Operating System
CS-2006
Lecture 7

Mahzaib Younas

Lecturer Department of Computer Science

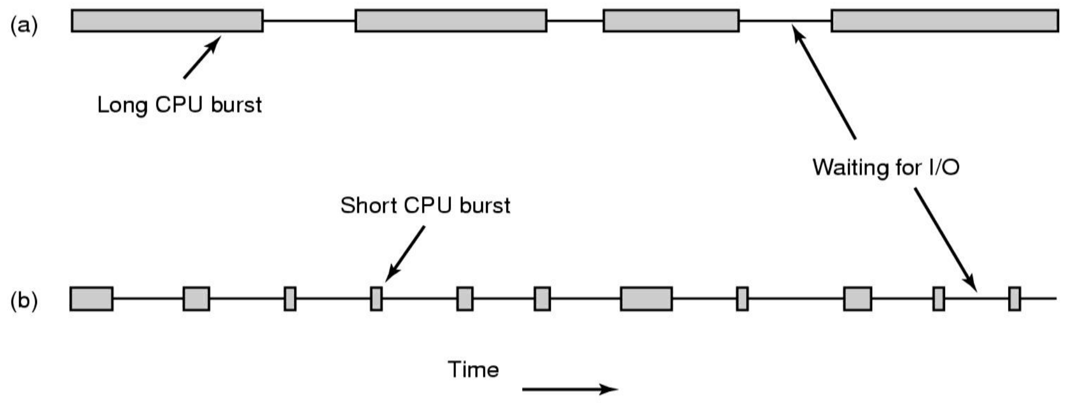
FAST NUCES CFD

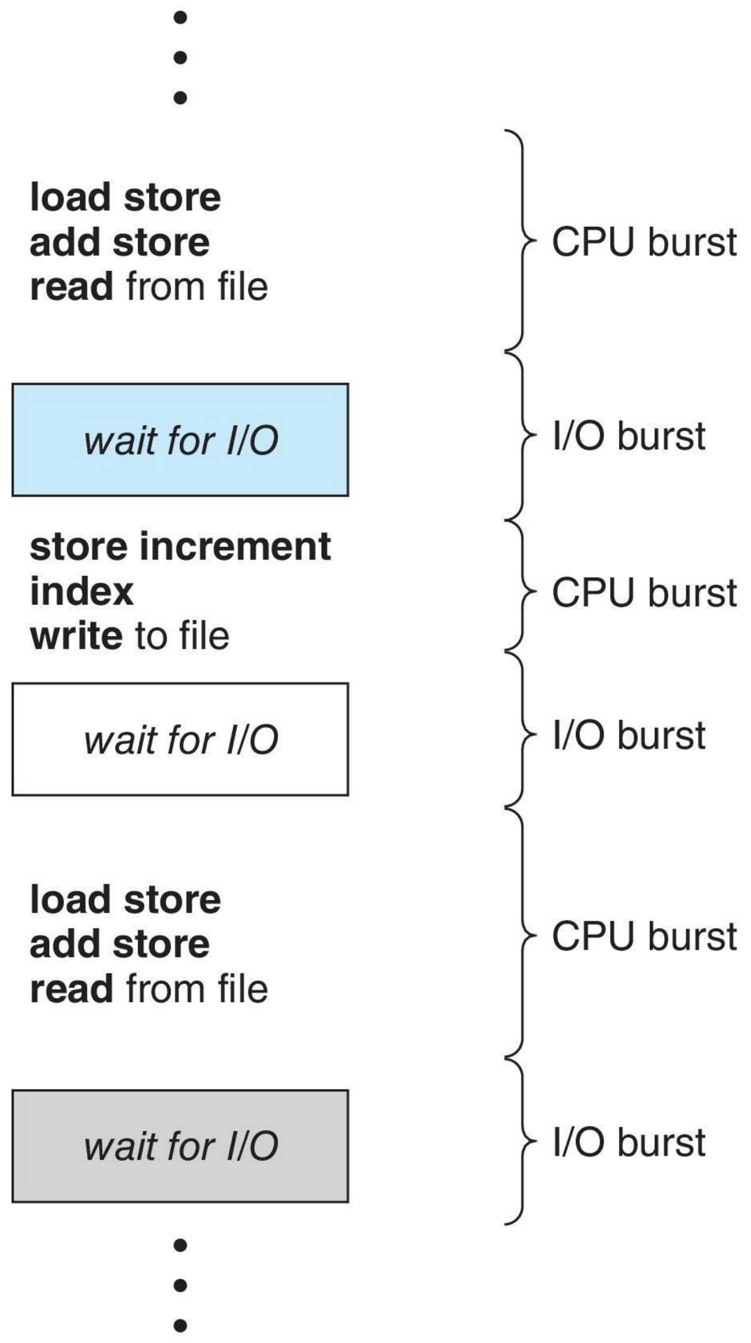
Basic Concepts

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling (Briefly)
- Operating Systems Examples (Homework)

Basic Concepts

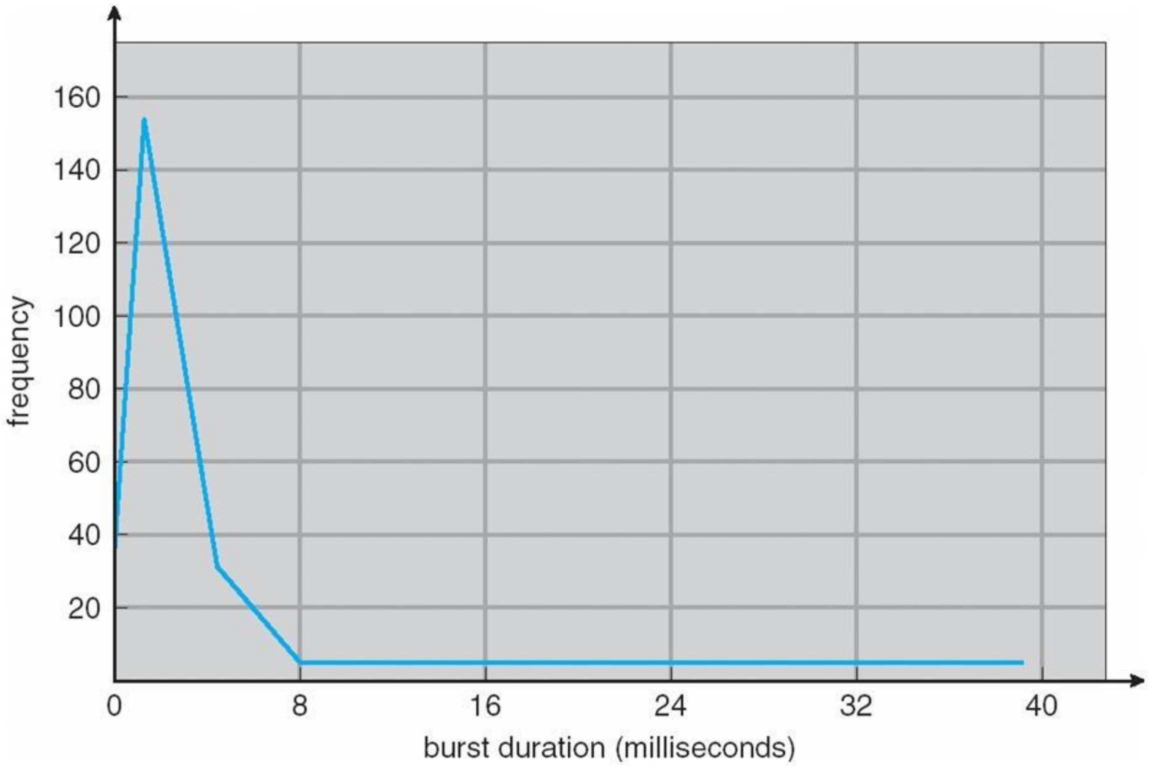
- **Maximum** CPU **utilization obtained with multiprogramming**
- **CPU always running a process**
- No idle CPU
- **CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait**
  - Run instructions
  - Wait for I/O
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern


(a)
Long CPU burst
Waiting for I/O
Short CPU burst
(b)
Time


load store
add store
read from file
CPU burst
wait for I/O
I/O burst
store increment
index
write to file
CPU burst
wait for I/O
I/O burst
load store
add store
read from file
CPU burst
wait for I/O
I/O burst

Histogram of CPU Brust Time

- **CPU burst distribution** is of main concern
- **CPU bursts vary from process to process,** but an extensive study shows frequency patterns similar to the diagram:
- Scheduling is aided by knowing the length of these bursts



CPU Schedular Queues

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
- Queue may be ordered in various ways:
  - FIFO
  - LIFO
  - Random, Priority

Types of Scheduling

- Types of scheduling
- Preemptive scheduling
  - Running process may be interrupted and moved to the Ready queue
  - Preemptive scheduling **allows a running process to be interrupted by a high priority process,**
- Non-preemptive scheduling
  - once a process is running, it continues to execute until it terminates or blocks for I/O
  - any new process has to **wait until the running process finishes its CPU cycle**

Scheduling Decision Modes

- The scheduler **runs when it needs to select a process to run on the processor**.
- The scheduling decision mode specifies the times at which the selection of a process to run is made.

