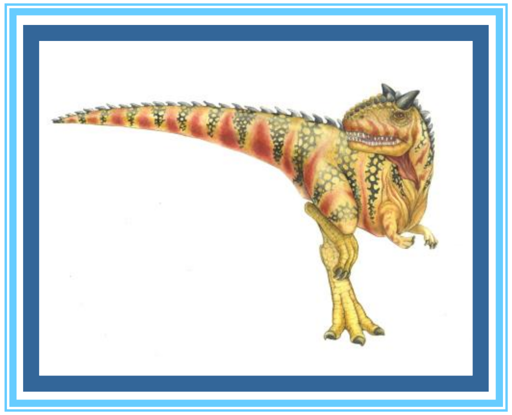
Xi'an Jiaotong-Liverpool University
西交利物浦大学

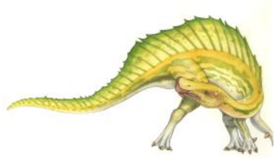
CPT104 - Operating Systems Concepts

CPU Scheduling I



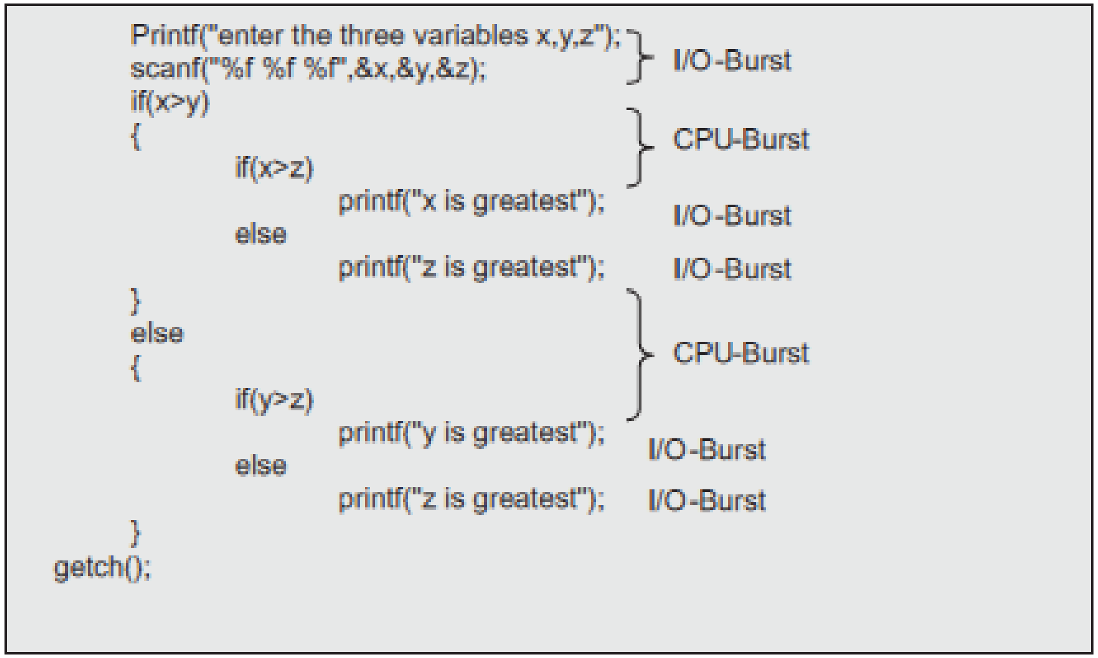
CPU Scheduling I

- Basic Concepts
- Scheduling Criteria
- Estimating future CPU burst time
- Scheduling Algorithms

