CPU Scheduling

Xin Liu

xl24j@fsu.edu

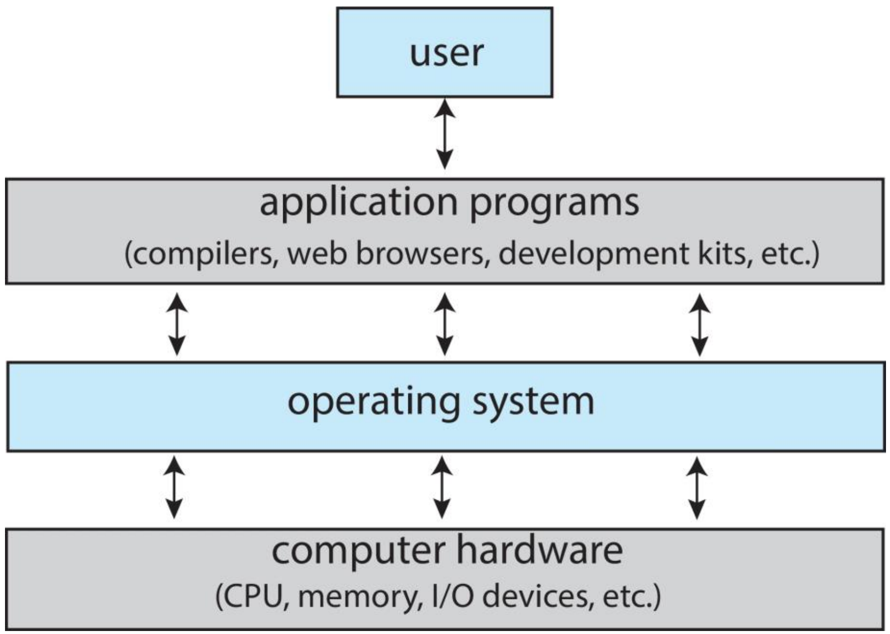
COP 4610 Operating Systems

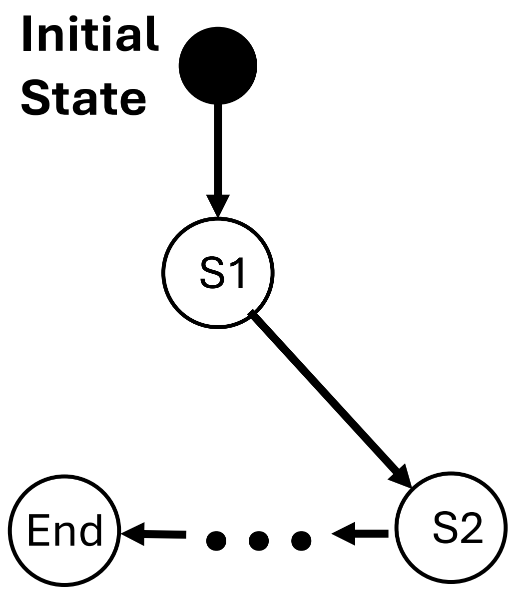
Outline

- How is the first process created?
- How is the second process created?
- How to manage processes?

Recap

- CPU is a state machine.
- A program (whether an OS or application) running on a CPU is inevitably a state machine.





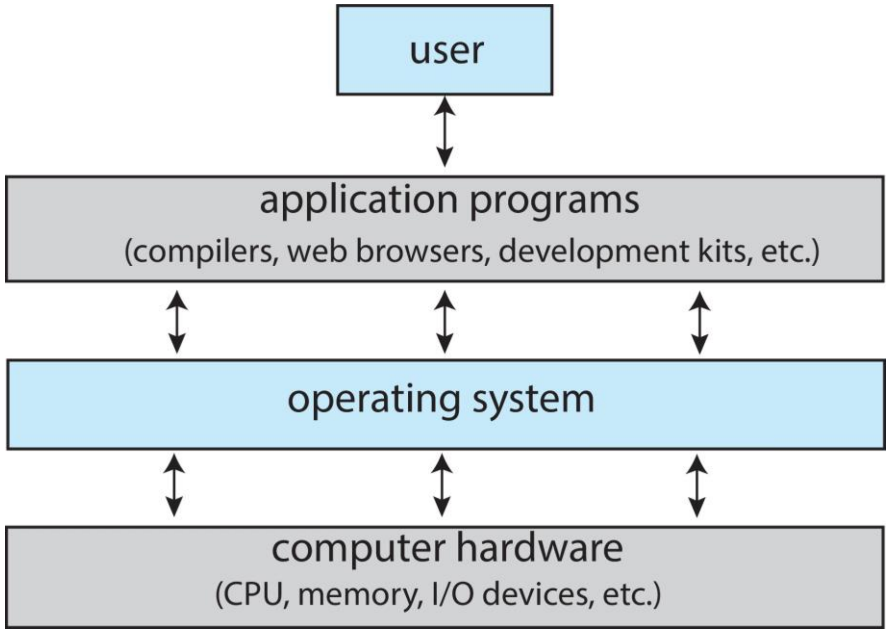
If we are given the initial state, we can deduce what the next state will be. The key is knowing:

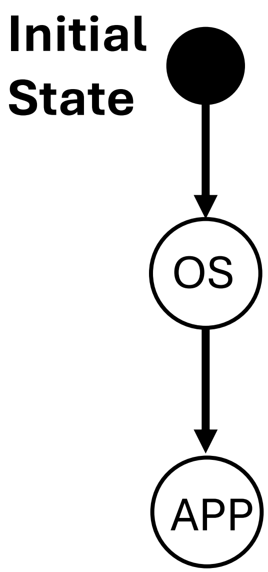
Where is the initial state?

The Beginning of Everything

- An application runs on the OS

  -> OS creates the application's initial state





Who creates the initial state of OS?

Who creates the initial state of OS?

- Well…it's a long story…
  - It starts with a simple computing machine

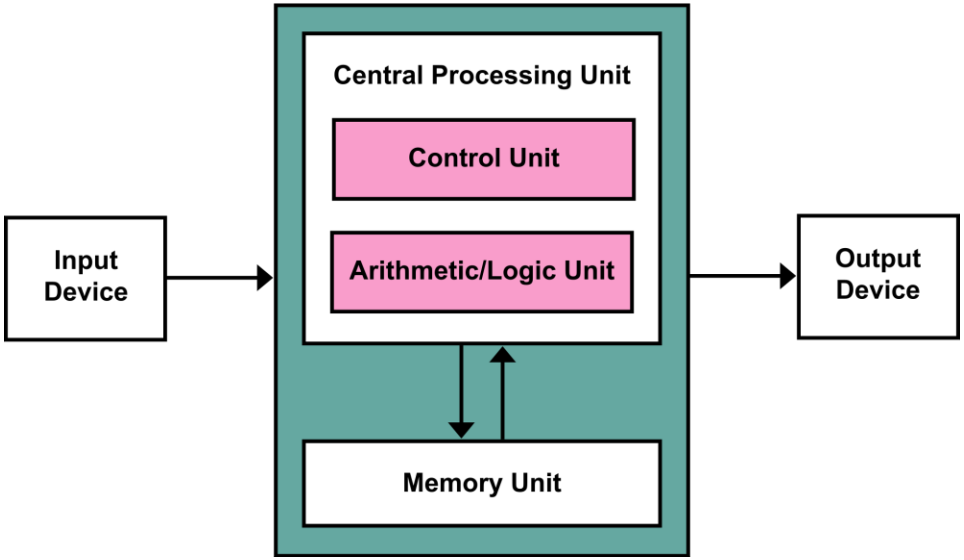
Long, Long,

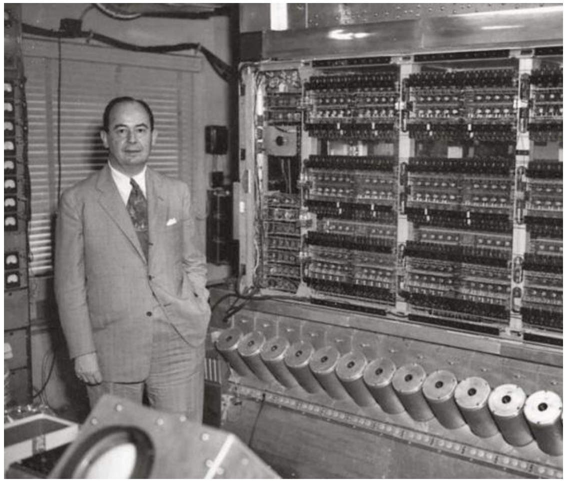
Long Ago…

(During the 1940s)

Long, Long, Long Ago…(during the 1940s)

- John von Neumann invented ***von Neumann computer architecture***
  - A CPU
  - A memory unit
  - I/O devices (e.g., disks and tapes)



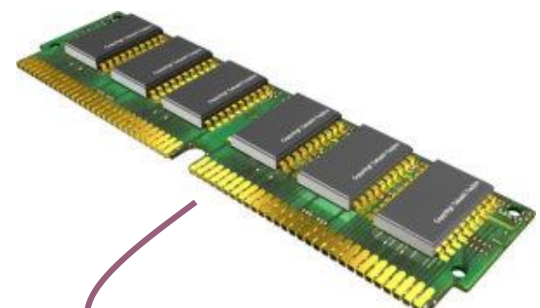


von Neumann Architecture

In von Neumann Architecture,

- Programs are stored on storage devices

- Programs are copied into memory for execution



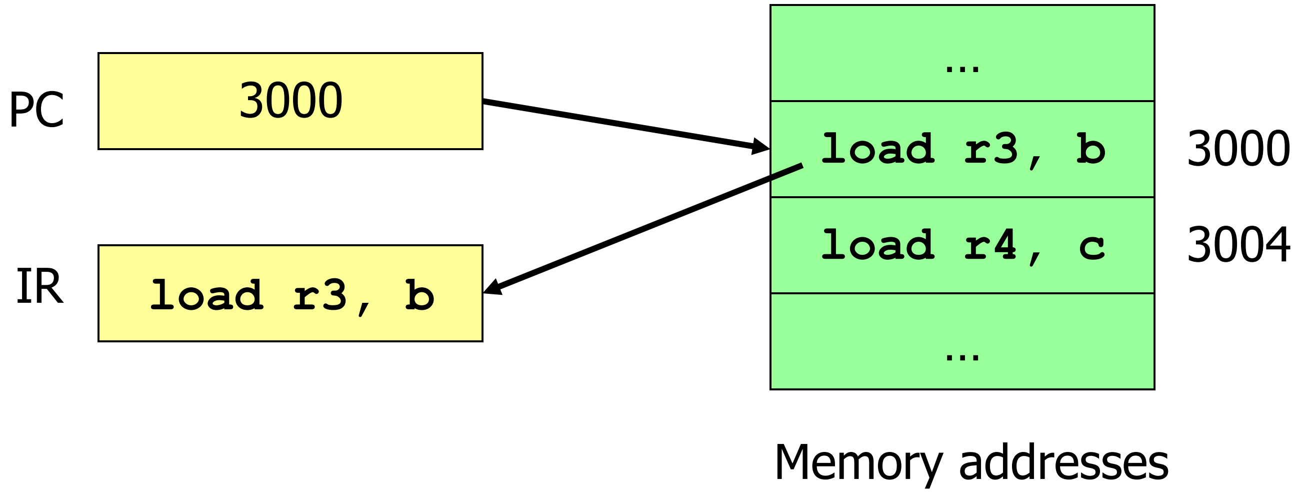
- CPU reads each instruction in the program and executes accordingly

A Simple CPU Model

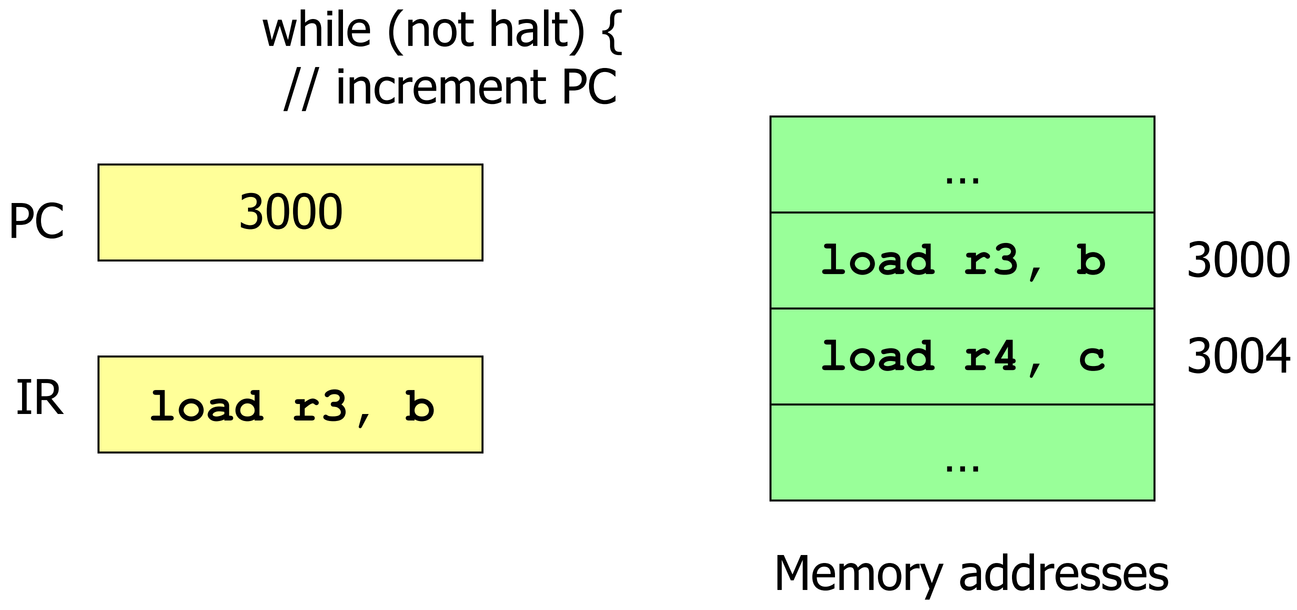
- ***Fetch-execute algorithm***
- The CPU fetches the instruction:
  - The program counter (PC) is loaded with the address of the instruction
  - The instruction register (IR) is loaded with the instruction from the address
- The CPU decodes the instruction.
- The CPU executes the instruction.