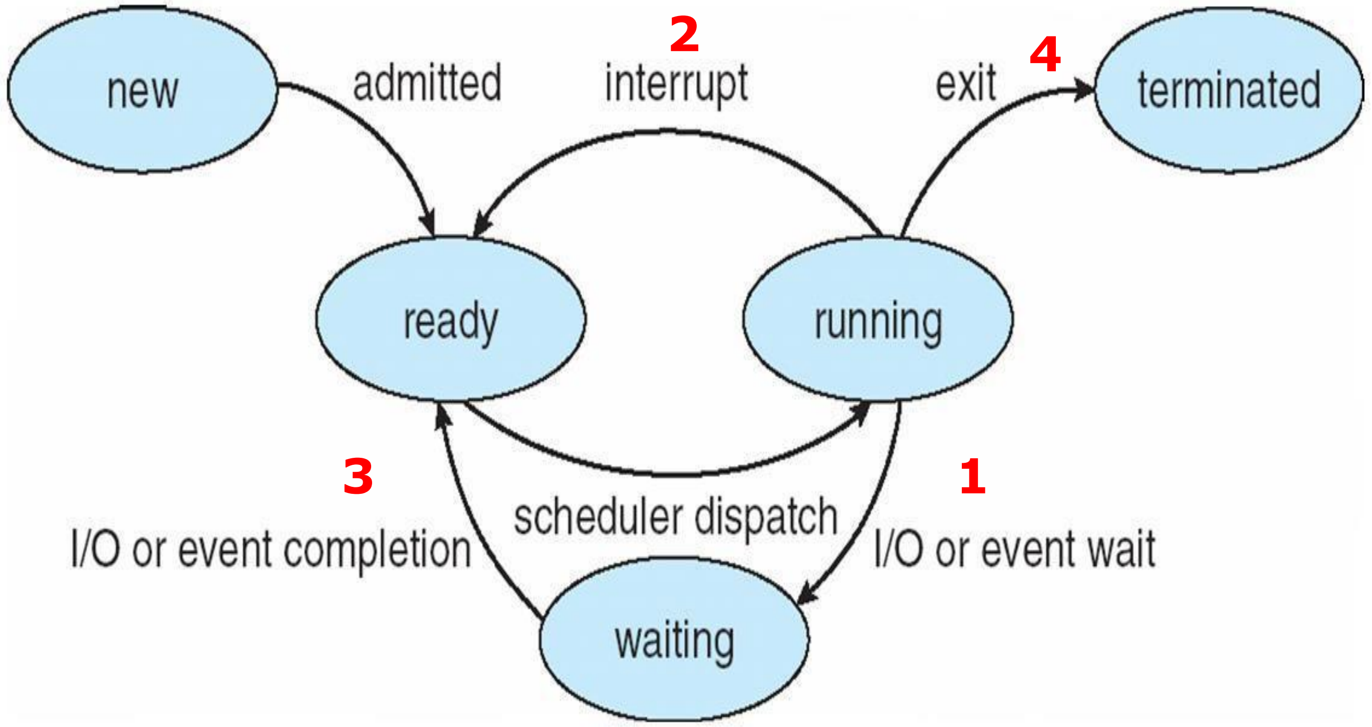
This decision can take place at the following times:

1. Switches from running to waiting state;
ex: as result of I/O request or wait()

2. Switches from running to ready state;
ex: when an interrupt occurs

3. Switches from waiting to ready
at completion of I/O. or because a new process was created and was placed in the queue.

4. Terminates



CPU Schedular

- Scheduling taking place only under **circumstances 1 and 4 is non preemptive**
- Process keeps the **CPU until it either terminates or switches** to waiting state
- No choice in terms of scheduling;
  - new process must be selected for CPU

CPU Schedular

- In **circumstances 2 and 3(running to ready, waiting to ready)**, the OS scheduler has a choice:
  - it can either allow the current process to **continue running**, or it could step in and put the current process to **sleep and select a different process to run**. The latter operation is called "preempting" the current process and scheduling a new one to run.
- Preemption can result in race conditions (see Chap-6)
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in **<u>race conditions</u>** when data are **<u>shared among several processes</u>**.

- Consider the case of two processes that share data.
  - one **<u>process is updating the data</u>**, it is **<u>preempted</u>**
  - second process can run. The second process then tries to read the data, which are in an **<u>inconsistent state</u>**.

What is Schedular

- Process Scheduling is an OS task that **schedules processes of different states** like **ready, waiting, and running**
- Scheduling Queues: Three types of operating system queues are:
  - **Job queue**
    - It helps you to store all the processes in the system.
  - **Ready queue**
    - This type of queue helps you to set every process residing in the main memory, which is ready and waiting to execute.
  - **Device queues**
    - It is a process that is blocked because of the absence of an I/O device.

- **Types of Schedular**
  - Long Term Schedular
  - Short Term Schedular
  - Medium Term Schedular

Long Term Scheduler

- Long-Term Scheduler is also called **Job Scheduler** and is responsible for **controlling the Degree of Multiprogramming**
  - **Example**: the total number of processes that are present in the ready state.

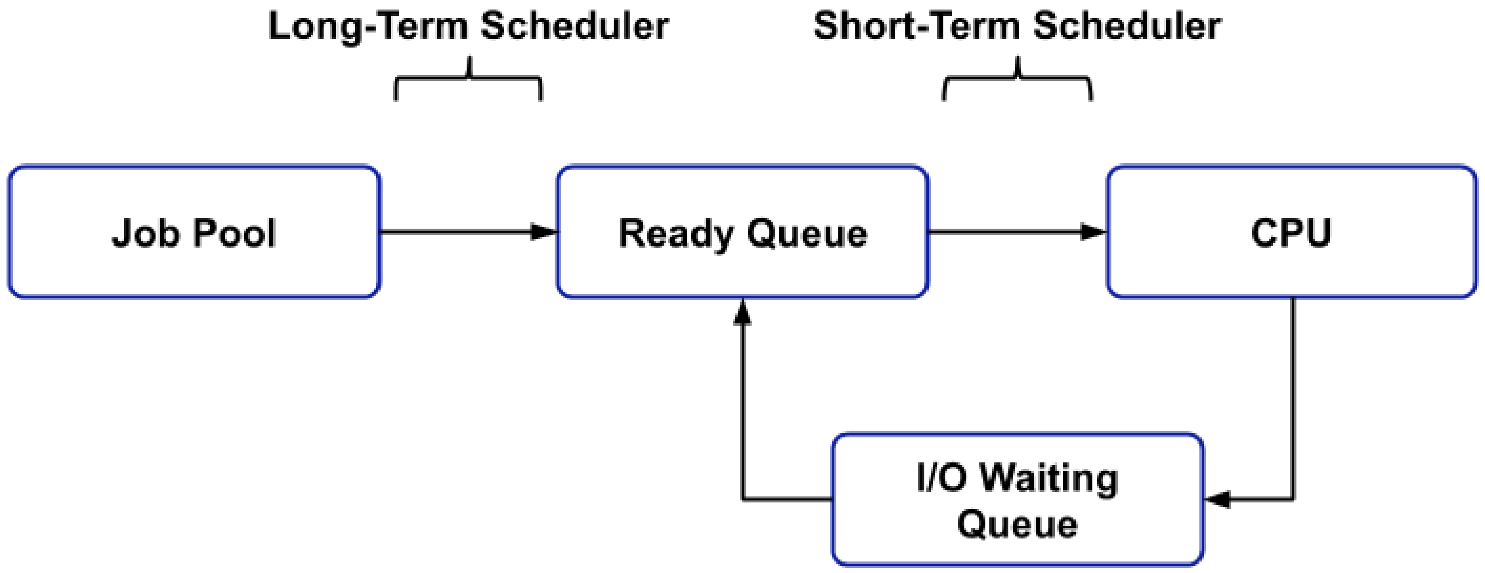
- Long-Term schedulers are those schedulers whose decision will have a **long-term effect on the performance**.
- The duty of the long-term scheduler is to bring the process from the **JOB pool to the Ready state for its execution**.

Medium Term Scheduler

- Medium-term schedulers are those schedulers whose decision will have a mid-term effect on the performance of the system.
- It is responsible for swapping of a process from the **Main Memory to Secondary Memory and vice-versa.**

Short Term Memory Schedular

- Short-Term Scheduler is also known as CPU scheduler and is responsible for selecting one process from the **ready state for scheduling it on the running state.**



The Dispatcher

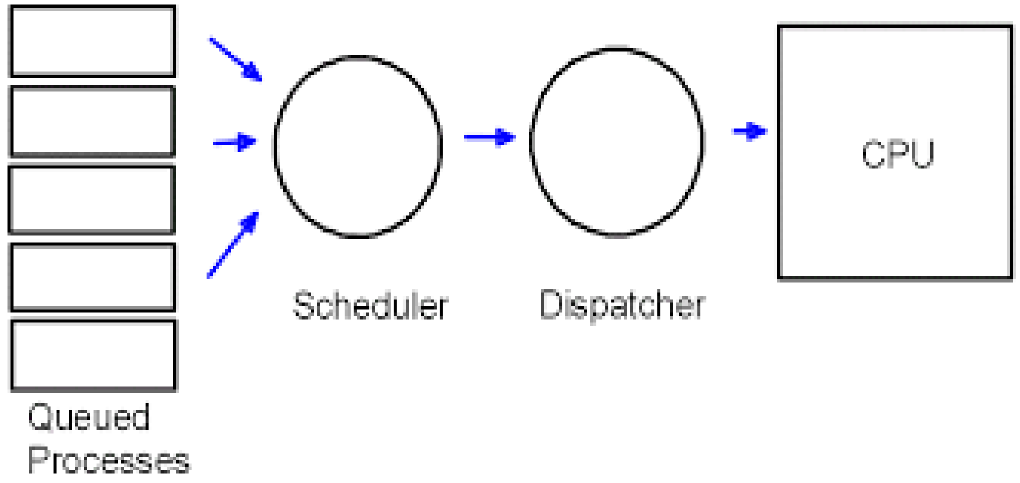
- The dispatcher is the kernel routine that performs the context switch.

