FIT1047 - Week 7

Operating Systems (Part 2)

MONASH University

Recap

An Operating System

- Makes computers easier to use
  - For the end user
  - For the programmer
- Provides ***illusion*** of multiple processes running simultaneously
  - By ***virtualising*** the resources (CPU, memory, disk, etc.)
  - By ***protecting*** the system from malicious or buggy programs

Today’s goals

Process scheduling

- When to switch between processes

Virtual memory

- How to virtualise the RAM

Process scheduling (when to switch)

Scheduling policies

We've seen the **mechanisms** for process switching:

- User and kernel modes in the CPU
- Context switching
- Timer interrupts (for preemptive timesharing)

Now we need to look at **policies:**

- How long is each process allowed to run?

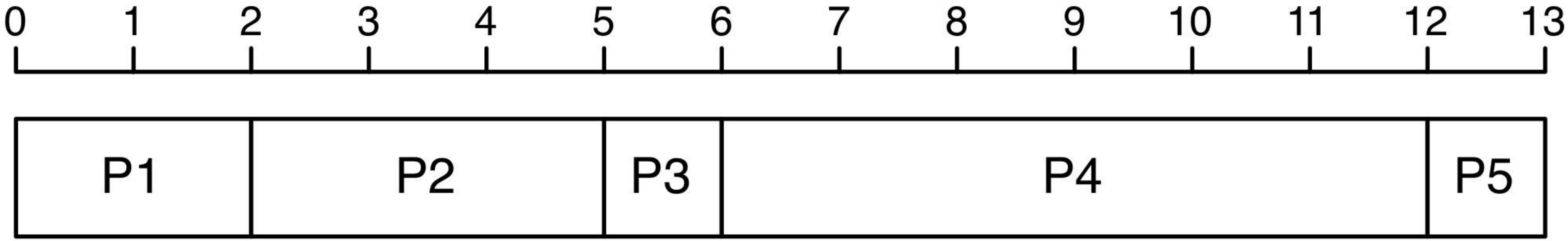
Scheduling

Can aim for different goals:

- Turnaround time:
  How long does a process take from arrival to finish?
- Fairness:
  All processes get a fair share of processing time

Poor turnaround

Schedule processes in some arbitrary order. E.g. first come first served:

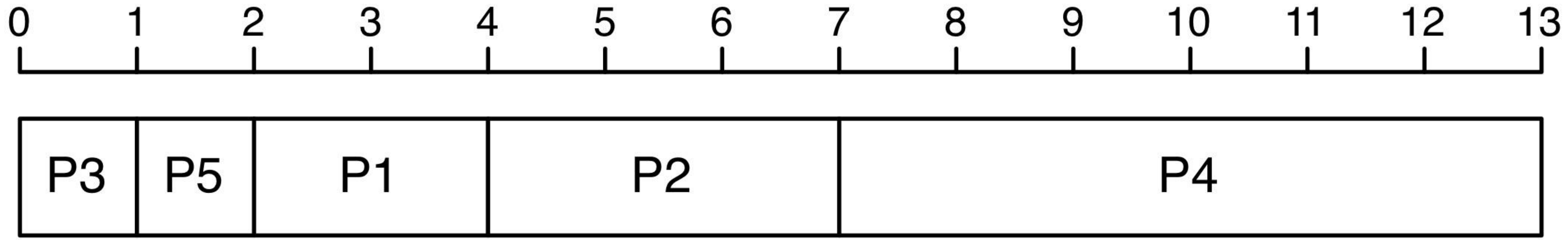


$$\frac{2+5+6+12+13}{5} = 7.6$$

On average, each process waits 7.6 time units to complete.

Good turnaround

Order processes from shortest to longest:



$$\frac{1+2+4+7+13}{5} = 5.4$$

Average goes down to 5.4 time units!
(this is in fact optimal)

Turnaround scheduling

Works well if

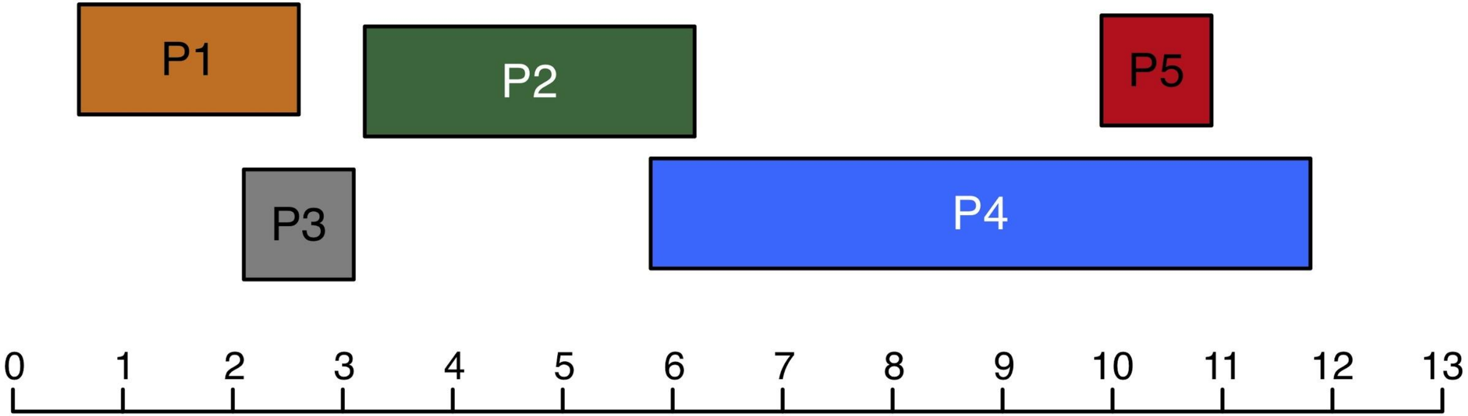
- All processes have known duration
- All processes are ready to run at the same time
- The goal is to reduce average turnaround

