Chap 2: Process & Threads

BY PRATYUSA MUKHERJEE, ASSISTANT PROFESSOR (I)

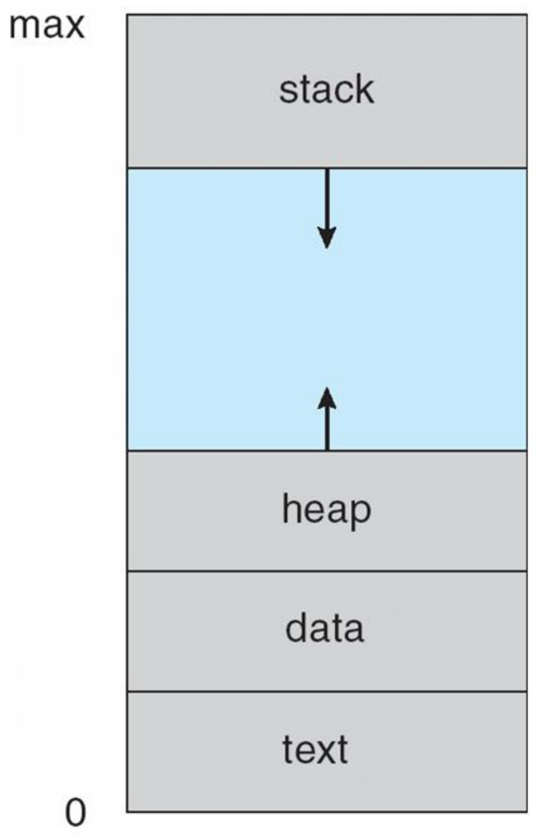
KIIT DEEMED TO BE UNIVERSITY

Process

A program in execution is called a process. It is executed in a sequential manner

It includes

- The program code, also called **text section**.
- **Current activity** which is represented by value of program counter and contents of processor registers
- **Stack** containing temporary data (Function parameters, return addresses, local variables)
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



Process vs Program

Process	Program
Active Entity	Passive Entity
Instructions in machine code only	Instruction in any language
Present in Main Memory only	Present in Secondary Storage

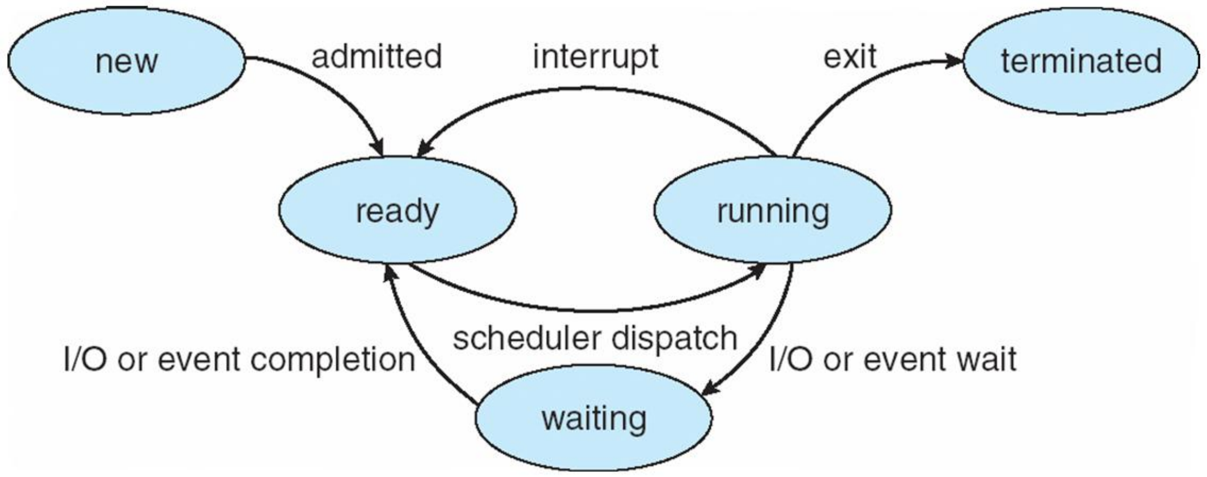
Program becomes process when executable file loaded into memory

One program can be several processes

Process State

As a process executes, it changes its **State**

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur (I/O completion or signal reception)
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution



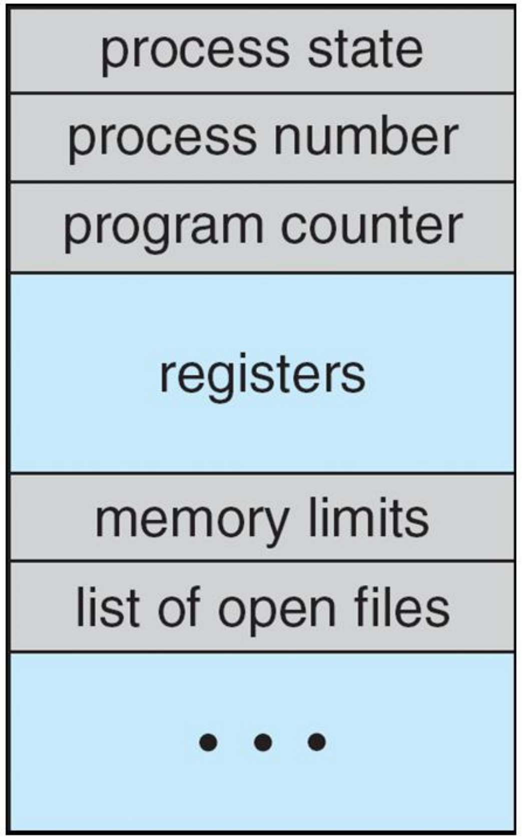
Only 1 process can run on 1 processor but several others may be ready and waiting

Process Control Block (PCB)

Each process is represented in the OS by a PCB/ task control block. It is a repository for any information that varies from process to process.

Information associated with each process

- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information**- priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files

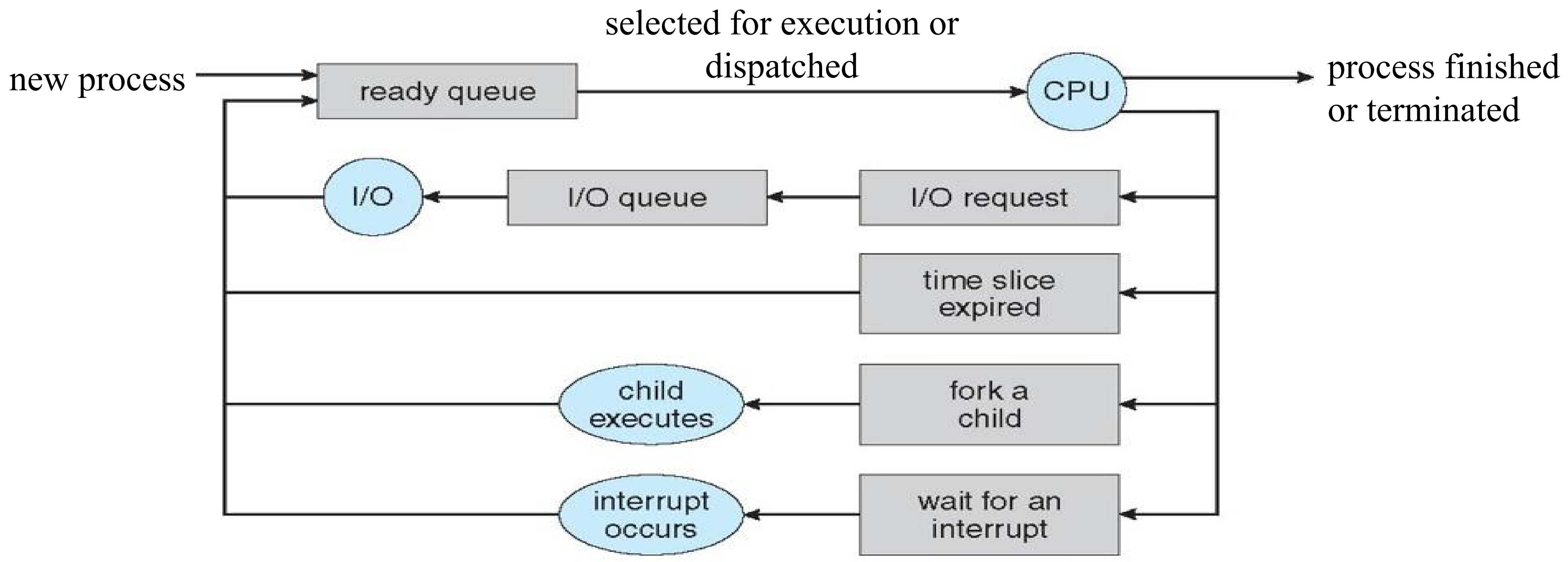


Process Scheduler

- To maximize CPU usage, processes must be switched quickly onto the CPU for time sharing.
- **Process scheduler** selects one process among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - ☐**Job queue** – set of all processes in the system. All processes that enter the system, enter here
  - ☐**Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - ☐**Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Representation of Process Scheduling

