	4	2

Round-robin

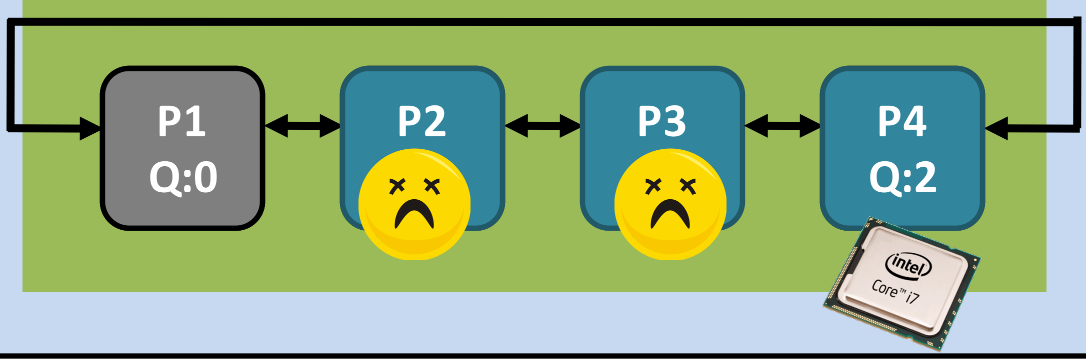
Animation; don't print



Time = 11

P1's quantum is 0;
P4 is selected!

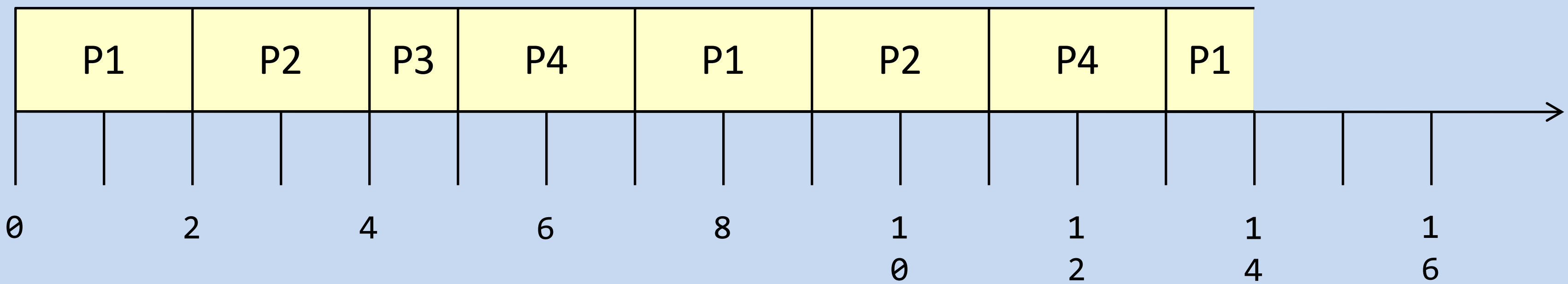
Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	0
P3	4	1	0
P4	5	4	2