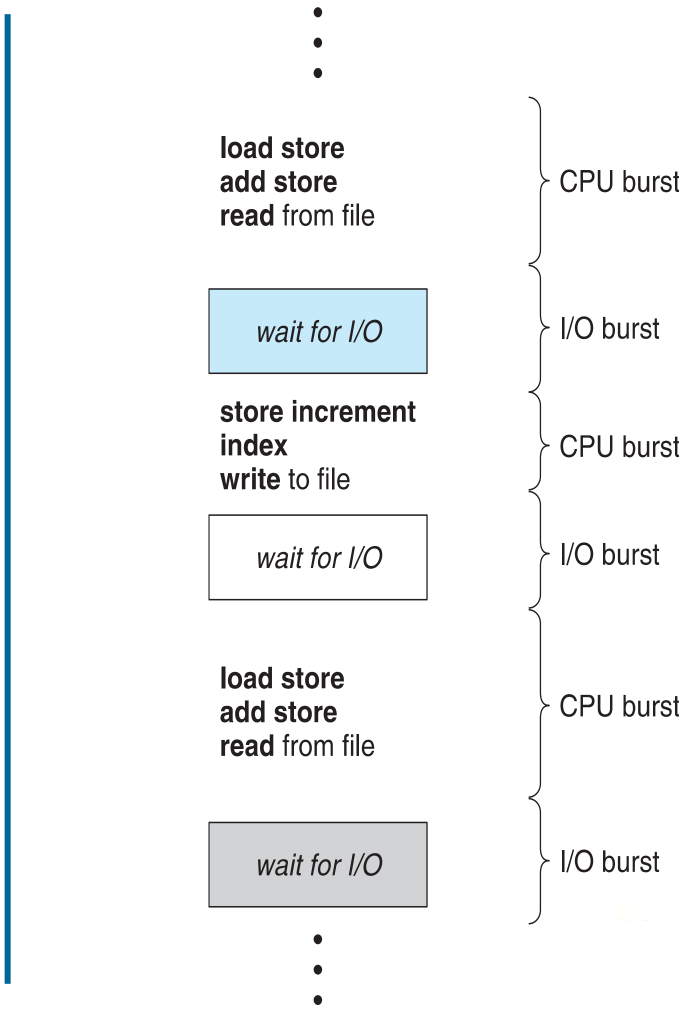


Basic Concepts

Execution phases of a process



```
        Printf("enter the three variables x,y,z");   } I/O-Burst
        scanf("%f %f %f",&x,&y,&z);
        if(x>y)
        {                                             } CPU-Burst
                if(x>z)
                        printf("x is greatest");        I/O-Burst
                else
                        printf("z is greatest");        I/O-Burst
        }
        else                                          } CPU-Burst
        {
                if(y>z)
                        printf("y is greatest");      I/O-Burst
                else
                        printf("z is greatest");      I/O-Burst
        }
    getch();
```

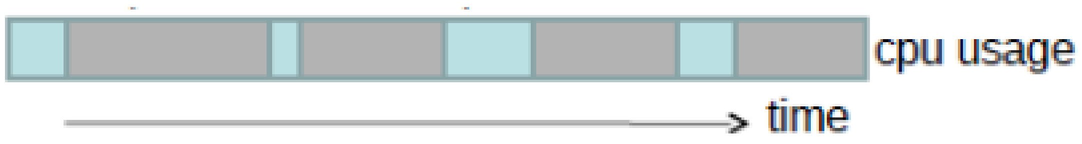


CPU burst and I/O burst in a process

Types of Processes

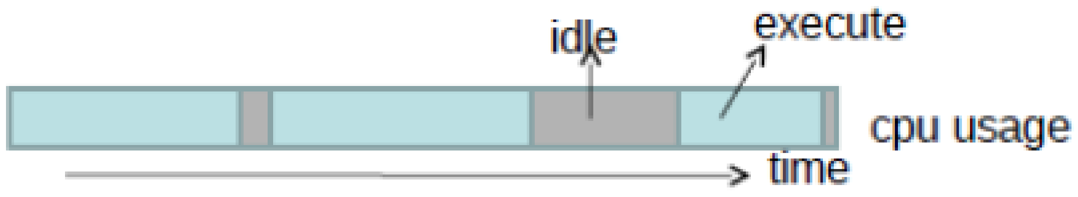
I/O bound

- Has small bursts of CPU activity and then waits for I/O (e.g. *Word processor)*
- Affects user interaction (*we want these processes to have highest priority*)



CPU bound

- Mostly CPU activity (e.g. GCC, scientific modeling, 3D rendering, etc.)
- Useful to have long CPU bursts
- Could do with lower priorities



Basic Concepts

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time.