Unrealistic in modern operating systems:

- We don’t know how long a process will take to finish
- Some processes don’t finish at all as long as the computer is running
- We want multiple processes to run simultaneously

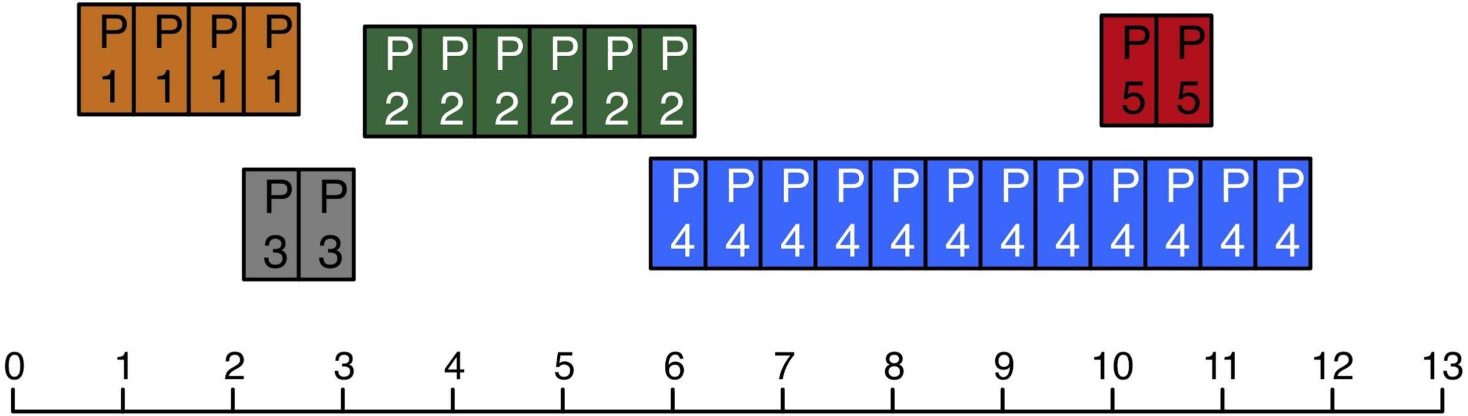
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



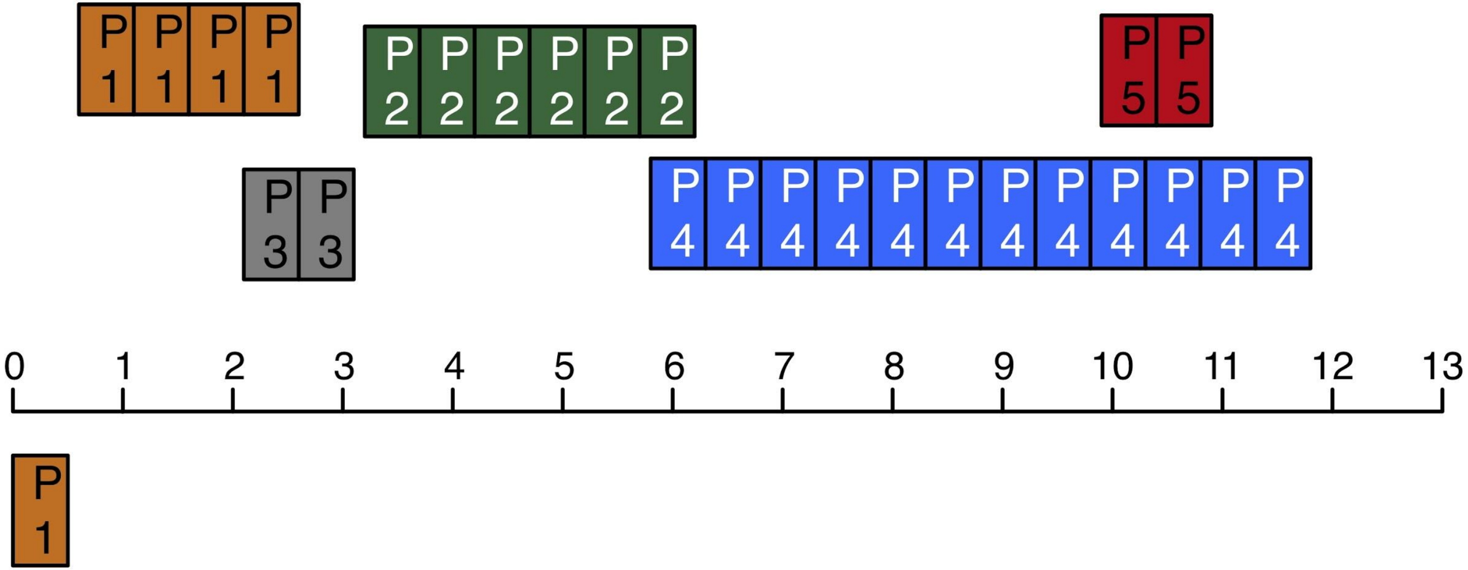
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