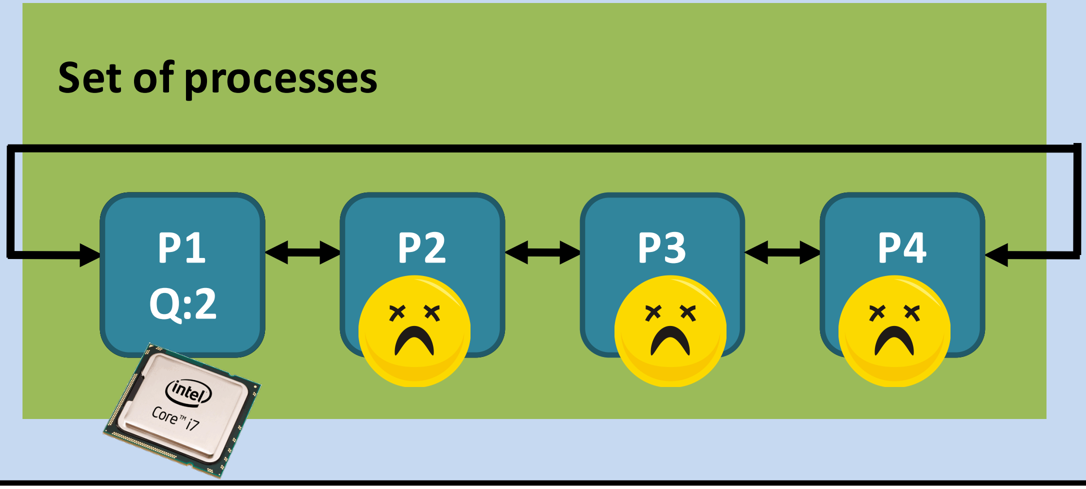
Round-robin

Animation; don't print



Time = 13

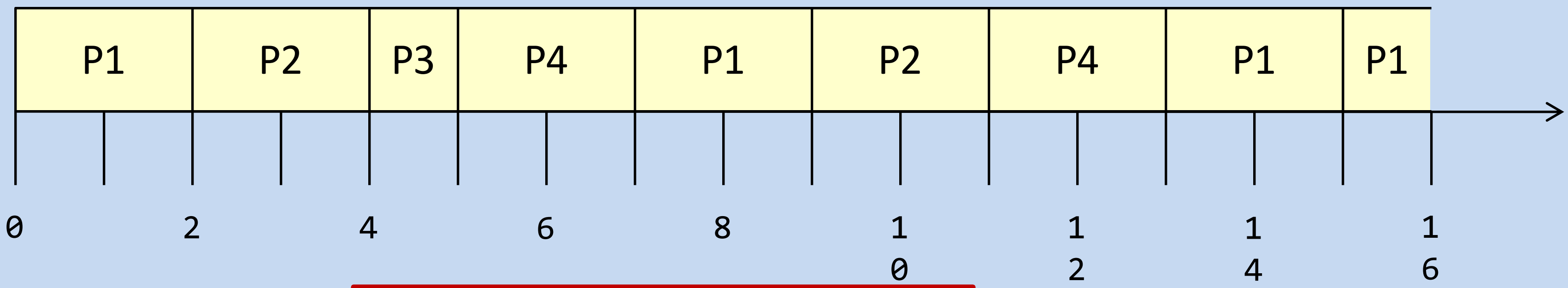
Now, recharge is needed.
P1 is selected.



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	0
P3	4	1	0
P4	5	4	0

Round-robin

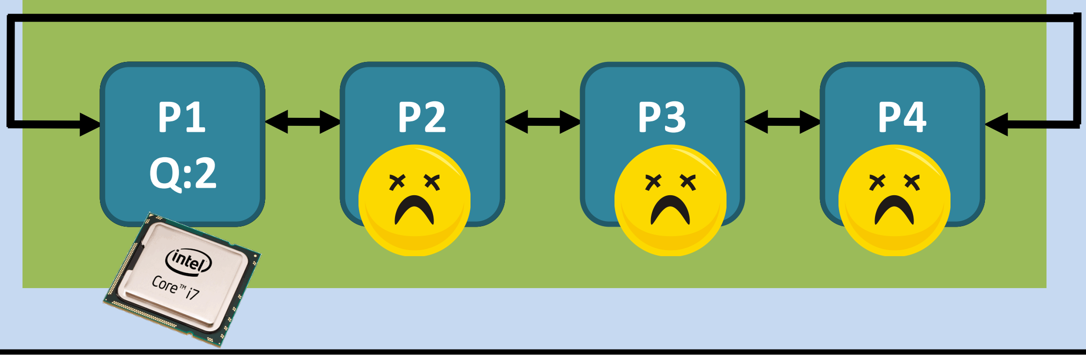
Animation; don't print



Time = 15

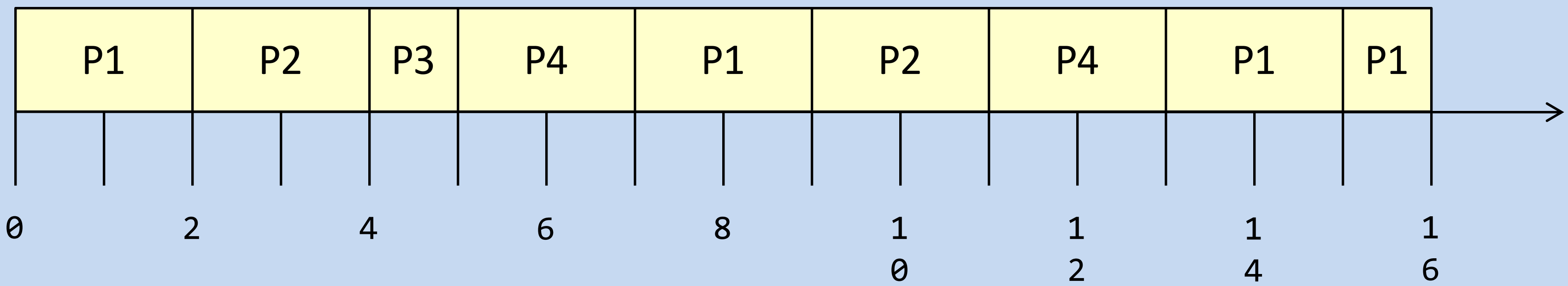
Now, recharge is needed.
P1 is selected.

Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	1
P2	2	4	0
P3	4	1	0
P4	5	4	0

Round-robin




```
Waiting time:

   P1 = 9; P2 = 5; P3 = 0; P4 = 4;

   Average = (9 + 5 + 0 + 4) / 4 = 4.5
```

```
Turnaround time:

   P1 = 16; P2 = 9; P3 = 1; P4 = 8;

   Average = (16 + 9 + 1 + 8) / 4 = 8.5
```

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

RR VS SJF

	Non-preemptive SJF	Preemptive SJF	RR
Average waiting time	4	**3**	**4.5 (largest)**
Average turnaround time	8	**7**	**8.5 (largest)**
# of context switching	**3**	5	**8 (largest)**

So, the RR algorithm gets all the bad! Why do we still need it?

The responsiveness of the processes is great under the RR algorithm. E.g., you won't feel a job is "frozen" because every job is on the CPU from time to time!

