CSCC69 Week 2 Notes

Program vs Process:

- **Program:** Static data on some storage.
- **Process:** Instance of a program execution.
- Several processes of the same program can run concurrently.
  E.g. You can open two or more instances of the same browser.

Running processes concurrently:

- A CPU core will run multiple processes concurrently by running each process for a little amount of time before switching to another one. This is called **limited direct execution**.
- The CPU will switch to another process when either the running process runs out of time slice (system clock interrupt) or the running process initiates an I/O that will take some time.

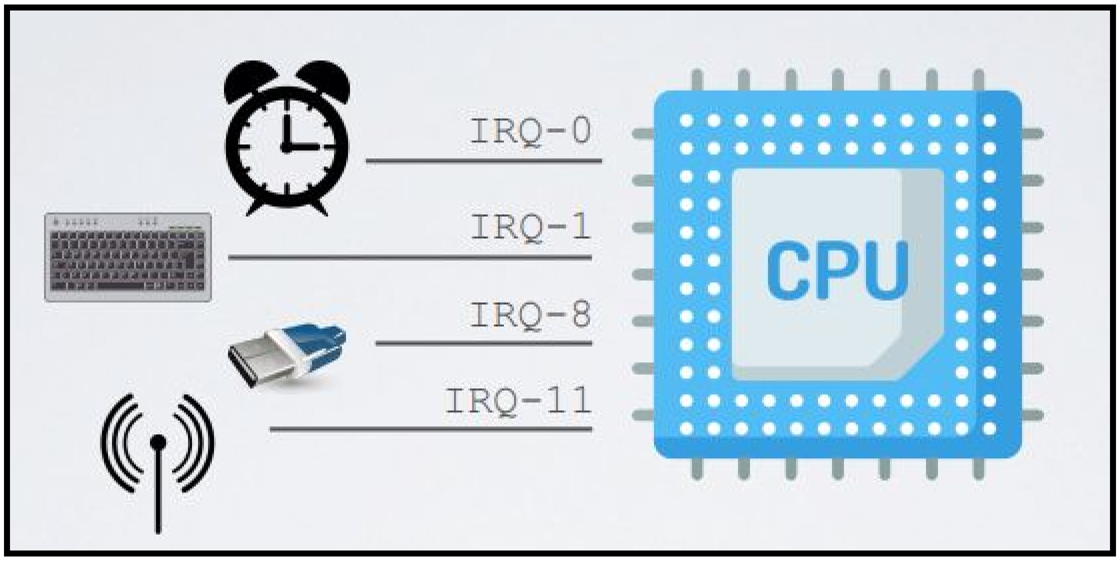
Advantages of concurrency:

- From the system perspective, there is better CPU usage resulting in a faster execution overall (but not individually).
- From the user perspective, programs seem to be executed in parallel.
- However, it requires some mechanisms to manage and schedule these concurrent processes.

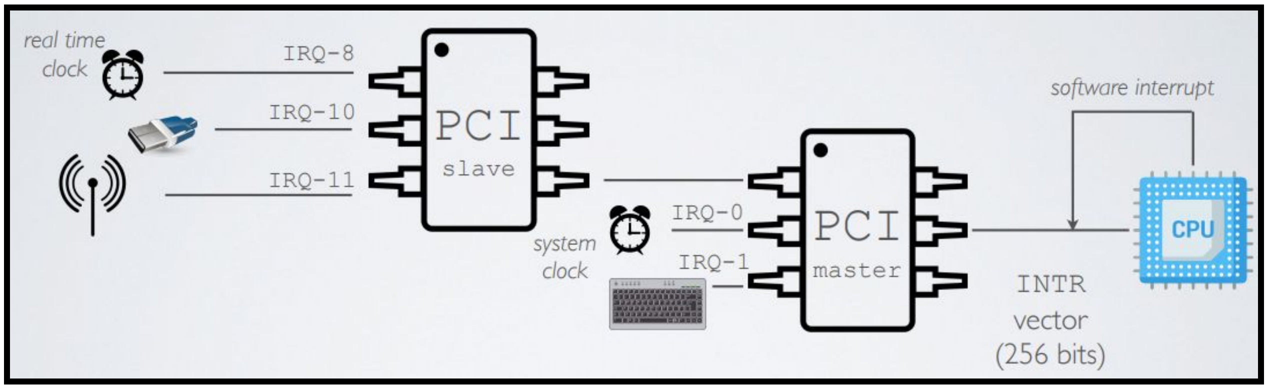
Managing Interrupts:

