OS: Lecture 8

Process scheduling

Review - What and Why

Computers often do several things concurrently, even if it has only one CPU. •

- It's called **multiprogramming**.

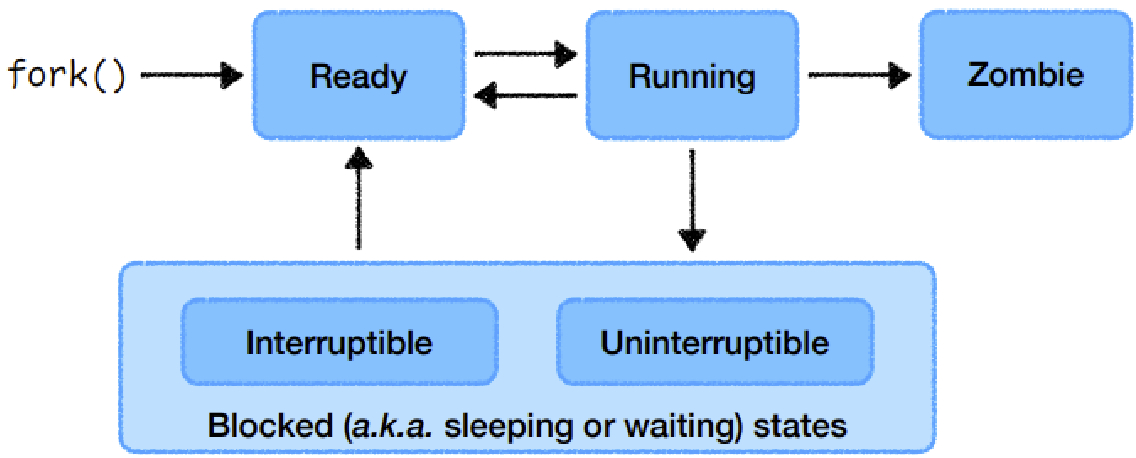
The CPU switches from process to process quickly, running each for a few *ms* .

- It's called **multitasking**.

The OS needs to choose which process to run next.

- It's called **scheduling**.
- The part of the OS that makes the choice is called the **scheduler**.

Process states

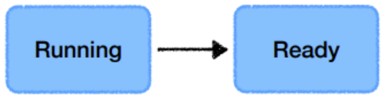


Some questions:

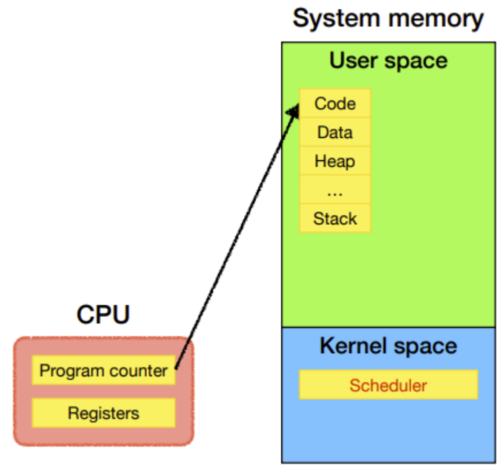
- If we just created a new process, what state is the process in?
  - Ready
- If we only have 1 core, how many process can be in the running state?
  - One
- What's the state after a process called `waitPID()`
  - blocked
  - because it has to wait for some child process to terminate
- What's a zombie
  - The child terminates, but the parent process hasn't called `wait()`

- Context switching

When it's time for a process to give up running on the CPU…



The scheduler in the kernel will choose the next process to run.

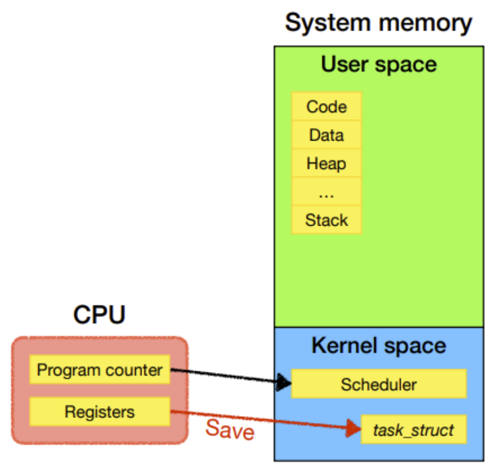


Before the scheduler can take up the CPU, it has to back up register values.

- Where should the backup be stored?