The CPU fetches the instruction

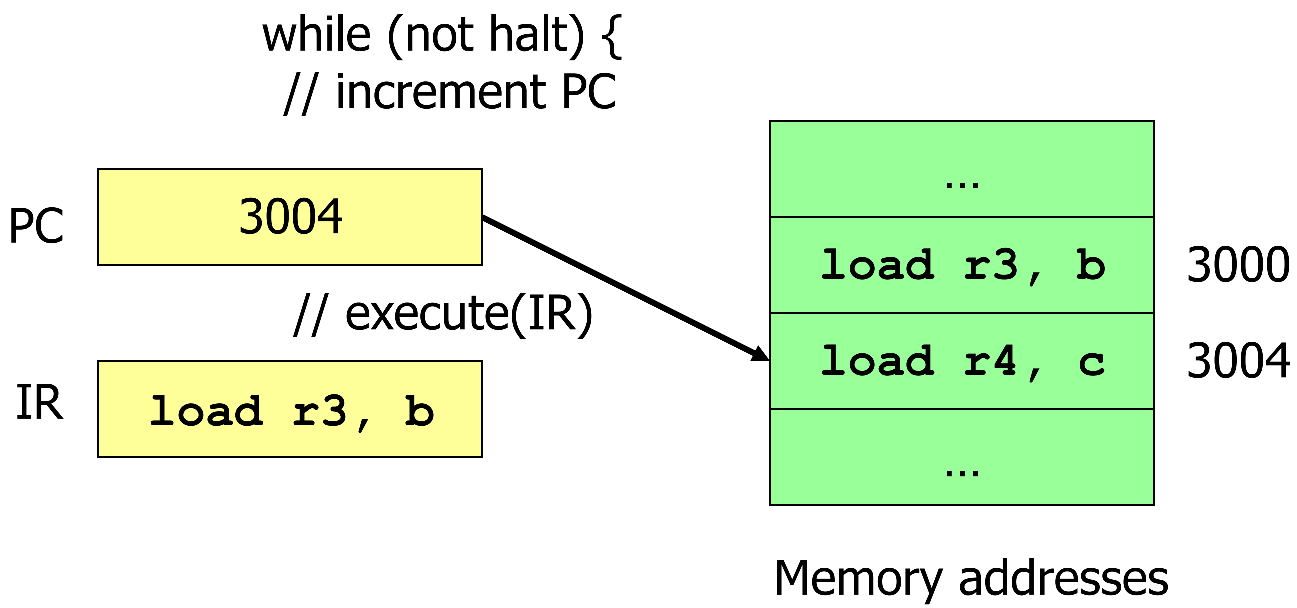
PC = <address of the first instruction>



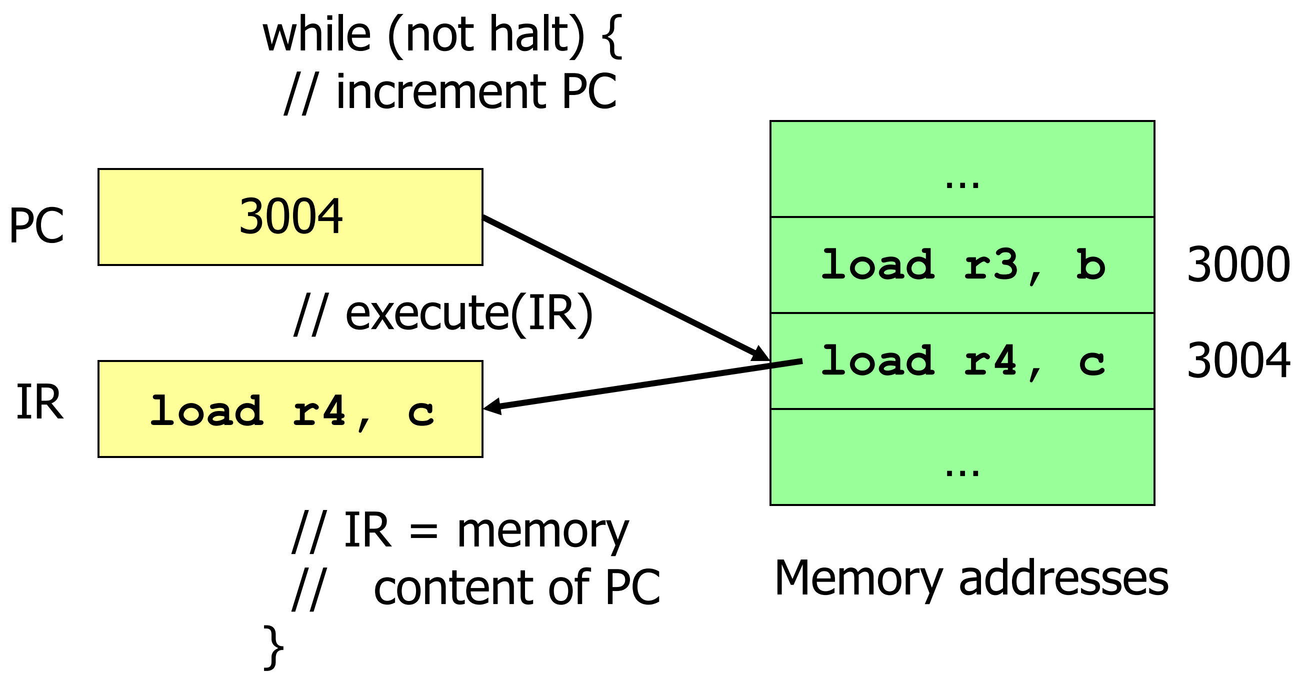
The CPU fetches the instruction



Fetch-Execute Algorithm



The CPU fetches the instruction



OS Booting Sequence

- The address of the first instruction is fixed
- It is stored in read-only-memory (ROM)

Booting Procedure for i386 Machines

- On i386 machines, ROM stores a ***Basic Input/Output System (BIOS)***
  - BIOS contains information on how to access storage devices
- Being replaced with ***United Extended Firmware Interface (UEFI)***
  - To access storage > 2TB

BIOS Code

- Performs Power-On Self Test (POST)
  - Checks memory and devices for their presence and correct operations
  - If you are old as I am, you might remember:
    - For ancient computers, you will hear memory counting, which consists of noises from the hard drive and CDROM, followed by a final beep

After the POST

- The ***master boot record (MBR)*** is loaded from the ***boot device*** (e.g., a hard drive configured in BIOS)
- The MBR is stored at the first logical sector of the boot device that
  - Fits into a single 512-byte disk sector (***boot sector***)
  - Describes the physical layout of the disk (e.g., number of tracks)
- MBR is being replaced by GUID Partition Table (***GPT***) for 64-bit addressing

After Getting the Info on the Boot Device

- BIOS loads a more sophisticated loader from other sectors on disk
  - Under old Linux, this sophisticated loader is called ***LILO (Linux Loader)***
  - It has nothing to do with Lilo and Stitch
  - Linux uses ***GRUB (GRand Unified Bootloader)*** nowadays

- The more sophisticated loader loads the operating system

More on OS Loaders

- LILO
  - Partly stored in MBR with the disk partition table
    - A user can specify which disk partition and OS image to boot
    - Windows loader assumes only one bootable disk partition
  - After loading the kernel image, LILO sets the kernel mode and jumps to the entry point of an operating system

Booting Sequence After A CPU Reset

- A CPU jumps to a fixed address in ROM,
- Loads the BIOS (UEFI),
- Performs POST,
- Loads MBR (GPT) from the boot device (hard drive),
- Loads an OS loader (LILO, GRUB),
- Loads the kernel image,
- Sets the kernel mode, and
- Jumps to the OS entry point -- ***init***

Read Latest Linux Kernel Code:
https://elixir.bootlin.com/linux/v6.10.9/source/init/main.c#L1523

Important

Can We Really See Every Instruction Executed After a CPU Reset?

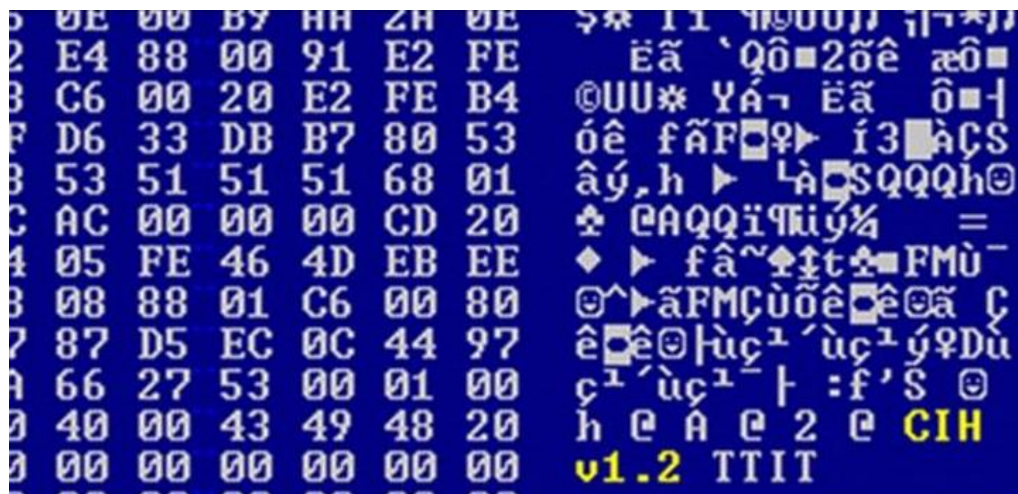
- **"Talk is cheap. Show me the code."**— Linus Torvalds
- **Computer System Axiom:**
  - *If you can imagine it, someone has already done it.*

- **Simulation Option: QEMU**
  - Createed by legendary hacker and genius programmer **Fabrice Bellard**
  - **QEMU**:A fast and portable dynamic translator *(USENIX ATC'05)*
  - Powers Android Virtual Devices, VirtualBox, and more *(All built on QEMU)*

- **Real Machine Option: JTAG** (Joint Test Action Group) **Debugger**
  - A series of **physical debugging registers**
  - Allows integration with **gdb** (!!!)

A Side Story: Firmware Virus (1998)

- **Firmware is usually read-only** (though…)
  - The Intel 430TX (Pentium) chipset allows writing to Flash ROM.
    - By writing a specific sequence to Flash BIOS, the Flash ROM becomes writable.
    - This provides a channel for firmware updates.
  - Getting this sequence isn’t too difficult.
    - It seems the documentation even provides it. 🤔 Boom…
- Chen Ing-Hau, the author of CIH, was arrested but not convicted.



Linux Initialization

- Set up a number of things:
  - **Trap table**: Handles system calls and exceptions.
  - **Interrupt handlers**: Manage external hardware interrupts.
  - **Scheduler**: Decides which processes run and when.
  - **Clock**: Manages system time and scheduling.
  - **Kernel modules**: Loads additional functionality for the kernel.
  - …
  - **Process manager:** Controls process creation, termination, and state changes.

How is the first process created?

Process 1:

- Is instantiated from the ***init*** (now ***systemd*** for parallelism) program
- Is the ancestor of all processes
- Controls transitions between ***runlevels***
- Executes startup and shutdown scripts for each runlevel

Runlevels

- Level 0: shutdown
- Level 1: single-user
- Level 2: multi-user (without network file system)
- Level 3: full multi-user
- Level 5: X11
- Level 6: reboot

Process Creation

- Via the ***`fork`*** system call family

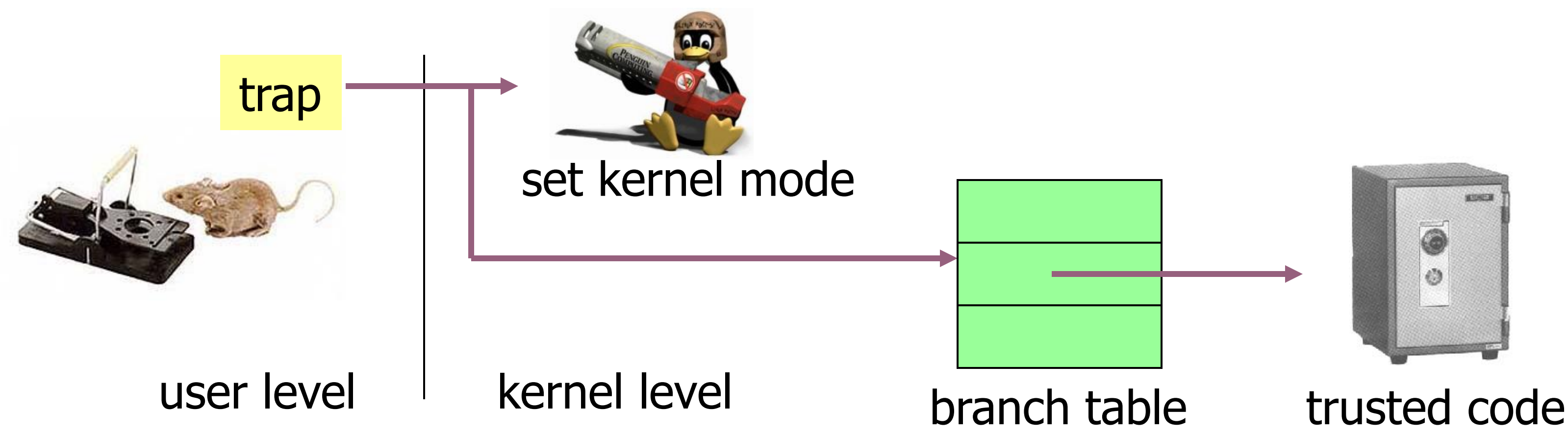
Before we discuss process creation, a few words on system calls…

System Calls

- ***System calls*** allow processes running at the ***user mode*** to access kernel functions that run under the ***kernel mode***
- Prevent processes from doing bad things, such as
  - Halting the entire operating system
  - Modifying the MBR

UNIX System Calls

- Implemented through the ***trap*** instruction
  - The program in user mode makes a system call request (like read).
  - trap instruction is triggered, switching the CPU to kernel mode.
  - The kernel takes control and looks up the system call number.
  - It uses a branch table (system call table) to find the correct service.
  - The trusted code is executed to perform the requested operation.
  - The system then returns to user mode, and the program continues running.



Important

This Is How Simple Operating Systems Are!

CPU Reset → Firmware (BIOS/UEFI) → Boot Loader (MBR, LILO/GRUB)

↓

Kernel_start()

↓

Process 1

↓

Application Program (state machine)
+
system call

↓

Process Management | Memory Management | File Management

Use **ls /sbin/init -l** to check if systemd is being used

Read Latest Linux Kernel Code:

https://elixir.bootlin.com/linux/v6.10.9/source/init/main.c#L1523

- Process Management
  - fork, exec, and exit
- Memory Management
  - mmap –virtual address space
- File management
  - open, close, read, write
  - mkdir, link, unlink

You can use system call to create the world!

fork()

- Creates a complete copy of the state machine
  - Copies memory, register states, and other process information
  - int fork(); - system call used to create a new process

- How fork() Works:
  - Immediately copies the state machine, including the entire memory space
  - The newly created process (child) returns 0
  - The process that calls fork() (parent) returns the child process ID (PID)

A **`fork`** Example, **`Nag.c`**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
  pid_t pid;
  if ((pid = fork()) == 0) {
    while (1) {
      printf("child's return value %d:  I want to play…\n", pid);
    }
  } else {
    while (1) {
      printf("parent's return value %d:  After the project…\n", pid);
    }
  }
  return 0;
}
```

A `fork` Example, `Nag.c`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
  pid_t pid;
  if ((pid = 3128) == 0) {
    while (1) {
      printf("child's return value %d:  I want to play…\n", pid);
    }
  } else {
    while (1) {
      printf("parent's
    }
  }
  return 0;
}
```

Parent process

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
  pid_t pid;
  if ((pid = 0) == 0) {
    while (1) {
      printf("child's return value %d:  I want to play…\n", pid);
    }
  } else {
    while (1) {
      printf("parent's return value %d:  After the project…\n", pid);
    }
  }
  return 0;
}
```

Child process

`Nag.c` Outputs

```
>a.out
child’s return value 0:  I want to play…
child’s return value 0:  I want to play…
child’s return value 0:  I want to play…
…// context switch
parent’s return value 3218:  After the project…
parent’s return value 3218:  After the project…
parent’s return value 3218:  After the project…
…// context switch
child’s return value 0:  I want to play…
child’s return value 0:  I want to play…
child’s return value 0:  I want to play…
^C
>
```

Why clone a process?

Why clone a process?

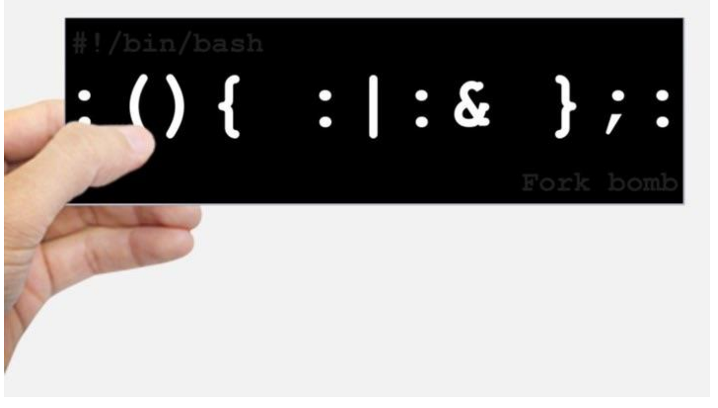
- Simplifies parameter passing
  - Environmental variables, permissions, etc.
- Performance optimization
  - Copy on write

Also,...don't try it (or try it in docker)

Fork Bomb:

```
while (1) {
	fork();
}
```

Or



In bash, colons are allowed as identifiers...
the symbol for fork is a colon.

