Unit-3

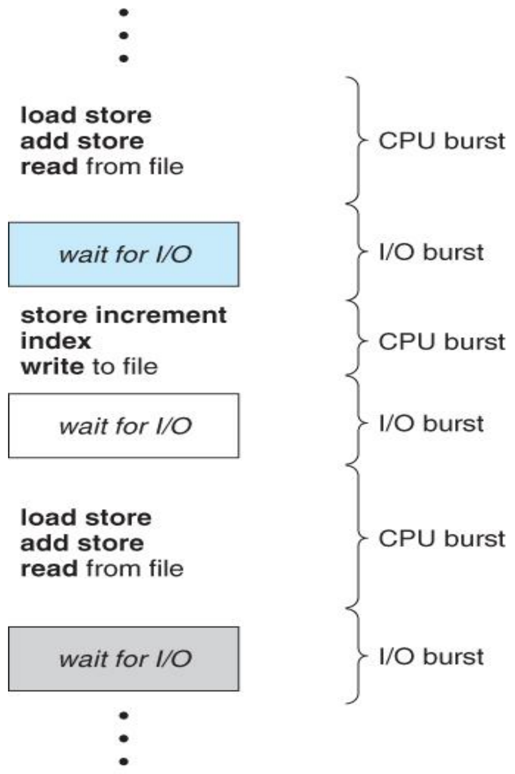
Process Scheduling and Scheduling algorithms

Foundation and Scheduling objectives

- Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)
- In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.
- A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
- The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

