Process management

What are we going to learn?

- ***Processes :*** Concept of processes, process scheduling, co-operating processes, inter-process communication.

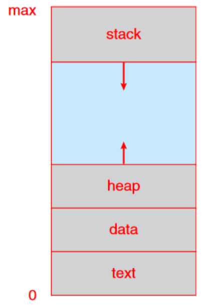
- ***CPU scheduling :*** scheduling criteria, preemptive & non-preemptive scheduling, scheduling algorithms (FCFS, SJF, RR, priority), algorithm evaluation, multi-processor scheduling.

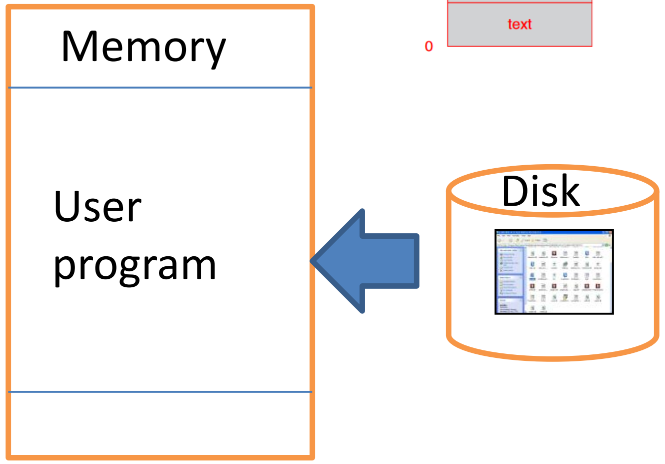
- ***Process Synchronization :*** background, critical section problem, critical region, synchronization hardware, classical problems of synchronization, semaphores.

- ***Threads :*** overview, benefits of threads, user and kernel threads.

- ***Deadlocks :*** system model, deadlock characterization, methods for handling deadlocks, deadlock prevention, deadlock avoidance, deadlock detection, recovery from deadlock.