```
fork() {fork | fork &} ; fork
```

The **`exec`** System Call Family

- A **`fork`** by itself is not interesting
- To make a process run a program that is different from the parent process, you need ***`exec`*** system call
- exec starts a program by overwriting the current process

E.g.,

- int execve(const char *filename, char * const argv[], char * const envp[]);
  - Executes a program named filename
  - Allows setting parameters argv (arguments) and environment variables envp
  - Directly corresponds to the arguments passed to main()!

The `exit` System Call Family

- Destroy the current state machine and allow for a return value.
- The termination of the child process will notify the parent process (explained in later lessons)

Thread Creation

- Use **`pthread_create()`** instead of **`fork()`**
- A newly created thread will share the address space of the current process and all resources (e.g., open files)

\+ Efficient sharing of states

\- Potential corruptions by a misbehaving thread

Address Space

- Contains all states necessary to run a program
  - Code, data, stack
  - Program counter
  - Register values
  - Resources required by the program
  - Status of the running program
- A mechanism to protect one app from crashing another app

- Inspect the address space of a process
  - pmap - report memory of a process

Understanding Address Space through Game Cheats

- Pioneers of esports: Real-Time Strategy (RTS) Games
- Command and Conquer (Westwood), Starcraft (Blizzard), ...
- What if we wanted to 'tamper with' the execution of the game...?
  - ps aux | grap starcraft
  - pmap <pid>
  - change data:
    - gdb -p <process_id>
    - x /10x 0x<address>
    - set *(int *)0x<address> = <new_value>

Understanding Address Space through Game Cheats

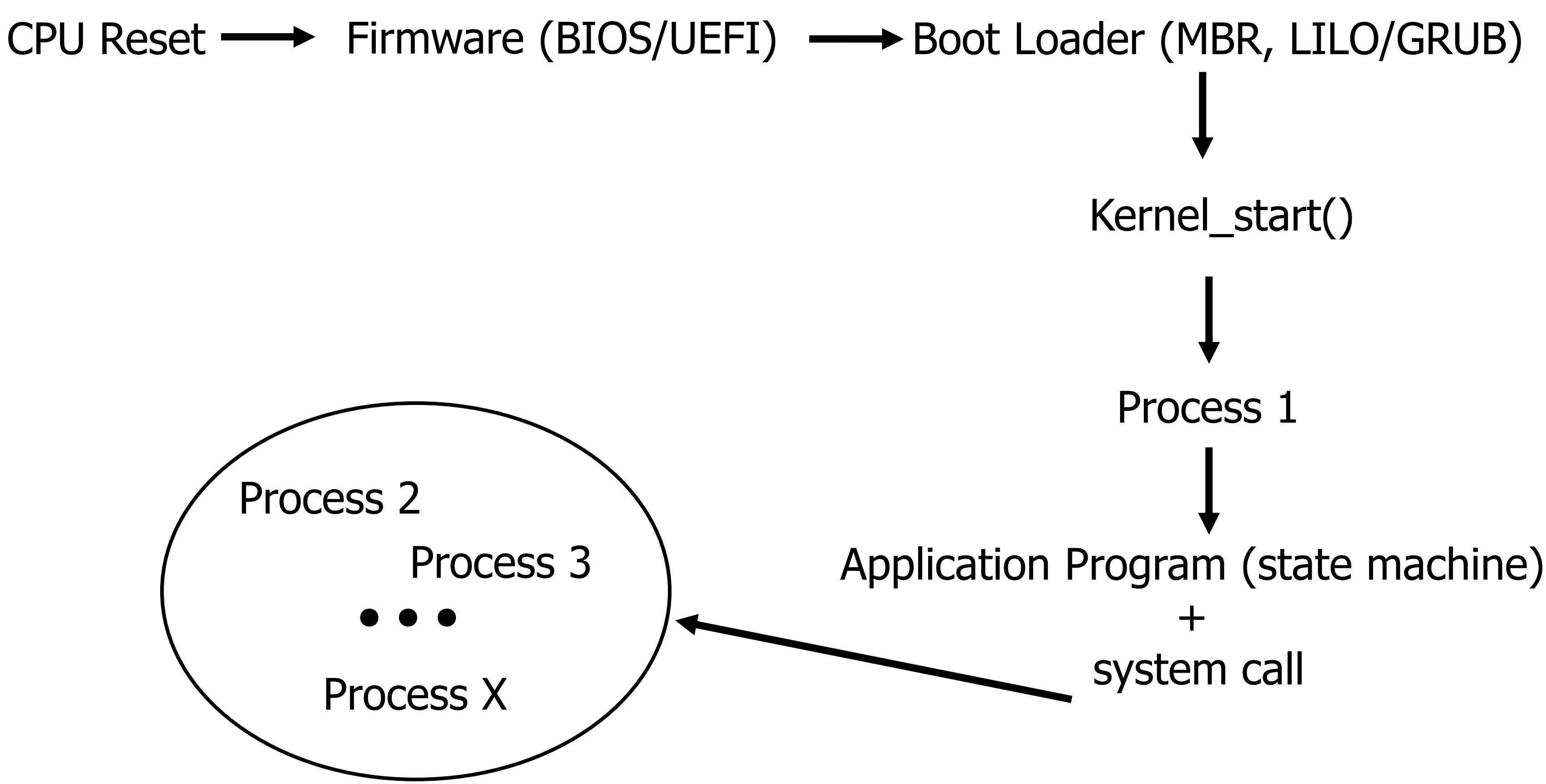
- Game is also state machine.