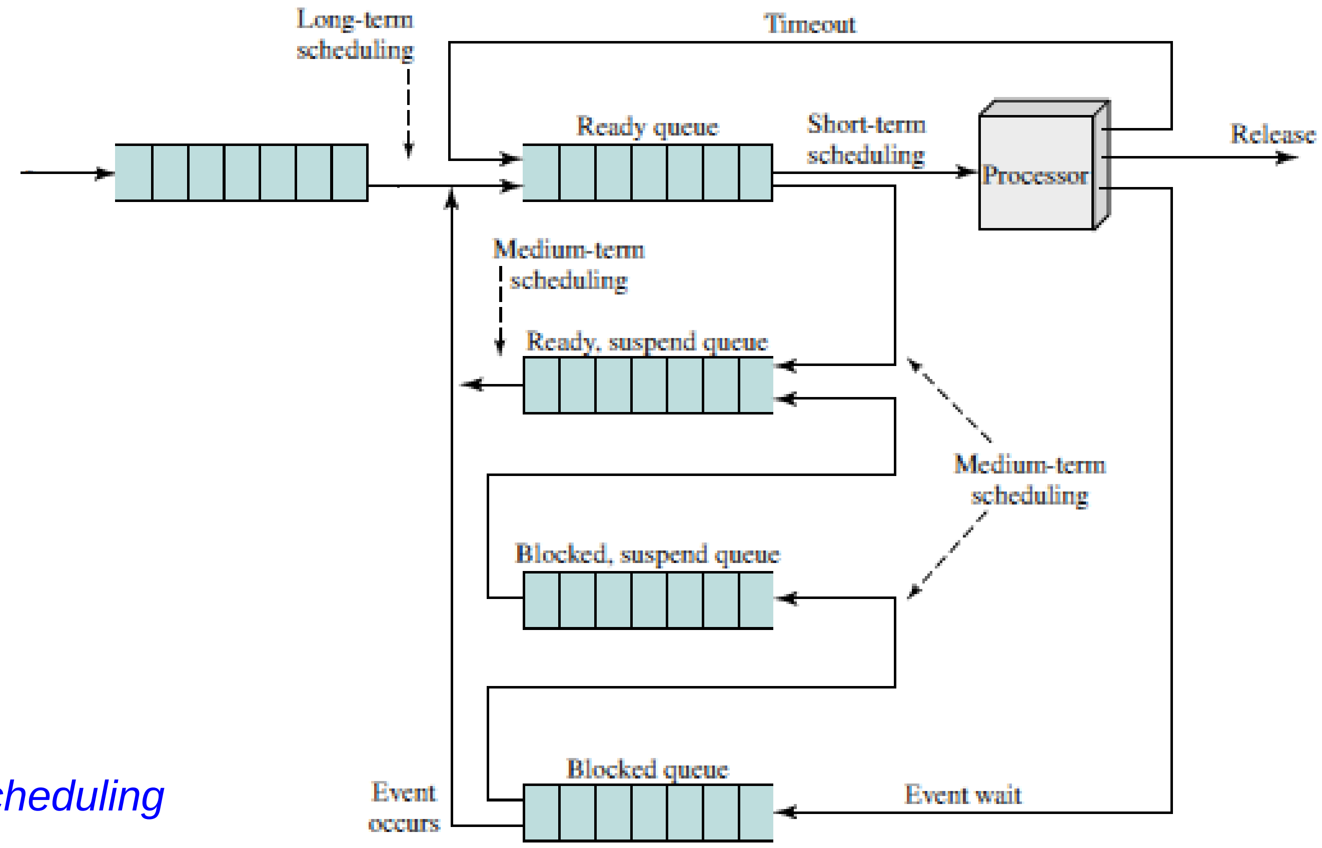
In many systems, this scheduling activity is broken down into three separate functions:

- **Long-Term Scheduler** is performed when a new process is created. This is a decision whether to add a new process to the set of processes that are currently active.
- **Short-Term Scheduler** is the actual decision of which ready process to execute next.
- **Medium-Term Scheduler** is a part of the swapping function. This is a decision whether to add a process to those that are at least partially in main memory and therefore available for execution.

Basic Concepts

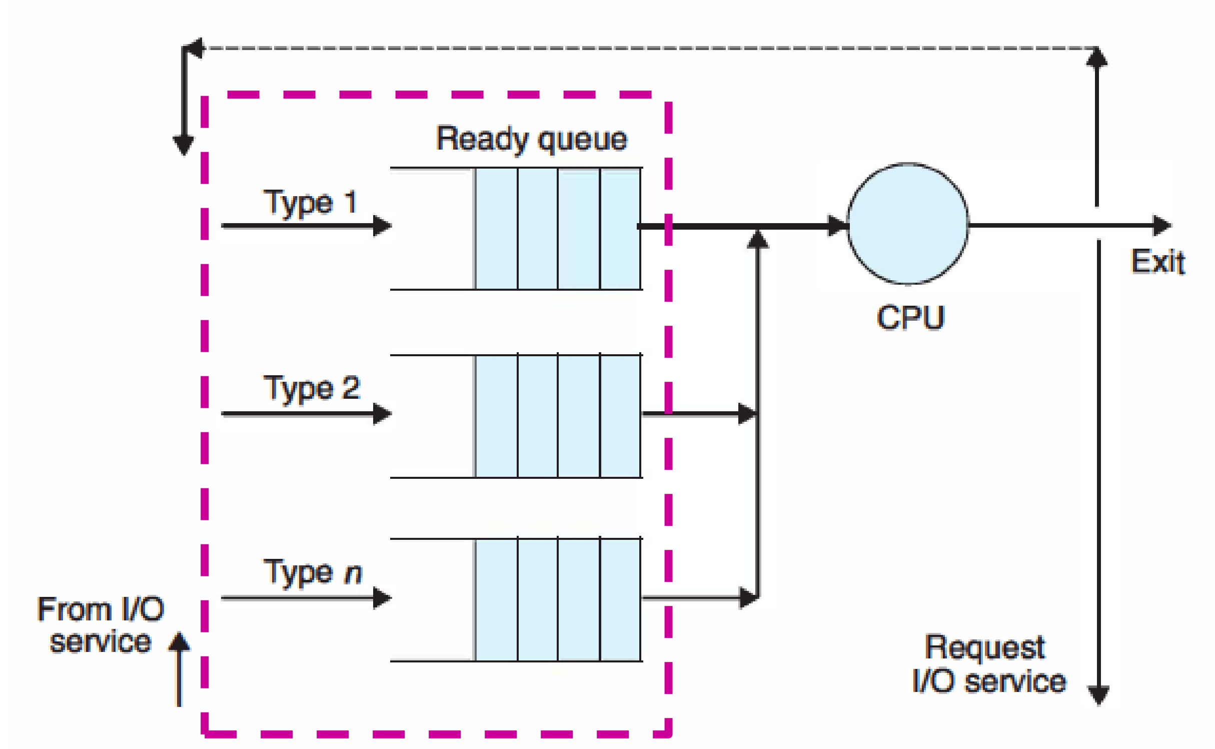
The (CPU) **scheduler** is the mechanism to select which process has to be executed next and allocates the CPU to that process.