Process concept



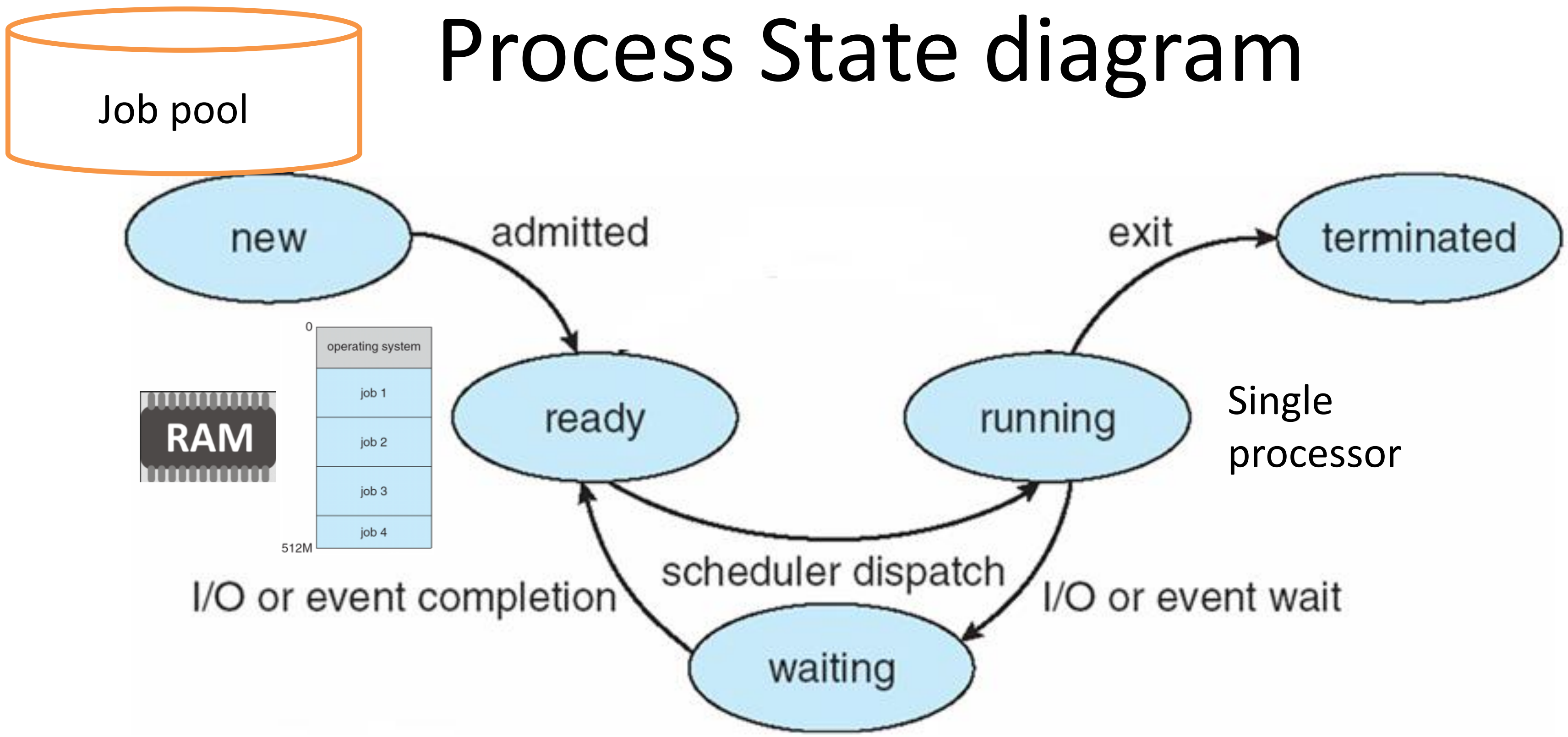


- Process is a dynamic entity
  - Program in execution
- Program code
  - Contains the text section
- Program becomes a process when
  - executable file is loaded in the memory
  - Allocation of various resources
    - Processor, register, memory, file, devices
- One program code may create several processes
  - One user opened several MS Word
  - Equivalent code/text section
  - Other resources may vary

Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution

Process State diagram

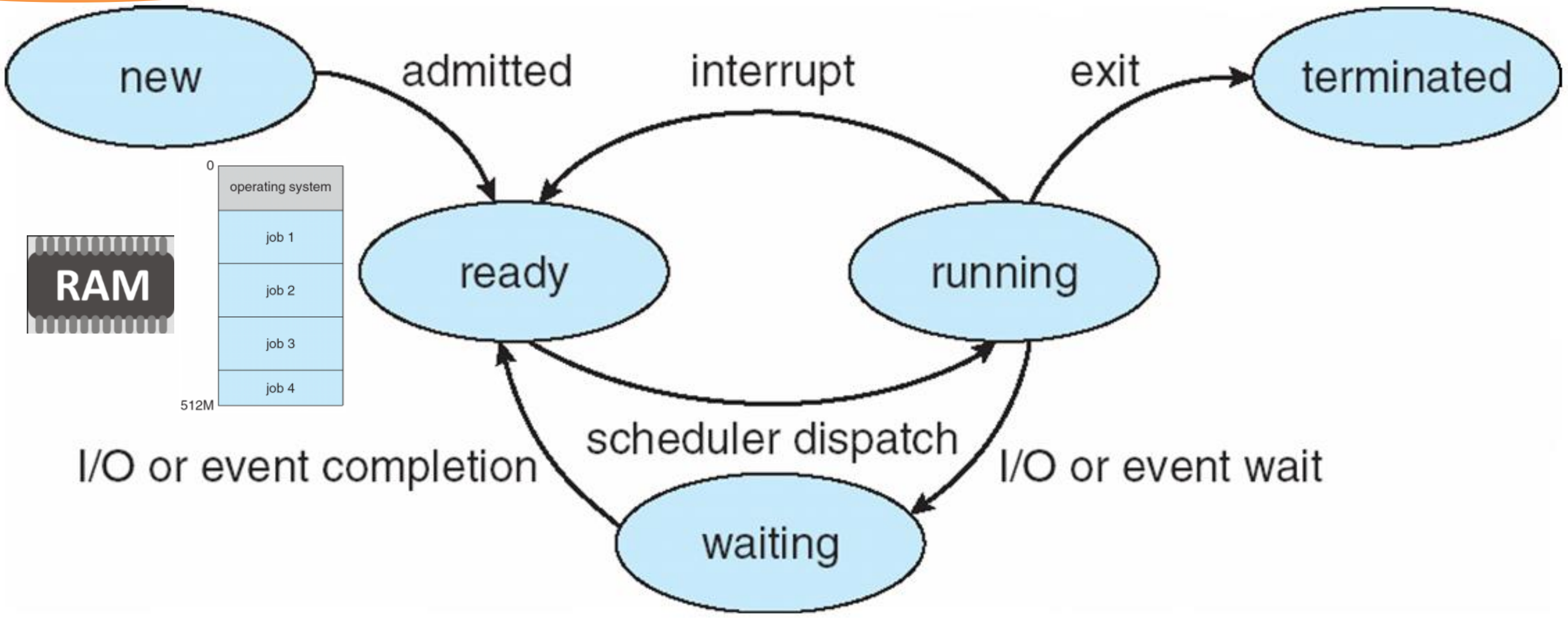


Multiprogramming

As a process executes, it changes *state*

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Process State diagram



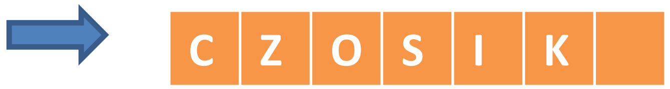
Multitasking/Time sharing

As a process executes, it changes *state*

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

How to represent a process?