  A dispatcher is an **essential component of an operating system (OS**) that manages and allocates system **resources to different processes**.

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching to user mode by changing the processor mode to user mode
  - switching context from the currently running process to the new one.

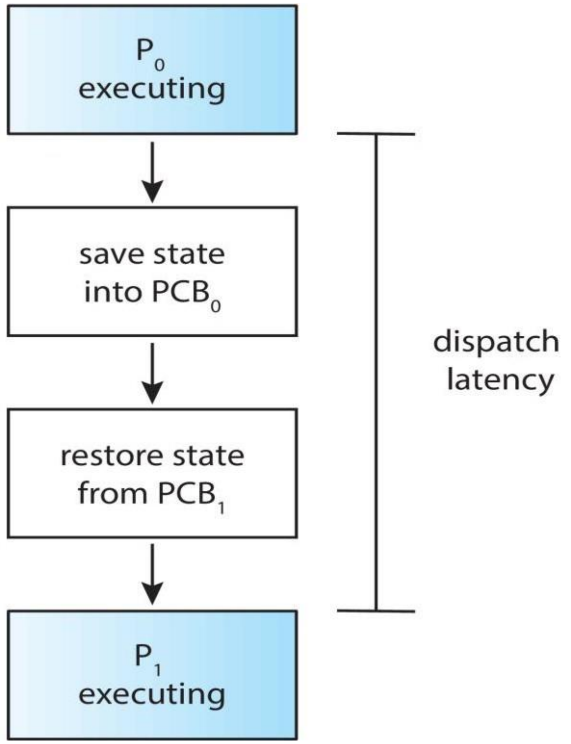
The Dispatcher (Cont…)

- 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**



The Dispatcher

- The dispatcher is invoked by the CPU scheduler and does the following:
  - It switches the context from the **currently running process to the new one.**
  - It then changes the **processor mode to user mode**, since it runs in kernel mode.
  - It loads the instruction register with the location in the new process at which it should start or continue its execution



voluntary and nonvoluntary context switches.

Scheduling Criteria

- CPU utilization
  - keep the CPU as busy as possible
- Throughput
  - of processes that complete their execution per time unit
- Turnaround time
  - amount of time to execute a particular process
- 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.

Scheduling Criteria

CPU utilization keep the CPU as busy as possible

Ranges from 40% to 90% i.e., from light to heavy loaded

Scheduling algorithm optimization criteria: Max CPU utilization

Throughput of processes that completing per time unit

Ranges from 10 processes/second to 1 process/hour

Scheduling algorithm optimization criteria: Max throughput

Turnaround time time from the submission of a job until it completes;

Sum of times spent in job pool + ready queue + CPU execution + doing I/O

Scheduling algorithm optimization criteria: Min turnaround time

Waiting time amount of time a process has been waiting in the ready queue

Sum of times spent waiting in the ready queue

Scheduling algorithm optimization criteria: Min waiting time

Response time amount of time it takes from when a request was submitted until the first response is produced, not output

The time it takes to start responding to the user; in an interactive system

Scheduling algorithm optimization criteria: Min response time

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling Algorithms

- First come first serve
- Shortest Job First
- Shortest Remaining Time First
- Round Robin
- Multilevel Queue Scheduling
- Multilevel Feedback Queueing

First Come First Serve Algorithm (FCFS)

- FCFS is very simple
- Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.

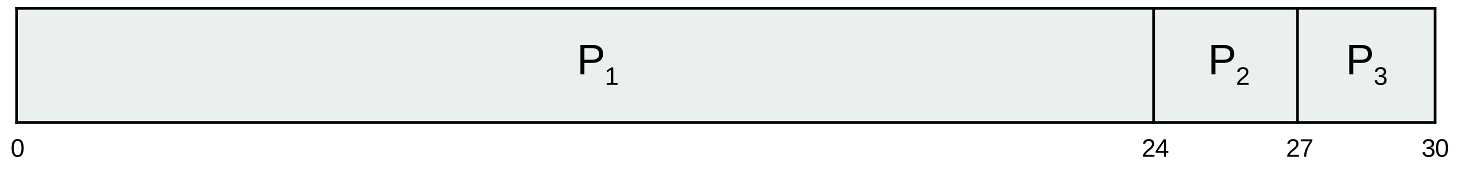
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Solution:

- Suppose that the processes arrive in the order: ***P_1 , P_2 , P_3***

The Gantt Chart for the schedule is:[includes start and finish time of each process]



Throughput: 3 jobs/30 seconds = 0.1 jobs/second

Solution: (Cont..)

- Turnaround Time: **P1 : 24, P2 : 27, P3 : 30**
  - Average TT: (24 + 27 + 30)/3 = 27
- Waiting time for ***P_1 = 0; P_2 = 24; P_3 = 27***
  - Average waiting time: (0 + 24 + 27)/3 = 17
- The simplest scheduling algorithm but usually very bad average waiting time

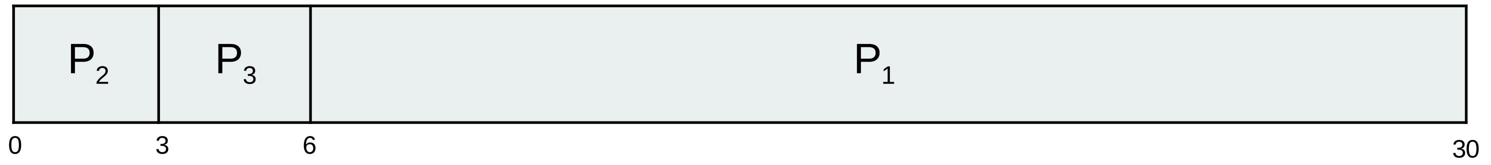
Example 2: FCFS

- **Now suppose that the processes arrive in the order: P2 , P3 , P1, (instead of P1,P2,P3) where Burst times,**
  - **P2 = 3**
  - **P3=3**
  - **P1 = 24**

- The Gantt chart for the schedule is: ??
- Throughput: ?
- Turnaround time: ?
- Average TT: ?
- Waiting time for P1 = ?; P2 = ?; P3 = ?
- Average waiting time: ?

Solution:

- Suppose that the processes arrive in the order:
- P2 , P3 , P1 , where Burst times, for P2 = 3, P3=3 and P1 = 24.
- The Gantt chart for the schedule is:



- Throughput: **3 jobs / 30 sec = 0.1 jobs/sec**
- Turnaround time: **P1 = 30, P2 = 3, P3 = 6**
  - Average TT: (30 + 3 + 6)/3 = 13 much less than 27
- Waiting time for **P1 = 6; P2 = 0; P3 = 3**
  - Average waiting time: (6 + 0 + 3)/3 = 3; 3
  - instead of 17, substantial reduction in wait time

Much better than previous case

Lesson From FCFS

- Lesson: scheduling algorithm can reduce TT
- Minimizing waiting time can improve TT
- Can a scheduling algorithm improve throughput?
- Yes, if jobs require both computation and I/O

FCFS Convoy Effect

Convoy effect - short processes maybe stuck behind long processes

- CPU-bound processes **hold CPU** while I/O-bounds **wait in ready queue** for the CPU
- **I/O devices are idle until CPU released**… then **CPU is idle.. Then … this repeats**

- Consider one long CPU-bound and many I/O-bound processes
  - I/O-bound processes spend most of the time waiting for CPU-bound to release CPU
  - Result in lower CPU and device utilization
  - FCFS is non-preemptive: process holds CPU until termination or I/O request

Shortest Job First Scheduling

The preceding examples/scheduling showed that when **short processes are scheduled before long processes**, the average waiting times are smaller than if they were to follow after it.