Queueing diagram represents queues, resources, flows



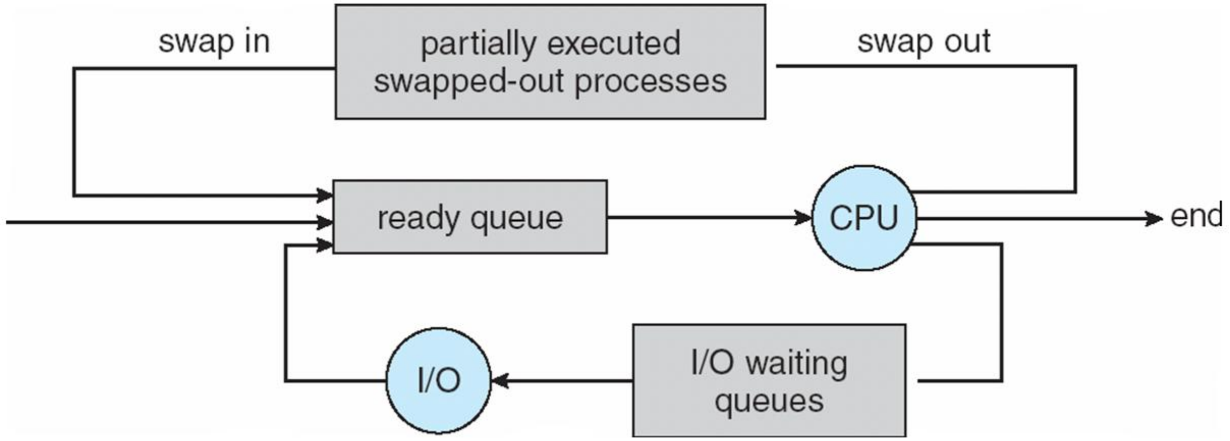
Schedulers

OS selects processes from various scheduling Queues by an appropriate scheduler.

- **Long-term scheduler (or job scheduler)** – selects which processes should be brought from mass storage and load them into memory / ready queue for execution. It is invoked infrequently (seconds, minutes) hence may be slow. It controls the degree of multiprogramming.
- **Short-term scheduler (or CPU scheduler)** – selects a process that is ready to execute next and allocates CPU to it. Sometimes it is the only scheduler in a system. It is invoked frequently (milliseconds) thus must be fast.
- Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; few very long CPU bursts
- Schedulers strives for good process mix

Mid Term Scheduler

- Medium-term scheduler can be added if degree of multiple programming needs to decrease
- It implements a swapping algorithm to decide which partially run programs to swap to disk and when to reinstate them.
- Thus, it removes process from memory, stores on disk, brings back in from disk to continue execution.
- It is an intermediate scheduler.
- **Swapping: Remove process from memory temporarily and reduce degree of multiprogramming. The process can be reintroduced into memory at a later time.**

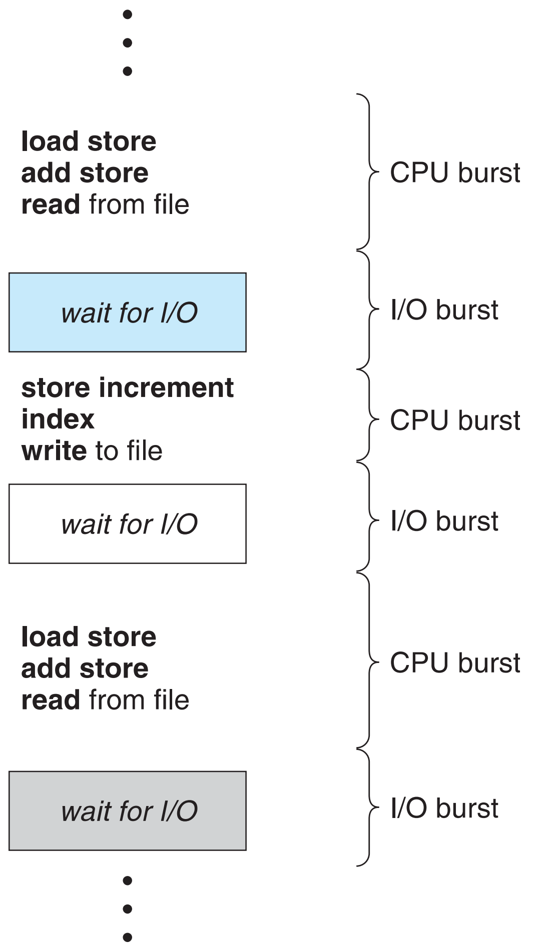


Process Address Space

- **Stack Space** : For function and system calls.
- **Data Space**: for variables
- **Text Space**: for program code

The Foremost Concept

- Maximum CPU utilization can be obtained by multiprogramming
- It is a combination of **CPU–I/O Burst Cycle**. Thus Process execution consists of a cycle of CPU execution and I/O wait
- **Remember it is CPU burst followed by I/O burst**
- **CPU burst distribution is of main concern**



When are Scheduling Decisions Taken

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
- Scheduling decisions are taken when a Process
  1. switches from running to waiting state
  2. switches from running to ready state
  3. switches from waiting to ready state
  4. is terminated
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Process Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – Number of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process. Time of submission to completion.
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Can you guess the optimal scheduling criterias??

Dispatcher

- It is module that gives control of CPU to the selected process by short term scheduler.
- It includes
  - switching context,
  - switching to user mode,
  - jumping to proper location in program to restart process

- **Dispatch Latency**: Time to stop a currently running process and start another.