Observations on RR

- Modified versions of round-robin are implemented in (nearly) every modern OS.
  - Users run a lot of **interactive jobs** on modern OS-es.
  - Users' priority list:
    - Number one - Responsiveness;
    - Number two - Efficiency;
  - In other words, "ordinary users" expect a fast GUI response than an efficient scheduler running behind.

- A joke for you to think about:
  - Will you doubt the correctness of your browser when it returns you something after it has been frozen for a while?

Observations on RR

- Modified versions of round-robin are implemented in (nearly) every modern OS.
  – With the round-robin deployed, the scheduling **looks like random**.
  – It also looks like "***fair to all processes***".

- Class discussion:
  – Does RR give fair treatment to both CPU-bound and I/O-bound processes?
  – If yes, how? If not, can you suggest any modifications? (there is no model answers)

Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-serve or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern (Skipped in lecture)
Priority scheduling with multiple queues.	The real implementation!	

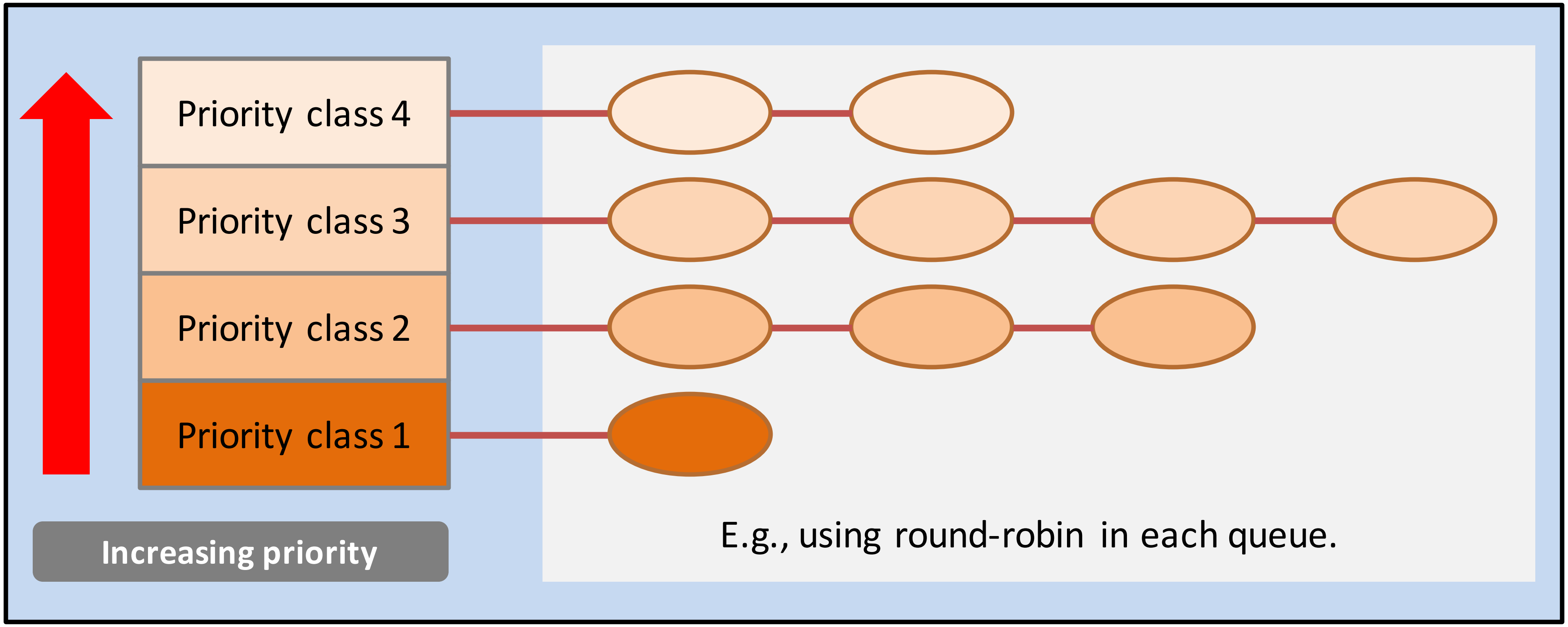
Priority Scheduling

- Some basics:
  - A task is given a priority (and is usually an integer).
  - A scheduler selects the next process based on the priority.
    - ***A typical practice***: the highest priority is always chosen.

2 Classes	
Static priority	**Dynamic priority**
Every task is given a fixed priority.	Every task is given an initial priority.
The priority is **fixed** throughout the life of the task.	The priority is **changing** throughout the life of the task.