- Associate with **each process the length of its next CPU burst**
  - Use these lengths to schedule the **process with the shortest time**
  - Assign the CPU to the process **that has the smallest next CPU burst**
  - What if two process have the same next CPU burst length?
  - FCFS breaks the tie

Shortest Job First

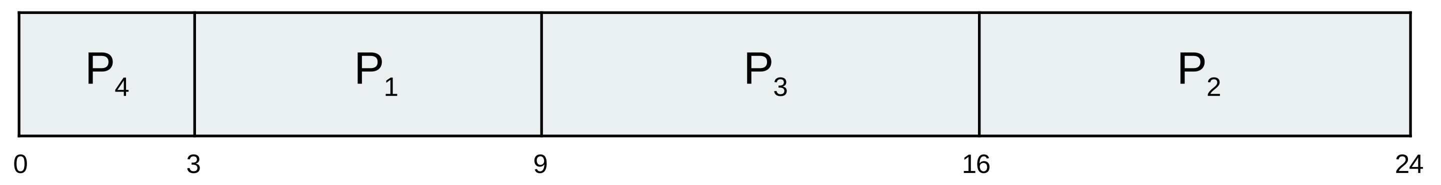
- SJF is optimal – gives minimum average waiting time for a given set of processes
- The difficulty is knowing **the length of the next CPU request**
- Could ask the user to estimate their processes' time limits; for job scheduling

SJF Example:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Example:

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
  - With FCFS it would be 10.25 time units with this order P1 P2 P3 P4
- Moving a short process before a long one decreases its waiting time more than it increases the long process's waiting time. Thus, decrease of average waiting time

Determining Length of Next CPU Burst

- Burst time can be estimated using lengths of past bursts: **next = average of all past bursts**
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential weighted moving average; **next = average of (past actual + past estimate)**

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

$$S_{n+1} = (1-\alpha)\, S_n + \alpha\, T_n$$

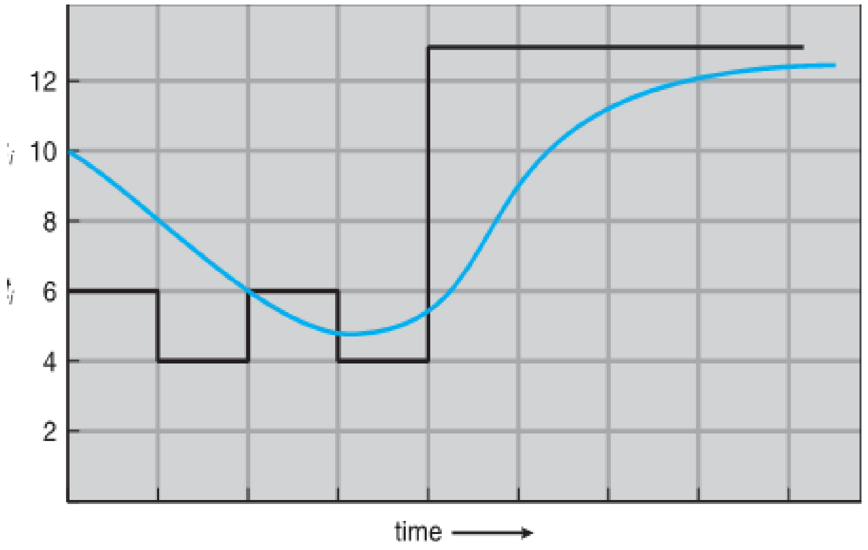
- Relative weight of recent history and past history. α = ½ usually but anything in the range [0, 1] is acceptable
- Commonly, α set to ½

Prediction of Next CPU Brust

- How to find next estimated burst, if initial estimate is

$S_0 = 10$

$T_0 = 06$



CPU Brust		6						
Guess	10							

Prediction of Next CPU Brust (S_1)

- Formula

$$S_{N+1} = (1 - \alpha)\, S_N + \alpha\, T_N$$

- Calculations:

$$S_{0+1} = \left(1 - \frac{1}{2}\right) 10 + \frac{1}{2}(6)$$
$$S_1 = (0.5)(10) + 3$$
$$S_1 = 5 + 3$$
$$S_1 = 8$$

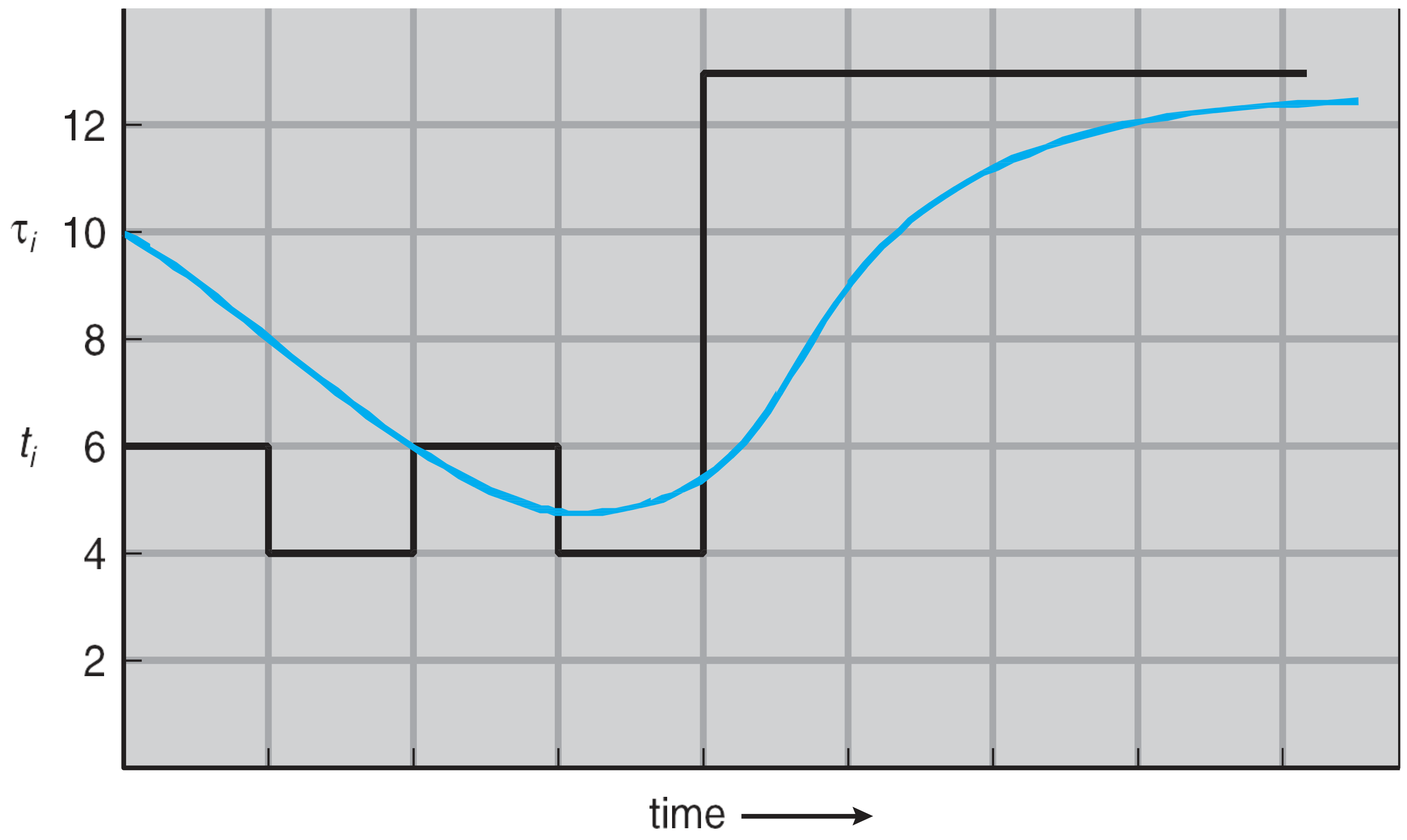
CPU Brust		**6**						
Guess	10	8						

Prediction of Next CPU Brust (S_1)

- $S_{(2)} = 0.5 \times 8 + 0.5 \times 4 = 4 + 2 = 6$
- $S_{(3)} = ?$
- $S_{(4)} = ?$

CPU Brust		**6**	**4**					
Guess	10	8	6					

Prediction of Next CPU Brust



CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Shortest Job First Example:

- Preemptive version of SJF is called shortest-remaining-time-first
- The next burst of new process may be shorter than that left of current process
- Currently running process will be preempted
- Now we add the concepts of varying arrival times and preemption to the analysis

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

- Draw Gantt chart for above mentioned processes:

Limitations of SJF

- SJF **doesn’t always minimize average Turnaround Time** (time from entering the system till completion)
- Only **minimizes waiting time**
- Can lead to unfairness or starvation newly arriving tasks have **shorter bursts**
- In practice, **can’t actually predict the future**
- But can **estimate CPU burst length based on past**

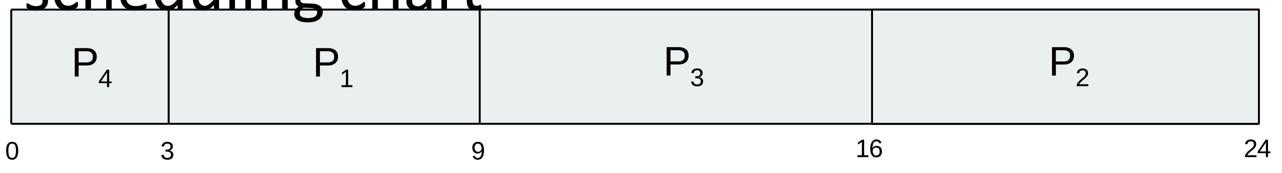
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these **lengths to schedule the process with the shortest time**
- SJF is optimal – gives minimum average waiting time for a given set of processes
- **Preemptive version called shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