The **context** of a process consists of…

- Its user-space memory;
- Register values.



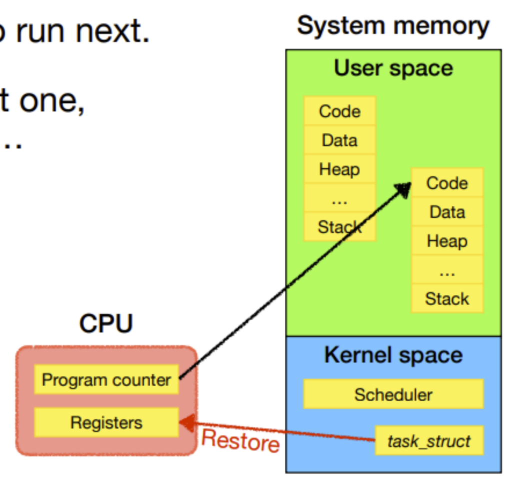
Then, the scheduler decides which process to run next.

If the next process is different from the current one, the OS performs a **context switch**, including…

- Saving and restoring registers;
- Switching memory maps;
- Flushing and reloading the cache;
- …

Context switching may be expensive…

…and the system isn't doing any useful work.



- CPU-bound processes vs. I/O-bound processes

Most processes' execution alternates between CPU execution and I/O wait.

CPU-bound processes spend most of their running time on the CPU.
- **Examples:** compiling a program, rendering a video, mining Bitcoin, scientific programming…
- If you run the `time` command, you will find: *user* time > *sys* time.

I/O-bound processes spend most of their running time on I/O.
- **Examples:** `/bin/ls`, downloading a file from the Internet, printing an image…
- If you run the `time` command, you will find: *sys* time > *user* time.

- When do we need to schedule?

A new process is created.
- It's up to the scheduler to decide whether to run the parent or the child.

An existing process is terminated.
- The scheduler should choose another process to run. If no process is ready, an "idle" process is run.

A process starts waiting for I/O (or something else).
- The process is blocked. The scheduler should choose another process to run.

A process finishes waiting for I/O.
- The process becomes ready to run again. It's up to the scheduler to decide whether to run it.

Or just periodically…

- Nonpreemptive vs. preemptive scheduling

Nonpreemptive scheduling
- When a process is scheduled, it keeps running until…
  - It starts waiting for I/O (or something else); or
  - It voluntarily relinquishes the CPU (`man 2 sched_yield`).

Preemptive scheduling
- A process can run for a particular period of time.
- When the time is up or some particular events occurs (*e.g.*, I/O completion), the process is suspended and another process may run.
- The hardware clock interrupt gives control of the CPU back to the scheduler.

- **Nonpreemptive Scheduling:**
  - We can't force a process to give up the CPU unless it's waiting for an I/O or it voluntarily gives up the CPU.
- **Preemptive Scheduling:**

- A process can run for a particular amount of time. If the time is up or it's waiting for I/O, then the process is suspended and we can switch to run another process.

What's a good scheduling?

General goals

<u>Fairness</u>

- Each process should have a fair share of the CPU.

<u>Policy enforcement</u>

- The system's policies should be carried out.
- **Example**: the admin may state that certain processes have a higher priority.

<u>Balance</u>

- All parts of the system should be kept busy all the time.
- **Example**: mix CPU-bound and I/O-bound processes.

Batch systems

A **batch system** collects a set of jobs and then process them **in batches**.

- **Examples**:
  - bank systems, data analytics, or even your laundry basket.
  - You "submit" your dirty clothes to be washed at the end of the ~~year~~ week.

Batch jobs can run **without user interaction**, so nonpreemptive scheduling or preemptive scheduling with large time slices are acceptable.

Scheduling goals for batch systems:

- **High throughput**: maximize the number of jobs per hour.
- **Short turnaround time:** minimize time between submission and completion.
- **High CPU utilization:** keep the CPU busy all the time. (*Is it a good metric* ?)

Interactive systems

An **interactive system** may be…

- A computer with an interactive user;
- A server that serves multiple interactive users.

Preemption is essential to keep one process from denying service to others.

Scheduling goals for interactive systems:

- **Short response time**:
  - minimize the time between request and response.
  - Interactive jobs may take precedence over background jobs.
- **Proportionality**:
  - meet users' expectations.
  - You think it's OK to load a video game for minutes, but not OK if it reacts a second after you press A.