```
ps aux | grap pugb
pmap <pid>
change data:
    gdb -p <process_id>
    x /10x 0x<address>
    set *(int *)0x<address> = <new_value>
```

PUBG: BATTLEGROUNDS

CPU Scheduler

- A ***CPU scheduler*** is responsible for
  - Removal of running process from the CPU
  - Selection of the next running process
    - Based on a particular strategy



Goals for a Scheduler

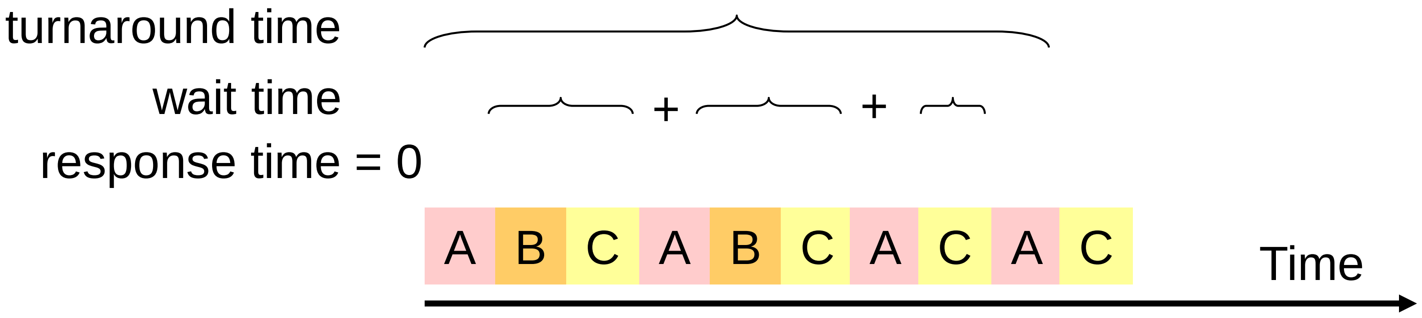
- Maximize
  - ***CPU utilization***: keep the CPU as busy as possible
  - ***Throughput***: the number of processes completed per unit time

Goals for a Scheduler

- Minimize
  - ***Response time***: the time of submission to the time the first response is produced
  - ***Wait time***: total time spent waiting in the ready queue
  - ***Turnaround time***: the time of submission to the time of completion

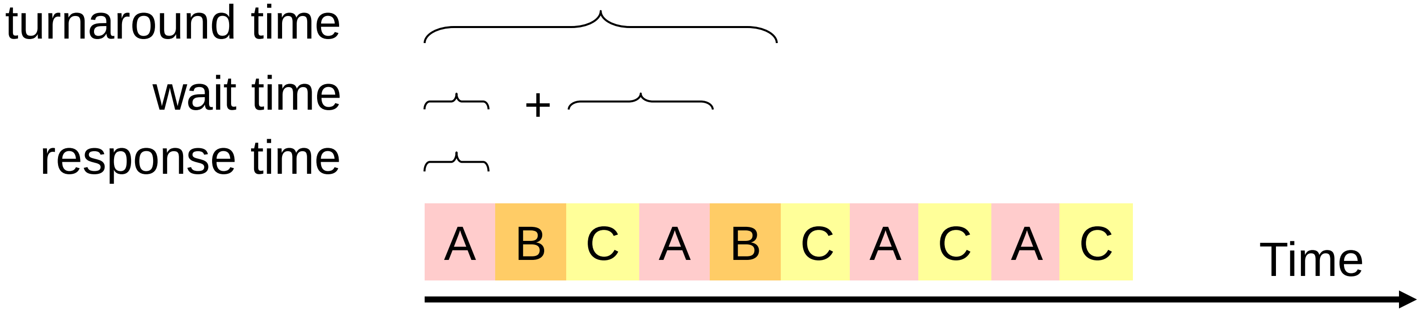
Goals for a Scheduler

- Suppose we have processes A, B, and C, submitted at time 0
- We want to know the response time, wait time, and turnaround time of process A



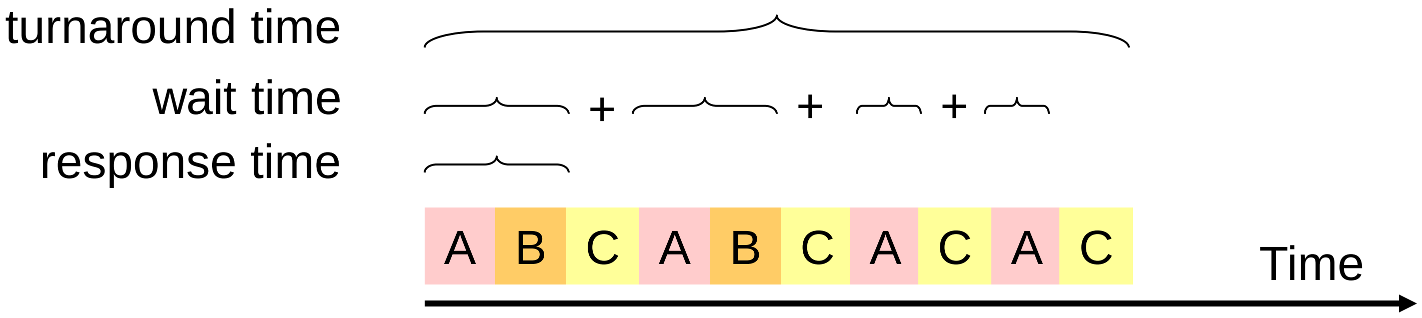
Goals for a Scheduler

- Suppose we have processes A, B, and C, submitted at time 0
- We want to know the response time, wait time, and turnaround time of process B



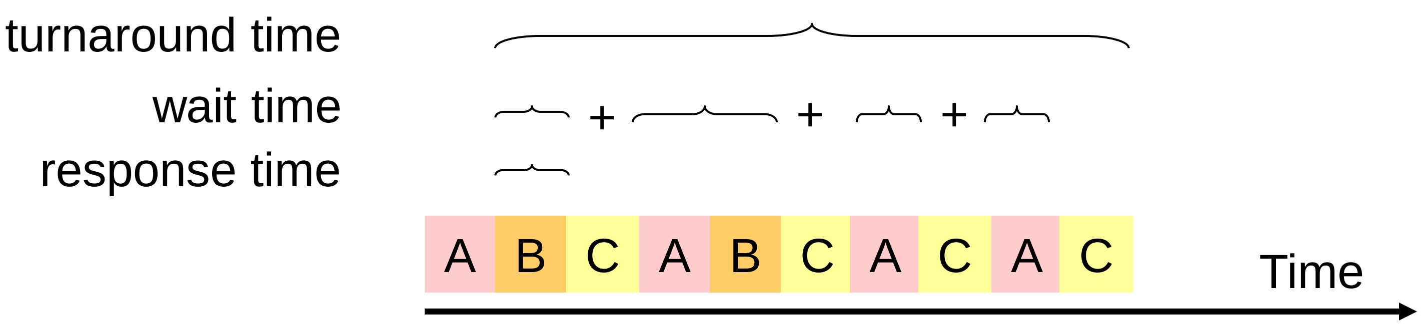
Goals for a Scheduler

- Suppose we have processes A, B, and C, submitted at time 0
- We want to know the response time, wait time, and turnaround time of process C



Goals for a Scheduler

- Suppose we have processes A and B submitted at time 0; process C, time 1
- We want to know the response time, wait time, and turnaround time of process C



Goals for a Scheduler

- Achieve ***fairness***
  - What is fair?
    - Guaranteed to have at least 1/n share
    - How do two people divide a cake in a fair way?
- There are tensions among these goals

Assumptions

- Each user runs one process
- Each process is single threaded
- Processes are independent

- They are not realistic assumptions; they serve to simplify analyses

Scheduling Policies

- FIFO (first in, first out)
- Round robin
- SJF (shortest job first)
- Multilevel feedback queues
- Lottery scheduling

FIFO

- ***FIFO***: assigns the CPU based on the order of requests
  - ***Nonpreemptive***: A process keeps running on a CPU until it is blocked or terminated