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

Determining the length of NEXT CPU Burst

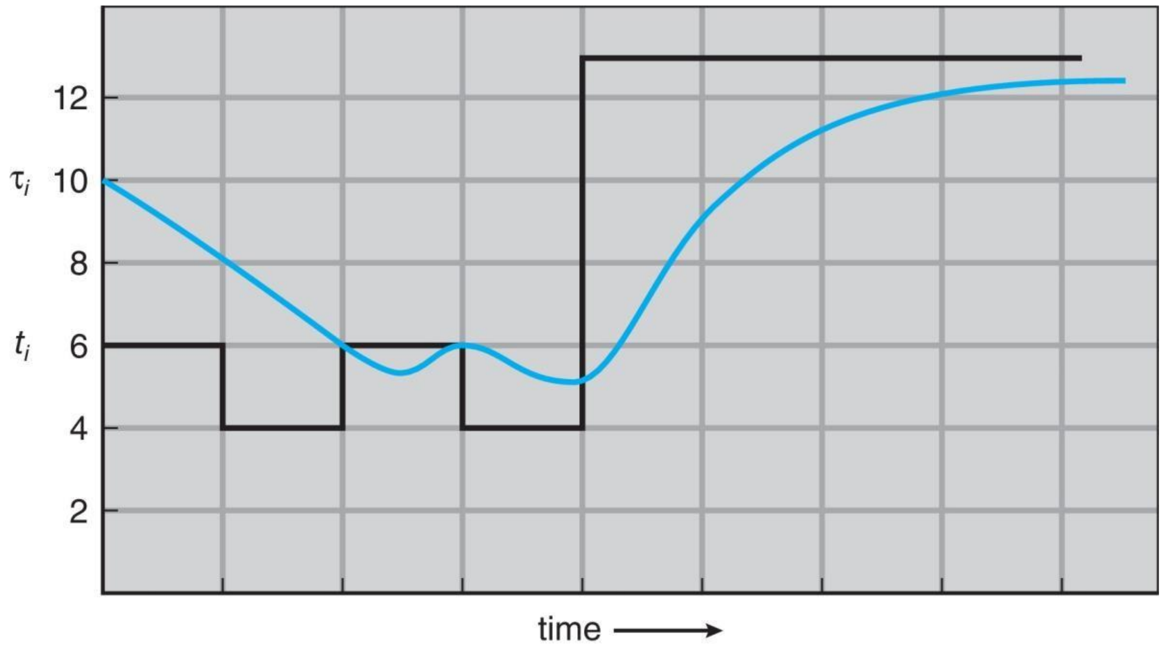
- Can only estimate the length – should **be similar to the previous one**
  - Then pick process **with shortest predicted next CPU burst**
- Can be done by using the length of previous CPU bursts, using exponential averaging

  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \leq \alpha \leq 1$
  4. Define:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

- Commonly, α set to ½

Prediction the length of NEXT CPU Brust



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successor predecessor term has less weight than its predecessor

Shortest Remaining Time First Scheduling

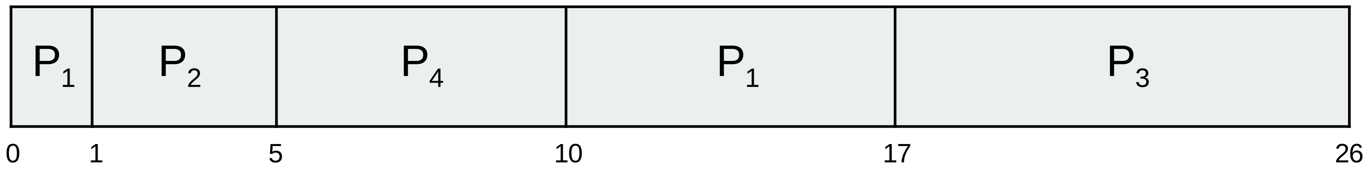
- **Preemptive version of SJN**
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJN algorithm.
- **Is SRT more “optimal” than SJN in terms of the minimum average** waiting time for a given set of processes?

Example : Shortest Remaining Time First Scheduling

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



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

Round Robbin Scheduling

- Each process gets a small unit of **CPU time (time quantum q),** usually **10-100 milliseconds**.
  - After this time has elapsed, the **process is preempted and added to the end of the ready 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.

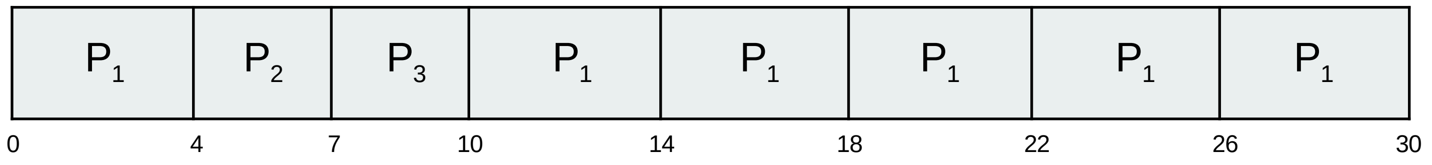
Round Robbin Scheduling

- Timer interrupts every quantum to schedule next process
- Performance
  - **q large F→IFO (FCFS)**
  - **q small → RR**
- Note that q must be large with respect to context switch, otherwise overhead is too high

Example: Round Robbin Scheduling T = 4

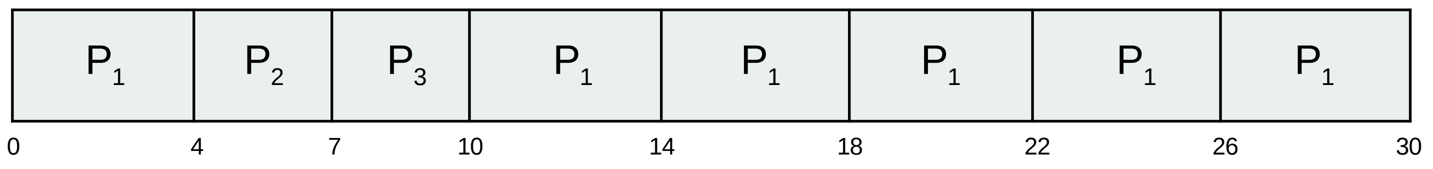
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



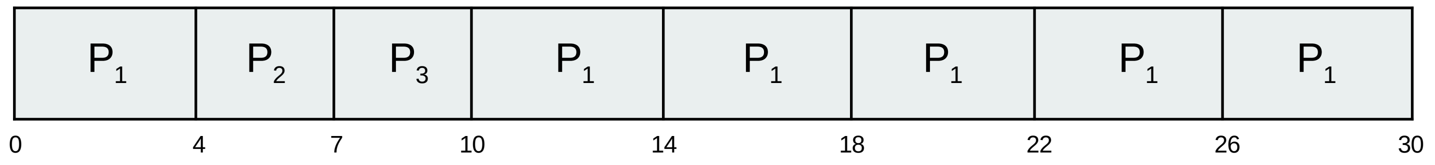
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds
  - T represent the Time Quantum

Example: Round Robbin Scheduling T = 4



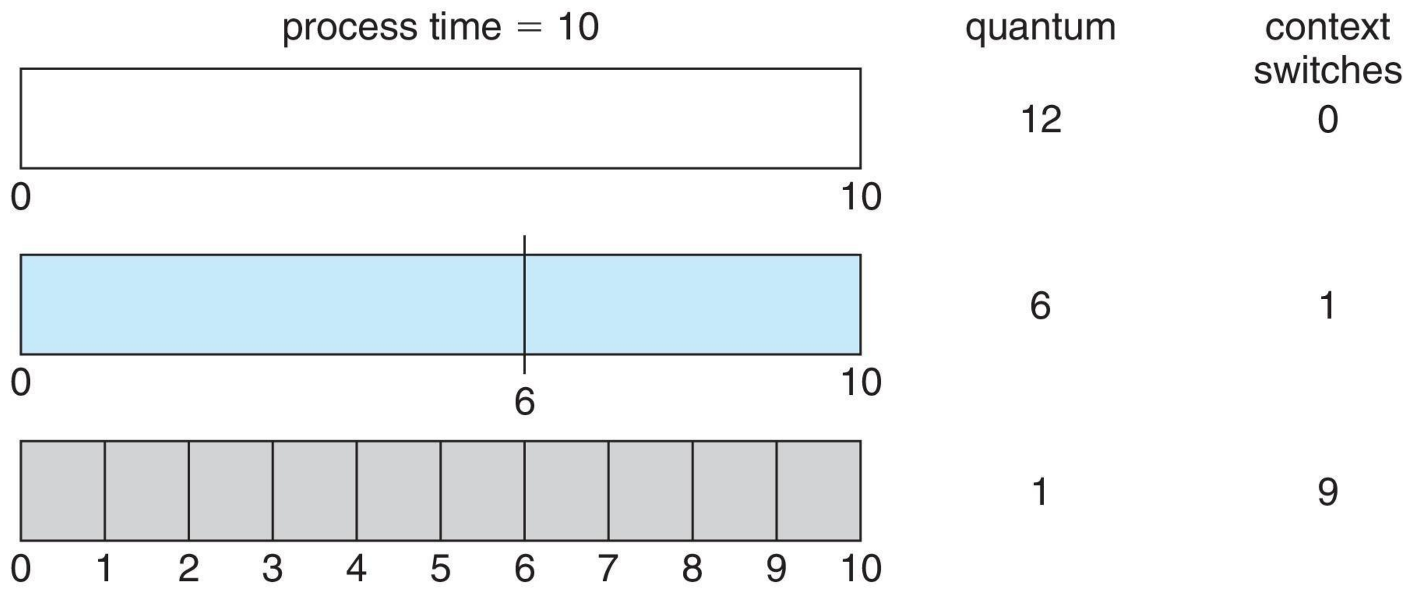
Processes	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	24	30	30	6
P2	0	3	7	7	4
P3	0	3	10	10	7
Turnaround Time = Completion Time – Arrival Time					
Waiting Time = Turnaround Time – Burst Time					