- There are 2 kinds of interrupts:
  1. **Hardware interrupts (asynchronous)**. This is caused by an I/O device that needs some attention.
  2. **Software interrupts/exceptions (synchronous)**. This is caused by executing instructions. When you execute a program, there are 2 types of things that can come up:
     a. **Fault**. This is a program or error.
        E.g. Divide by 0.
        E.g. Page Fault.
     b. **Trap**. A trap is a low level assembly for throwing an exception. The instruction is an int. Each exception can have a number. This is useful for system calls.
        E.g. int $0x80 for Linux system call trap.
        E.g. int $0x30 for Pintos system call trap.
- **Naive Implementation:**
- The naive implementation is having the CPU receive interrupts from different devices.
- The devices are all wired to **Interrupt Request lines (IRQs)**.
- However:
  - This isn't flexible (hardwired).
  - The CPU might get interrupted all the time.
  - How to handle interrupt priority.

- E.g. The system clock, keyboard, usb, network are all different I/O devices and they all have their own IRQs and each one will generate an interrupt.

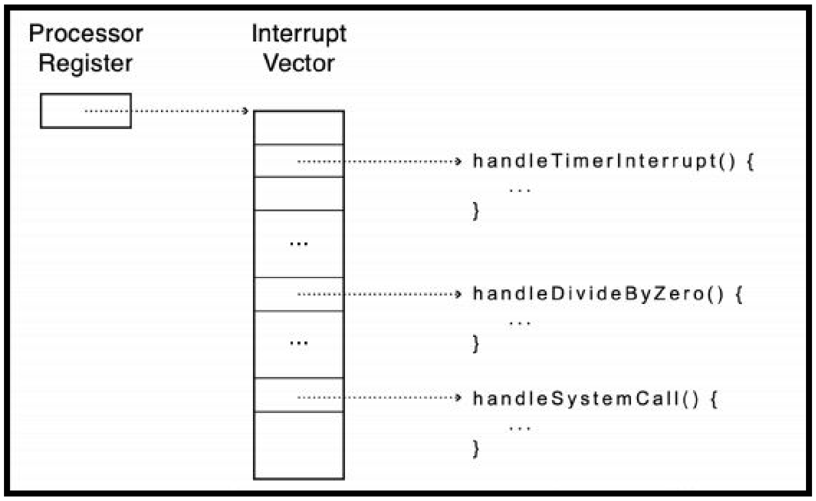


- **Real Implementation:**
- I/O devices have unique or shared IRQs that are managed by two **Programmable Interrupt Controllers (PIC)**. There are 2 of them, a PCI slave and a PCI master and they manage the interrupts.
- The PICs are responsible to tell the CPU when and which devices wish to interrupt through the INTR vector. There are 16 lines of interrupt (IRQ0 - IRQ15), interrupts have different priority and interrupts can be masked.
- E.g.



- **Handling an Interrupt:**
- Here are the steps for handling an interrupt:
    1. The CPU receives an interrupt on the INTR vector.
    2. The CPU stops the running program and transfers control to the corresponding handler in the **Interrupt Descriptor Table (IDT)**.
    3. The handler saves the current running program state.
    4. The handler executes the functionality.
    5. The handler restores (or halts) the running program.

- E.g.