- Process is a dynamic entity
  - Program in execution
- Program code
  - Contains the text section
- Program counter (PC)
- Values of different registers
  - Stack pointer (SP) (maintains process stack)
    - Return address, Function parameters
  - Program status word (PSW)
  - General purpose registers
- Main Memory allocation
  - Data section
    - Variables
  - Heap
    - Dynamic allocation of memory during process execution

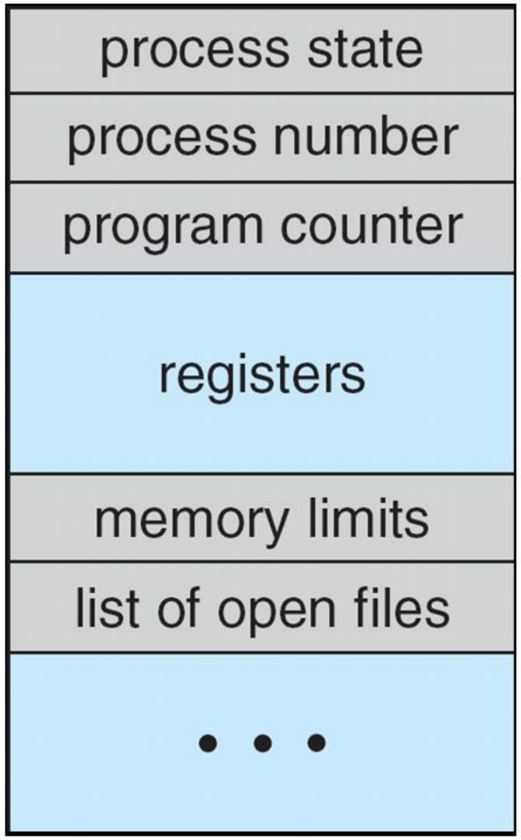


Process Control Block (PCB)

- Process is represented in the operating system by a Process Control Block

Information associated with each process

- Process state
- Program counter
- CPU registers
  - Accumulator, Index reg., stack pointer, general Purpose reg., Program Status Word (PSW)
- CPU scheduling information
  - Priority info, pointer to scheduling queue
- Memory-management information
  - Memory information of a process
  - Base register, Limit register, page table, segment table
- Accounting information
  - CPU usage time, Process ID, Time slice
- I/O status information
  - List of open files=> file descriptors
  - Allocated devices