Example: Round Robbin Scheduling T = 4

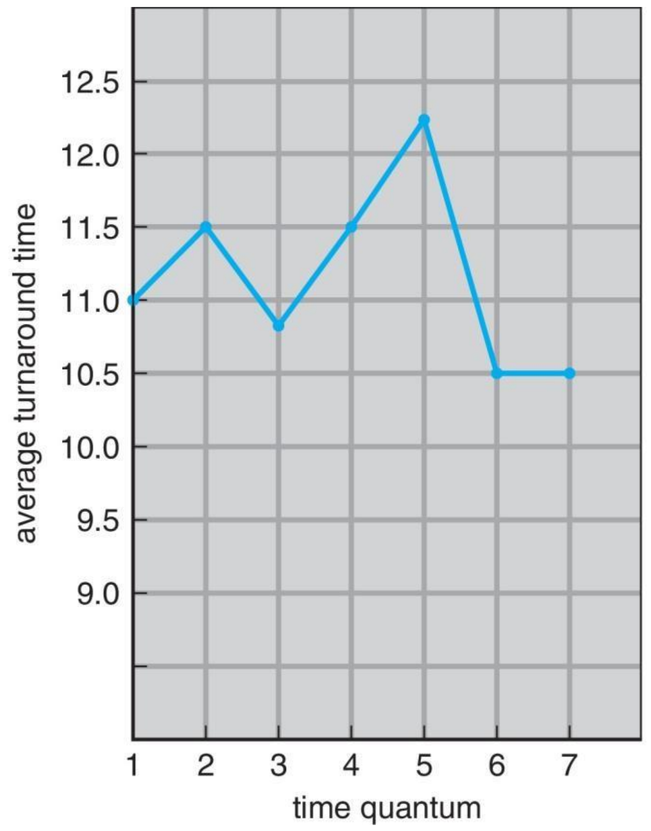


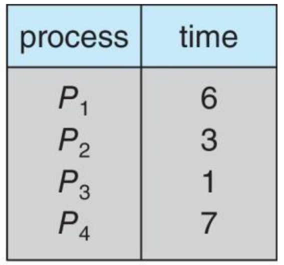
Processes	Arrival Time	Completion Time	Turnaround Time	Waiting Time
P1	0			
P2	0			
P3	0			
p4	0			

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

80% of CPU bursts
should be shorter than q

Priority Scheduling

- A **priority number (integer) is associated with each process**
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - **Preemptive**
  - **Non preemptive**
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

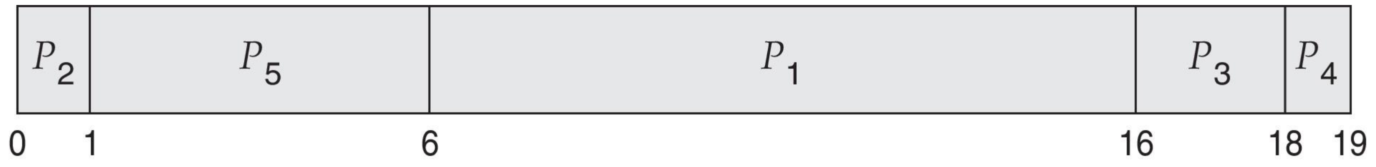
Priority Scheduling

- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

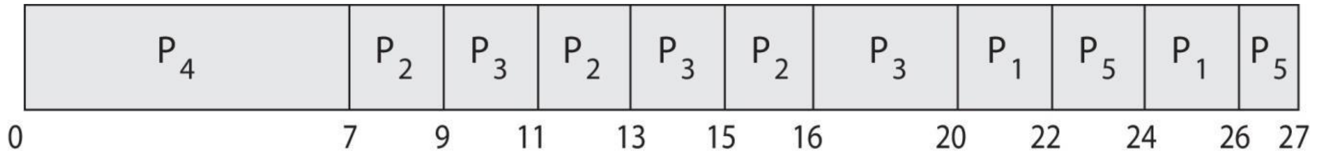
- Priority scheduling Gantt Chart



- Average waiting time = 8.2

Priority Scheduling with Round Robbin

- Run the process with the highest priority. **Processes with the same priority run round-robin**
- Example:



Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

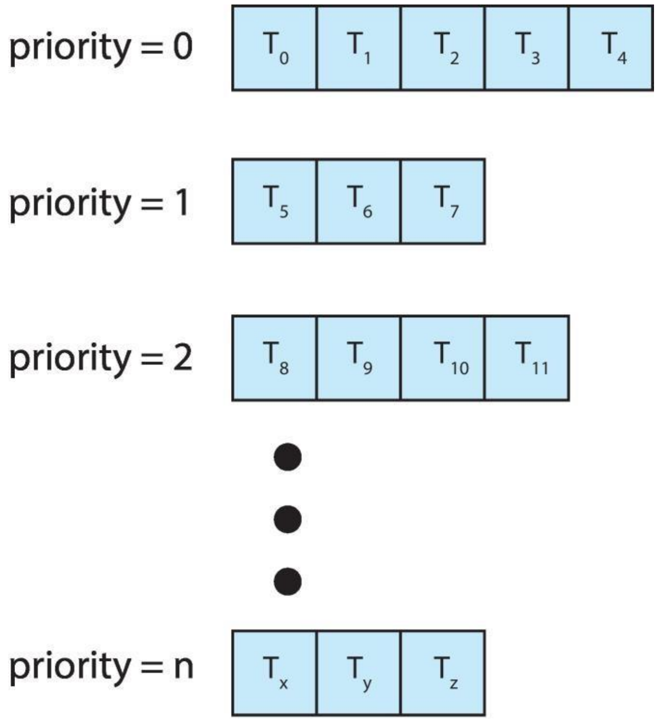
- Gantt Chart with time quantum = 2

Multilevel Queue

- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues

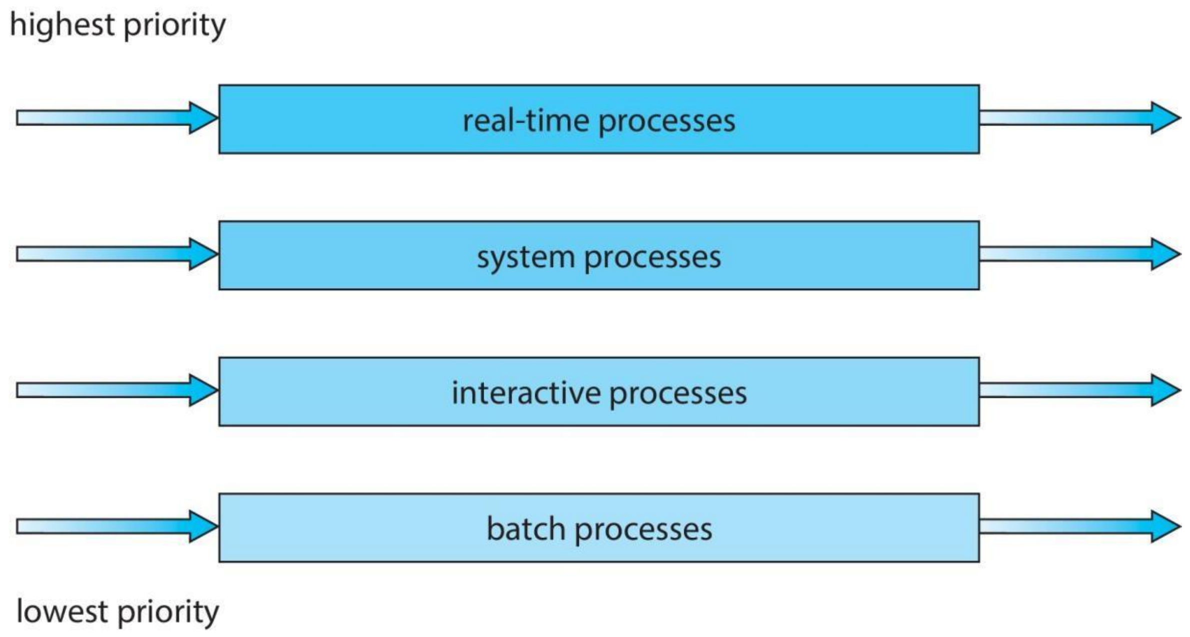
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue

- Prioritization based upon process type

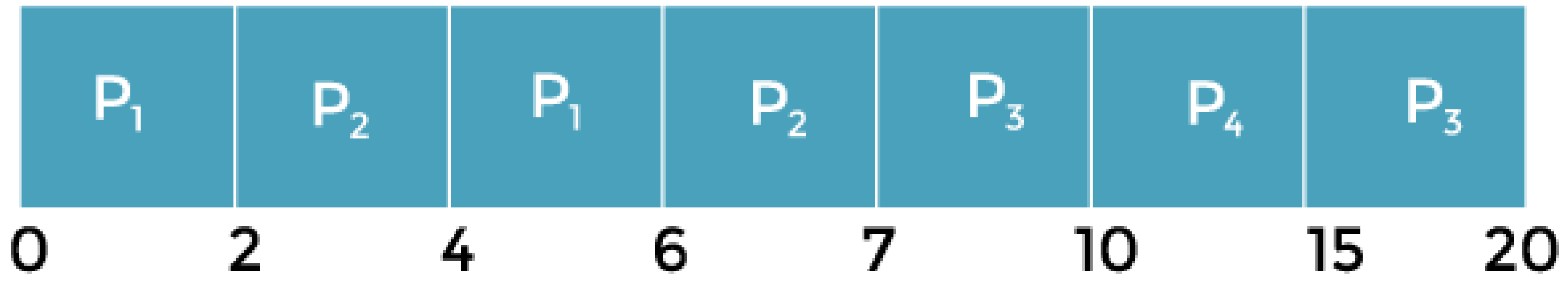


Example:

Process	Arrival Time	Brust Time	Queue
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

Queue 1 has a higher priority than queue 2. Round Robin is used in queue 1 (Time Quantum = 2), while FCFS is used in queue 2

Solution



1. Both queues have been processed at the start. Therefore, **queue 1 (P1, P2)** runs first (due to greater priority) in a **round-robin way and finishes after 7 units**.

2. The process in **queue 2 (Process P3)** starts running (since there is no process in queue 1), but while it is executing, **P4 enters queue 1 and interrupts P3, and then P3 takes the CPU and finishes its execution**.

Multilevel Feedback Queue

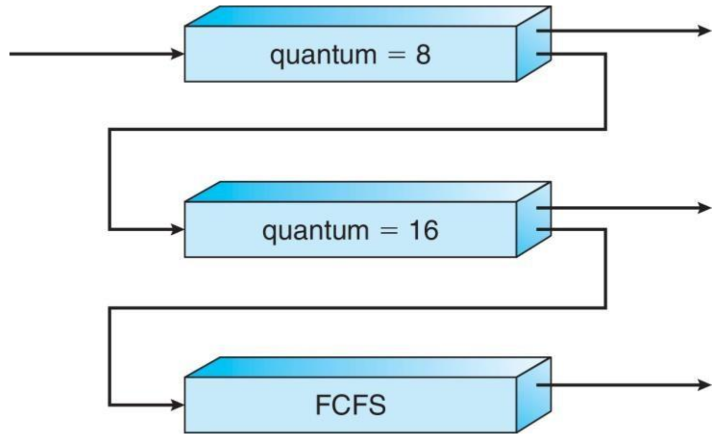
- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - **Method used to determine which queue a process will enter when that process needs service**
- Aging can be implemented using multilevel feedback queue

Multilevel Feedback Queue Parameters

- In general, a multilevel feedback queue scheduler is defined by the following parameters:
  - The number of queues
  - The scheduling algorithm for each queue
  - The method used to determine when to upgrade a process to a higher priority queue
  - The method used to determine when to demote a process to a lower priority queue
  - The method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Thread Scheduling

- Distinction between **user-level and kernel-level threads**
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as since scheduling competition is within the process-contention scope (PCS) ess
  - Typically done **via priority set by programmer**
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling

- Can be limited by OS – Linux and macOS only allow **PTHREAD_SCOPE_SYSTEM**

Pthread Scheduling API

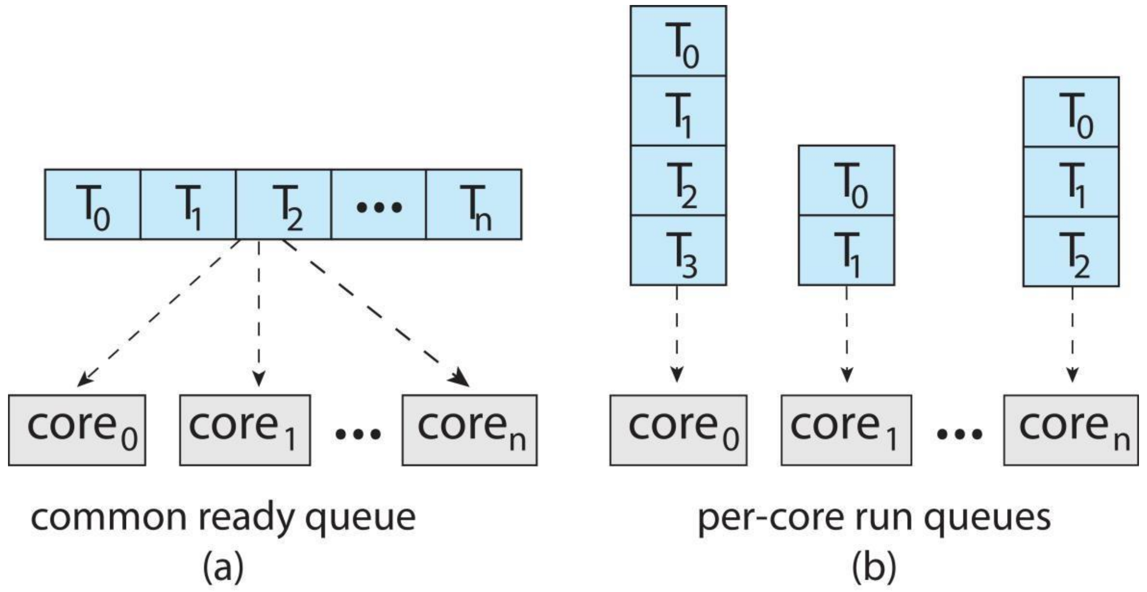
```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) !=
0)
      fprintf(stderr, "Unable to get scheduling
scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope
value.\n");
   }
```