Context Switching

- When CPU switches to another process, the **system must save the state of the old process** and **load the saved state for the new process** via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- The more complex the OS and the PCB, the longer the context switch
- Time dependent on hardware support
- Some hardware provides multiple sets of registers per CPU leads to multiple contexts loaded at once

Process Scheduling Algorithms

- ❖First Come First Serve (FCFS) Scheduling
- ❖Shortest Job First (SJF) Scheduling
- ❖Shortest Remaining Time First (SRTF) Scheduling
- ❖Priority Scheduling
- ❖Round Robin (RR) Scheduling
- ❖Highest Response Ratio Next (HRRN) Scheduling

Detailed Numericals discussed over whiteboard along with an illustration of Least Job First, Longest Job First, FCFS with Overhead, Processes involving I/O

Additional Points Scheduling Algorithms

- **Convoy Effect**: In FCFS Priority if a lengthy process arrives before other short processes, all these other processes wait for that lengthy process to get off the CPU as it is non-preemptive. Thus the CPU Utilization and throughput reduces drastically. The Waiting and Response Times of the processes increase to great extent.

Solution: Better to send shorter processes first (SJF Scheduling).

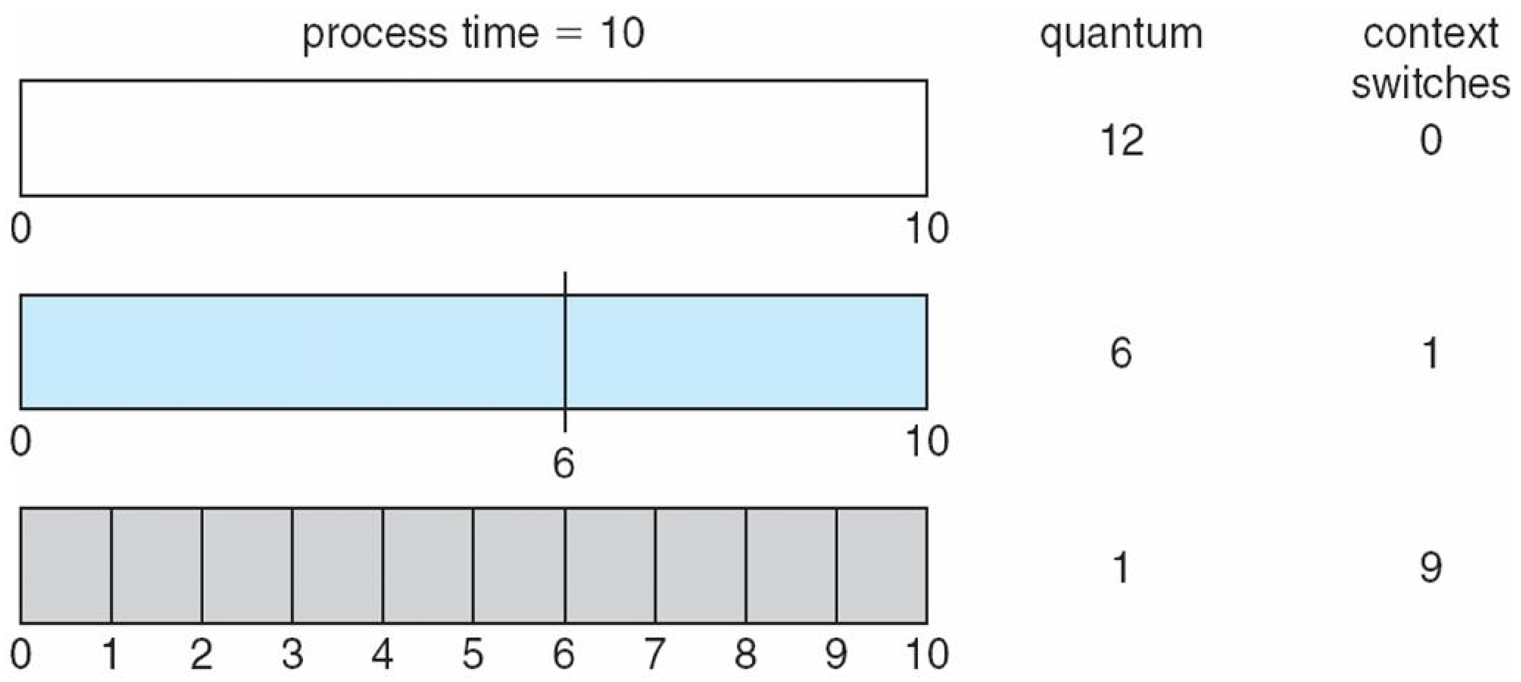
- **Starvation / Indefinite Blocking**: In Priority Scheduling, a low priority process can wait indefinitely to get access of the CPU as everytime it can be deprived by other high priority processes. This again drastically affects the Waiting and Response Times of this low priority process.

Solution: **Aging** which means gradually increase the priority of processes that wait in the system for a long time.