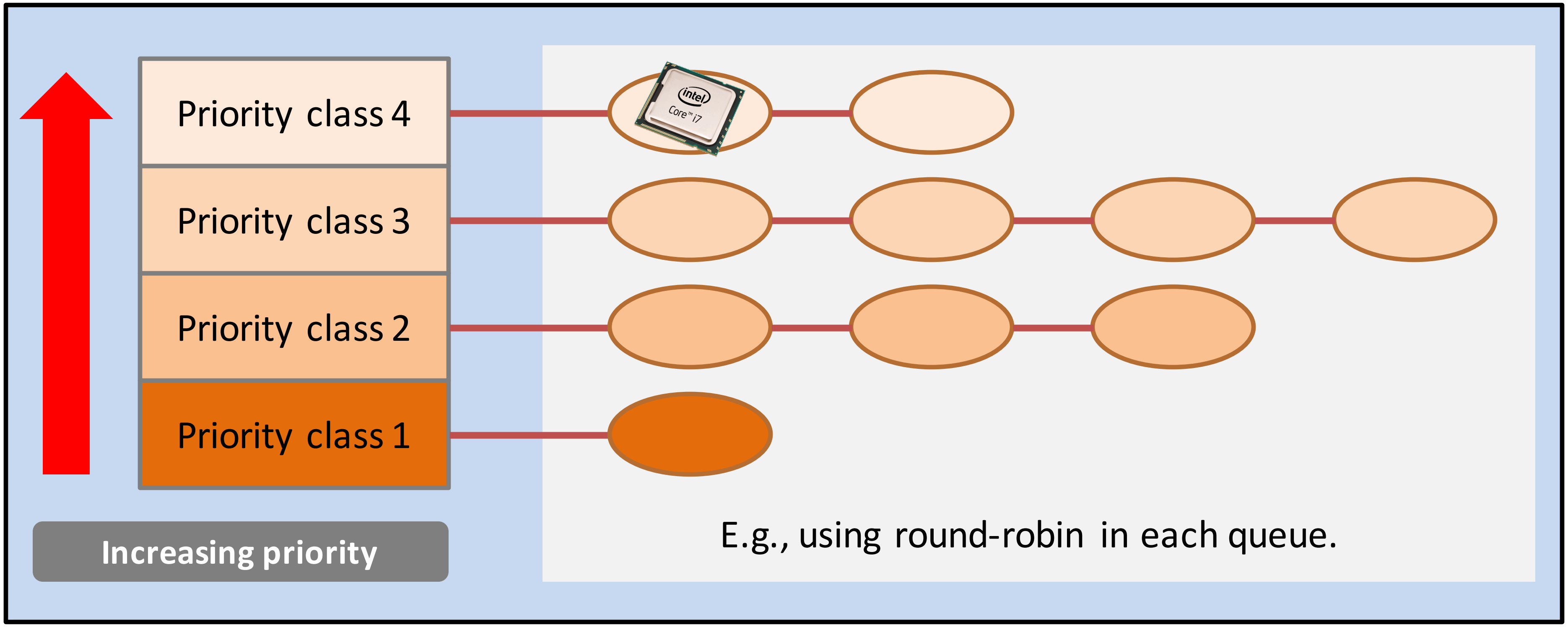
Static priority scheduling – an example

- **Properties**: process is assigned a fix priority when they are submitted to the system.
  - E.g., Linux kernel 2.6 has 100 priority classes, [0-99].



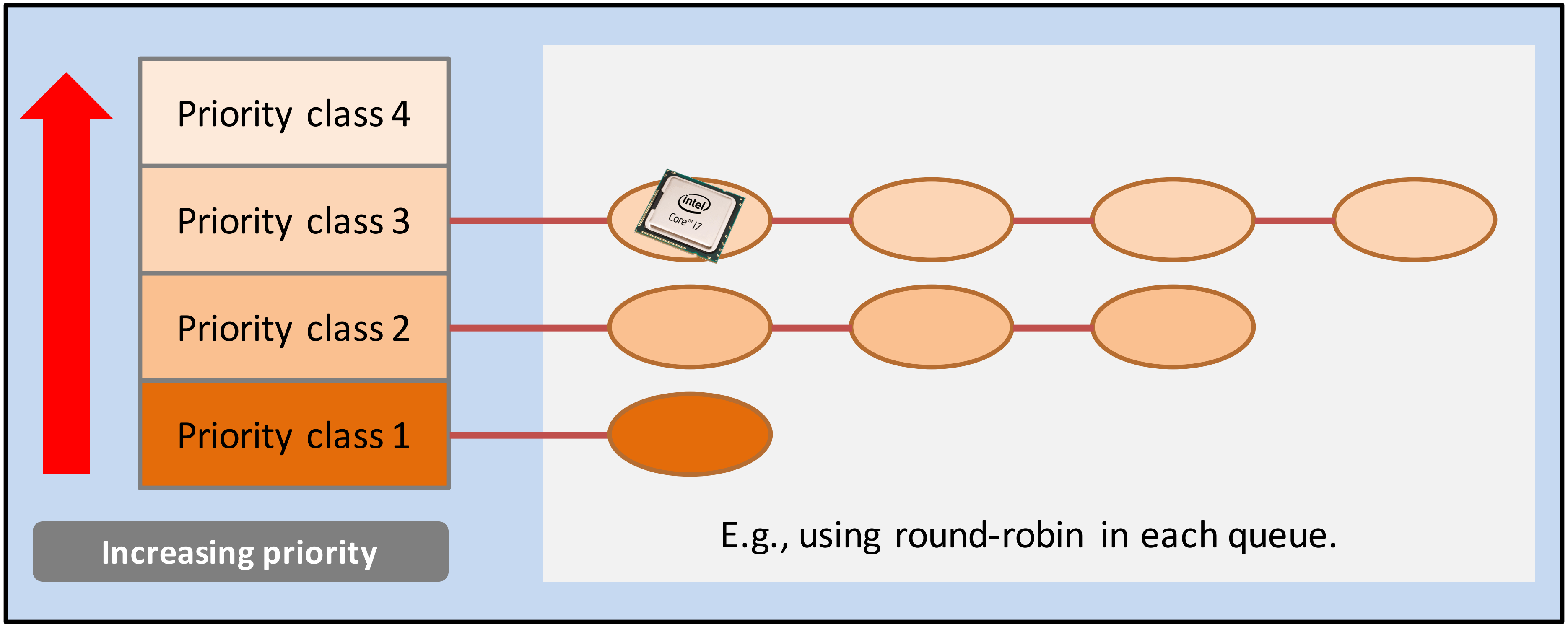
Static priority scheduling – an example

- The highest priority class will be selected.
  - The tasks are usually short-lived, but important;
    - To prevent high-priority tasks from running indefinitely.