Real-time systems

Real-time systems must guarantee response **within specified time constraints**.

- There's a precise deadline
- You have to meet the deadline for each process.

Example: a robot welding cars moving down an assembly line.

Scheduling goals for real-time systems:

- **Meeting deadlines**:
  - avoid losing data (or a car, or a nuclear reactor…).
  - If a device produces data at a regular rate, you must run the data-collection process on time.
- **Predictability**:
  - avoid quality degradation (in multimedia systems).
  - If the audio process runs too erratically, the sound quality will deteriorate rapidly (i.e., jitter).

Scheduling Algorithms

- **First-come, first-served**
  - It's nonpreemptive.
- **Shortest job first**
  - It can be nonpreemptive or preemptive.
- **Round-robin**
  - It's preemptive.
- **Priority scheduling**
  - It can be nonpreemptive or preemptive.

Metrics

Wait time

- The duration that the job is in the system but not running

Turnaround time

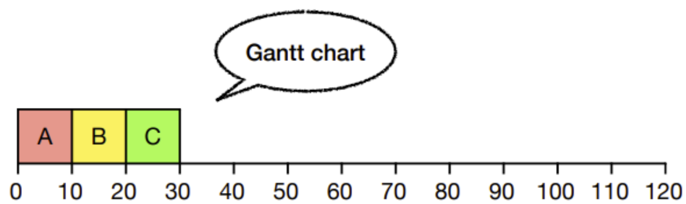
- The duration from when the job arrives in the system to the time it completes.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

First-come, first-served (FCFS)

First-come, first-served (FCFS), a.k.a. **first-in, first-out (FIFO)**, is a nonpreemptive scheduling algorithm in batch systems.

Job	Arrival time	CPU requirement
A	0	10
B	0	10
C	0	10



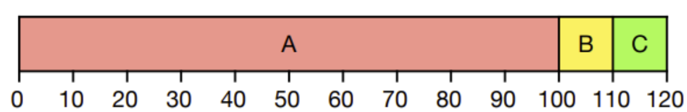
Wait time: $A = 0,\ B = 10,\ C = 20$

Average wait time $= \dfrac{0 + 10 + 20}{3} = 10$

Turnaround time: $A = 10,\ B = 20,\ C = 30$

Average turnaround time $= \dfrac{10 + 20 + 30}{3} = 20$

Job	Arrival time	CPU requirement
A	0	100
B	0	10
C	0	10



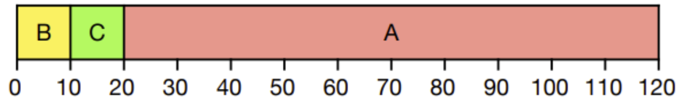
Wait time: $A = 0,\ B = 100,\ C = 110$

Average wait time $= \dfrac{0 + 100 + 110}{3} = 70$

Turnaround time: $A = 100,\ B = 110,\ C = 120$

Average turnaround time $= \dfrac{100 + 110 + 120}{3} = 110$

Job	Arrival time	CPU requirement
B	0	10
C	0	10
A	0	100



Wait time: $B = 0,\ C = 10,\ A = 20$

Average wait time $= \dfrac{0 + 10 + 20}{3} = 10$

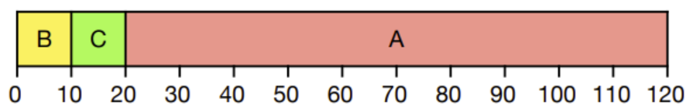
Turnaround time: $B = 10,\ C = 20,\ A = 120$

Average turnaround time $= \dfrac{10 + 20 + 120}{3} = 50$

• Shortest job first (SJF)

Shortest job first (SJF) has both nonpreemptive and preemptive versions. Let's look at the nonpreemptive version first.