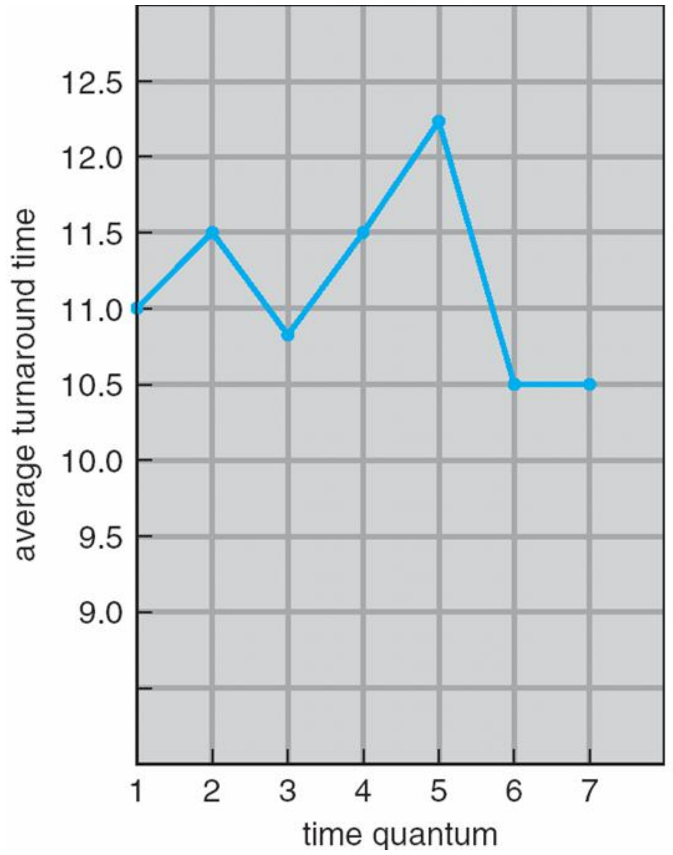
Additional Points contd.

- ☐In Round Robin Scheduling 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.**
- ☐The performance of Round Robin Scheduling is largely impacted by the value of the Time quantum *q*. **Hence it is necessary to choose *q* optimally**.
  - If *q* very large, it resembles a FCFS scheduling.
  - If *q* too small, too high overhead due to high context switching. Hence it is important that *q* must be large with respect to time to context switch.

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Thus, 80% of CPU bursts should be shorter than q

Determining Length of Next CPU Burst

- **All algorithms especially SJF faces another difficulty in knowing the length of the next CPU request.**
- We can only estimate the length somewhat like it should be similar to the previous one. Then for SJF we pick process with shortest predicted next CPU burst.
- This can be done by using the length of previous CPU bursts, using exponential averaging

$$T_{n+1} = \alpha t_n + (1 - \alpha) T_n$$

where $t_n =$ actual length of n^{th} CPU burst

$T_{n+1} =$ predicted value for the next CPU burst

$\alpha, 0 \le \alpha \le 1$

numericals further discussed over whiteboard

Few Important Points

α is the **Smoothing Factor**. It controls the relative weitage of recent and past history in our predictions.

- If $\alpha = 0$, $\tau_{n+1} = \tau_{n.}$ Thus Recent history does not count. No change in value of initial predicted BT.
- If $\alpha = 1$, $\tau_{n+1} = t_{n.}$ Thus only the actual last CPU burst counts. Predicted BT of new process will vary according to the actual BT of nth process.
- If we expand the formula, we get:

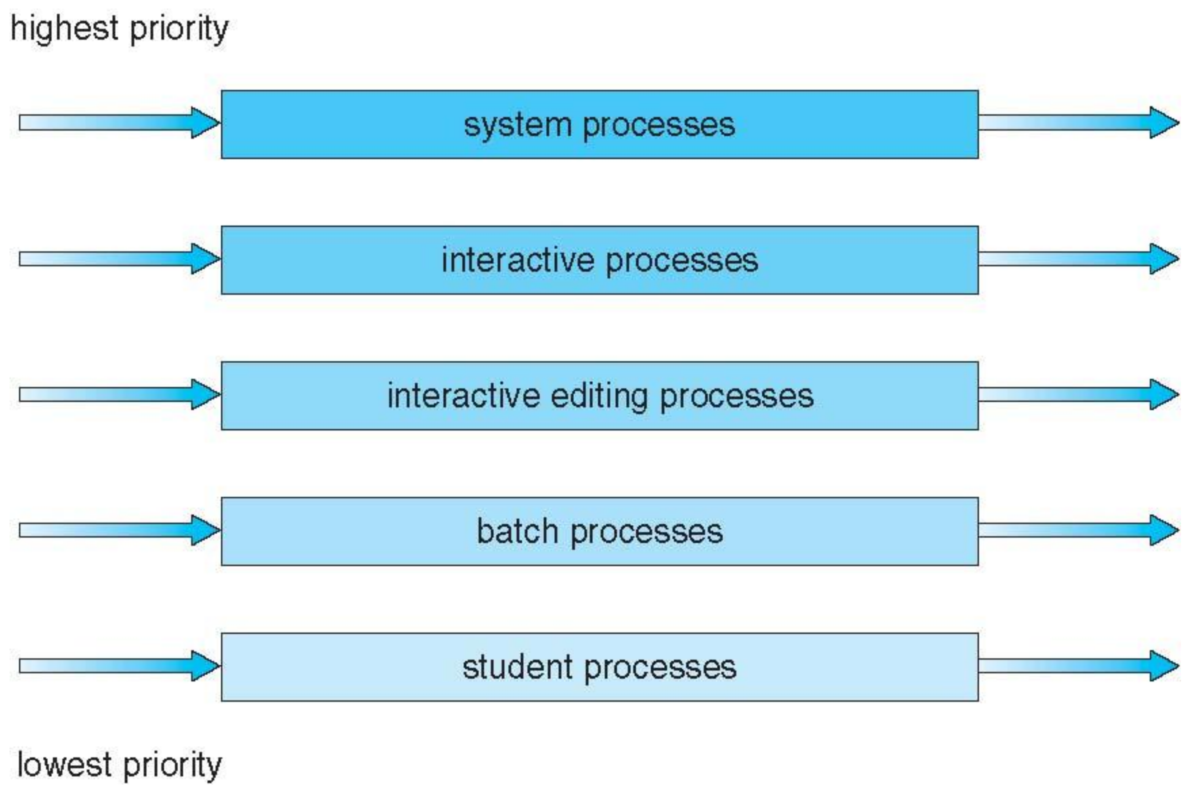
$$\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\alpha\ t_{n-1} + \ldots\ldots + (1 - \alpha)^j\ \alpha\ t_{n-j} + \ldots + (1 - \alpha)^{n+1}\ \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.
- If $\alpha = 1/2$, both recent and past history are equally important.

Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues, for example foreground or interactive processes and background or batch processes. They have different scheduling requirements. Foreground processes may have high prority than background processes.
- All process are permanently assigned to one queue based on some property - memory requirement, priority, type etc.
- Each queue has its own scheduling algorithm. For eg, foreground queue – RR and background queue – FCFS
- Scheduling must be done between the queues:
  - One possibility is Fixed priority scheduling; (i.e., serve all from foreground then from background). This may lead to starvation.
  - Another possibility is Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR and 20% to background in FCFS

Example of Multilevel Queue Scheduling



Multilevel Feedback Queue Scheduling

- In Multilevel Queue Scheduling, processes are permanently assigned to a queue when they enter the system. Processes do not move from one queue to another as they do not change their nature. This setup reduces the scheduling overhead but is inflexible.
- In Multilevel Feedback Queue, 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

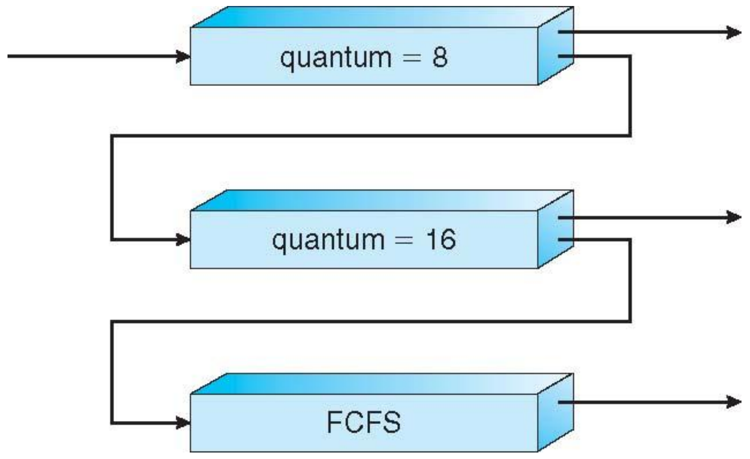
Example of Multilevel Feedback Queue

Consider we have Three queues in this order of priority:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

Scheduling Process:

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



Two more important terms

- **Prefetching:**
  - Method of overlapping the I/O of a job with that job's own computation.
  - It can e initiated by programmers
  - It helps to minimise WT for an individual process