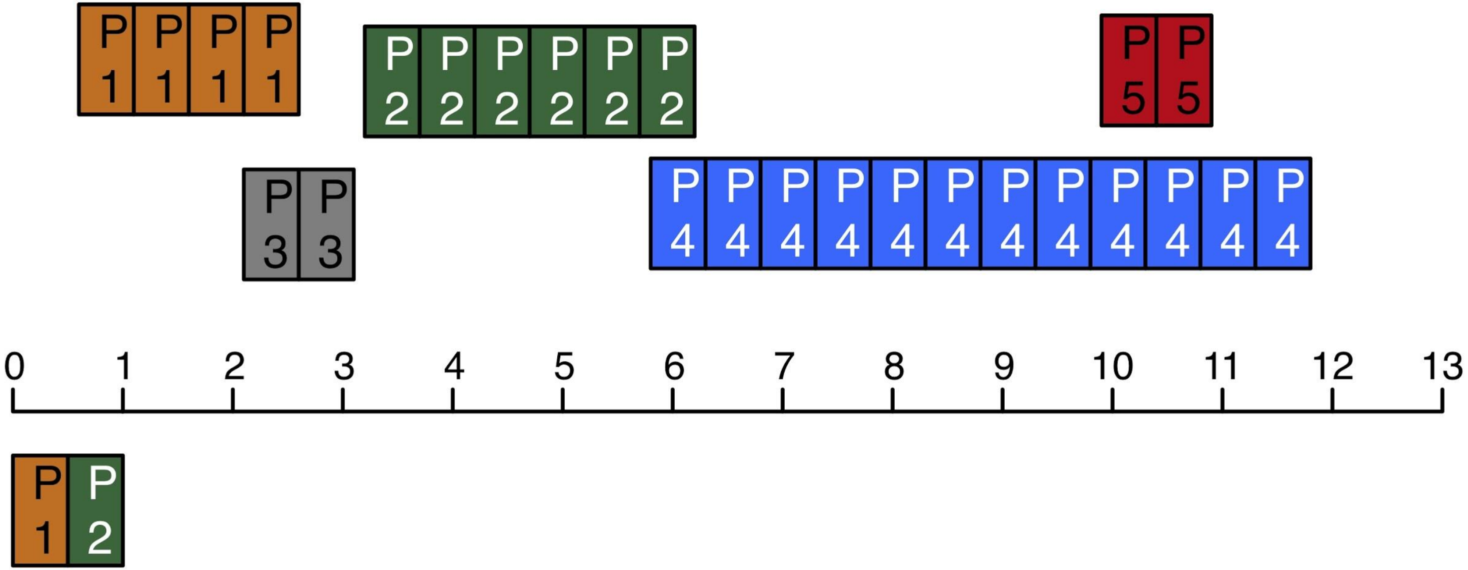
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



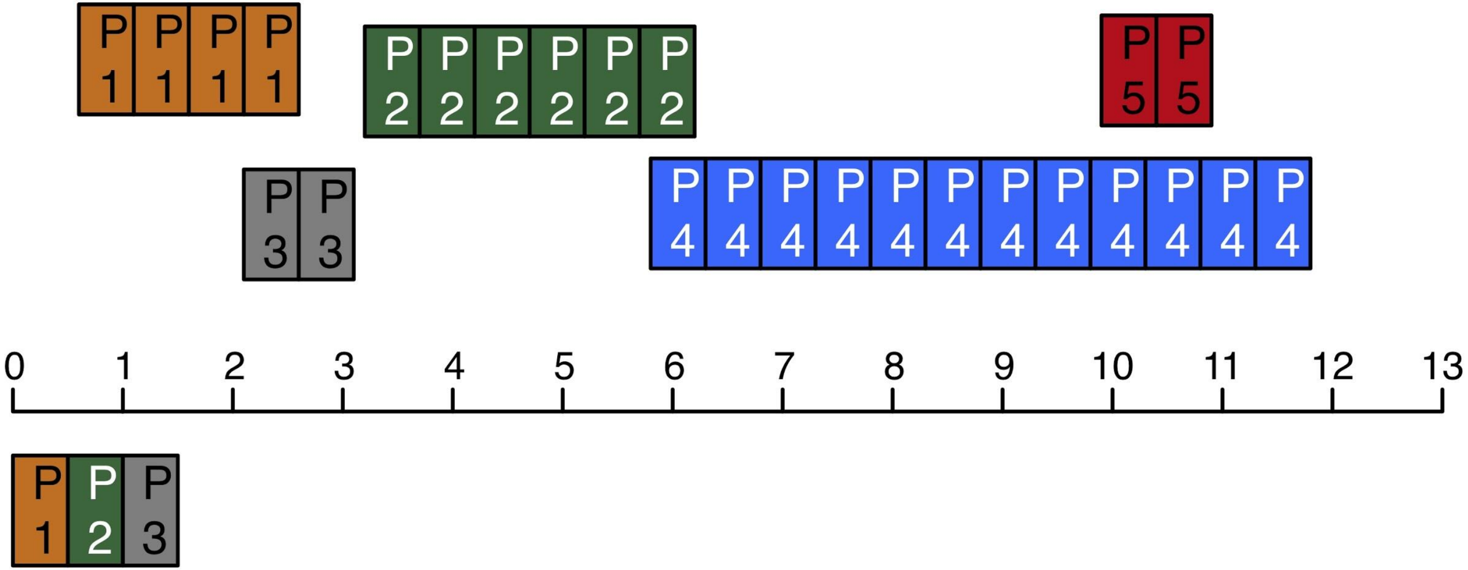
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