Job	Arrival time	CPU requirement
A	0	100
B	0	10
C	0	10



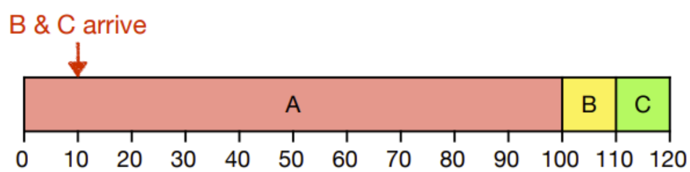
Wait time: $B = 0,\ C = 10,\ A = 20$

Average wait time $= \dfrac{0 + 10 + 20}{3} = 10$

Turnaround time: $B = 10,\ C = 20,\ A = 120$

Average turnaround time $= \dfrac{10 + 20 + 120}{3} = 50$

Job	Arrival time	CPU requirement
A	0	100
B	10	10
C	10	10



Wait time: $A = 0,\ B = 90,\ C = 100$

Average wait time $= \dfrac{0 + 90 + 100}{3} = 63.33$

Turnaround time: $A = 100,\ B = 100,\ C = 110$

Average turnaround time $= \dfrac{100 + 110 + 120}{3} = 103.33$

Even though B and C arrived shortly after A, they have to wait until A completes.

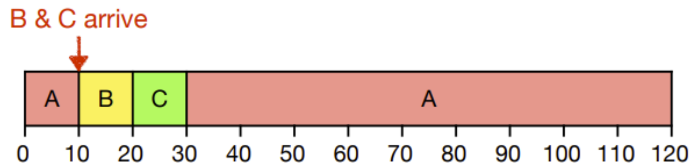
This is called the **convoy effect**, where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.

To address this concern, let's look at the **preemptive shortest job first (PSJF)**.

Whenever a new job arrives, the PSJF scheduler determines which job has the shortest remaining time, and schedules that one.

- **Preemptive shortest job first (PSJF)**
    - Preemptive shortest job first is like shortest job first
    - but if a new job comes in with a shorter runtime than the total runtime of the current job, it is run instead.

Job	Arrival time	CPU requirement
A	0	100
B	10	10
C	10	10



Wait time: $A = 20,\ B = 0,\ C = 10$

Average wait time $= \dfrac{20 + 0 + 10}{3} = 10$

Turnaround time: $A = 120,\ B = 10,\ C = 20$

Average turnaround time $= \dfrac{120 + 10 + 20}{3} = 50$

- **A new metric - response time**

PSJF looks great for batch systems.

However, in interactive systems, users would demand a short response time.

Let's define **response time** as the duration from when the job arrives in the system to the first time it is scheduled.

$$T_{response} = T_{first\_run} - T_{arrival}$$

- **Round-robin (RR)**

Round-robin (RR) is a preemptive scheduling algorithm in interactive systems.

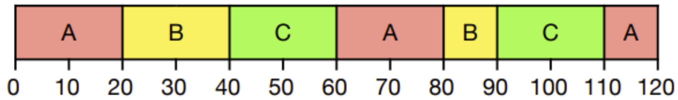
Each job is assigned a **time slice** (*a.k.a.* **quantum**).

- The time slice is the amount of time the job is allowed to run.
- At the end of the time slice, the CPU is preempted and given to the next job.
- For simplicity, let's assume all jobs have the same time slice.

Jobs are running one by one in a queue, which is called the **run queue**.

Job	Arrival time	CPU requirement
A	0	50
B	0	30
C	0	40

Time slice = 20



Wait time: $A = 70,\ B = 60,\ C = 70$

Average wait time $= \dfrac{70 + 60 + 70}{3} = 66.67$

Turnaround time: $A = 120,\ B = 90,\ C = 110$

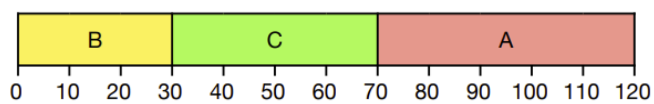
Average turnaround time $= \dfrac{120 + 90 + 110}{3} = 106.67$

Response time: $A = 0,\ B = 20,\ C = 40;$ average response time = 20

SJF vs. RR

- Recall SJF

Job	Arrival time	CPU requirement
A	0	50
B	0	30
C	0	40



Wait time: $A = 70,\ B = 0,\ C = 30$

Average wait time $= \dfrac{70 + 0 + 30}{3} = 33.33$

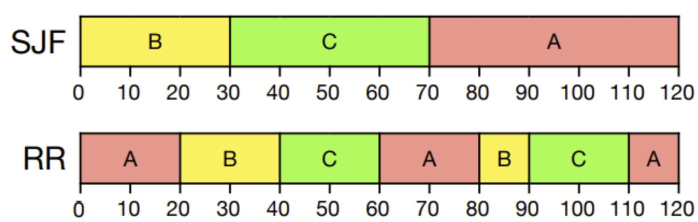
Turnaround time: $A = 120,\ B = 30,\ C = 70$

Average turnaround time $= \dfrac{120 + 30 + 70}{3} = 73.33$

Response time: $A = 70,\ B = 0,\ C = 30;$ average response time = 33.33

- SJF vs. RR

Job	Arrival time	CPU requirement
A	0	50
B	0	30
C	0	40



	SJF	RR
Average wait time	33.33	66.67
Average turnaround time	73.33	106.67
Average response time	33.33	20
Number of context switches	2	6

Typically, RR has worse CPU efficiency than SJF.