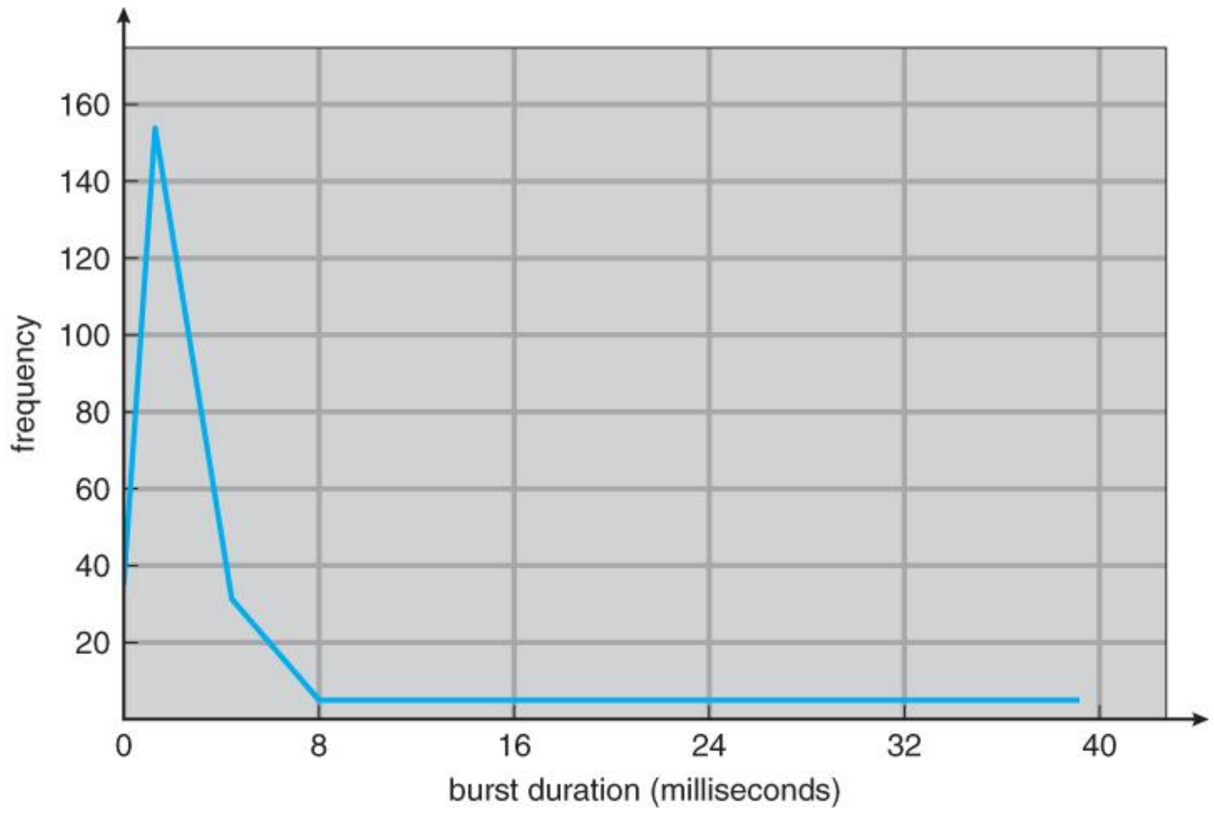
CPU-I/O Burst Cycle

- Almost all processes alternate between two states in a continuing cycle, as shown in the figure below:
    - A CPU burst of performing calculations, and
    - An I/O burst, waiting for data transfer in or out of the system.



Alternating sequence of CPU and I/O bursts.

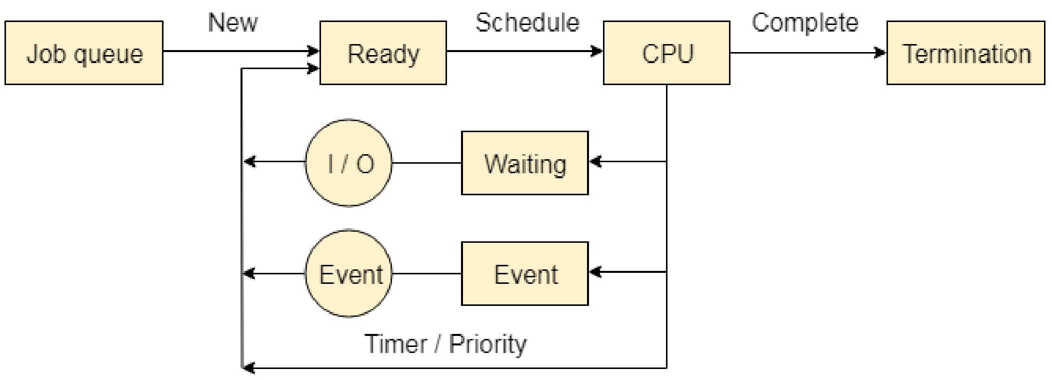
- CPU bursts vary from process to process, and from program to program, but an extensive study shows frequency patterns similar to that shown in figure below:



Histogram of CPU-burst durations

Scheduling Queues

- All processes, upon entering into the system, are stored in the **Job Queue**. It is maintained in the secondary memory. The **long term scheduler (Job scheduler)** picks some of the jobs and put them in the primary memory.
- Processes in the Ready state are placed in the **Ready Queue**. Ready queue is maintained in primary memory. The **short term scheduler (CPU Scheduler)** picks the job from the ready queue and dispatch to the CPU for the execution.
- Processes waiting for a device to become available are placed in **Device Queues / Waiting Queues**. There are unique device queues available for each I/O device.



A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

- The process could issue an I/O request, and then be placed in the I/O queue.
- The process could create a new sub process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

CPU Scheduler

- Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue. There are several alternatives to choose from, as well as numerous adjustable parameters for each algorithm, which is the basic subject of this entire chapter.

Types of Schedulers

A scheduler is a type of system software that allows you to handle process scheduling.

There are mainly three types of Process Schedulers:

1. Long Term
2. Short Term
3. Medium Term

Long Term Scheduler

Long term scheduler is also known as a **job scheduler**. This scheduler regulates the program and select process from the job queue and loads them into memory for execution. It also regulates the degree of multi-programing.

However, the main goal of this type of scheduler is to offer a balanced mix of jobs, like Processor, I/O jobs., that allows managing multiprogramming.