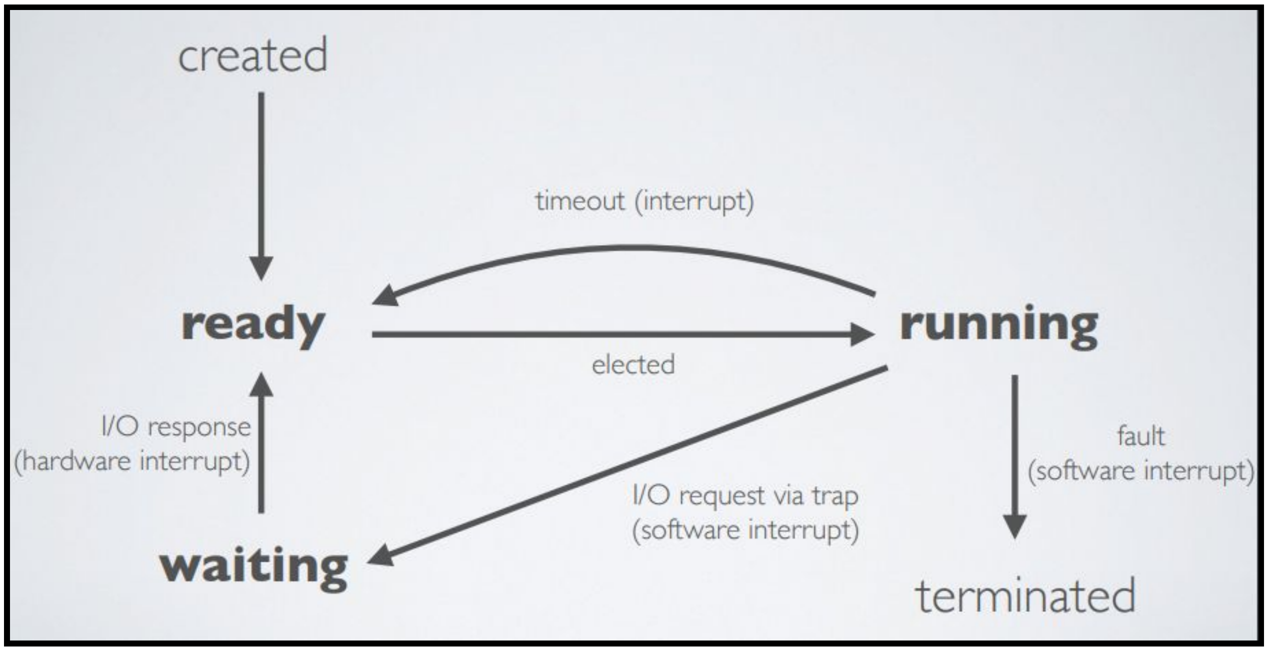
- Interrupts are defined on:
    - Linux: /proc/interrupt
    - Windows: msinfo32.exe
- E.g. When a key on the keyboard is pressed:
    1. The Keyboard controller tells PIC to cause an interrupt on IRQ #1.
    2. The PIC, which decides if the CPU should be notified.
    3. If so, IRQ 1 is translated into a vector number to index into the CPU's interrupt table.
    4. The CPU stops the current running program.
    5. The CPU invokes the current handler.
    6. The handler talks to the keyboard controller via IN and OUT instructions to ask what key was pressed.
    7. The handler does something with the result (E.g. Write to a file in Linux).
    8. The handler restores the running program.

Context Switching:

- **Context switching** is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.
  I.e. A **context switch** is a procedure that a computer's CPU follows to change from one task (or process) to another while ensuring that the tasks do not conflict. Effective context switching is critical if a computer is to provide user-friendly multitasking.
  I.e. Everytime the CPU switches the running program, it is performing a context switch.
- When the CPU runs processes concurrently:
    - Only one process at a time is running (on one core).
    - Several processes might be waiting for an I/O response.
    - Several processes might be ready to be executed.

CSCC69 Week 2 Notes

- Different states of a process:



- Context switching when:
    - When the OS receives a fault:
        1. Suspends the execution of the running process.
        2. Terminates the program.
    - When the OS receives a System Clock Interrupt or a System Call Trap (I/O request):
        3. Suspends the execution of the running process.
        4. Saves its execution context.
        5. Changes its state to ready or to waiting.
        6. Elects a new process from the ones in the ready state.
        7. Changes its state to running.
        8. Restores its execution context.
        9. Resumes its execution.

    When the OS receives any other I/O interrupt:
        1. Executes the I/O operation.
        2. Switches the process that was waiting for that I/O operation, into the ready state.
        3. Resumes the execution of the current program.
    - For each process, the OS needs to keep track of its state (running, waiting) and its execution context (registers, stack, heap and so on).
- **Process Control Block (PCB):**
- It is a data structure to record process information.
- Some of the most important information that it stores are:
    - Pid (process id) and ppid (parent process)
    - State (as either running, ready, waiting)

  - Registers (including eip and esp)
  - User
  - Address space
  - Open files
- **State Queues:**
- The OS maintains a collection of queues with the PCBs of all processes.
- There is one queue for the processes in the ready state.
- There are multiple queues for the processes in the waiting state (one queue for each type of I/O request).