However, jobs on a RR scheduler are more responsive.

- You won't feel that a job freezes because it's on the CPU from time to time.
- Therefore, it's more suitable for interactive systems.

Typically, **RR** has **worse CPU efficiency** than **SJF**.

However, jobs on a RR scheduler are **more responsive**.

- You won't feel that a job **freezes** because it's on theCPU from time to time.
- Therefore it's more suitable for **interactive systems**.

- Trade-offs

This is an inherent trade-off between performance and fairness.

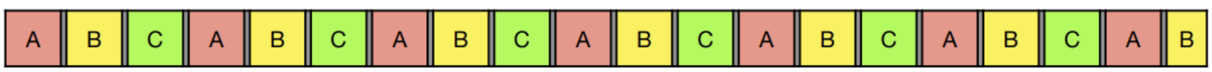
A fair scheduler (such as RR) evenly divides the CPU among active jobs on a small time scale, at the cost of turnaround time.

Most ordinary users run a lot of interactive jobs on modern operating systems.

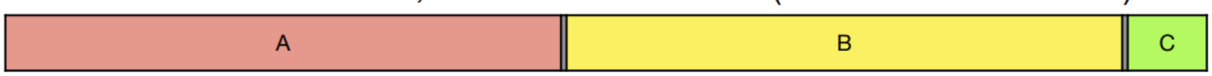
- They value responsiveness more than CPU efficiency.

Another trade-off comes from context switching. It's relatively slow.

Case 1: time slice = 10ms, context switch = 1ms. (~10% of time is wasted.)



Case 2: time slice = 100ms, context switch = 1ms. (<1% of time is wasted.)



- Short time slice ⇒ many context switches ⇒ low CPU efficiency.
- Long time slice ⇒ poor response ⇒ "sluggish."

- Priority scheduling

Each job is assigned a priority.

The scheduler always chooses the job with the highest priority to run.

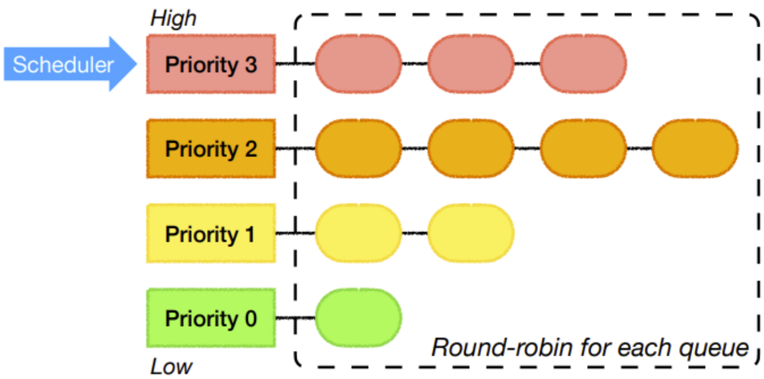
Priorities can be static or dynamic.

Static priority means that a job is assigned a fixed priority when it is submitted to the system.

- **Example:** a background email process should get a lower priority than a real-time video game process.

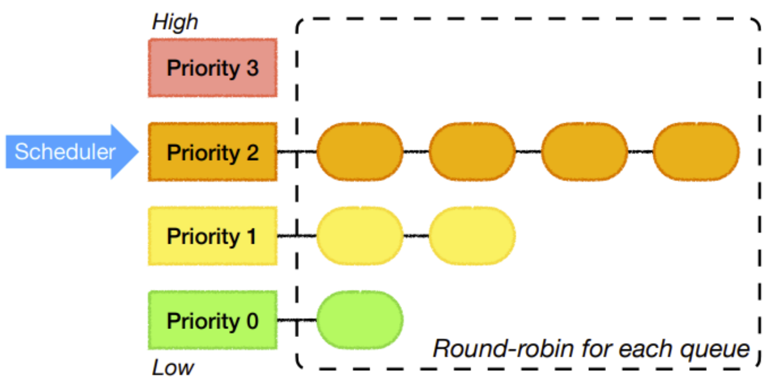
Dynamic priority means that a job's priority may be changing throughout its life in the system.