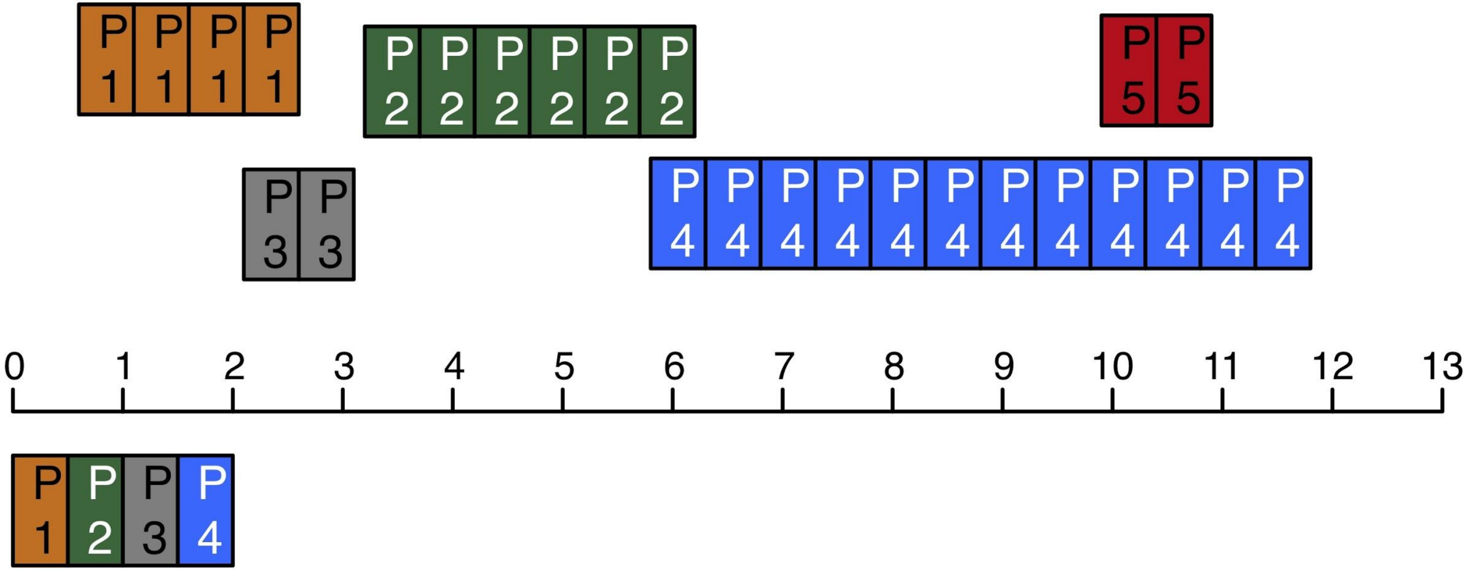
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



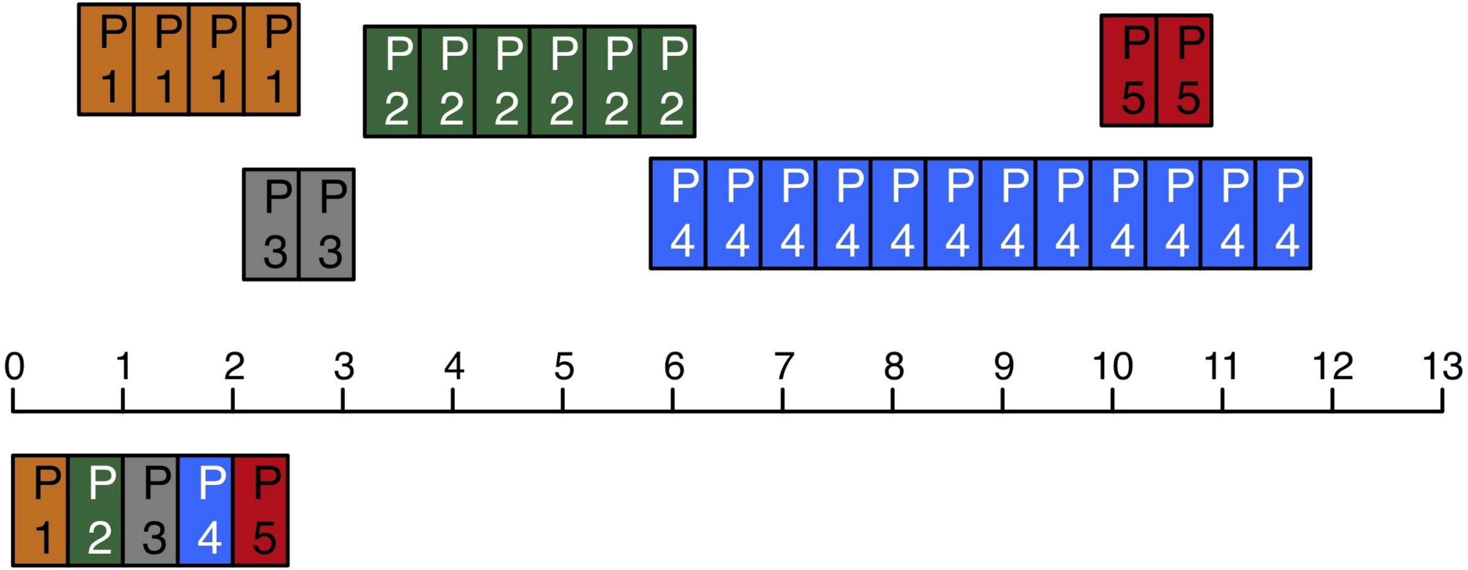
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