Static priority scheduling – an example

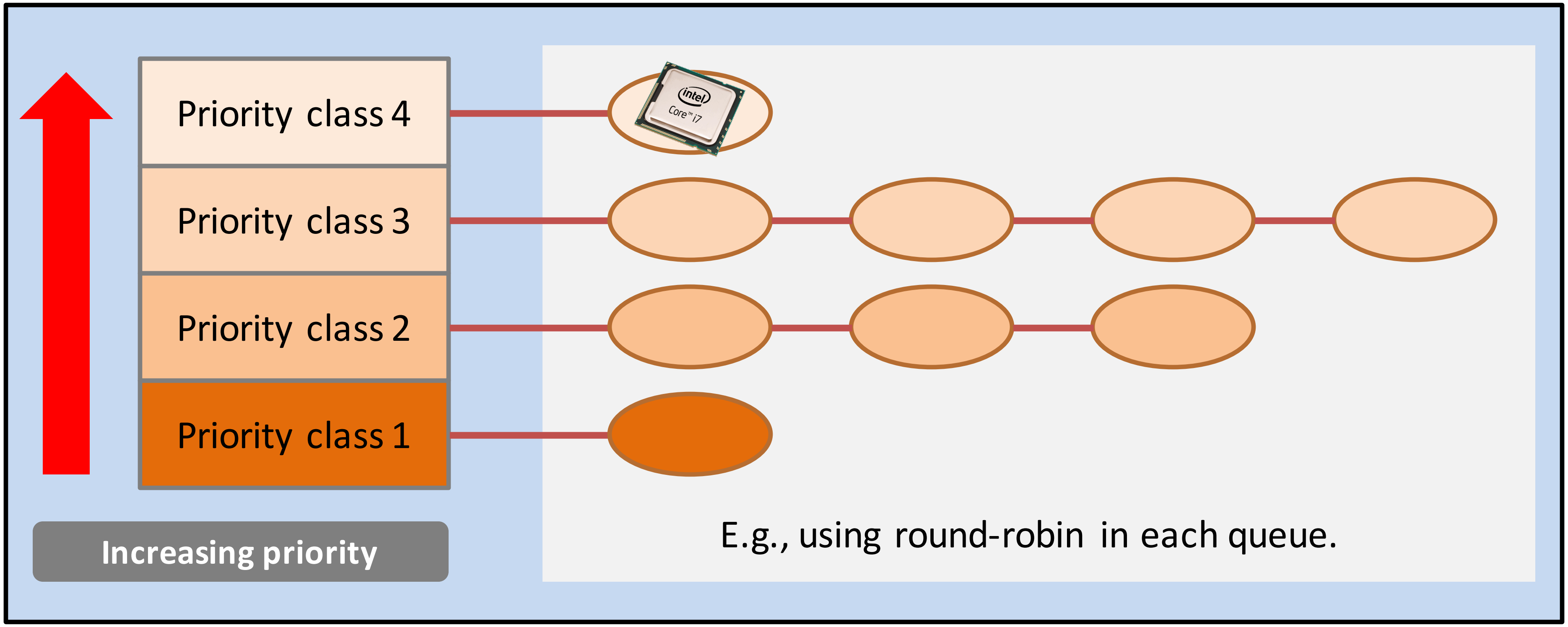
- Lower priority classes will be scheduled only when the upper priority classes has no tasks.



Static priority scheduling – an example

- Of course, it is a good design to have a high-priority task preempting a low-priority task.

  (conditioned that the high-priority task is short-lived.)



Multiple queue priority scheduling

- **Definitions.**
  - It is still a priority scheduler.
  - But, at each priority class, **different schedulers** may be deployed.
  - The priority can be a mix of static and dynamic.