  - Also known as FCFS (first come, first serve)

  \+ Simple

  \- Short jobs can get stuck behind long jobs

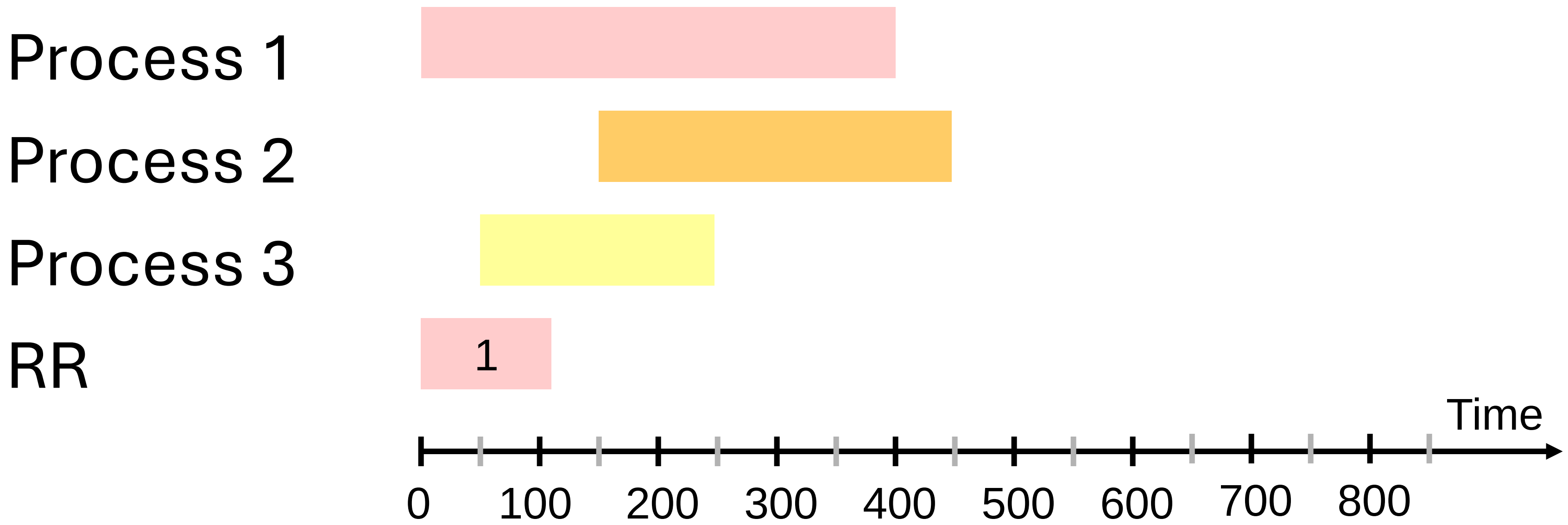
Round Robin

- ***Round Robin (RR)*** periodically releases the CPU from long-running jobs
  - Based on timer interrupts so short jobs can get a fair share of CPU time
  - ***Preemptive***: a process can be forced to leave its running state and replaced by another running process
  - ***Time slice***: interval between timer interrupts

More on Round Robin

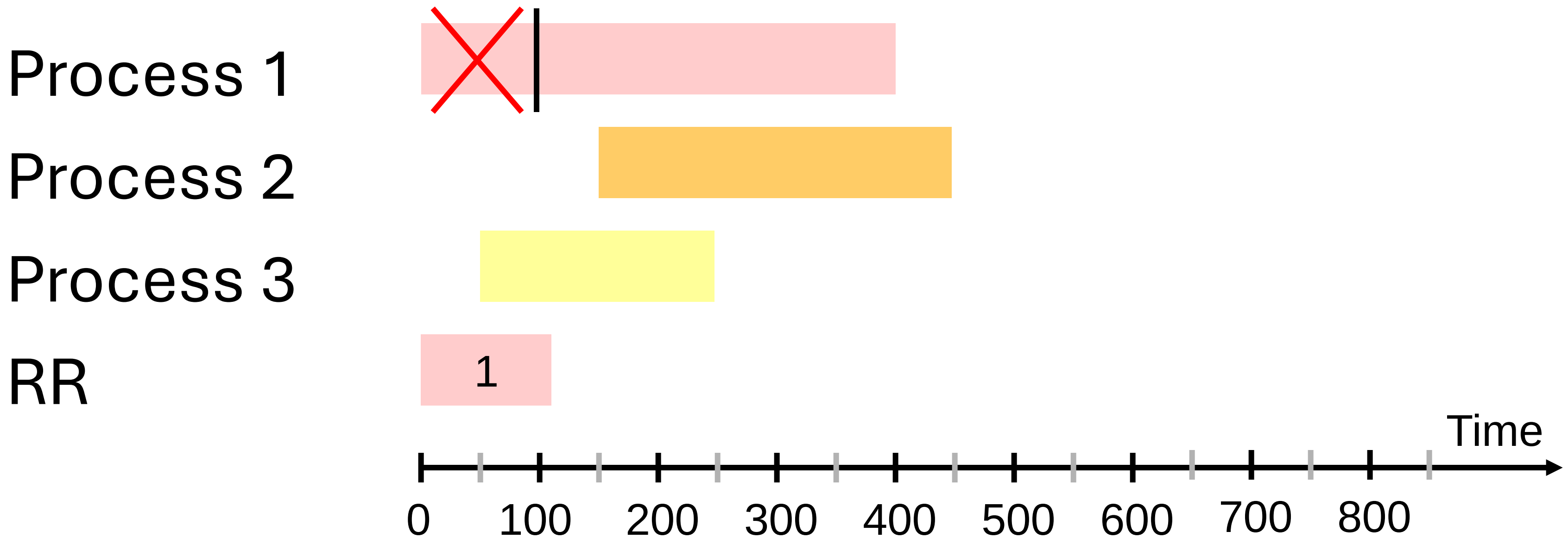
- If time slice is too long
  - Scheduling degrades to FIFO
- If time slice is too short
  - Throughput suffers
  - Context switching cost dominates

Round Robin (Time slice = 100)

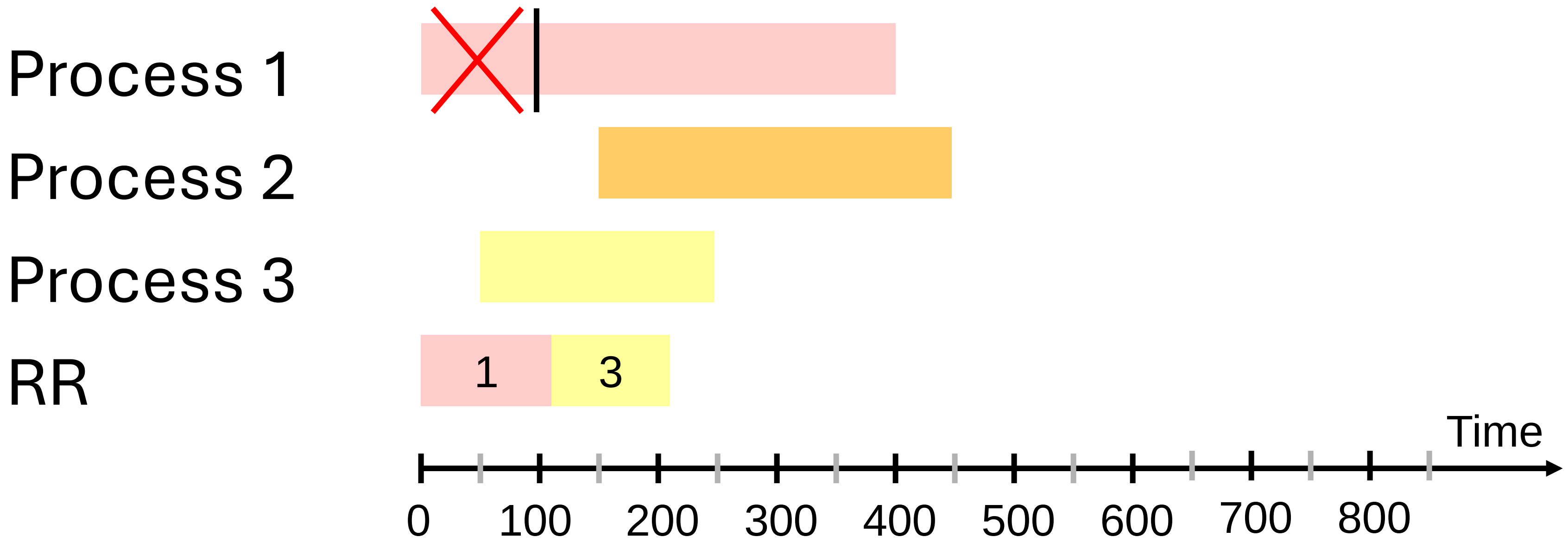


Important

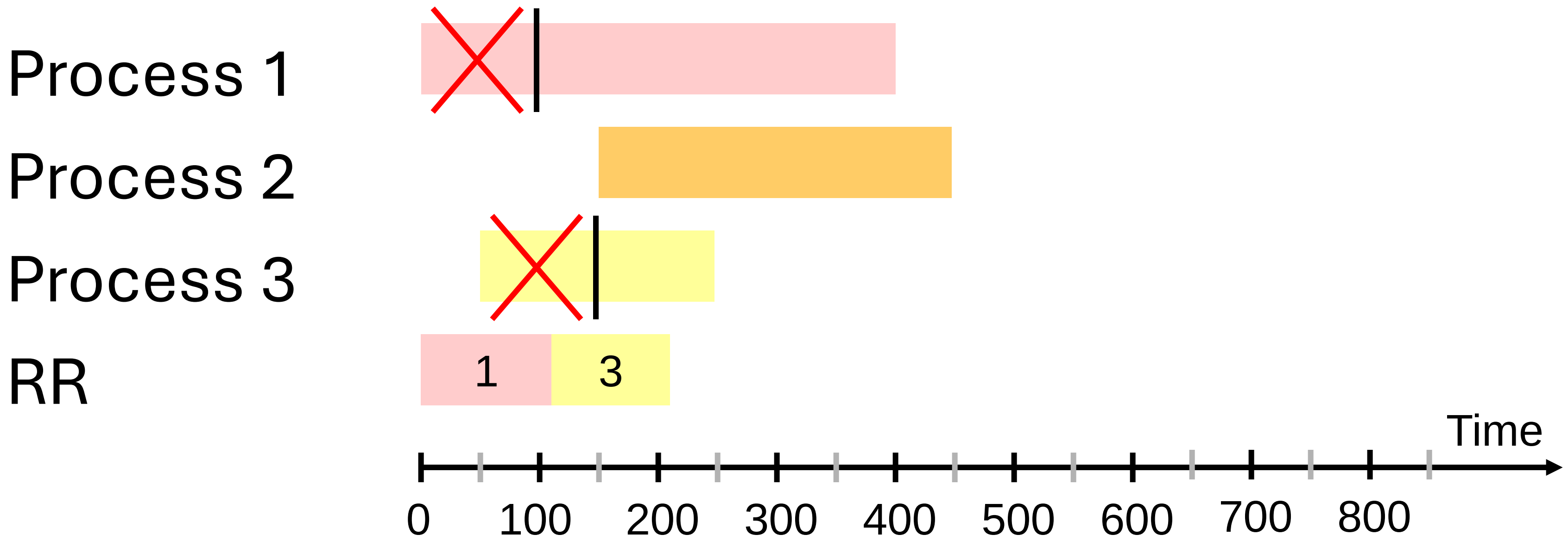
Round Robin (Time slice = 100)



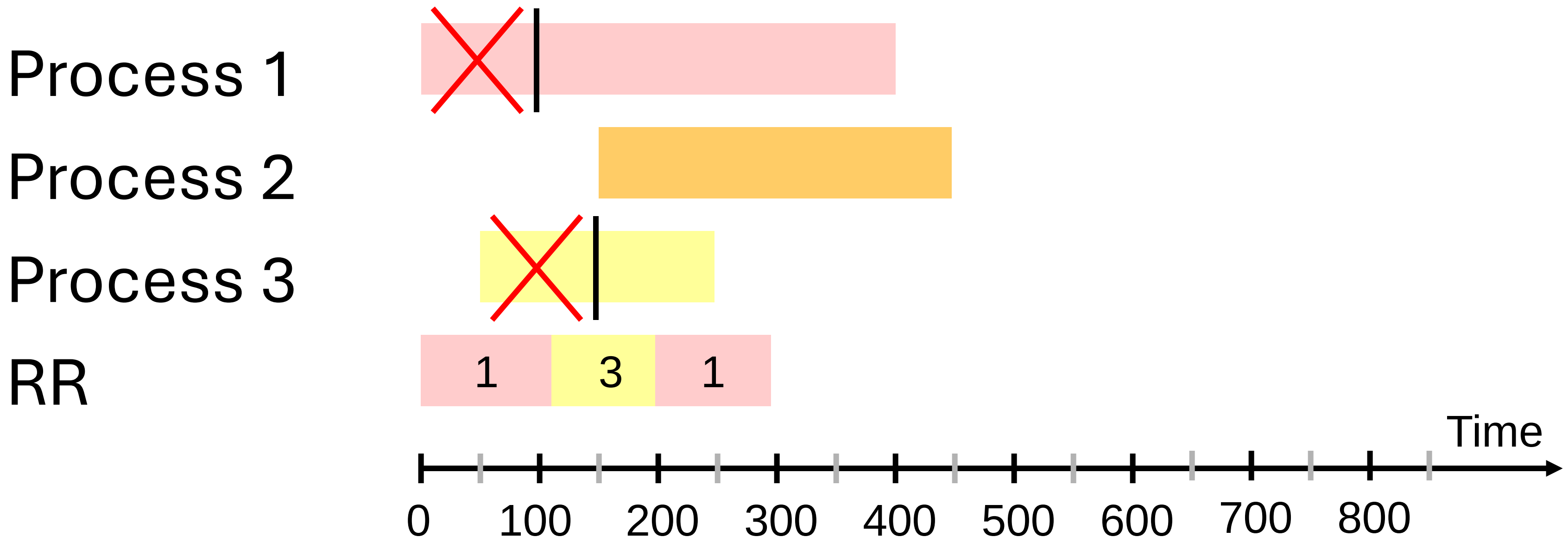
Round Robin (Time slice = 100)



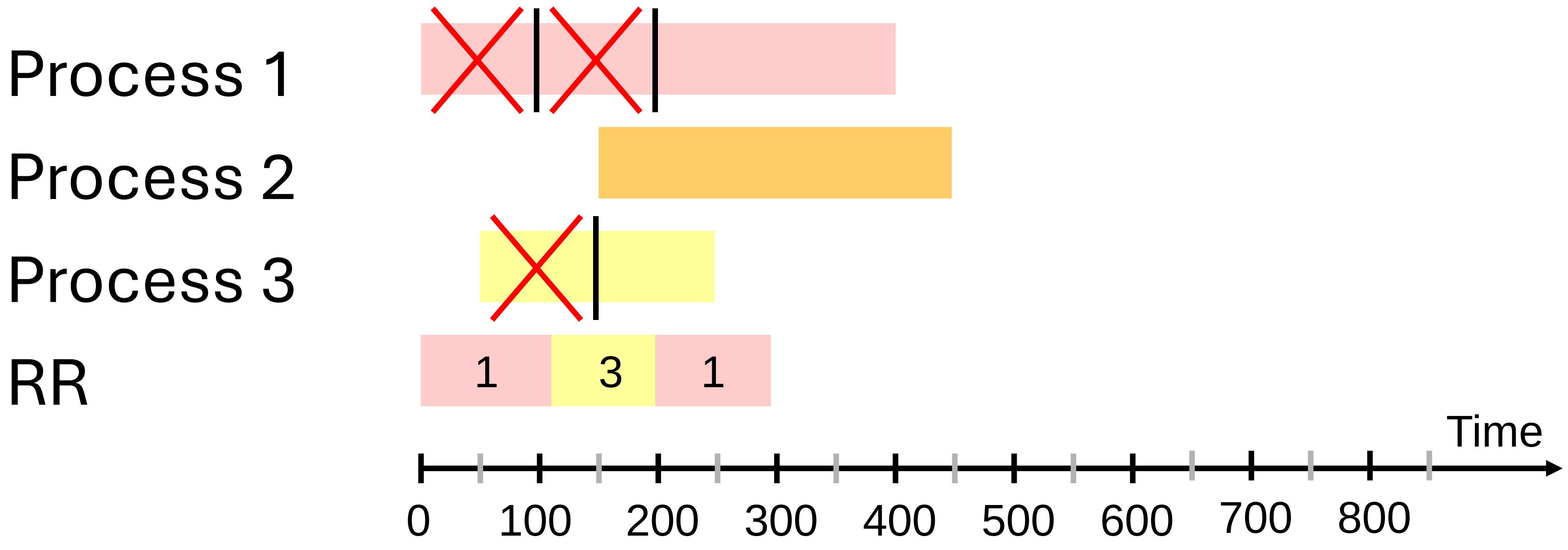
Round Robin (Time slice = 100)



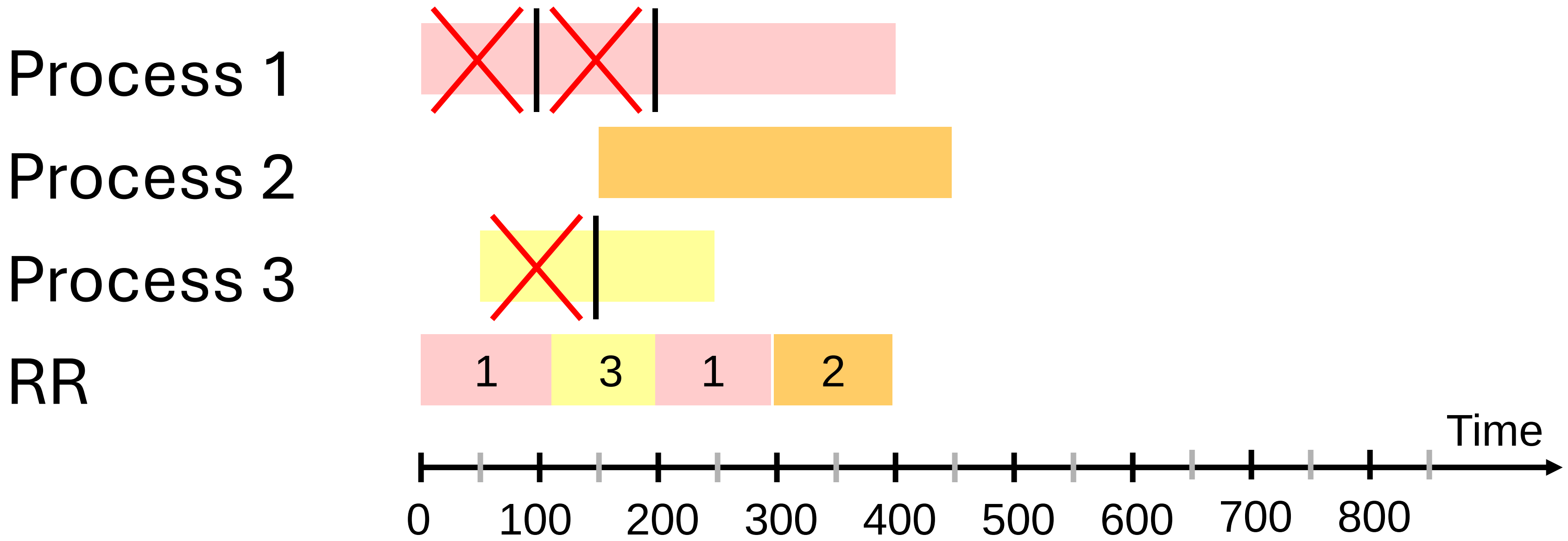
Round Robin (Time slice = 100)



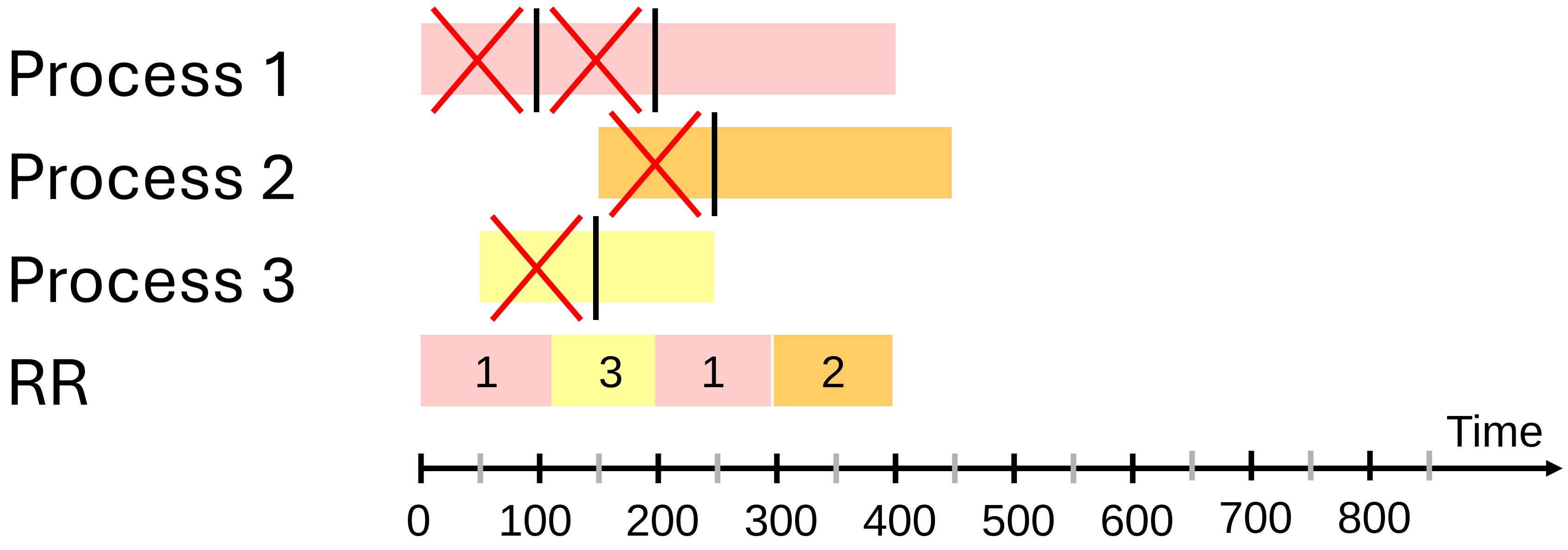
Round Robin (Time slice = 100)



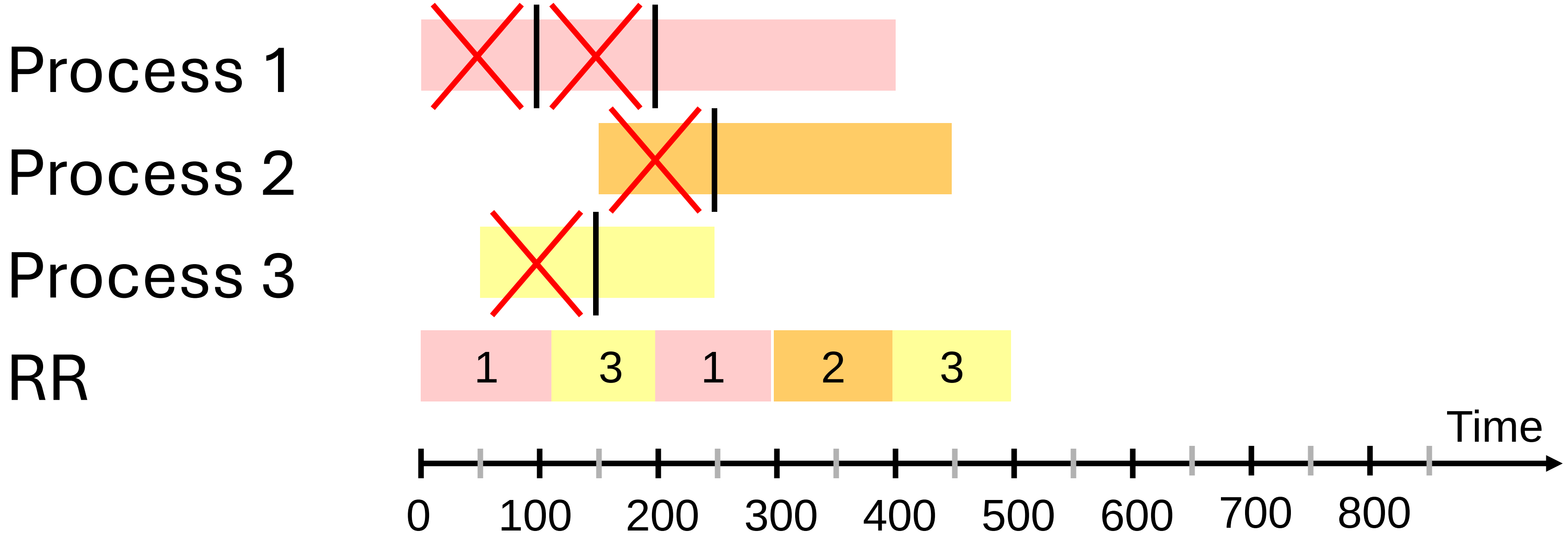
Round Robin (Time slice = 100)



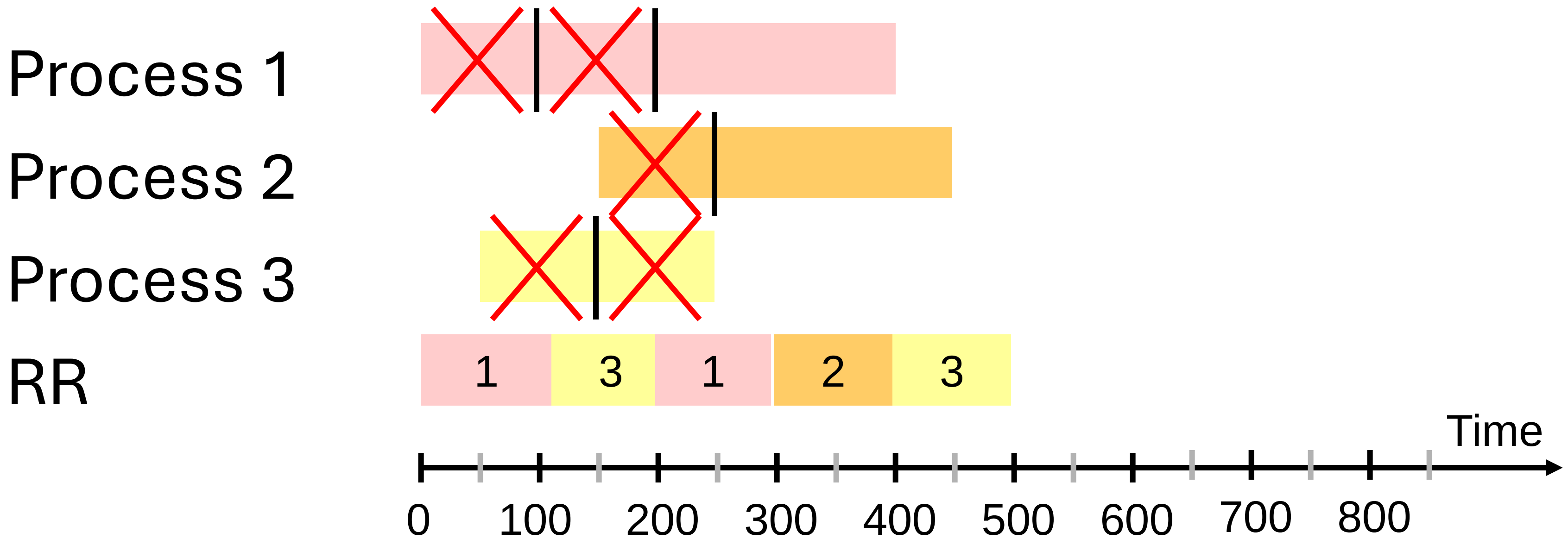
Round Robin (Time slice = 100)



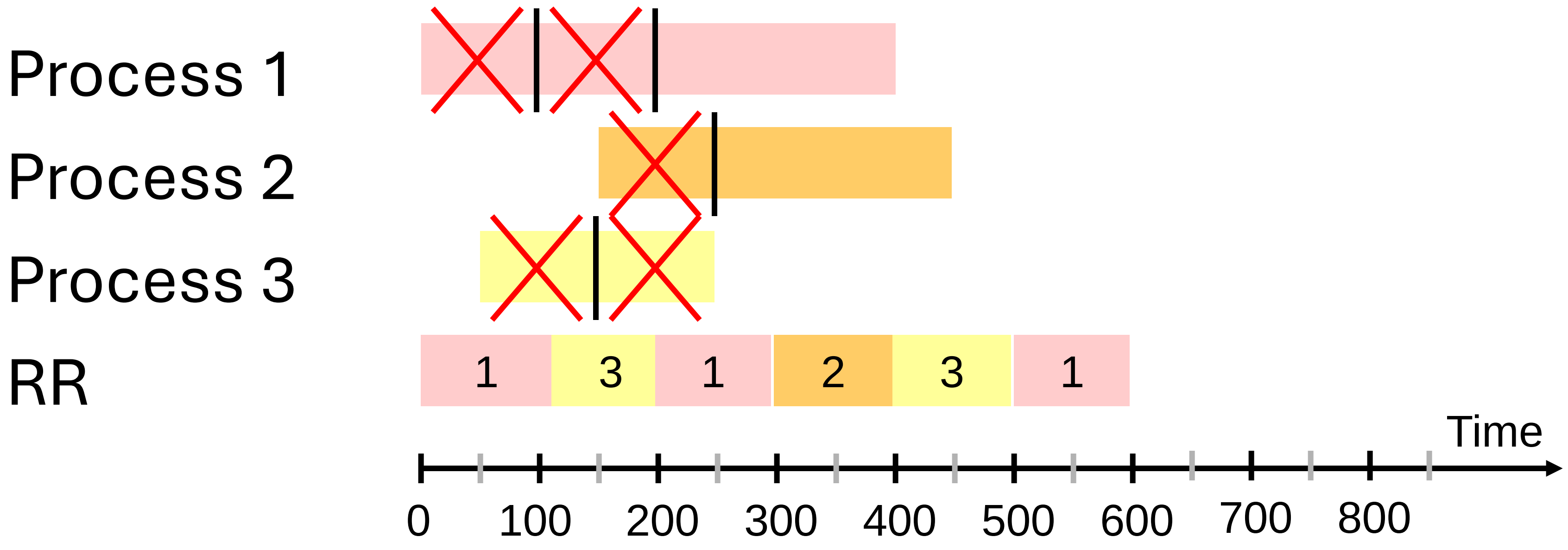
Round Robin (Time slice = 100)



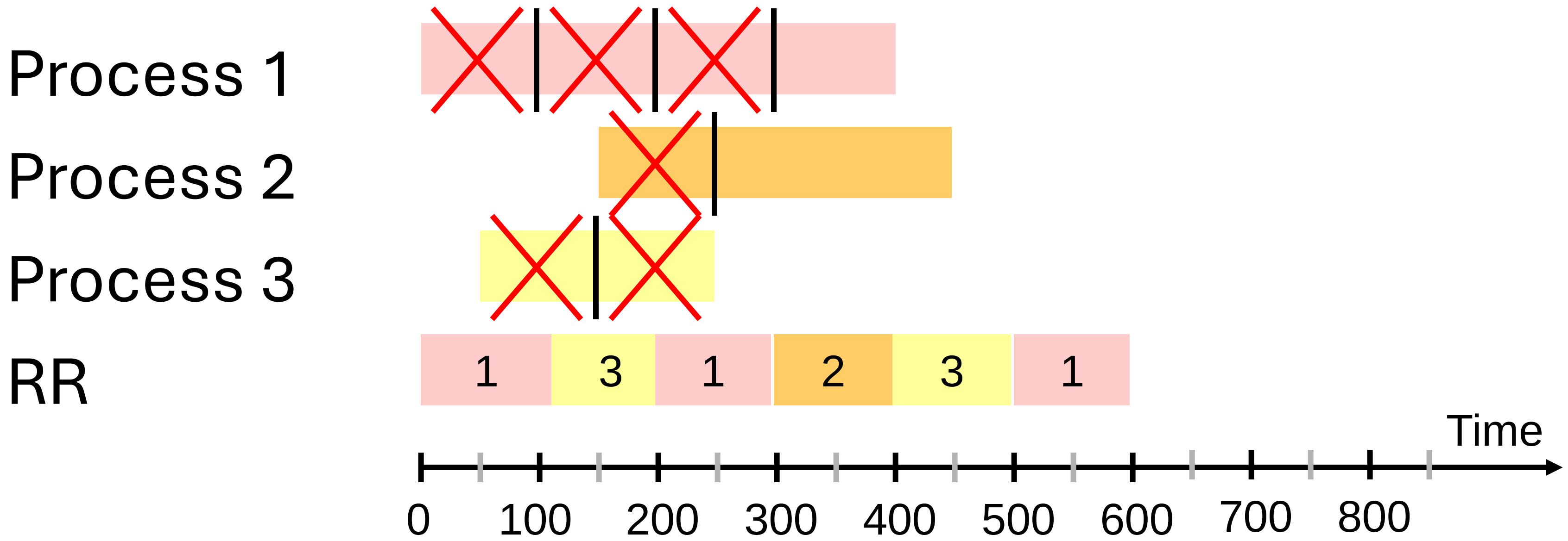
Round Robin (Time slice = 100)



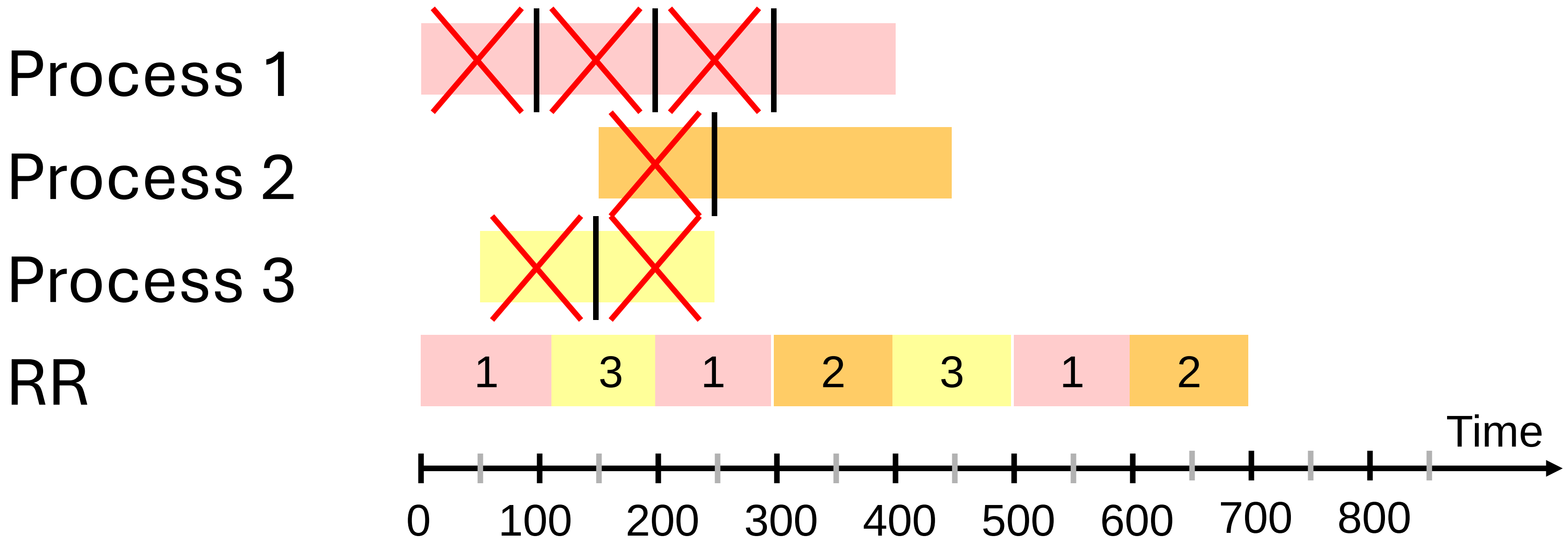
Round Robin (Time slice = 100)



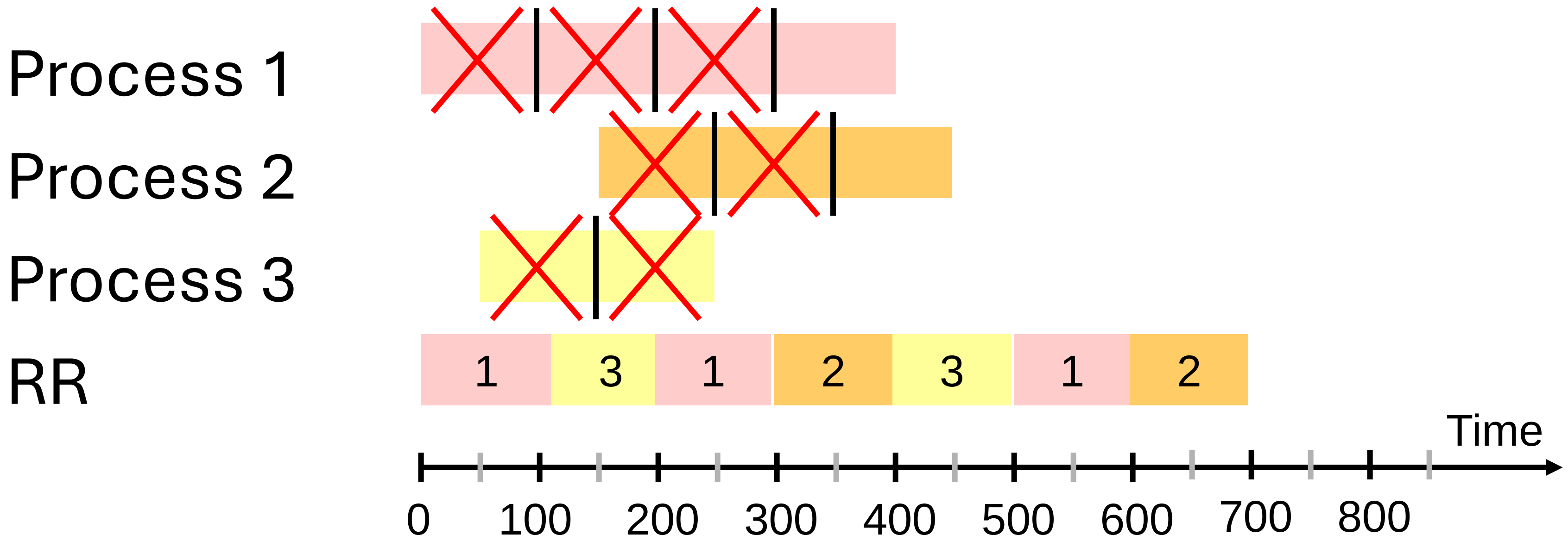
Round Robin (Time slice = 100)



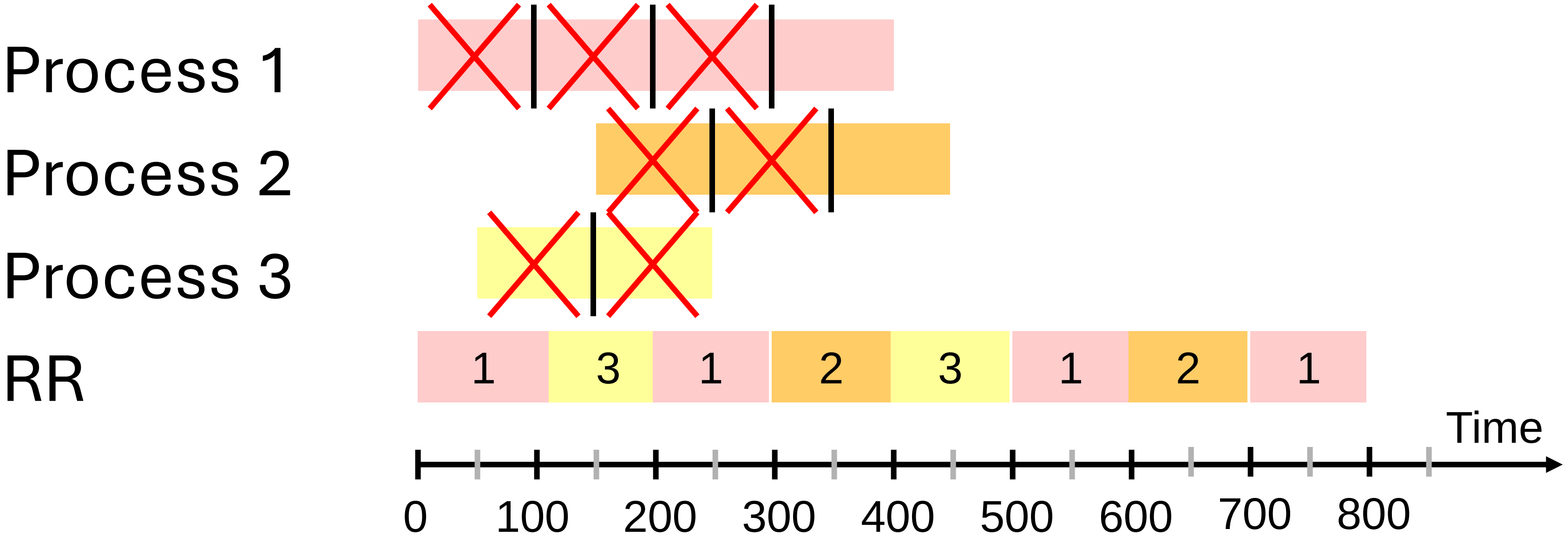
Round Robin (Time slice = 100)



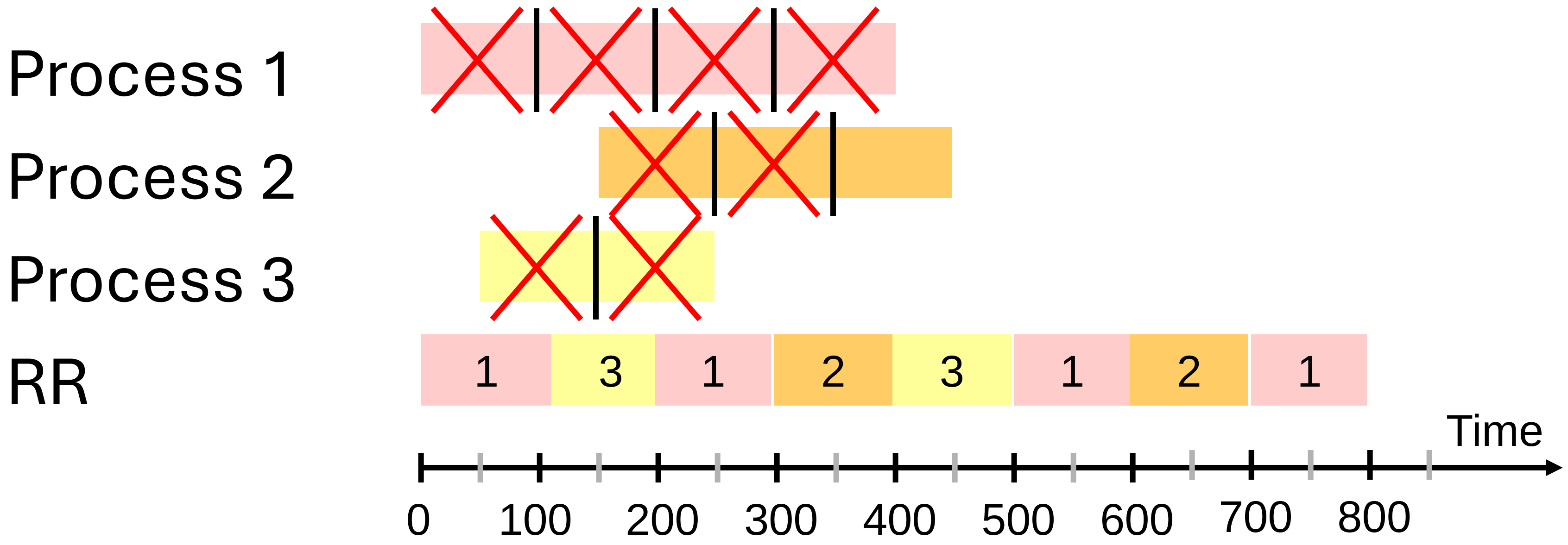
Round Robin (Time slice = 100)



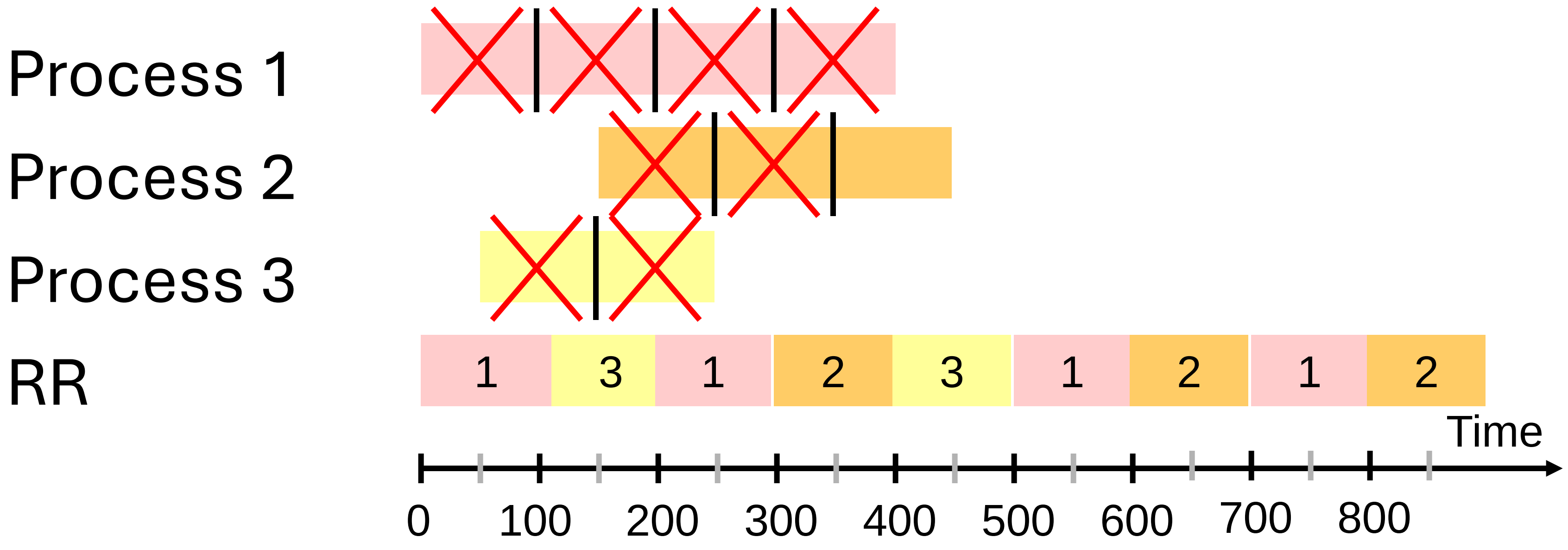
Round Robin (Time slice = 100)



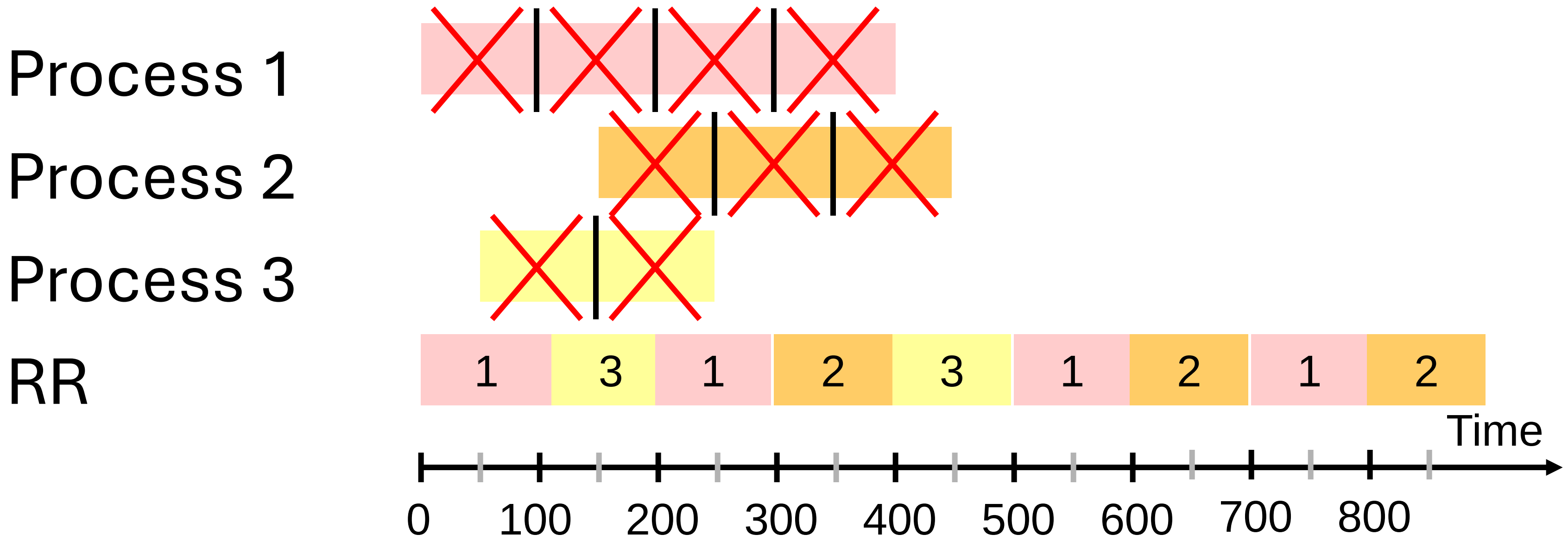
Round Robin (Time slice = 100)



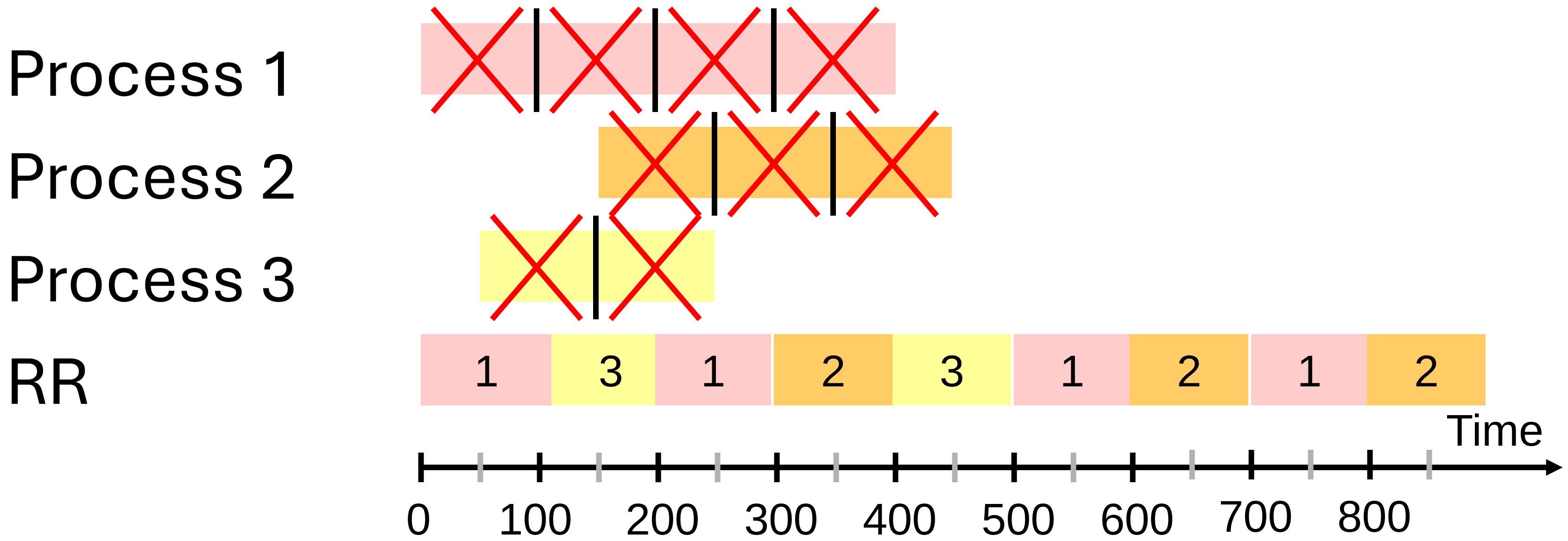
Round Robin (Time slice = 100)



Round Robin (Time slice = 100)



Round Robin (Time slice = 100)

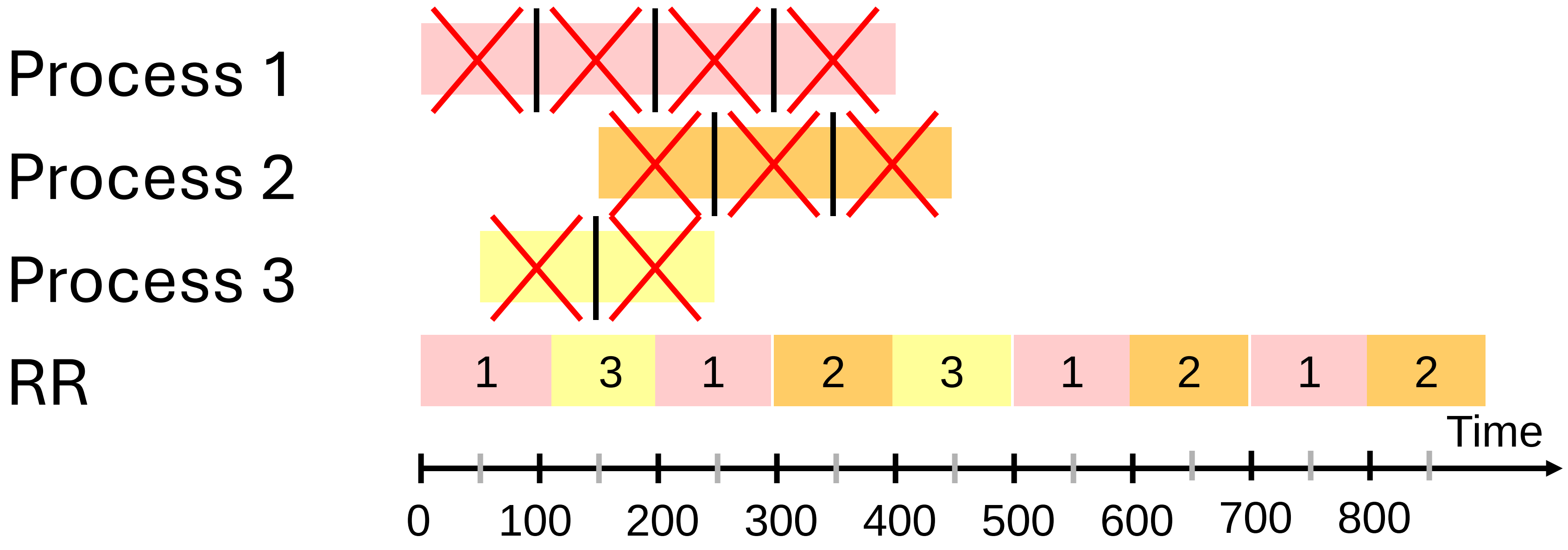


Response time for process 1: 0

Response time for process 2: 300 – 150 = 150

Response time for process 3: 100 – 50 = 50

Round Robin (Time slice = 100)

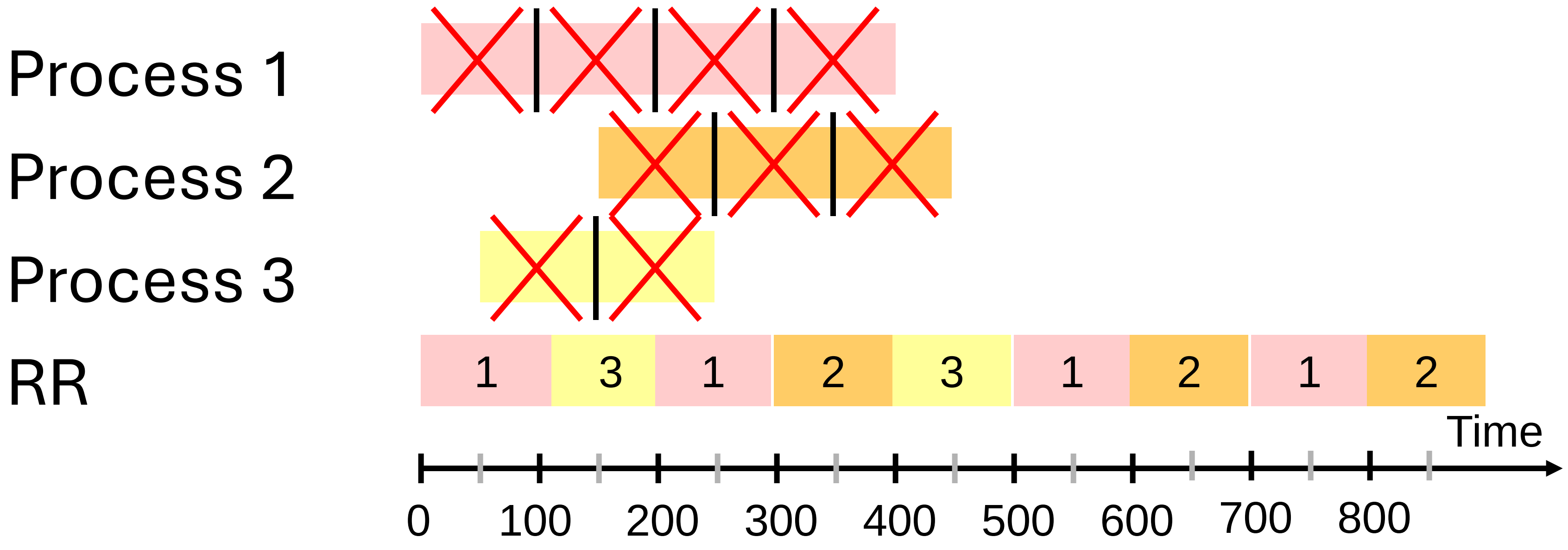