Schedulers are responsible for transferring a process from one state to the other.



Queueing Diagram for Scheduling

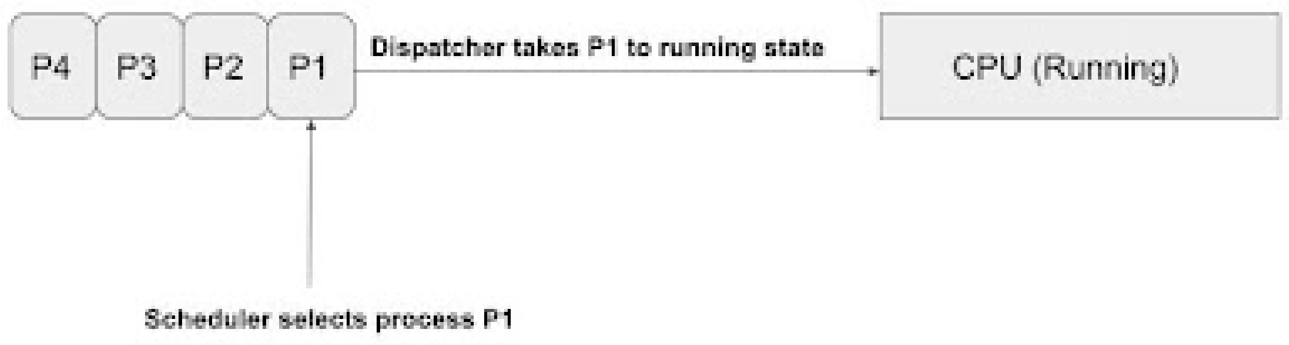
Basic Concepts



– The *ready list*, also known as a **ready queue**, in the operating system keeps a list of *all processes that are ready to run and not blocked on input/output or another blocking system request.*

– The entries in this list are pointers to a **Process Control Block**, which stores all information and state about a process.

Scheduler & Dispatcher


P4
P3
P2
P1
Dispatcher takes P1 to running state
CPU (Running)
Scheduler selects process P1

- **Schedulers** are special system software that handles process scheduling in various ways.
- **Dispatcher** is a module of the operating system that removes process from the ready queue and sends it to the CPU to complete.

CPU Scheduling Policies

PREEMPTIVE SCHEDULING = **the system may stop the execution** of the running process and after that, the context switch may provide the processor to another process**.**

The interrupted process is put back into the ready queue and will be scheduled sometime in future, according to the scheduling policy.

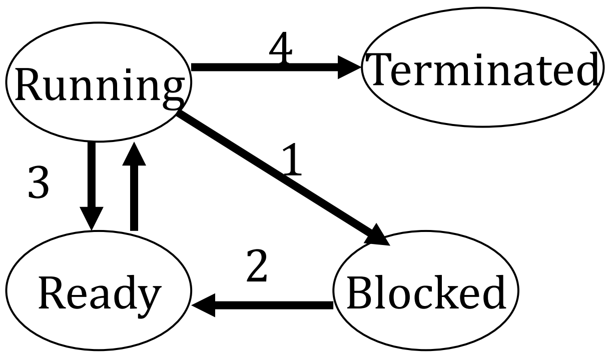
NON-PREEMPTIVE SCHEDULING = when a process is assigned to the processor, it is allowed to execute to its completion, that is, **a system cannot take away the processor from the process until it exits.**

Any other process which enters the queue has to wait until the current process finishes its CPU cycle.