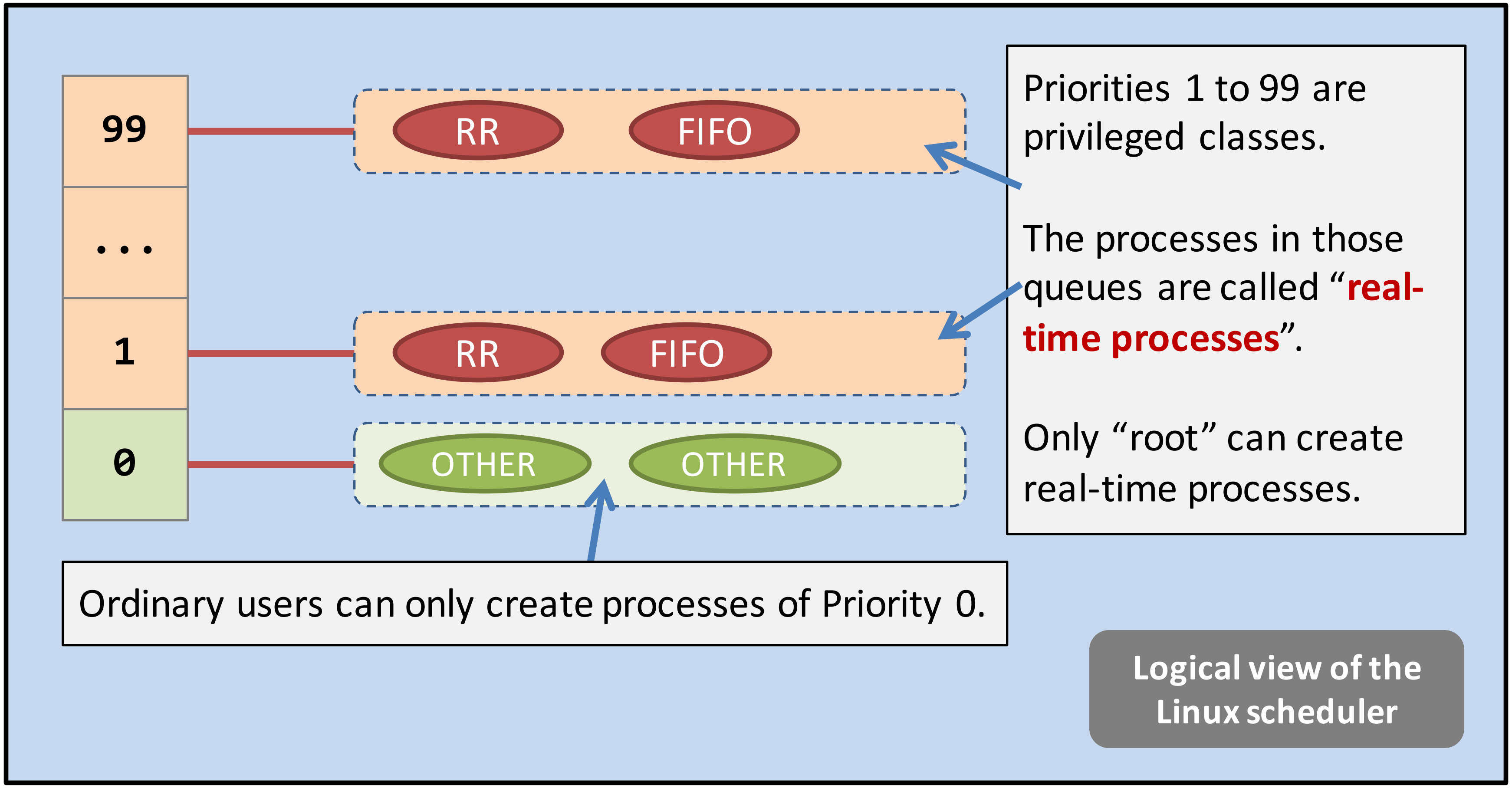
Priority class 5	Non-preemptive, FIFO
Priority class 4	Non-preemptive, SJF
Priority class 3	RR with quantum = 10 units.
Priority class 2	RR with quantum = 20 units.
Priority class 1	RR with quantum = 40 units.

Just an example.

A process drops to a lower priority class after it has used up its quantum and has the quantum recharged.