Pthread Scheduling API

```
    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

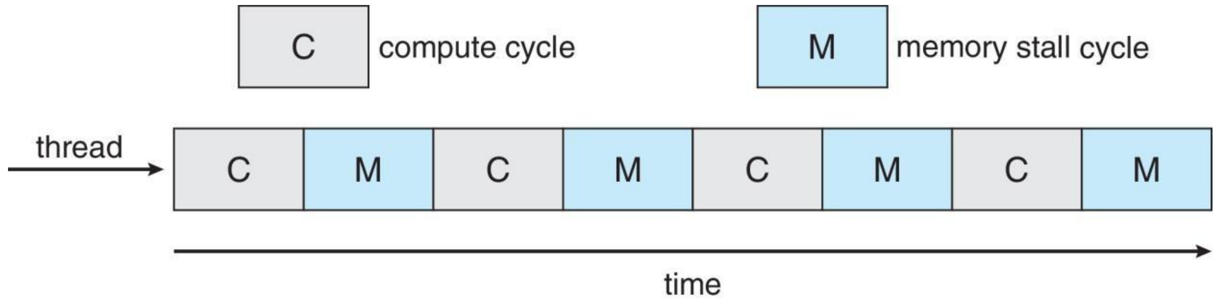
Multi Core Process Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)


T_0 T_1 T_2 ••• T_n
core_0 core_1 ••• core_n
common ready queue
(a)
T_0 T_1 T_2 T_3
T_0 T_1
T_0 T_1 T_2
core_0 core_1 ••• core_n
per-core run queues
(b)

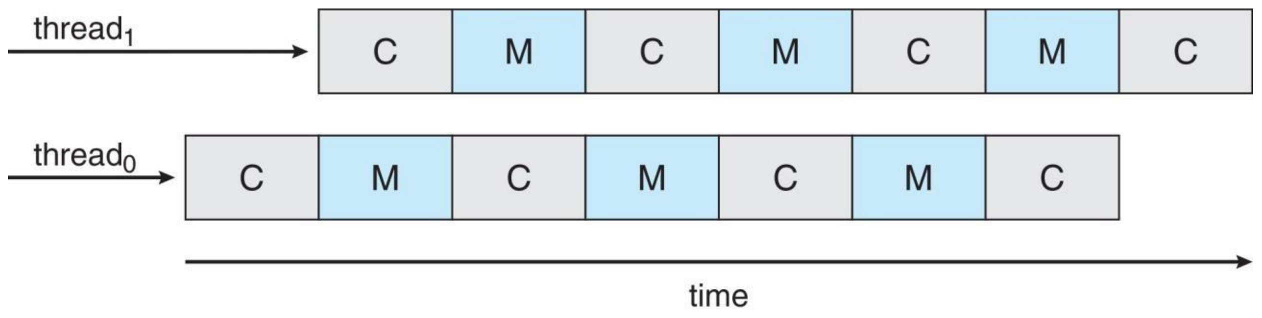
Multi Core Process Scheduling

- Recent trend to place multiple processor cores on same **physical chip**
- **Faster and consumes less power**
- **Multiple threads per core also growing**
  - Takes advantage of memory **stall to make progress on another thread while memory retrieve happens**
- Figure



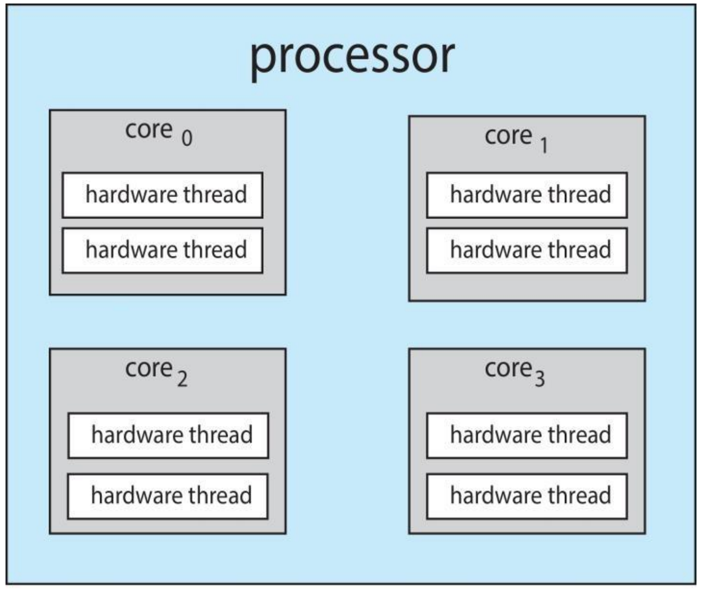
Multithreaded Multicore System

- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure

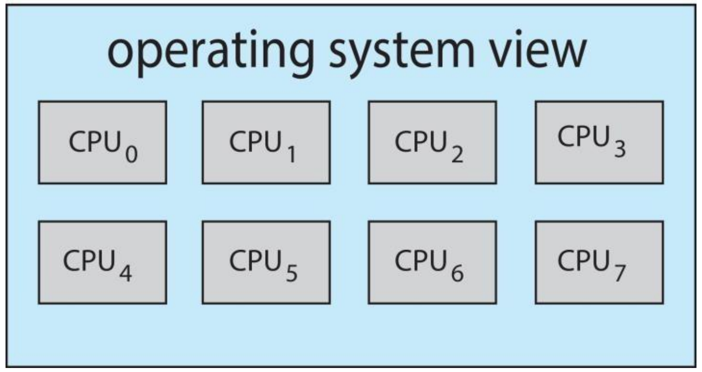


Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)



- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

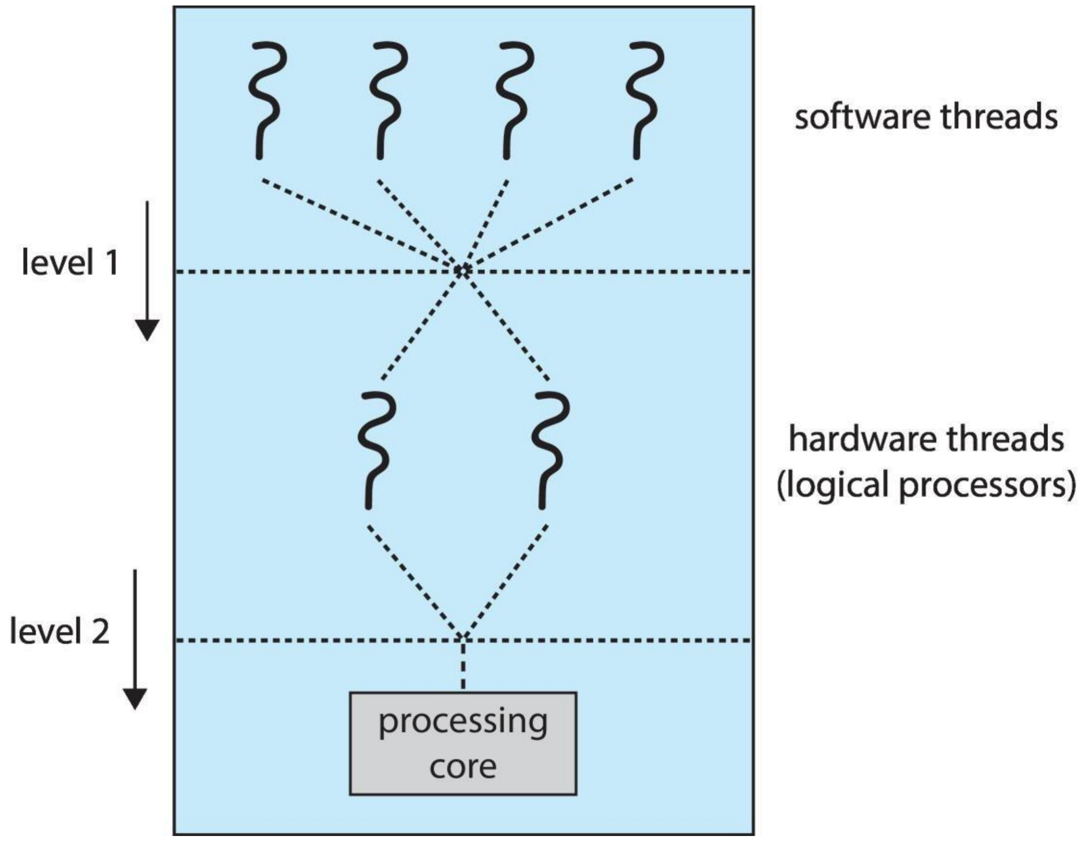


Multithreaded Multicore System

- Two levels of scheduling:

    1. The operating system deciding which software thread to run on a logical CPU

    2. How each core decides which hardware thread to run on the physical core.



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency

  - **Load balancing** attempts to keep workload evenly distributed
  - **Push migration –** periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - **Pull migration –** idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

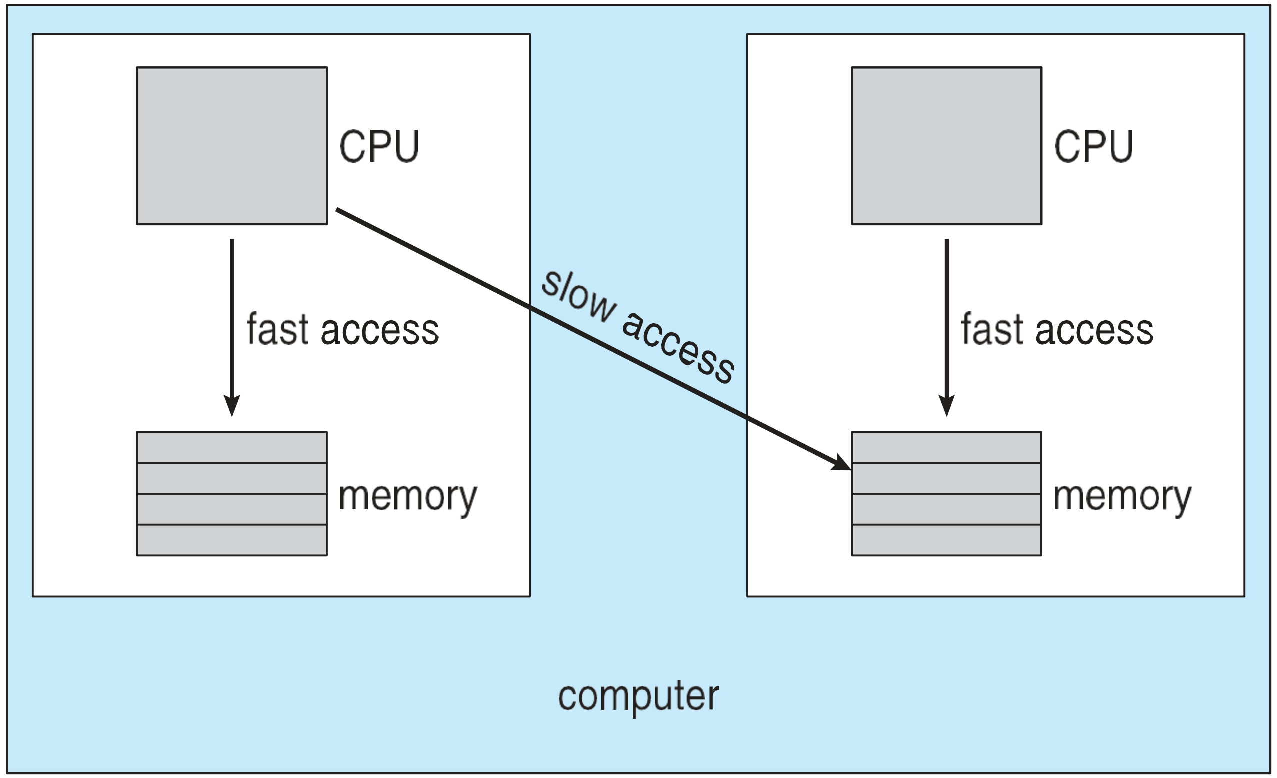
- When a **thread has been running on one processor,** the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)

Multiple-Processor Scheduling – Processor Affinity

- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- Soft affinity – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- Hard affinity – allows a process to specify a set of processors it may run on.

NUMA and CPU Scheduling

- If the operating system is NUMA-aware, it will assign memory closes to the CPU the thread is running on.


CPU
fast access
memory
slow access
CPU
fast access
memory
computer