Scheduling

Non-preemptive scheduling

- the CPU is allocated to the process until it terminates.
- the running process keeps the CPU until it voluntarily gives up the CPU
    - process exits
    - switches to blocked state
    - 1 and 4 only (no 3)



Preemptive scheduling

- the CPU is allocated to the processes for a specific time period
- the running process can be interrupted and must release the CPU (can be forced to give up CPU)

Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

– switching context

– switching to user mode

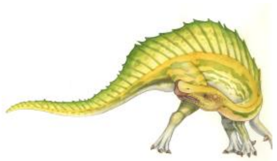
– jumping to the proper location in the user program to restart that program

• **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running.

Dispatcher is invoked during every process switch; hence it should be as fast as possible

CPU Scheduling

- Basic Concepts
- **Scheduling Criteria**
- Estimating future CPU burst time
- Scheduling Algorithms

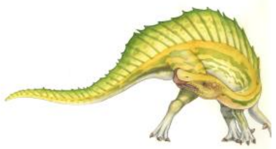


Scheduling Criteria

- **Max CPU utilization** – keep the CPU as busy as possible
- **Max Throughput** – complete as many processes as possible per unit time
- **Fairness** - give each process a fair share of CPU
- **Min Waiting time** – process should not wait long in the ready queue
- **Min Response time** – CPU should respond immediately

CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- **Estimating future CPU burst time**
- Scheduling Algorithms



Estimating future CPU burst time

The biggest problem with sorting processes this way is that we're trying to optimize our schedule using data that we don't even have!

We don't know what the CPU burst time will be for a process when it's next run.

It might immediately request I/O or it might continue running for minutes (or until the expiration of its time slice).

Computing an approximation of the length of the next CPU burst

Determining Length of Next CPU Burst

*Estimate by using lengths of past bursts: **next = average of all past bursts**.*

Let t_n = actual length of the n^{th} burst;

τ_{n+1} = predicted value for the next CPU burst;

a = weighing factor, $0 \leq a \leq 1$.

The estimate of the next CPU burst period is:

$$\tau_{n+1} = a t_n + (1 - a)\tau_n$$

Commonly, **a set to ½** -- determines the relative weight of recent and past history (τ_n)

- If **a=0**, then **recent history has no effect**

- If **a=1**, then only the **most recent CPU bursts matter**

Estimating future CPU burst time

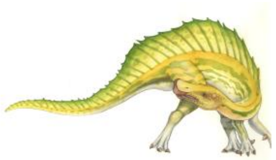
Advantage: Processes with short CPU bursts are given priority and hence run quickly (are scheduled frequently).

Disadvantages: Long-burst (CPU-intensive) processes are hurt with a long mean waiting time. In fact, if short-burst processes are always available to run, the long-burst ones may never get scheduled.

The situation where a process never gets scheduled to run is called ***starvation***.

Scheduling Algorithms

Order of scheduling matters



Terms the algorithms deal with…

Arrival Time (AT): Time at which the process arrives in the ready queue.

Completion Time: Time at which process completes its execution.

Burst Time: Time required by a process for CPU execution.

Turnaround Time (TT): the total amount of time spent by the process from coming in the ready state for the first time to its completion.

Turnaround time = Exit time - Arrival time.

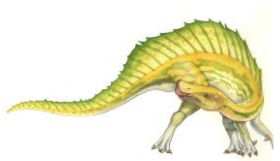
Waiting Time (WT): The total time spent by the process/thread in the ready state waiting for CPU.

Waiting Time = Turn Around Time – Burst Time

Response time: Time at which the process gets the CPU for the first time.

Scheduling Algorithms

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Shortest Remaining Time First (SRTF) Scheduling
- Priority Scheduling
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling
- Multilevel Feedback Queue Scheduling

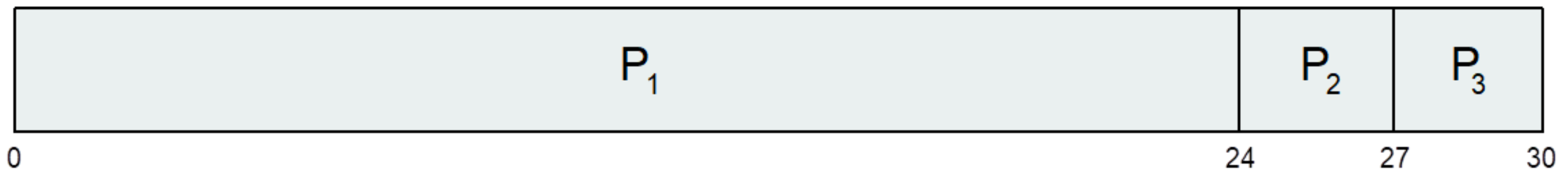


First- Come, First-Served (FCFS) Scheduling

- Processes are executed on **first come, first served** basis.
- Poor in performance as average wait time is high.

Process	**Burst Time (ms)**
P1	24
P2	3
P3	3

The Gantt chart



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17

Shortest Job First (SJF)
NO preemption

- Schedule first processes with a short burst time.
- the shortest burst time is scheduled first.