<u>The Process API:</u>

- The **Process API** lets you start, retrieve information about, and manage native operating system processes.
- A **process API** is where you keep your logic and orchestration, it does not 'talk' to end systems directly but instead connects to system API's to get it's data.
- **Creating a process:**
- In every os, Linux, Windows, etc, a process is created by another process.
  Note: There is a process 0 that is created by the bootloader.
- **Process creation on Unix using fork:**
- Running **int fork()** will:
  1. Create and initialize a new PCB.
  2. Create a new address space.
  3. Initialize the address space with a copy of the entire contents of the address space of the parent (with one exception).
  4. Initialize the kernel resources to point to the resources used by the parent (e.g., open files).
  5. Place the PCB on the ready queue.
- **Note:** fork will create a new process.
- fork is very useful when the child is cooperating with the parent and/or relies upon the parent’s data to accomplish its task.
- The main argument against fork is security.
- **Process creation on Unix using exec:**
- Running **int exec(char *prog, char *argv[])** will:
  1. Stop the current process.
  2. Load the program “prog” into the process’ address space.
  3. Initialize hardware context and args for the new program.
  4. Place the PCB onto the ready queue.
- **Note:** exec does not create a new process.
- Most calls to fork are followed by exec (**spawn**).
- **Process creation on Windows:**
- Running **CreateProcess: BOOL CreateProcess(char *prog, char *args)** will:
  1. Create and initialize a new PCB.
  2. Create and initialize a new address space.
  3. Load the program specified by “prog” into the address space.

4. Copy "args" into memory allocated in address space.
5. Initialize the saved hardware context to start execution at main (or wherever specified in the file).
6. Place the PCB on the ready queue.

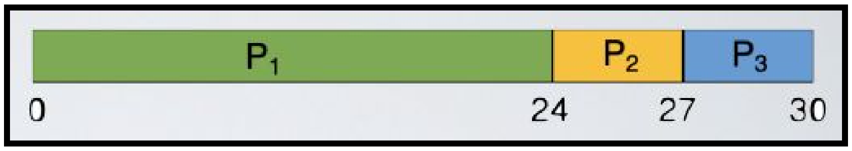
- **Wait for a process:**
- Unix: **wait(int *wstatus)**
- Windows: **WaitForSingleObject**
- **Terminate a process:**
- Unix: **exit(int status)**
- Windows: **ExitProcess(int status)**
- The OS will cleanup after the process:
    - Terminate all threads (coming next).
    - Close open files, network connections.
    - Allocated memory (and VM pages out on disk).
    - Remove PCB from kernel data structures, delete.

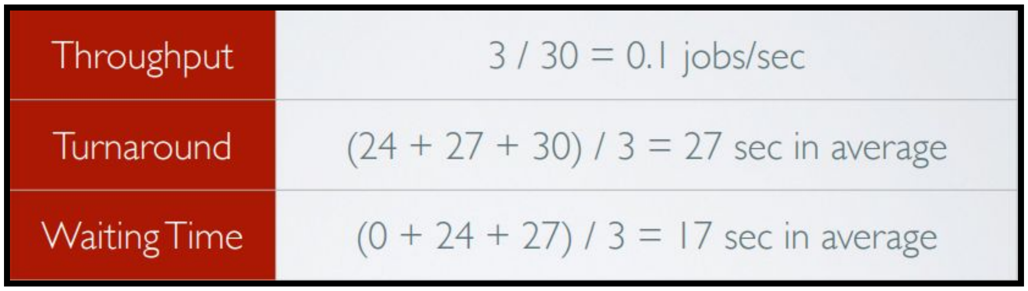
Scheduling:

- **The scheduling problem:** "Suppose you have multiple programs ready to run. Which jobs should we assign to which CPU(s)? and for how long?"
- We want to avoid starvation. **Starvation** is when a process is prevented from making progress because some other process has the resource it requires (could be CPU or a lock).
- Starvation is usually a side effect of the scheduling algorithm.
  E.g. A high priority process always prevents a low priority process from running.
  E.g. One thread always beats another when acquiring a lock.
- Starvation can be a side effect of synchronization.
  E.g. Constant supply of readers always blocks out writers.
- **Scheduling Criteria:**
- **Throughput:** This is the number of processes that complete per unit time. It is calculated by: number of jobs/time (Higher is better).
- **Turnaround time:** This is the amount of time for each process to complete.
  It is calculated by: Tfinish – Tstart (Lower is better)
  Tstart is when the process enters the queue.
- **Response time:** This is the amount of time from request to first response.
  I.e. The amount of time between waiting to ready transition and ready to running transition.
  It is calculated by: Tresponse – Trequest (Lower is better)
- The above criteria are affected by secondary criteria:
    1. **CPU utilization:** This refers to a computer's usage of processing resources, or the amount of work handled by a CPU.
       I.e. It is the amount of time a CPU is spent doing productive work.
    2. **Waiting time:** This is the amount of time each process waits in the ready queue.

- **How to balance the scheduling criterias:**
    1. **Batch Systems:**
        - Does not interact with the computer directly. There is an operator which takes similar jobs having the same requirement and groups them into batches. It is the responsibility of the operator to sort the jobs with similar needs.
        - It strives for job throughput, turnaround time (supercomputers).
        - **Advantages of Batch Operating System:**
        - It is very difficult to guess or know the time required by any job to complete. Processors of the batch systems know how long the job would be when it is in queue.
        - Multiple users can share the batch systems.
        - The idle time for the batch system is very small.
        - It is easy to manage large work repeatedly in batch systems.
        - **Disadvantages of Batch Operating System:**
        - The computer operators should be well known with batch systems.
        - Batch systems are hard to debug.
        - It is sometimes costly.
        - The other jobs will have to wait for an unknown time if any job fails.
    2. **Interactive systems:**
        - It strives to minimize response time for interactive jobs (PC).
        - However, in practice, users prefer predictable response time over faster but highly variable response time.
        - Often optimized for an average response time.
- **Two kinds of scheduling algorithm:**
    1. **Non-preemptive scheduling** (good for batch systems). Once the CPU has been allocated to a process, it keeps the CPU until it terminates.
    2. **Preemptive scheduling** (good for interactive systems). CPU can be taken from a running process and allocated to another.
- **Note:** Some computers have a mix of both scheduling types.
- **FCFS - First Come First Serve:**
- It is non-preemptive.
- It runs jobs in order that they arrive without interruption.

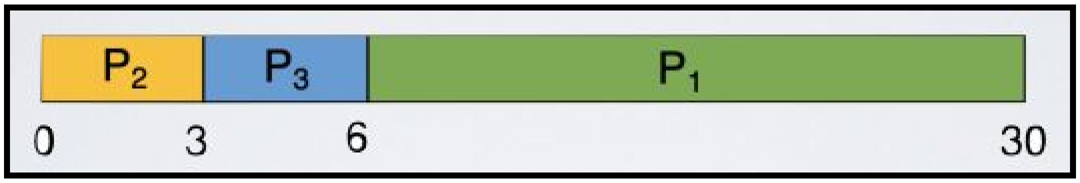
- E.g.

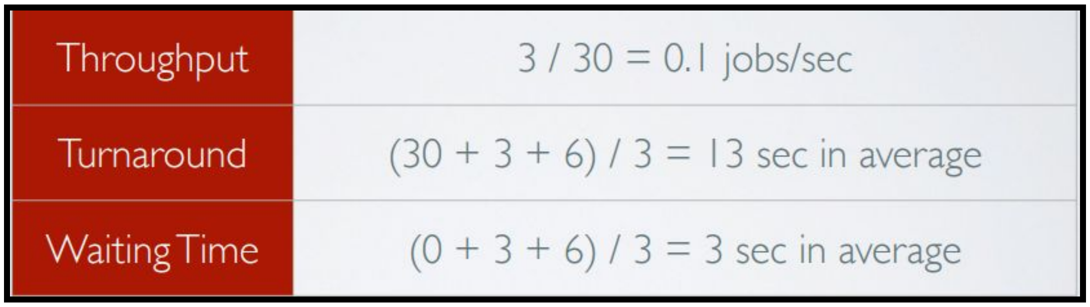




Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(24 + 27 + 30) / 3 = 27 sec in average
Waiting Time	(0 + 24 + 27) / 3 = 17 sec in average

- One problem with FCFS is the **convoy effect**, which means that all other processes wait for the one big process to release the CPU.
- **SJF - Shortest-Job-First:**
- It is non-preemptive.
- It chooses the process with the shortest processing time.
- E.g.