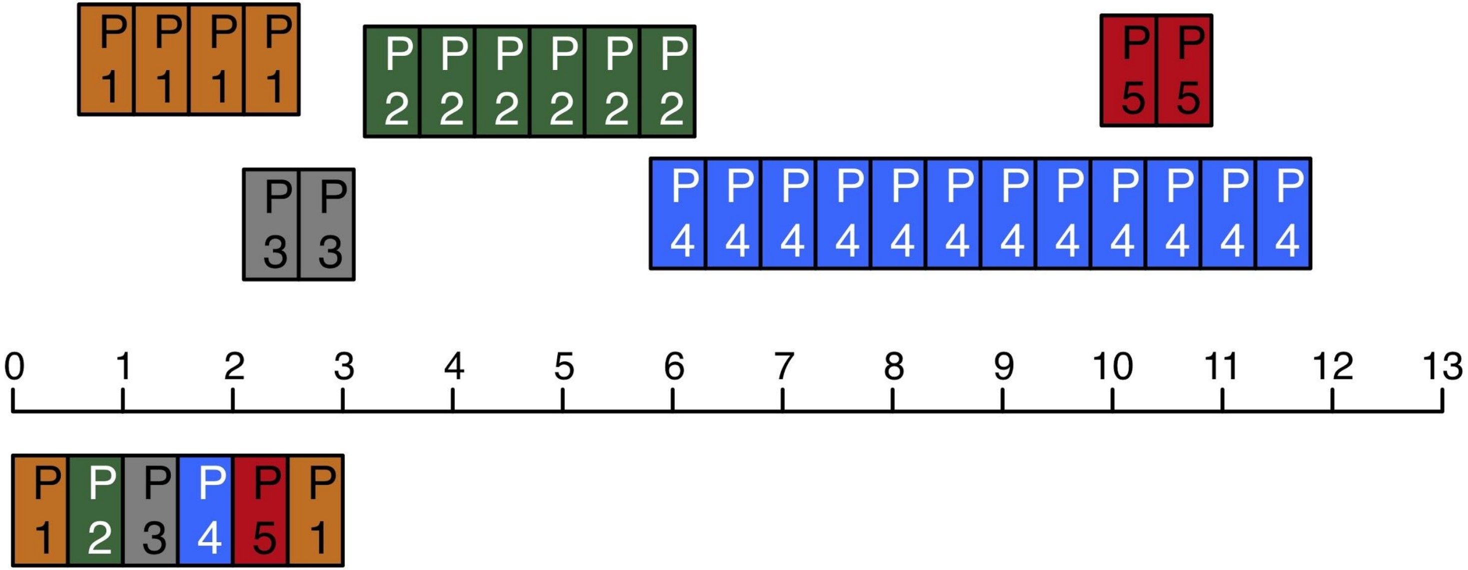
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



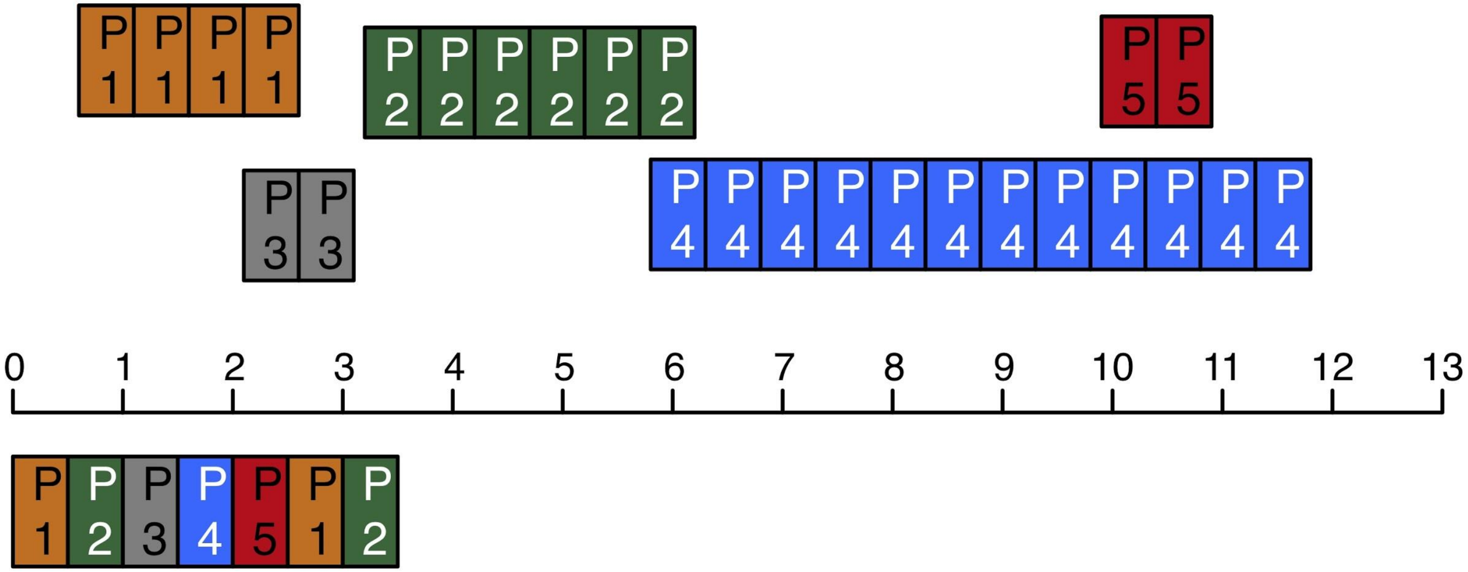
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