Wait time for process 1: 0 + (200 – 100) + (500 – 300) + (700 - 600) = 400

Wait time for process 2: (300 – 150) + (600 – 400) + (800 – 700) = 450

Wait time for process 3: (100 - 50) + (400 – 200) = 250

Round Robin (Time slice = 100)



Turnaround time for process 1: 800 – 0 = 800

Turnaround time for process 2: 900 – 150 = 750

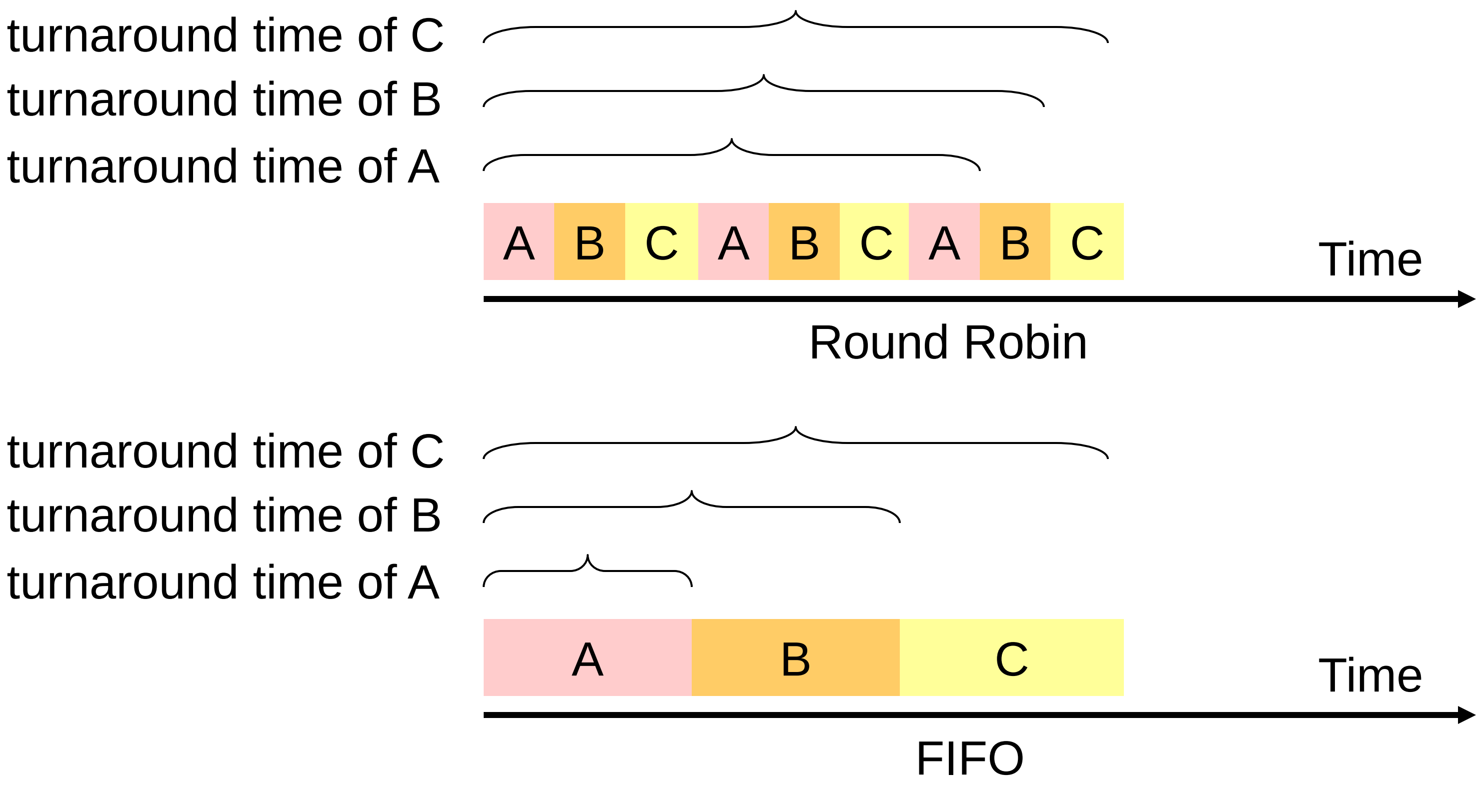
Turnaround time for process 3: 500 – 50 = 450

FIFO vs. Round Robin

- With zero-cost context switch, is RR always better than FIFO?

FIFO vs. Round Robin

- Suppose we have three jobs of equal length



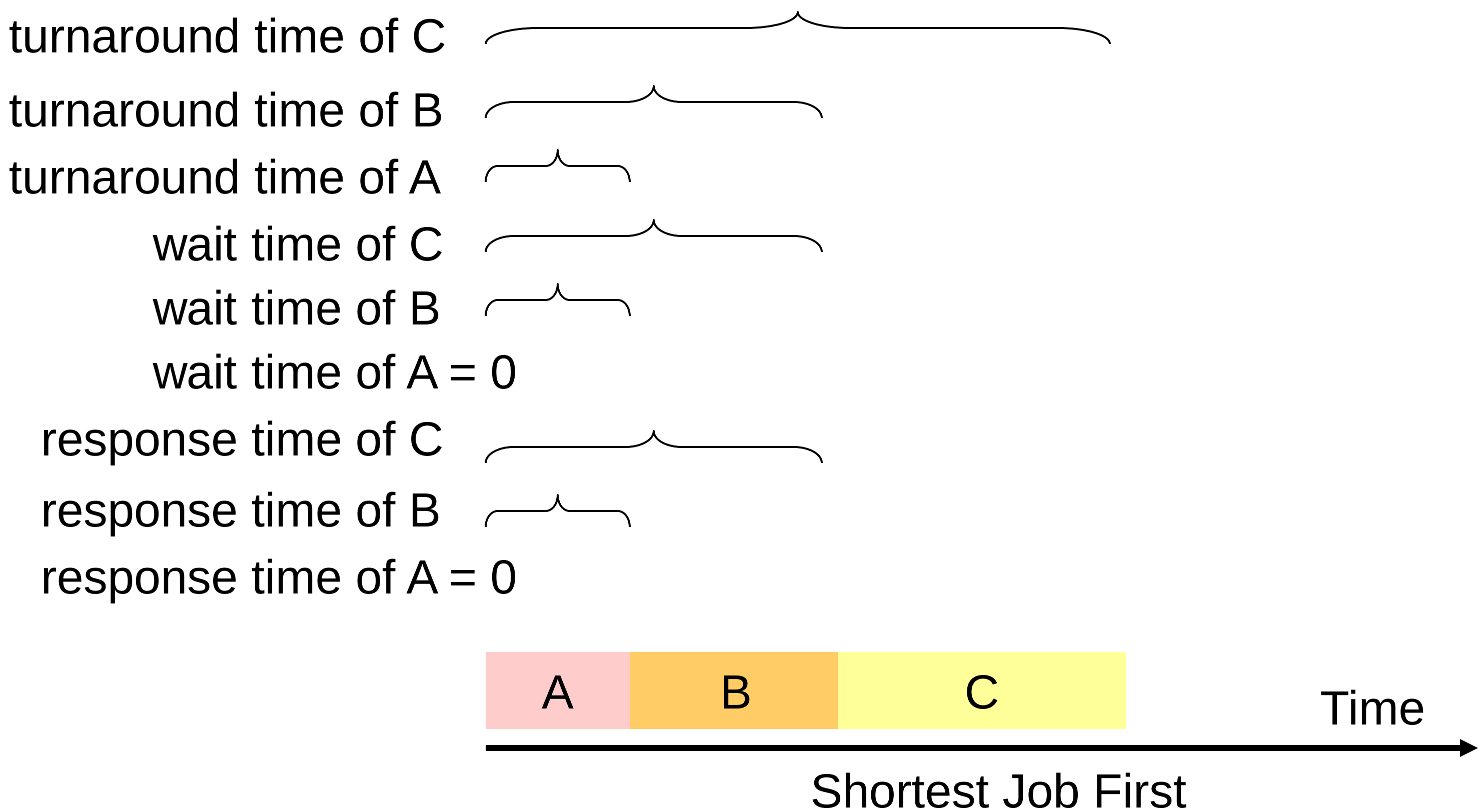
FIFO vs. Round Robin

- Round Robin
  + Shorter response time
  + Fair sharing of CPU
  - Not all jobs are preemptive
  - Not good for jobs of the same length

Shortest Job First (SJF)

- ***SJF*** runs whatever job puts the least demand on the CPU, also known as ***STCF (shortest time to completion first)***
  + Provably optimal
  + Great for short jobs
  + Small degradation for long jobs
- Real life example: supermarket express checkouts

SJF Illustrated



Shortest Remaining Time First (SRTF)

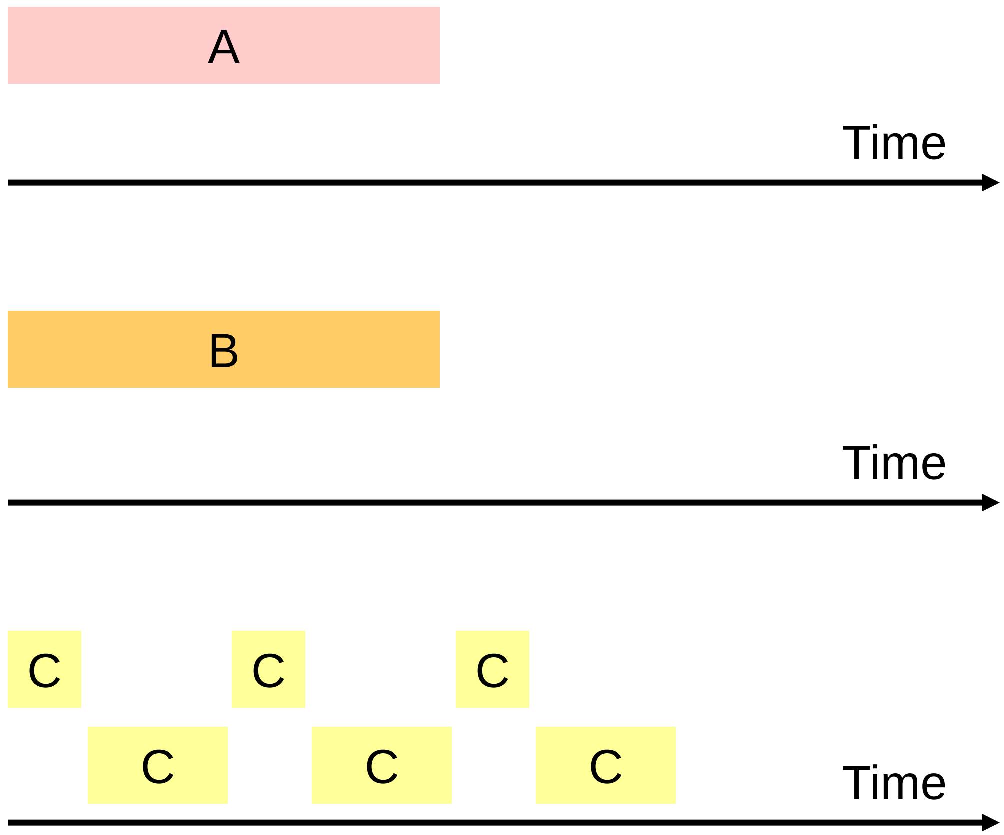
- ***SRTF***: a preemptive version of SJF
  - If a job arrives with a shorter time to completion, SRTF preempts the CPU for the new job
  - Also known as ***SRTCF (shortest remaining time to completion first)***
  - Generally used as the base case for comparisons

SJF and SRTF vs. FIFO and Round Robin

- If all jobs are the same length, SJF → FIFO
  - FIFO is the best you can do
- If jobs have varying length
  - Short jobs do not get stuck behind long jobs under SRTF

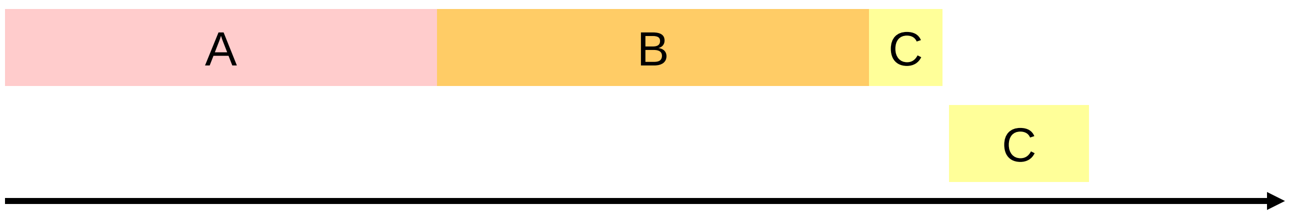
A More Complicated Scenario (Arrival Times = 0)

- Process A (6 units of CPU request)
  - 100% CPU
  - 0% I/O
- Process B (6 units of CPU request)
  - 100% CPU
  - 0% I/O
- Process C (infinite loop)
  - 33% CPU
  - 67% I/O



A More Complicated Scenario

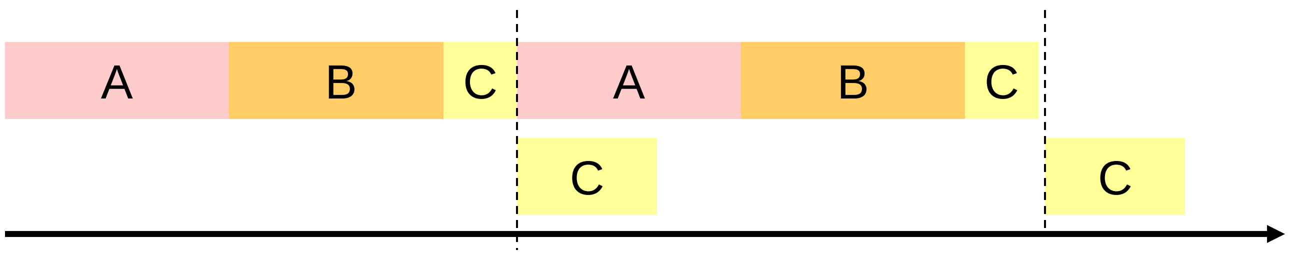
- FIFO
  - CPU
  - I/O



Poor response and wait time for process C

Time

- Round Robin with time slice = 3 units
  - CPU
  - I/O



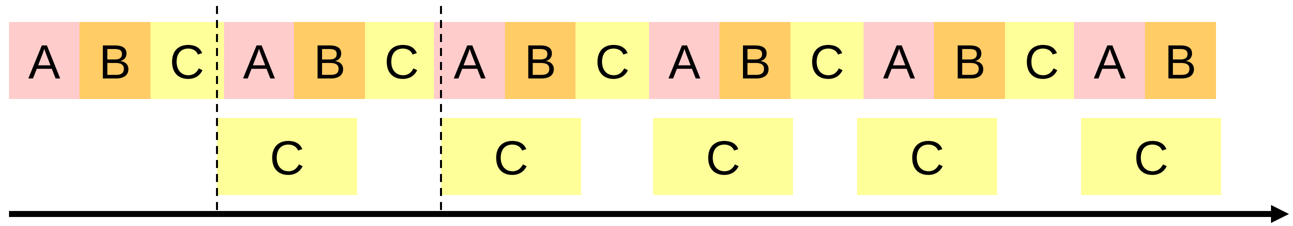
Disk utilization: 29% (2 out of 7 units)

Time

Important

A More Complicated Scenario

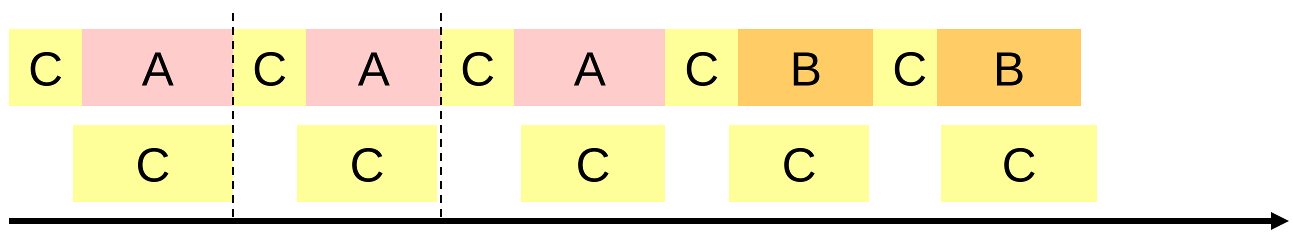