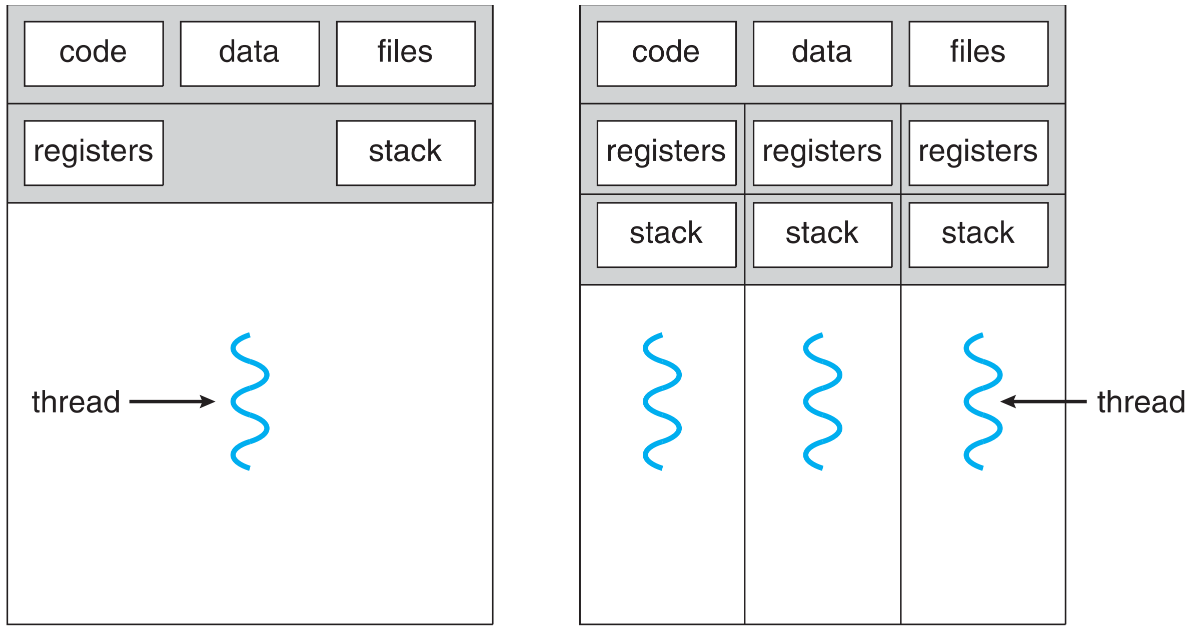
- **Spooling**:
  - CPU overlaps the input of one job with the computation and output display of other jobs
  - It is more effective and is done by the system itself.

Threads

- It is a **Light Weight (LWT) Process** (no separate memory, resources, system calls needed).
- It is the basic unit of CPU Utilization
- Comprises of Thread ID, PC, Register Sets and Stack.
- It shares its code section, data section and Resouces.
- Information specific to a thread: PC, SP, Execution Stack

Benefits of Multithreading

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces.
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures
- **Utilization** - CPU idle time reduces and CPU utilization increases.



single-threaded process   multithreaded process

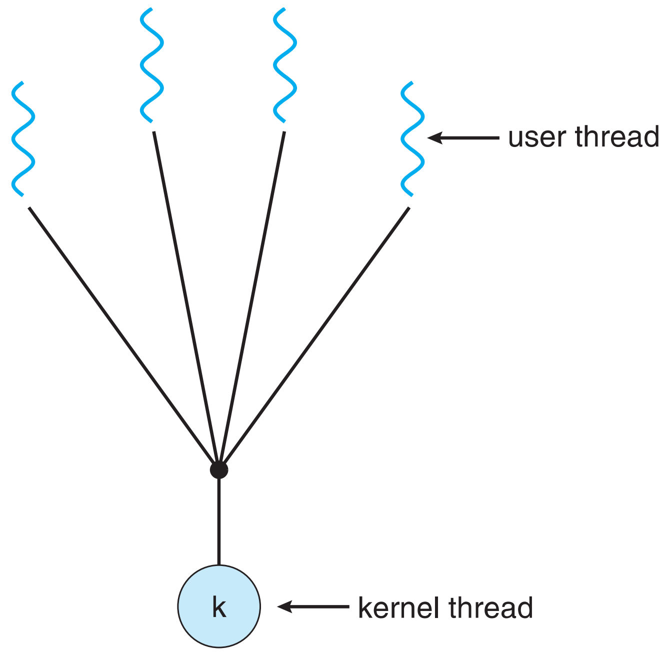
User Threads vs Kernel Threads

- Supported above the Kernel. Thus kernel is unaware of user threads (user threads are never transparent to the kernel).
- It is implemented by a thread library at user level.
- no kernel intervention, thus fast to create and east to manage.
- Eg: PThreads, Cthreads, UI threads

- Supported directly by the OS
- Kernel creates and manages these threads.
- kernel intervention exists thus slower and difficult to manage.
- Eg: Windows NT, Windows 2000, Solaris, BeOS

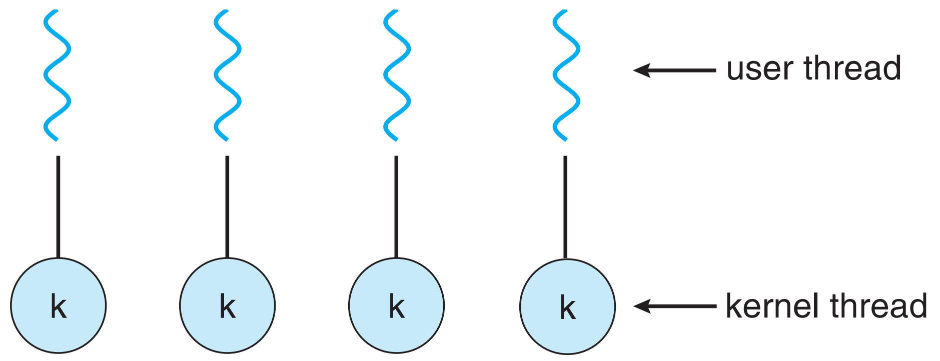
Many to One Thread Model

- **Many user-level threads mapped to single kernel thread**.
- Thread management is done at user space, hence efficient
- One thread blocking (by a blocking system call) causes all others to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples: Solaris Green Threads, GNU Portable Threads