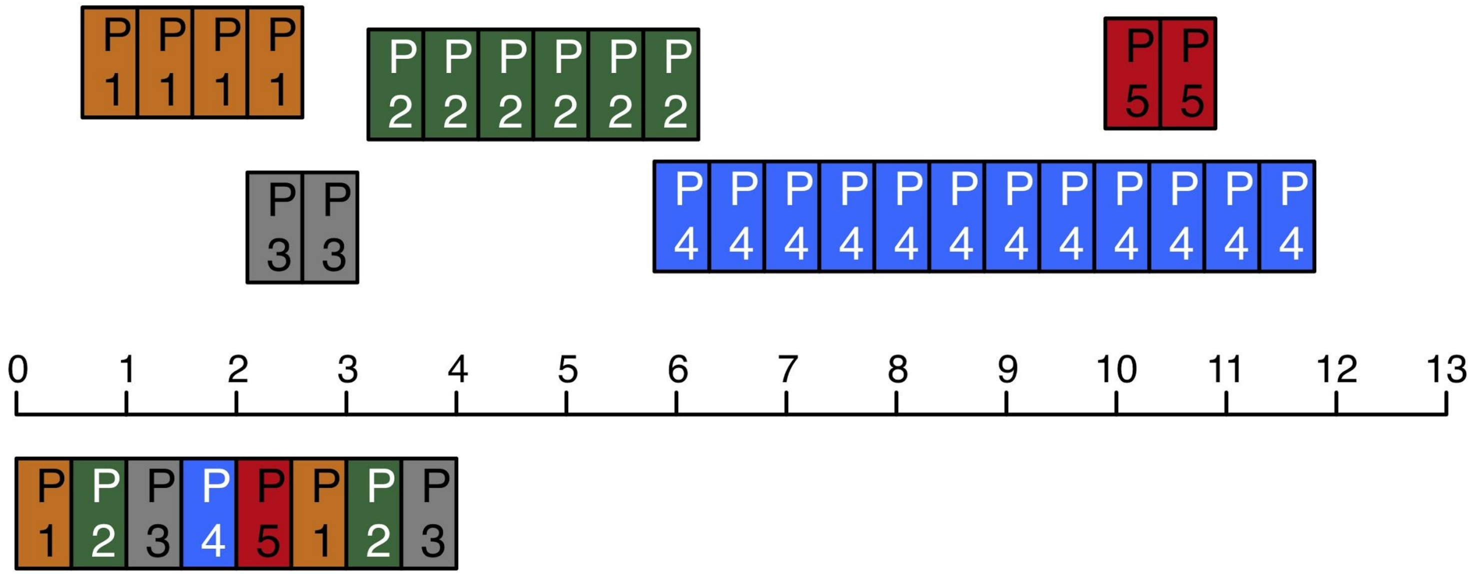
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



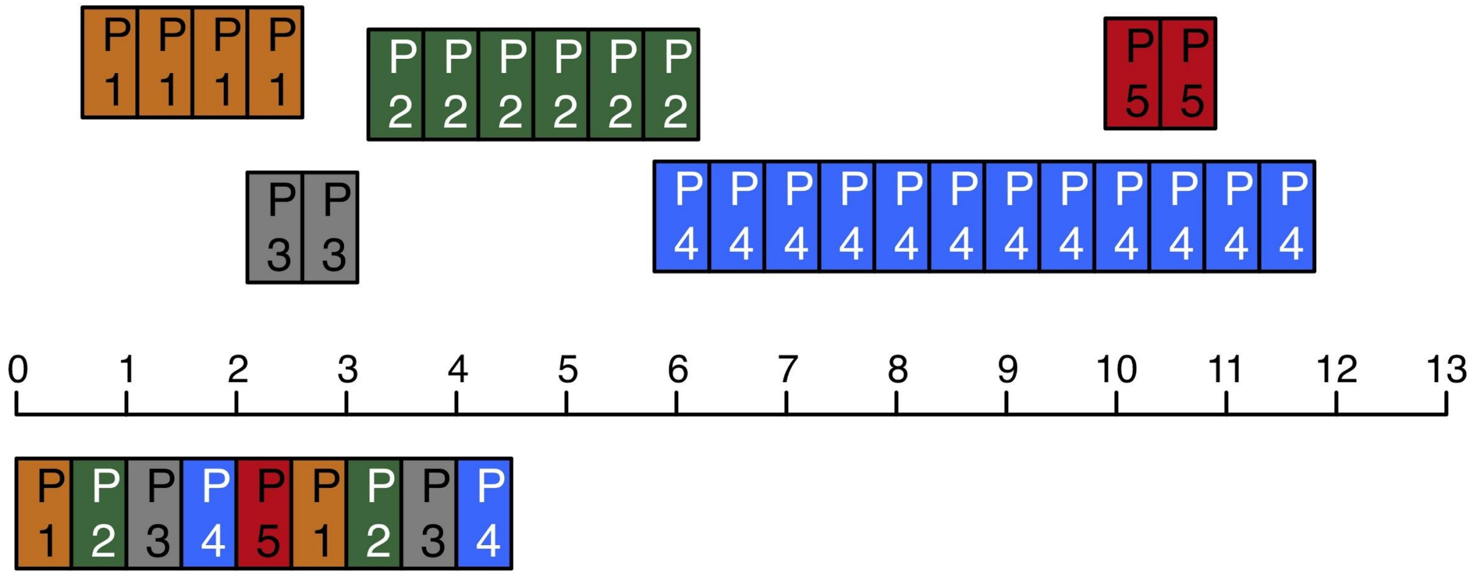
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