• **Advantages**

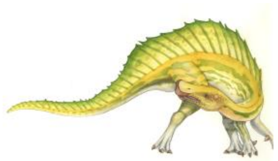
– Minimizes average wait time and average response time

• **Disadvantages**

– **Not practical** : difficult to predict burst time

• Learning to predict future

– May starve long jobs

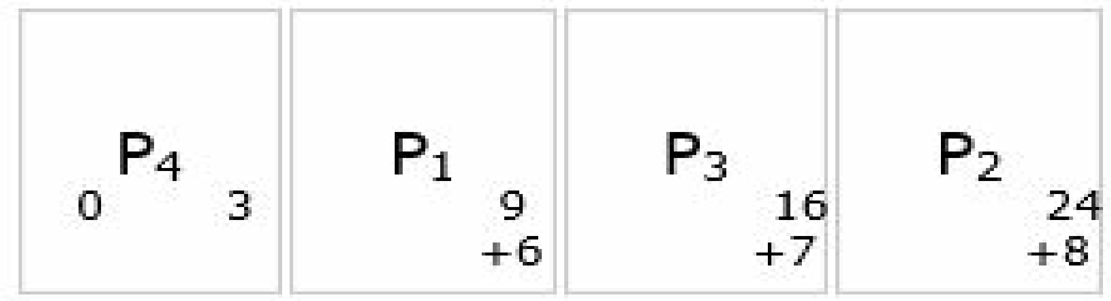


Shortest-Job-First (SJF) Scheduling (Example)

without preemption

Process	Burst Time (ms)
P1	6
P2	8
P3	7
P4	3

SJF scheduling **Gantt chart**



Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Shortest-Remaining-Time-First SRTF

(SJF with preemption)

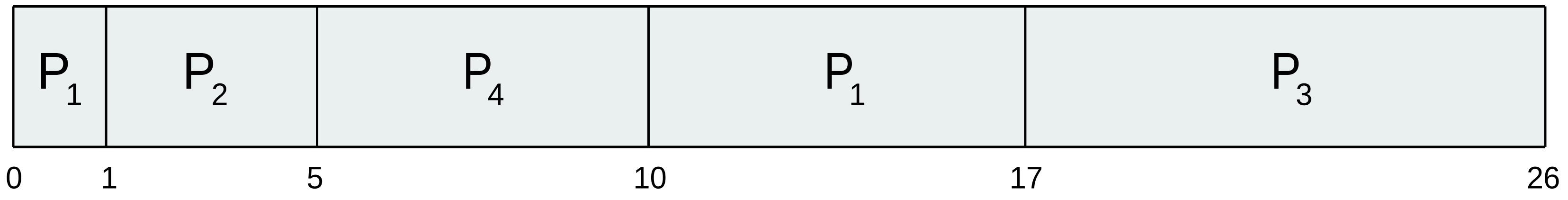
- **If a new process arrives with a shorter burst time than remaining of current process, then schedule new process**
- Further reduces average waiting time and average response time
- *Context Switch* - the context of the process is saved in the *Process Control Block PCB* when the process is removed from the execution and the next process is scheduled.
- This PCB is accessed on the **next execution** of this process.

Shortest-Remaining-Time-First SRTF

Example

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt chart



Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

- Each process is assigned a **priority** (an integer number).
- The CPU is allocated to the process with the highest priority (**smallest integer ≡ highest priority**)
- Priorities may be:
  - **Internal priorities** based on criteria within OS. *Ex: memory needs*.
  - **External priorities** based on criteria outside OS. *Ex: assigned by administrators.*

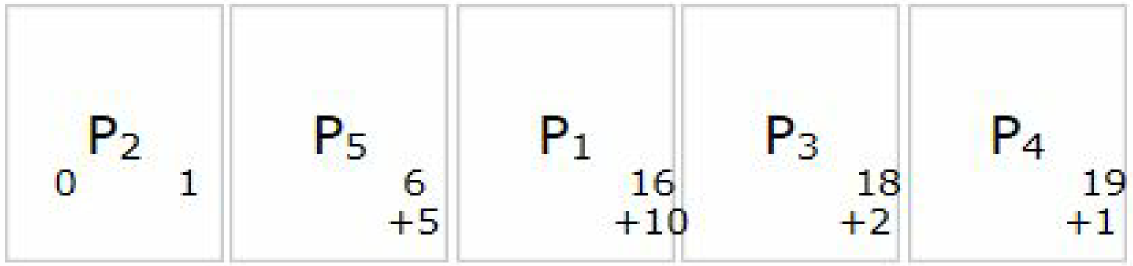
!!! PROBLEM ≡ **Starvation** – low priority processes may never execute

SOLUTION ≡ **Aging** – as time progresses increase the priority of the process *Example: do priority = priority + 1 every 15 minutes*

Priority Scheduling (Ex.1.)