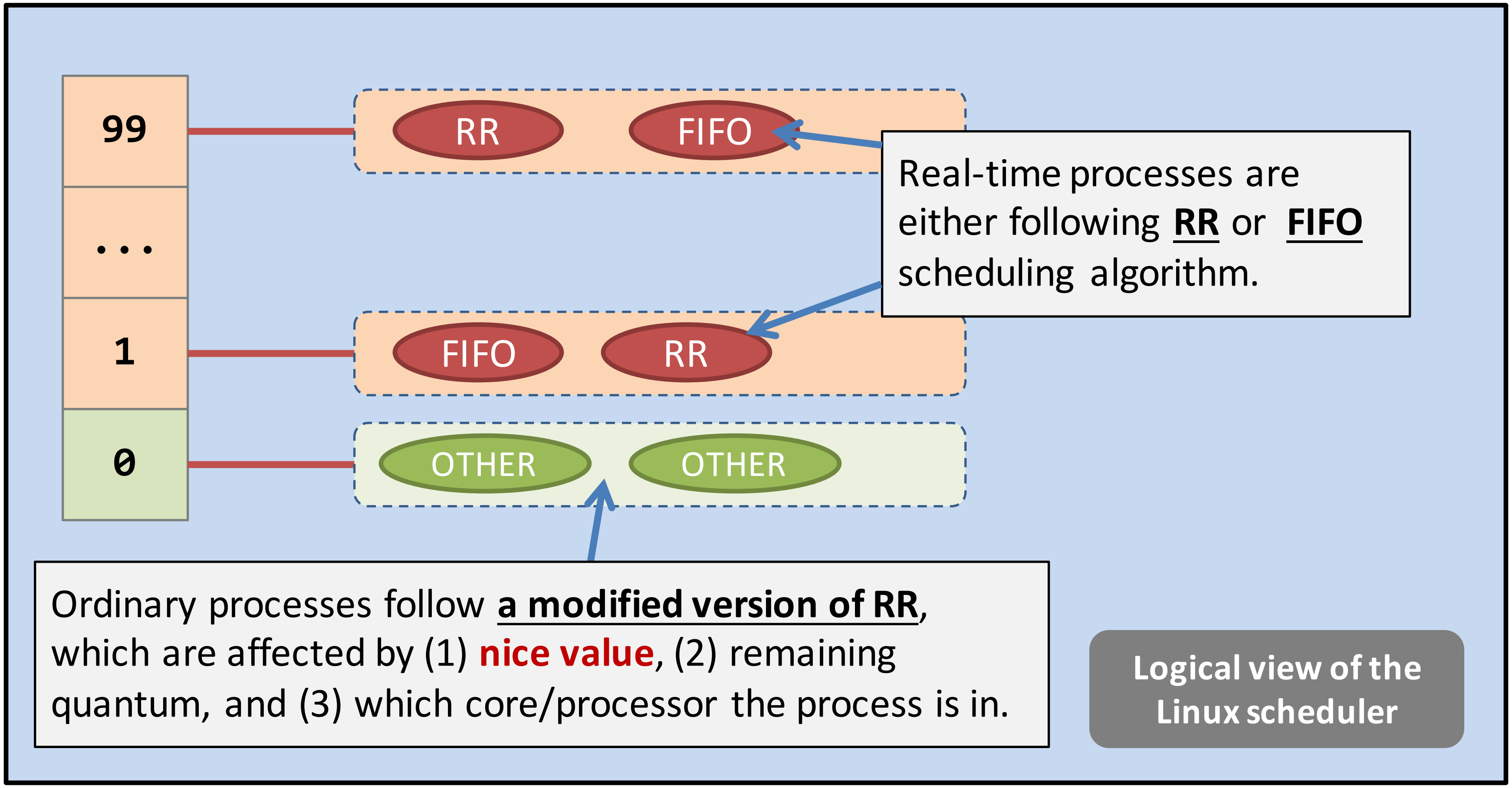
Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.



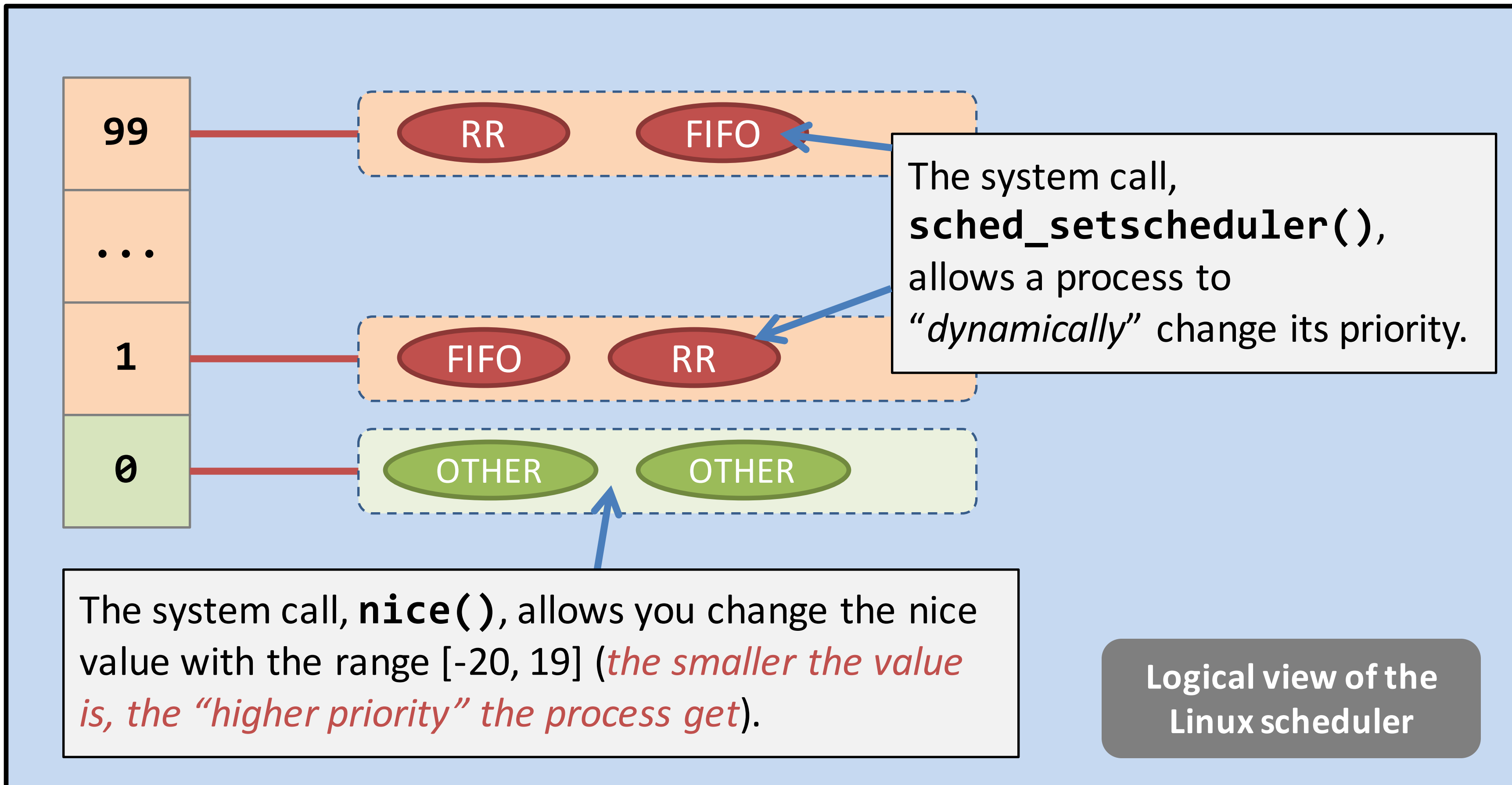
Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.