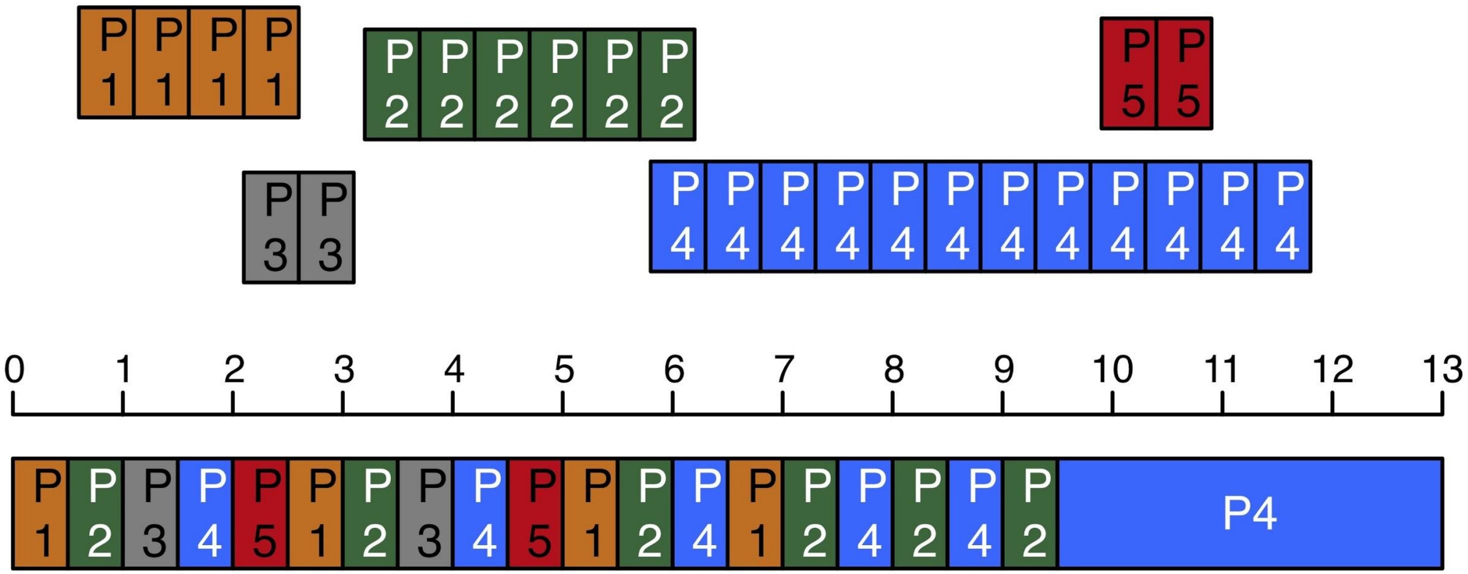
Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:

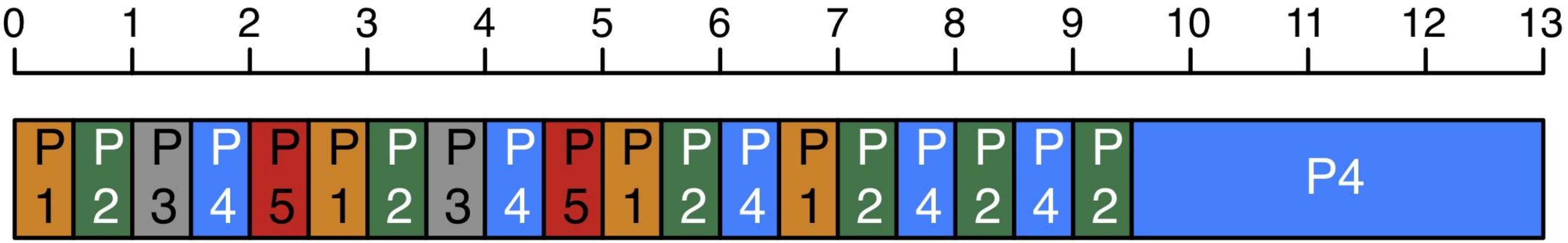


Fairness

Allocate ***time slices*** for each process, schedule them in a ***round-robin*** fashion:



Fairness



The shorter the time slice, the fairer the schedule!

But: each context switch takes time. OS needs to find the right compromise.

Summary: scheduling

OS needs to decide ***when*** to run ***which*** process.

Modern OSs use a form of round-robin scheduling.

Virtual Memory

Virtual Memory

Reminder: user programs run directly on the CPU.

User mode: programs can only access I/O through ***system calls***.

How can we prevent a process from accessing the memory of another process?

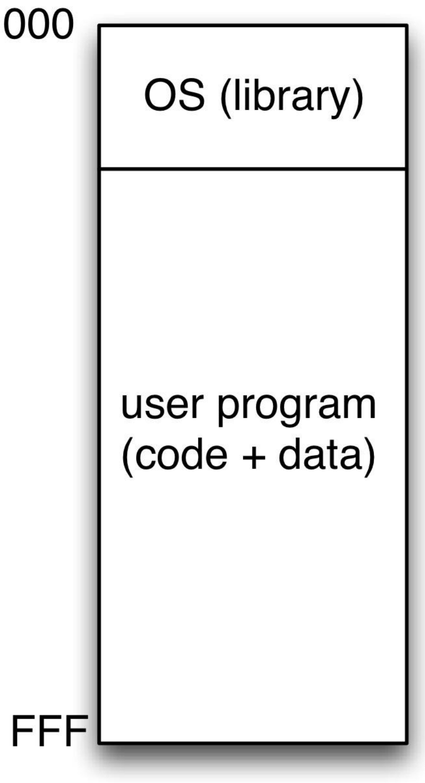
Virtualise the memory!