- Round Robin with time slice = 1 unit
  - CPU
  - I/O



Disk utilization 66% (2 out of 3 units)

Time

- SRTCF
  - CPU
  - I/O



Disk utilization: 66% (2 out of 3 units)

Time

Drawbacks of Shortest Job First

- ***Starvation***: constant arrivals of short jobs can keep long ones from running
- There is no way to know the completion time of jobs (most of the time)
  - Some solutions
    - Ask the user, who may not know any better
    - If a user cheats, the job is killed

Priority Scheduling (Multilevel Queues)

- ***Priority scheduling***: The process with the highest priority runs first
- Priority 0: C
- Priority 1: A
- Priority 2: B
- Assume that low numbers represent high priority

Priority Scheduling

Priority Scheduling

+ Generalization of SJF
    - With SJF, higher priority is inversely proportionally to requested_CPU_time

- Starvation
    - To prevent starvation, mechanisms like aging can be used, where the priority of a process increases the longer it waits.

Multilevel Feedback Queues

- ***Multilevel feedback queues*** use multiple queues with different priorities
  - Round robin at each priority level
  - Run highest priority jobs first
  - Once those finish, run next highest priority, etc
  - Jobs start in the highest priority queue
  - If time slice expires, drop the job by one level
  - If time slice does not expire, push the job up by one level

Multilevel Feedback Queues

time = 0

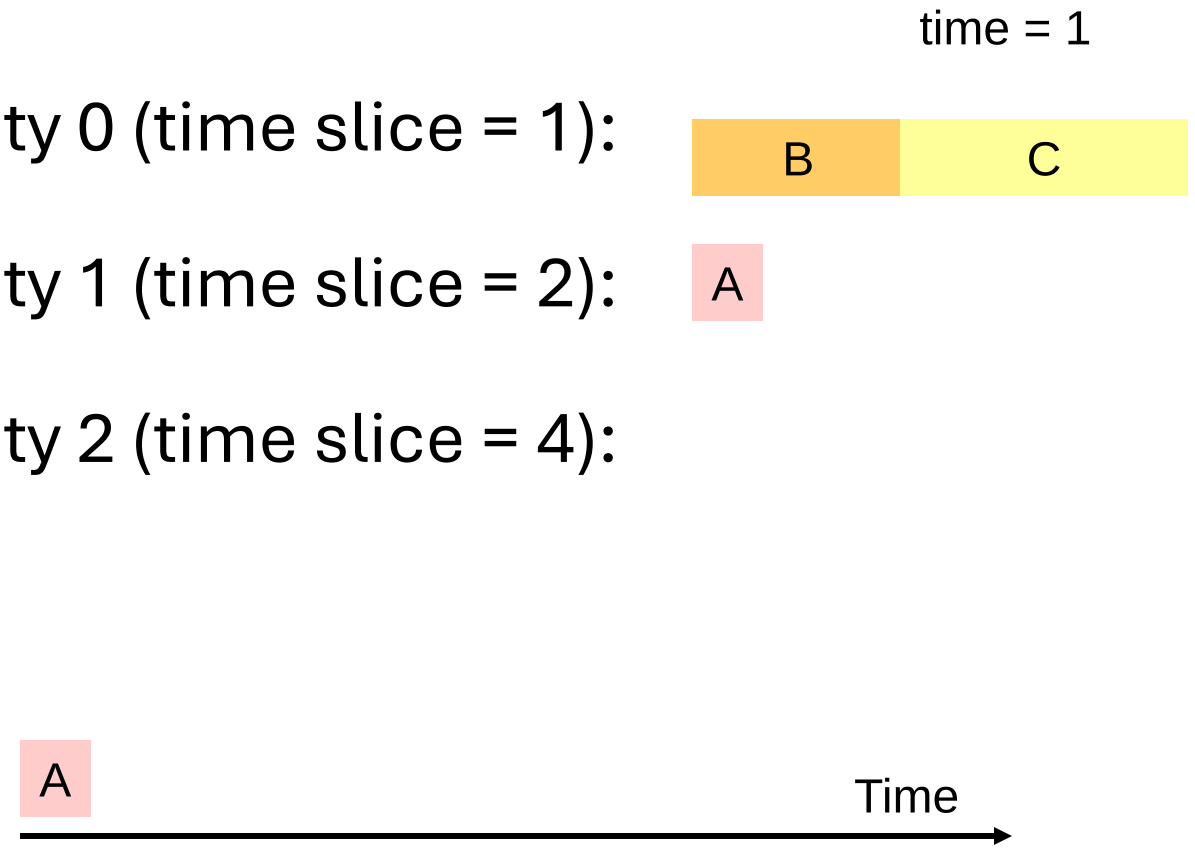
- Priority 0 (time slice = 1):

- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):

Important

Multilevel Feedback Queues

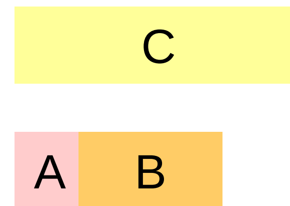
- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):



Multilevel Feedback Queues

time = 2

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):



Multilevel Feedback Queues

time = 3

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2): 

- Priority 2 (time slice = 4):

Multilevel Feedback Queues

time = 3

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2): 

- Priority 2 (time slice = 4):

suppose process A is blocked on an I/O

Multilevel Feedback Queues

time = 3

- Priority 0 (time slice = 1): A
- Priority 1 (time slice = 2): B C
- Priority 2 (time slice = 4):

suppose process A is blocked on an I/O

Multilevel Feedback Queues