Each process has assigned a priority (integer number).
Process with highest priority is to be executed first and so on..

Process	**Burst Time**	**Priority**
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



Average waiting time = = (0 + 1 + 6 + 16 + 18) / 5 = 8.2

Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (**time quantum** or **time-slice**), usually 10-100 milliseconds.

- After this time has elapsed, the process is preempted and added to the end of the ready queue *Ready queue is treated as a circular queue*

- If there are **n** processes in the ready queue and the time quantum is **q**, then each process gets **1/n** of the CPU time in chunks of at most **q** time units at once. No process waits more than **(n-1)q** time units.

- **Performance**
  - **q large** $\Rightarrow$ RR scheduling = FCFS scheduling
  - **q small** $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Round Robin (RR) Scheduling (Ex.1.)

Process	**Burst Time**
P1	24
P2	3
P3	3

The average wait time would be:

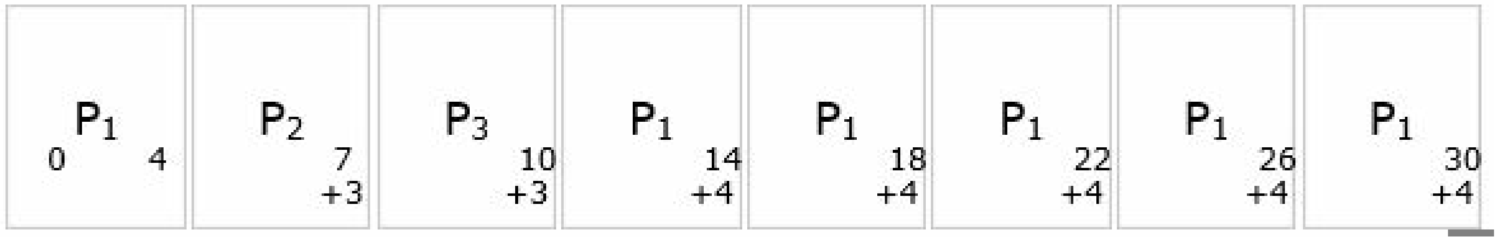
$P_{1_w} = 30 - 24 = 6$

$P_{2_w} = 7 - 3 = 4$

$P_{3_w} = 10 - 3 = 7$

AWT = (6 + 4 + 7) / 3 = 5.66ms

Time quantum = 4ms



P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1
0 4	7	10	14	18	22	26	30
	+3	+3	+4	+4	+4	+4	+4

Multilevel Queue Scheduling

- *Ready* queue is partitioned into separate queues;

EXAMPLE: Two queues containing

- **foreground** (interactive) **processes** . May have externally defined priority over background processes
- **background** (batch) **processes**

Process is permanently associated to a given queue; **no move to a different queue**

There are two types of scheduling in multi-level queue scheduling:

- Scheduling among the queues.
- Scheduling between the processes of the selected queue.

Must schedule among the queues too (not just processes):

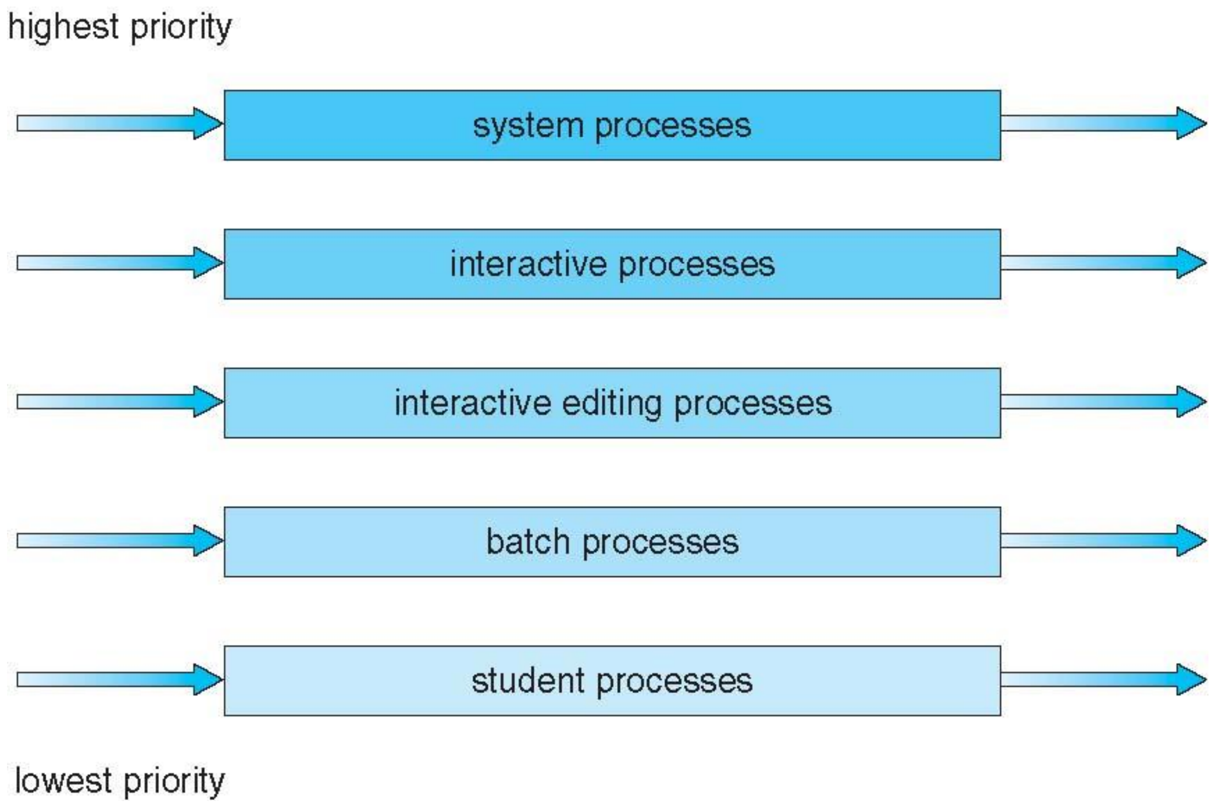
- **Fixed priority scheduling**; (i.e., serve all from foreground then from background). Possibility of starvation.
- **Time slice** – each queue gets a certain amount of CPU time which it can schedule amongst its processes;