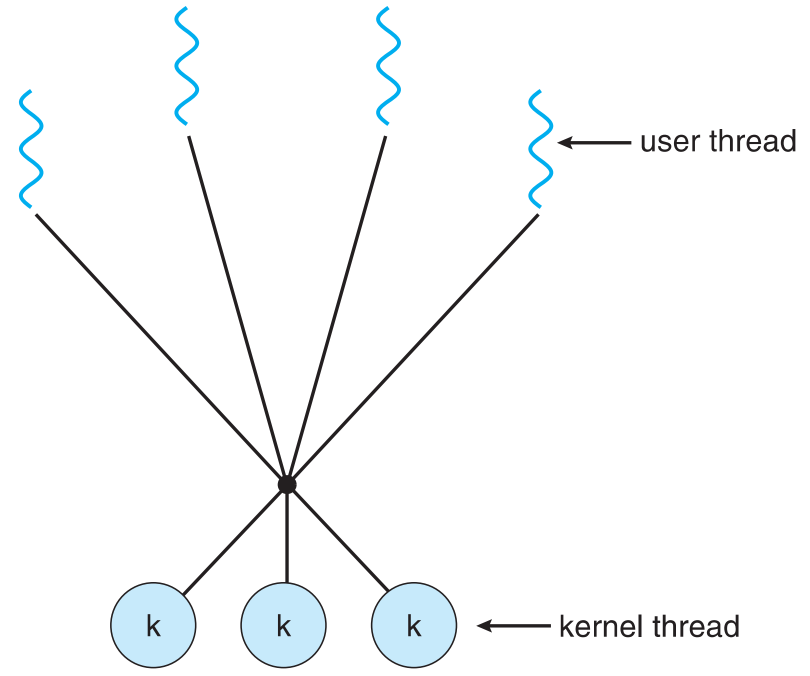
One to One Thread Model

- **Each user-level thread maps to kernel thread**
- Creating a user-level thread creates a kernel thread.
- Another thread can run even if any thread makes a blocking system call.
- More concurrency than many-to-one as multiple threads can run in parallel
- Number of threads per process sometimes restricted due to overhead
- Examples: Windows, Linux, Solaris 9 and later



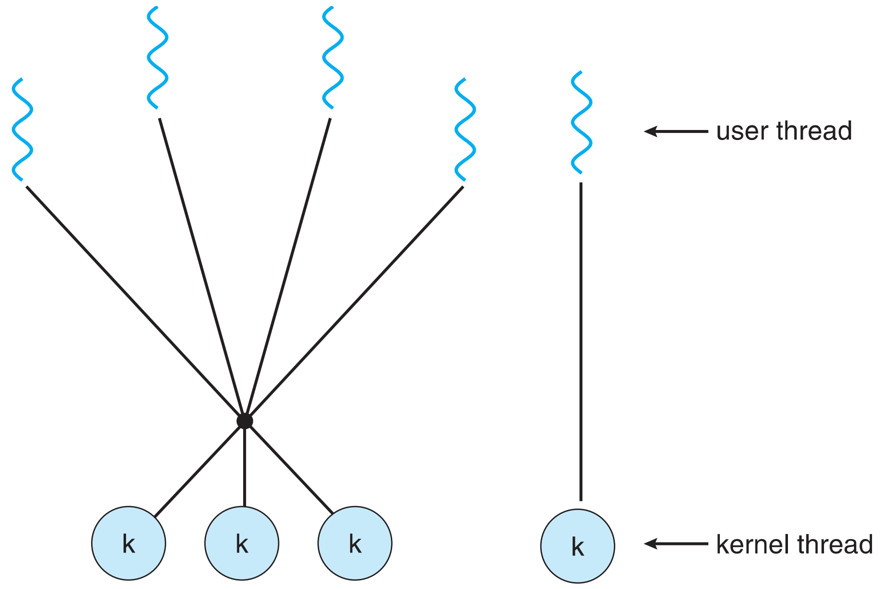
Many to Many Thread Model

- **Allows many user level threads to be mapped to many kernel threads**
- Allows the operating system to create a sufficient number of kernel threads.
- All these run in parallel in multiprocessor systems.
- Example: Solaris prior to version 9



Two Level Thread Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier



Thread Libraries

Thread library provides programmer with API for creating and managing threads

Two primary ways of implementing
- Library entirely in user space
- Kernel-level library supported by the OS

Process vs Thread

PROCESS	THREAD
Can’t share memory / files	Share memory / files
Need more time to create, execute, terminate	Need less time to create, execute, terminate
Need more context switching time	Need less context switching time
Require system calls to communicate	No system calls as they reside in same memory
Not suitable for parallel activities	Suitable for parallel activities
Need more resouces as they can’t share resources	Need less resouces as they can share resources