Medium Term Scheduler

Medium-term scheduling is an important part of **swapping**. It enables you to handle the swapped out-processes. In this scheduler, a running process can become suspended, which makes an I/O request.

A running process can become suspended if it makes an I/O request. A suspended processes can't make any progress towards completion. In order to remove the process from memory and make space for other processes, the suspended process should be moved to secondary storage.

Short Term Scheduler

Short term scheduling is also known as **CPU scheduler**. The main goal of this scheduler is to boost the system performance according to set criteria. This helps you to select from a group of processes that are ready to execute and allocates CPU to one of them. The dispatcher gives control of the CPU to the process selected by the short term scheduler.

Difference between Schedulers

Long-Term Vs. Short Term Vs. Medium-Term

Long-Term	**Short-Term**	**Medium-Term**
Long term is also known as a job scheduler	Short term is also known as CPU scheduler	Medium-term is also called swapping scheduler.

Long-Term	**Short-Term**	**Medium-Term**
It is either absent or minimal in a time-sharing system.	It is insignificant in the time-sharing order.	This scheduler is an element of Time-sharing systems.
Speed is less compared to the short term scheduler.	Speed is the fastest compared to the short-term and medium-term scheduler.	It offers medium speed.
Allow you to select processes from the loads and pool back into the memory	It only selects processes that is in a ready state of the execution.	It helps you to send process back to memory.
Offers full control	Offers less control	Reduce the level of multiprogramming.

Pre-emptive and non-pre-emptive Scheduling

Pre-emptive Scheduling

- CPU scheduling decisions take place under one of four conditions:
    - When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
    - When a process switches from the running state to the ready state, for example in response to an interrupt.
    - When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
    - When a process terminates.
- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be ***non-preemptive***, or ***cooperative***. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be ***preemptive.***
- Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.
- Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures.
- Preemption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the preemption occurs. Most modern UNIXes deal with this problem by making the process wait until the system call has either completed or blocked before allowing the preemption Unfortunately this solution is problematic for real-time systems, as real-time response can no longer be guaranteed.
- Some critical sections of code protect themselves from con currency problems by disabling interrupts before entering the critical section and re-enabling interrupts on exiting the section. Needless to say, this should only be done in rare situations, and only on very short pieces of code that will finish quickly, (usually just a few machine instructions).

Dispatcher

- The **dispatcher** is the module that gives control of the CPU to the process selected by the scheduler. This function involves:
  - Switching context.
  - Switching to user mode.
  - Jumping to the proper location in the newly loaded program.
- The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as **dispatch latency.**

Scheduling Criteria

There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:

- **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded)
- **Throughput** - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
- **Turnaround time** - Time required for a particular process to complete, from submission time to completion. (Wall clock time)
- **Waiting time** - How much time processes spend in the ready queue waiting for their turn to get the CPU.
  - (**Load average** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who")
- **Response time** - The time taken in an interactive program from the issuance of a command to the ***commence*** of a response to that command.

In general one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others). However sometimes one wants to do something different, such as to minimize the maximum response time.

Sometimes it is most desirable to minimize the ***variance*** of a criterion than the actual value. I.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

Formulas for calculation of various scheduling timings

Given: Process nos, Arrival Time (AT), Burst Time (BT)

Find: Completion Time (CT), Turn Around Time (TAT), Waiting Time (WT)

CT: Last occurrence of a job in the Gantt chart

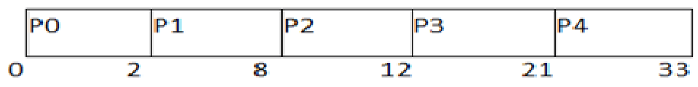
TAT = CT - AT

WT = TAT – BT

FCFS

FCFS

Process ID	Arrival Time	Burst Time
0	0	2
1	1	6
2	2	4
3	3	9
4	4	12



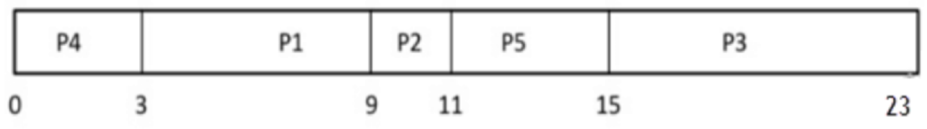
Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	8	4
3	3	9	21	18	9
4	4	12	33	29	17

SJF (Non-Pre-emptive)

NON PRE-EMPTIVE SJF

Consider the following five processes each having its own unique burst time and arrival time.

Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4



SJF (Pre-emptive)/SRTF

PRE-EMPTIVE SJF

Consider the following five process:

Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

P4	P1	P5	P2	P5	P1	P3	
0	3	4	5	7	10	15	23

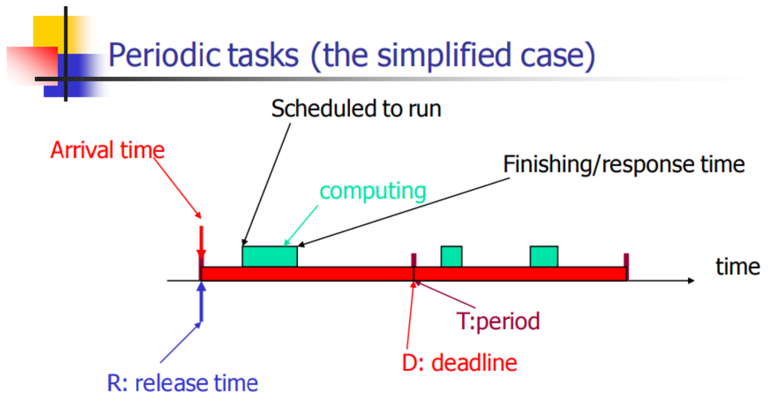
RR

Process ID	Arrival Time	Burst Time
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

P1	**P2**	**P3**	**P4**	**P5**	**P1**	**P6**	**P2**	**P5**	
0	4	8	11	12	16	17	21	23	24

Real Time scheduling: RM and EDF

- **Rate monotonic scheduling** is a priority algorithm that belongs to the static priority scheduling category of Real Time Operating Systems.
- It is pre-emptive in nature.
- The priority is decided according to the cycle time of the processes that are involved.
- If the process has small job duration, then it has the highest priority.
- Thus if a process with highest priority starts execution, it will pre-empt the other running processes. The priority of a process is inversely proportional to the period it will run for.

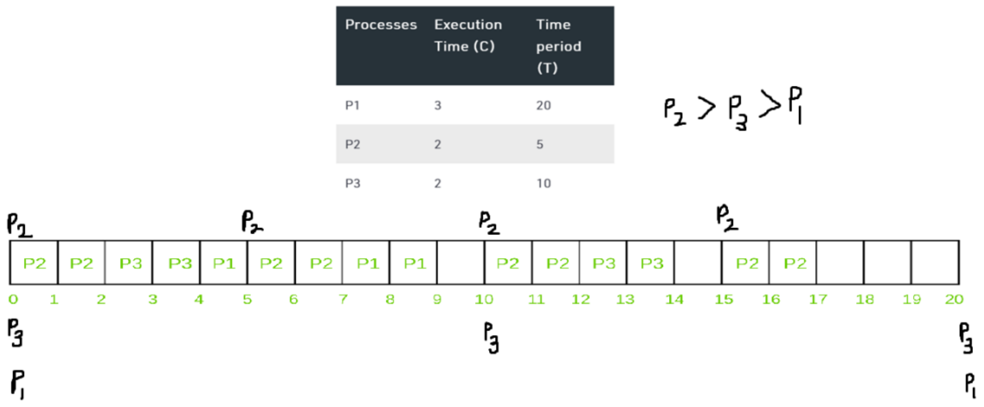


A set of processes can be scheduled only if they satisfy the following equation:

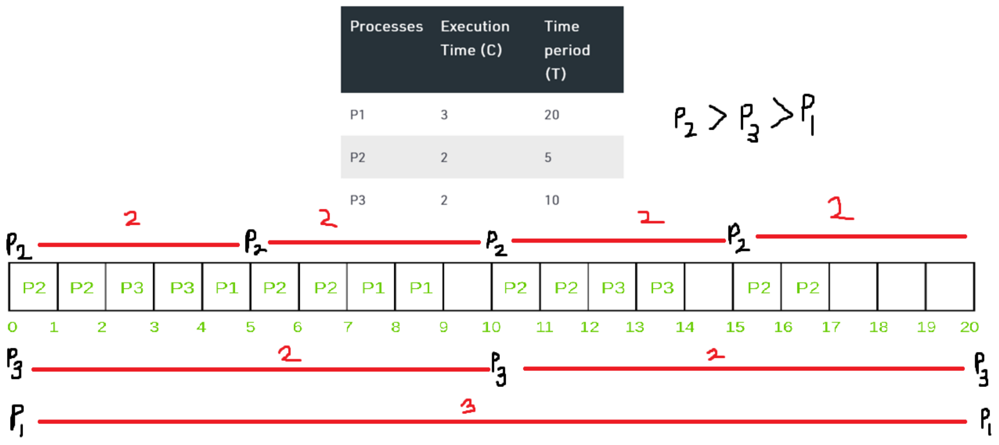
$$\sum_{k=1}^{n} \frac{Ci}{Ti} \leq U = n\left(2^{1/n} - 1\right)$$

Where n is the number of processes in the process set, Ci is the computation time of the process, Ti is the Time period for the process to run and U is the processor utilization.

Resource: https://www.geeksforgeeks.org/rate-monotonic-scheduling/



Processes	Execution Time (C)	Time period (T)
P1	3	20
P2	2	5
P3	2	10

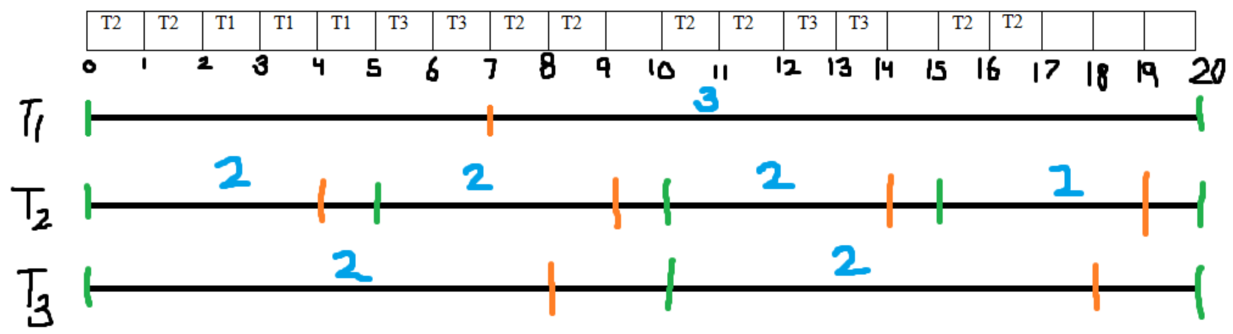


Processes	Execution Time (C)	Time period (T)
P1	3	20
P2	2	5
P3	2	10

- **Earliest deadline first** (**EDF**) or **least time to go** is priority-based preemptive scheduling policy
- **EDF is a** dynamic priority scheduling algorithm used in real-time operating systems to place processes in a priority queue.
- Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline.
- job with earliest (absolute) deadline has highest priority
- does not require knowledge of execution times
- is known to be an ***optimal*** policy for a single processor (?)
- With scheduling periodic processes that have deadlines equal to their periods, EDF has a utilization bound of 100%. Thus, the schedulability test for EDF is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1,$$

Tasks	Execution Time (C)	Deadline	Time Period (T)
T1	3	7	20
T2	2	4	5
T3	2	8	10



T2	T2	T1	T1	T1	T3	T3	T2	T2		T2	T2	T3	T3		T2	T2			