➢ 80% to foreground in RR, and 20% to background in FCFS

Multilevel Queue Scheduling

The various categories of processes can be:

- Interactive processes
- Non-interactive processes
- CPU-bound processes
- I/O-bound processes
- Foreground processes
- Background processes



Multilevel Feedback Queue Scheduling

- **Multilevel feedback queues** - *automatically place processes into priority levels* based on their CPU burst behavior.

- I/O-intensive processes will end up on **higher priority queues** and CPU-intensive processes will end up on **low priority queues**.

A process **can move between the various queues**

A **multilevel feedback queue** uses **two basic rules**:

1. A new process gets placed in the highest priority queue.

2. If a process does not complete its execution after a time quantum, there are two options: it will stay at the same priority level or it moves to the next lower priority level.

Multilevel Feedback Queue Scheduling

A process can move between the various queues;

- aging can be implemented this way.

• Multilevel-feedback-queue scheduler defined by the following **parameters**:

– number of queues

– scheduling algorithms for each queue

– method used to determine when to upgrade a process

– method used to determine when to demote a process

– method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queue Scheduling

EXAMPLE:

Three queues:

***Q_0* – RR with time quantum 8 milliseconds**

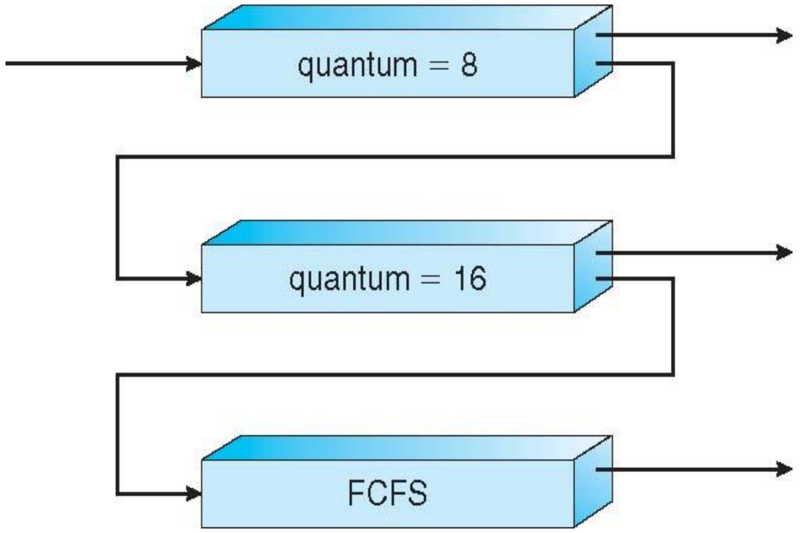
- Highest priority.

***Q_1* – RR time quantum 16 milliseconds**

- Medium priority.

***Q_2* – FCFS**

- Lowest priority.



Scheduling:

1. A new job enters queue Q_0 which is served first-come first served
   - When it gains CPU, job receives 8 milliseconds
   - If it does not finish in 8 milliseconds, job is moved to queue Q_1
2. At Q_1 job is again served RR and receives 16 additional milliseconds
   - If it still does not complete, it is preempted and moved to queue Q_2

End of Lecture

Summary

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms

Reading

- Textbook 9th edition, **chapter 6 of the module textbook**