- Static priority scheduling

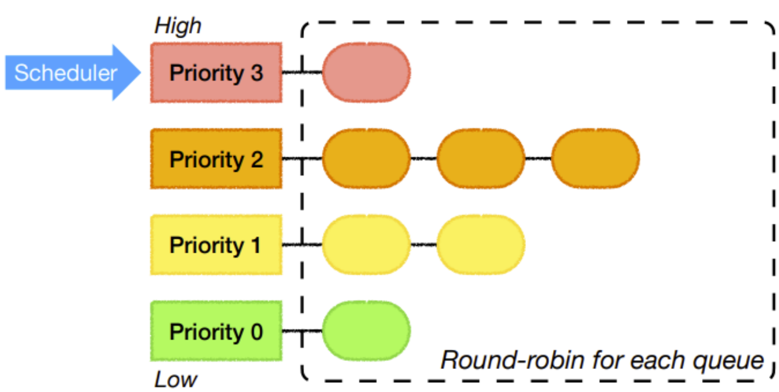


Jobs with Priority 3 are scheduled first.

They should be short-lived to prevent starvation of lower-priority jobs.



When there is no job with Priority 3 left, jobs with Priority 2 will be scheduled.



When a higher-priority job appears, the running job may or may not be preempted.

- Limitations

Limitations

High-priority jobs may run for a prolonged period, or even indefinitely.

Low-priority jobs may starve to death.

- **Rumor:** when the IBM 7094 mainframe at MIT was shut down in 1973, people found a low-priority process submitted in 1967 had not yet been run.

It does not differentiate between CPU-bound and I/O-bound jobs.

- I/O-bound jobs spend most of their time waiting for I/O to complete.
- When such a job wants the CPU, we'd better schedule it immediately to let it start its next I/O request.
- In that way, I/O requests can proceed in parallel with another process actually computing.

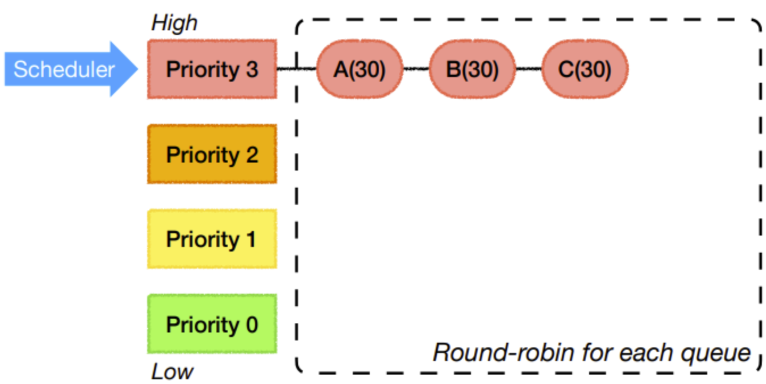
- Dynamic priority scheduling

There is no standard way to assign priorities dynamically.

Let's look at an **example** policy.

Rules:

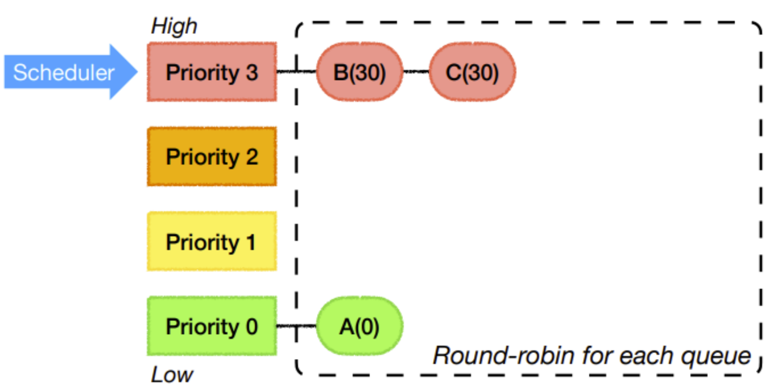
- All jobs start running at Priority 3 with time slice = 30ms.
- A job is preempted if its time slice is used up or it starts waiting for I/O.
- When a job is preempted, its priority is changed to $\left\lfloor \frac{\text{its time slice left}}{10\text{ms}} \right\rfloor$.