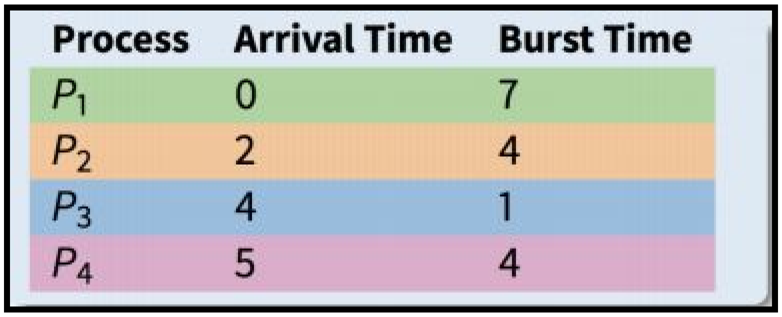




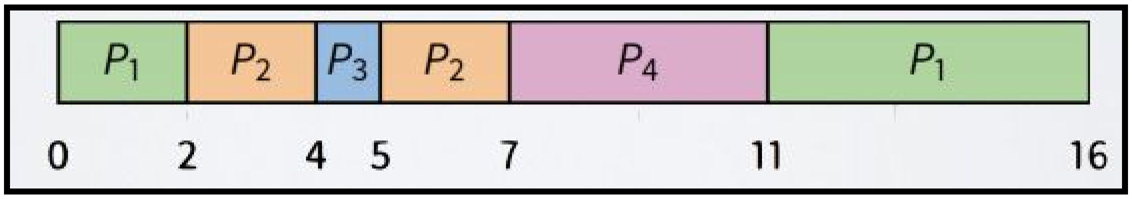
Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(30 + 3 + 6) / 3 = 13 sec in average
Waiting Time	(0 + 3 + 6) / 3 = 3 sec in average

- The problem is that we need to know processing time in advance.
- **SRTF - Shortest-Remaining-Time-First:**
- It is preemptive.
- In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute.
  I.e. The CPU executes a process, but if another process arrives that takes less time to be executed, the CPU will execute the new process first.
- If a new process arrives with CPU burst length (time it takes to be executed) less than remaining time of the current executing process, preempt the current process.

- E.g.

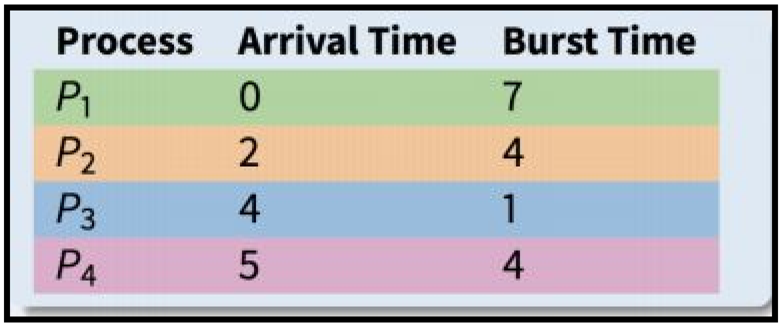


Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

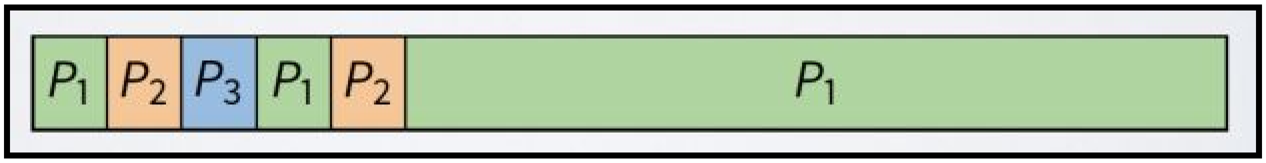


Here, the CPU starts executing P1 until P2 arrives. Since P2's execution time is less than P1's, the CPU executes P2 instead. It executes P2 for a bit, until P3 arrives. Since P3's execution time is less than P2's, the CPU executes P3 instead. After P3 has been terminated, the CPU executes P2 again. After P2 has been terminated, the CPU executes P4. After P4 has been terminated, the CPU executes P1.