Virtual Memory

Goal: no process can access any memory except its own.

Mechanism: each process has its own ***address space***.

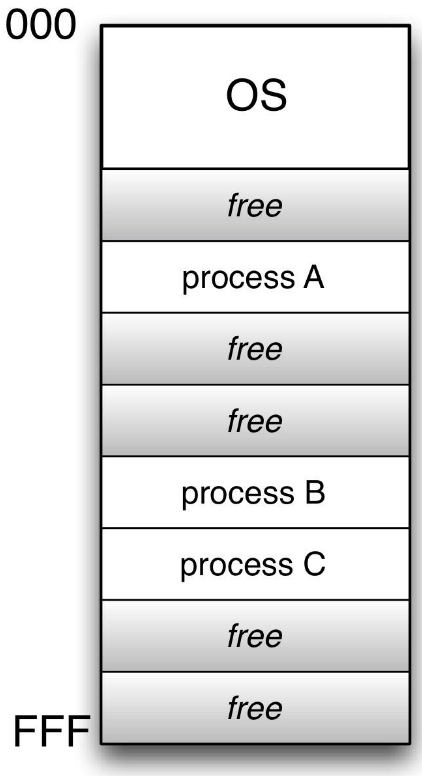
Single process



Early computers:

- OS is just a library of subroutines
- Process has entire memory to itself

Multiprogramming



- Every process gets a fixed block of memory
- But for the process it looks as if that block starts at address 0
- OS/hardware ensures *protection*

How can we make this work?

Virtual addresses

Each process ***"thinks"*** its addresses start at 0.

I.e., programmers can write code as if addresses start at 0.

OS/Hardware ***translates*** these ***virtual*** addresses to ***physical*** memory locations.

We call this abstraction ***virtual memory.***

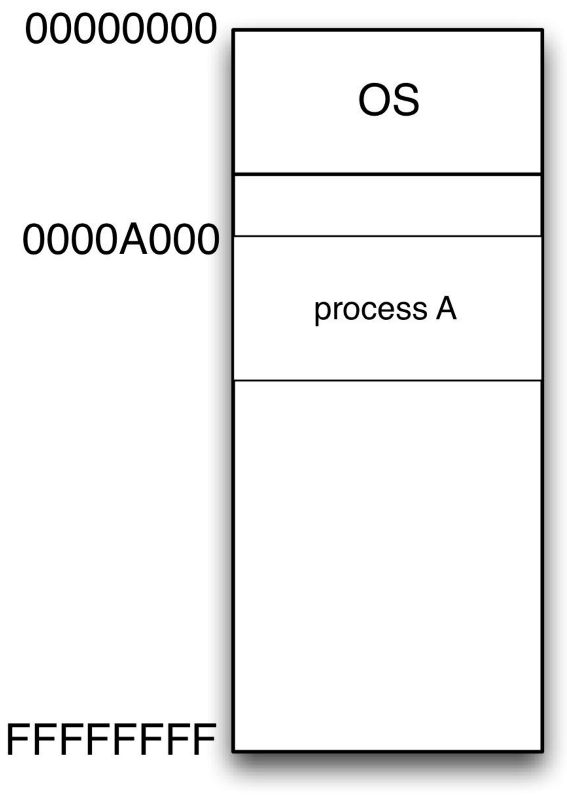
Virtual memory: simple approach

Each process has a ***base address*** X, the start address of its address space in physical memory.

When a process accesses a *virtual* address Y, the CPU must access the physical address X+Y.

- Add a new register ***B*** (base register)
- When OS switches between processes: set ***B*** to process base address
- All instructions use ***B***
- E.g. Load X now loads from address ***B***+X

Virtual memory: simple approach




```
Load  200
Add   300
Store 200
Halt
```

When switching to process A:

- Set base register ***B*** to 0000A000
- Load from 0000A000+200
- Add from 0000A000+300
- Store into 0000A000+200

Memory protection

Using base register:

- Process cannot access any memory lower than ***B***

How to protect from access beyond its address space?

Similarly simple fix:

- Add ***bounds*** register
  (largest address that the current process may access)
- All instructions check that memory access is between ***B*** and ***bounds***
- If access is outside of address space, raise an error
  (an interrupt that gives control back to the OS)

Virtual memory (realistic systems)

Problems with the simple approach:

- Each process needs to get a fixed block
- No way to dynamically shrink or enlarge

Modern approach:

- Hardware and OS allocate smaller chunks of memory (called ***pages***)
- Each process has a *set* of pages it can access
- Pages can be added and removed from processes
- Physical addresses of pages are mapped dynamically into process address space

Huge virtual memory

RAM is limited

- E.g. 8GB may not be enough for all processes *at the same time*

Virtual memory is the solution:

- Save currently *unused* pages to external storage (hard disk)
- When a process tries to access an unavailable page: hardware creates interrupt (called "page fault")
- OS handles interrupt and can load page back from external storage, ***swapping*** it for an unused page

Works very well if swapping is not too frequent!

Summary

Scheduling

- Round-robin style scheduling with time slices to achieve fairness

Virtual memory

- Programmers can write code as if their program had entire memory to itself
- OS/hardware map virtual addresses to physical addresses