All jobs start running at Priority 3 with (time slice = 30ms).

A is scheduled first.

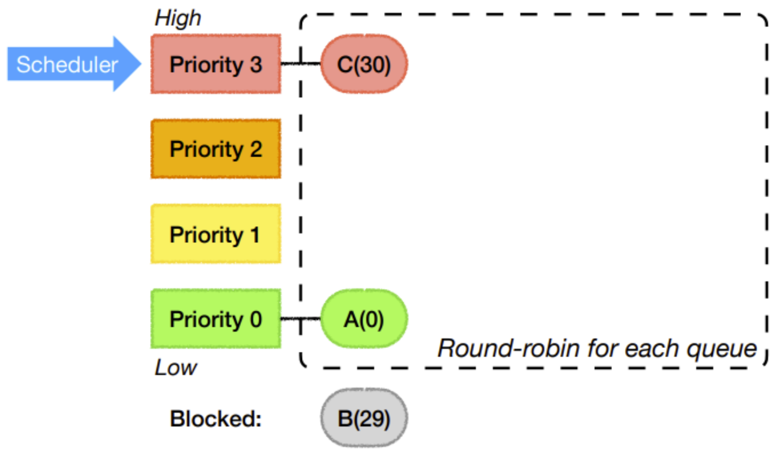
Let's say A is a CPU-bound process and will use up its time slice.



A has used up its time slice and its priority is changed to 0.

B is scheduled next.

Let's say B is an I/O-bound process, and it's blocked right after its execution starts.

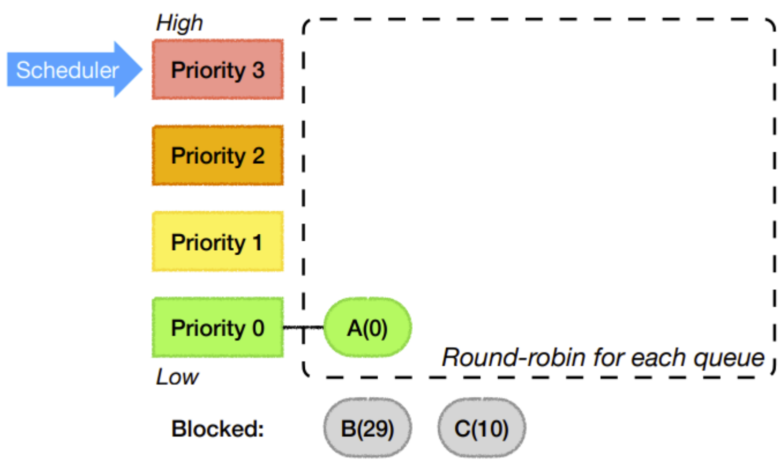


B is removed from the run queue until the I/O completes.

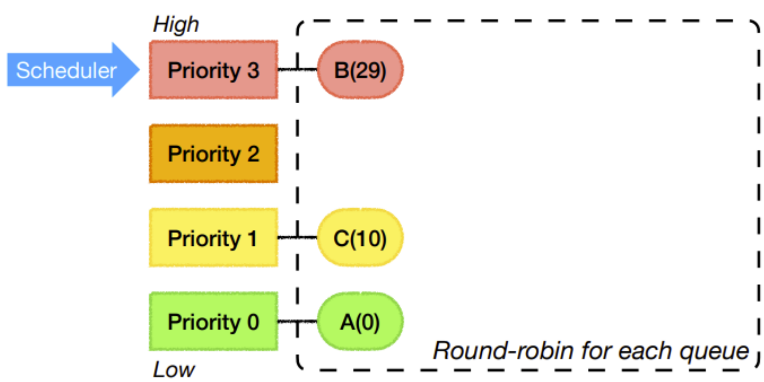
C is scheduled next.

Let's say C is a mixed process.

It runs on CPU for 20ms and starts waiting for I/O.



C's priority is changed to 1, and C is removed from the run queue.



When B comes back, since it has not used any of its time slice before, B will join the run queue of Priority 3.

When C comes back, it will join the run queue of Priority 1.

In this way, I/O-bound jobs will have a higher priority to access the CPU.

- **Can we do better?**