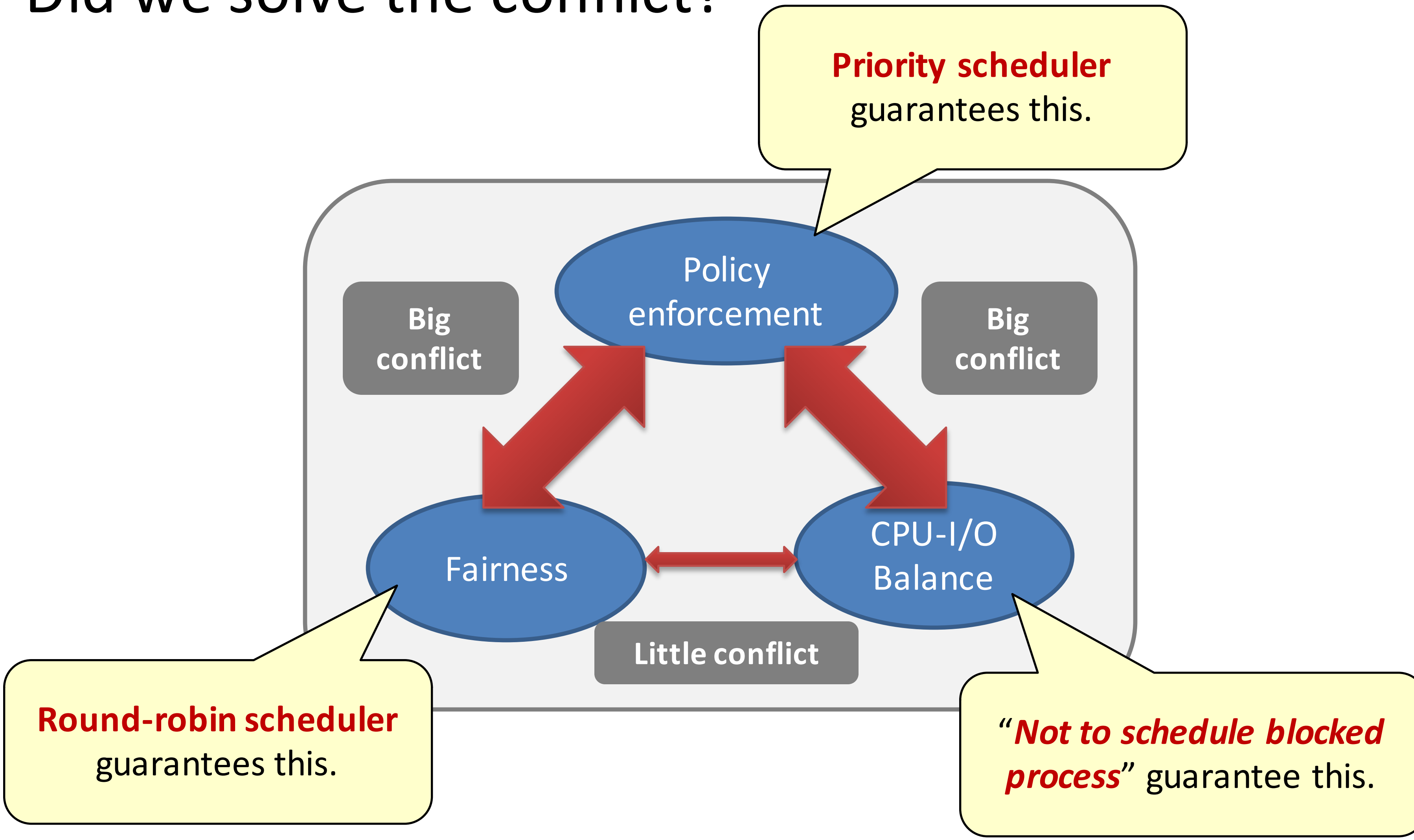
Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.



Summary

- Did we solve the conflict?



Summary

- So, you may ask:
  - "What is the **best** scheduling algorithm?"
  - "What is the **standard** scheduling algorithm?"

- There is **no best or standard** algorithm because of, *at least*, the following reasons:
  - No one could predict how many clock ticks does a process requires.
    - **Consider**: an infinite-loop program.
  - On modern OS-es, processes are submitted online.
    - Online scheduling is a NP-hard problem…