- Good: Optimizes waiting time.
- Problem: Can lead to starvation.
- **RR - Round Robin:**
- It is preemptive.
- Each job is given a time slice called a **quantum**. It preempts a job after the duration of quantum, and moves it to the back of the FIFO queue.
- E.g.

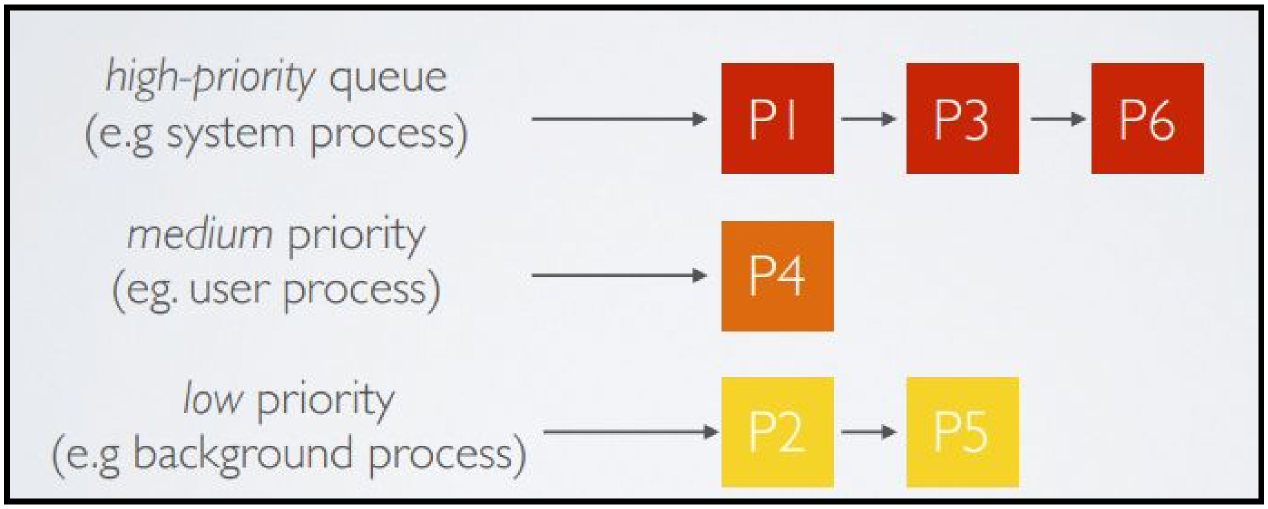


Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



- Good: Fair allocation of CPU and low waiting time (interactive).
- Problem: There is no priority between processes.
- **How to determine the time quantum:**
- **Note:** Context switches are frequent and need to be very fast.

- The quantum time should be much larger than context switch cost. Furthermore, the majority of bursts should be less than the quantum time, but not so large that the system reverts to FCFS.
- Typical values: 1–100 ms
- **Priorities:**
- We need to have priorities to:
    1. Optimize job turnaround time for “batch” jobs.
    2. Minimize response time for “interactive” jobs.
- To do this, we can have **Multilevel Queue Scheduling (MLQ)**.
- **MLQ - Multilevel Queue Scheduling:**
- It is preemptive.
- Associate a priority with each process and execute the highest priority process first. If multiple processes have the same priority, do round-robin.
- E.g.



- Problem 1: Starvation of low priority processes.
- Solution: To prevent starvation, change the priority over time by either increasing priority as a function of waiting time or decreasing priority as a function of CPU consumption.
- Problem 2: How to decide on the priority?
- Solution: To decide on the priority, observe and keep track of the process.
  E.g. If the process is old, look at past executions.
  E.g. If the process is new, look at the I/O. If the process is doing a lot of I/O, it means it doesn’t have a high priority. It doesn’t need to have a lot of CPU.
- **MLFQ - Multilevel Feedback Queue Scheduling:**
- It is preemptive.
- Running on most PCs, whether it’s Windows or Unix.
- It is the same as MLQ but changes the priority of the process based on observations/set of rules.

- Tables of rules:

Rule 1	If Priority(A) > Priority(B), A runs.
Rule 2	If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
Rule 3	When a job enters the system, it is placed at the highest priority (the topmost queue). We do this because we want to maximize response time.
Rule 4	Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (I.e. It moves down one queue).
Rule 5	After some time period S, move all the jobs in the system to the topmost queue.

- Good: Turing-award winner algorithm.