time = 5

- Priority 0 (time slice = 1): A
- Priority 1 (time slice = 2): C
- Priority 2 (time slice = 4):

suppose process A is returned from an I/O

Multilevel Feedback Queues

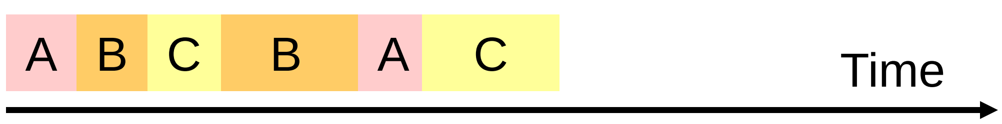
time = 6

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2): C
- Priority 2 (time slice = 4):

Multilevel Feedback Queues

time = 8

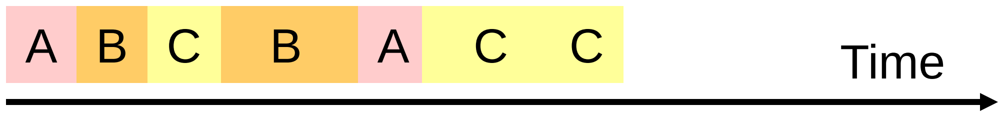
- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4): C



Multilevel Feedback Queues

time = 9

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):



Multilevel Feedback Queues

- Approximates SRTF
  - A CPU-bound job drops like a rock
  - I/O-bound jobs stay near the top
  - Still unfair for long running jobs
  - Counter-measure: ***Aging***
    - Increase the priority of long running jobs if they are not serviced for a period of time
    - Tricky to tune aging

Lottery Scheduling

- ***Lottery scheduling*** is an adaptive scheduling approach to address the fairness problem
  - Each process owns some tickets
  - On each time slice, a ticket is randomly picked
  - On average, the allocated CPU time is proportional to the number of tickets given to each job

Lottery Scheduling

- To approximate SJF, short jobs get more tickets
- To avoid starvation, each job gets at least one ticket

Lottery Scheduling Example

- short jobs: 10 tickets each
- long jobs: 1 ticket each

# short jobs/# long jobs	% of CPU for each short job	% of CPU for each long job

Important

Pros and Cons of Lottery Scheduling

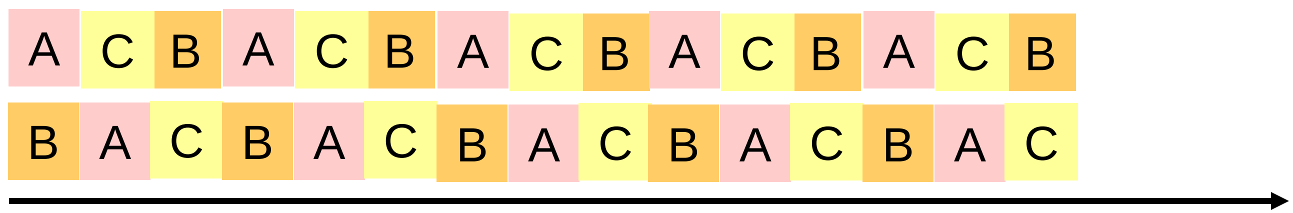
+ Good for coordinating computers with different computing power

+ Good for controlling the schedules for child processes

- Not as good for real-time systems

Multicore Scheduling

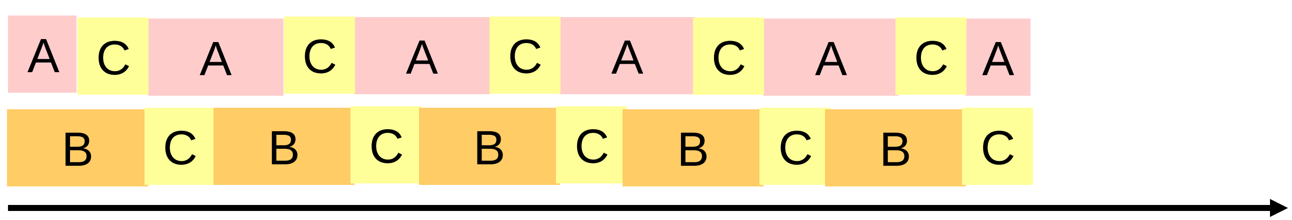
- Single-queue multiprocessor scheduling (SQMS)
  - CPU1
  - CPU2



Expensive to migrate jobs across CPUs
(Poor ***CPU affinity***)

Time

- Another SQMS
  - CPU1
  - CPU2



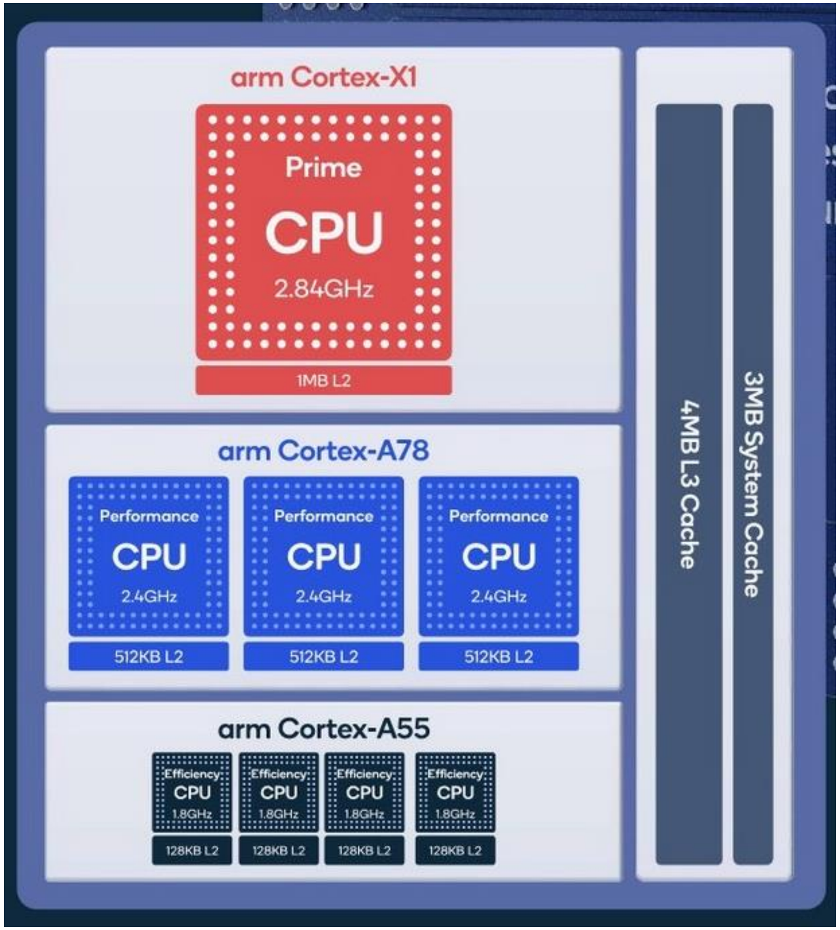
Time

More Schedulers

- Multi-queue scheduling
- O(1) scheduler
- Completely Fair Scheduler (CFS)

Real World

- Big.LITTLE
  - Snapdragon 888
- Others
  - Distance
  - Power
  - …



Takeaways

- OS boot sequence
- Process creation, and address space
- CPU scheduling policies