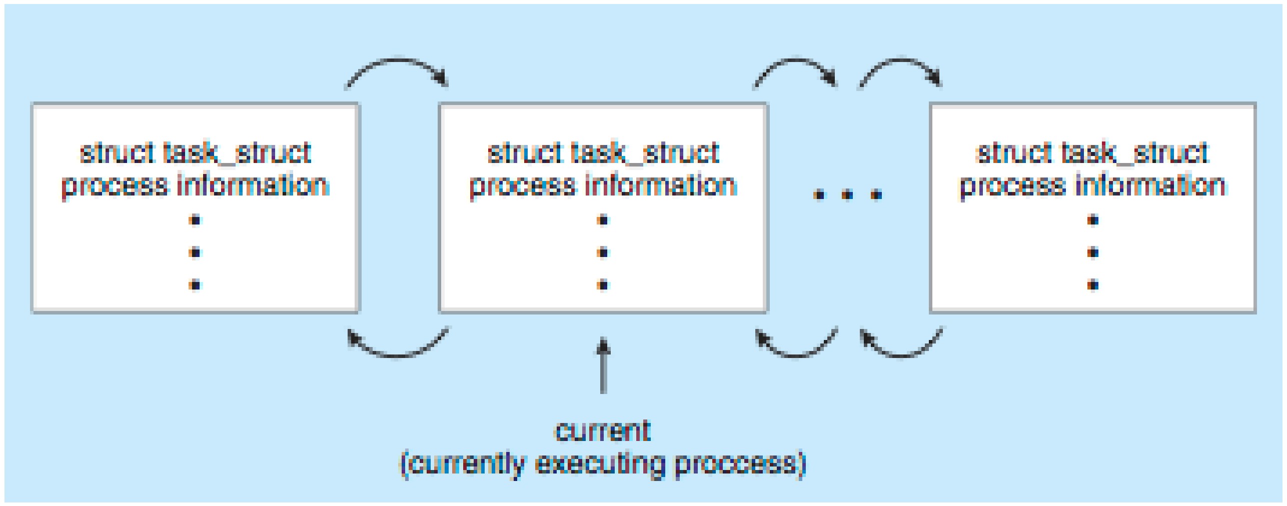


Process Representation in Linux

Represented by the C structure `task_struct`

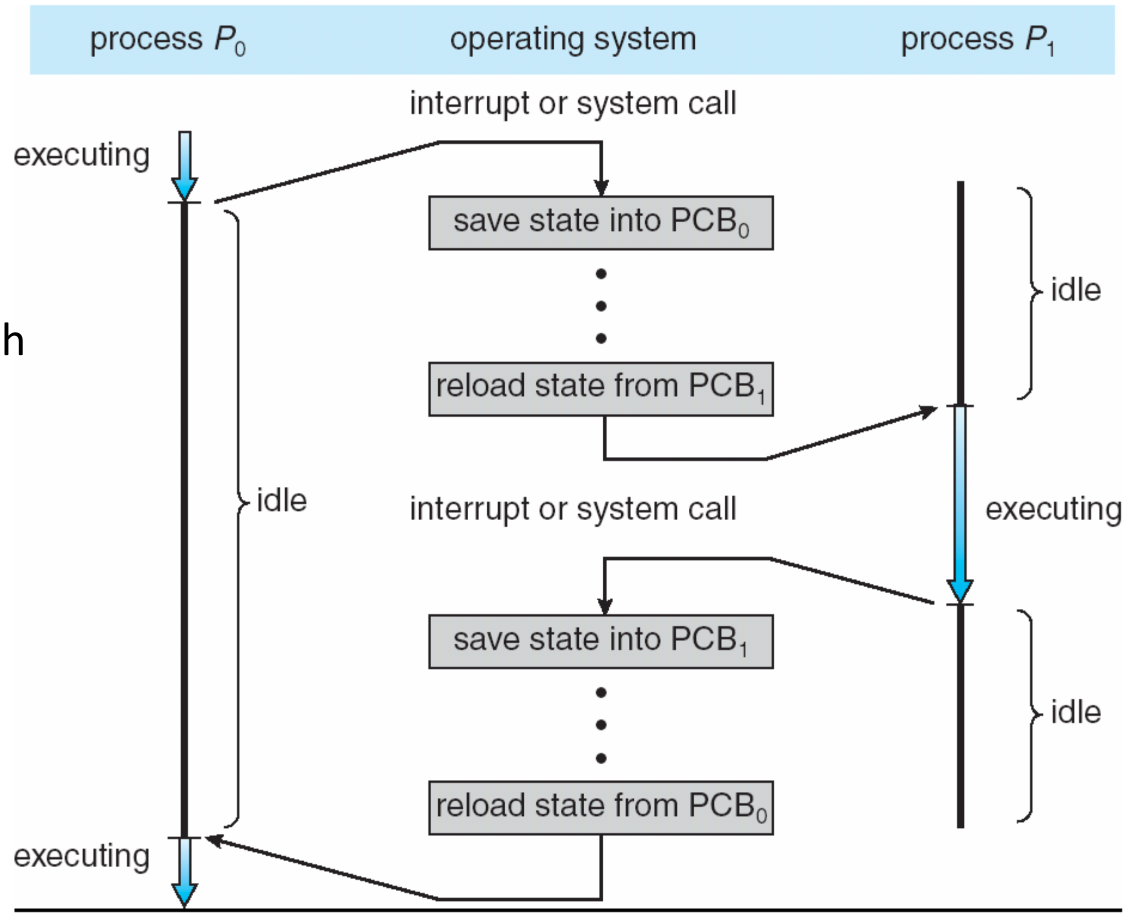
```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this pro */
```

Doubly linked list

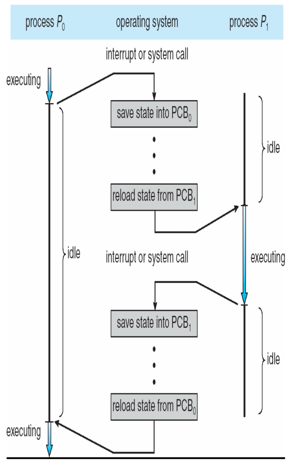


CPU Switch From Process to Process

Context switch



Context Switch

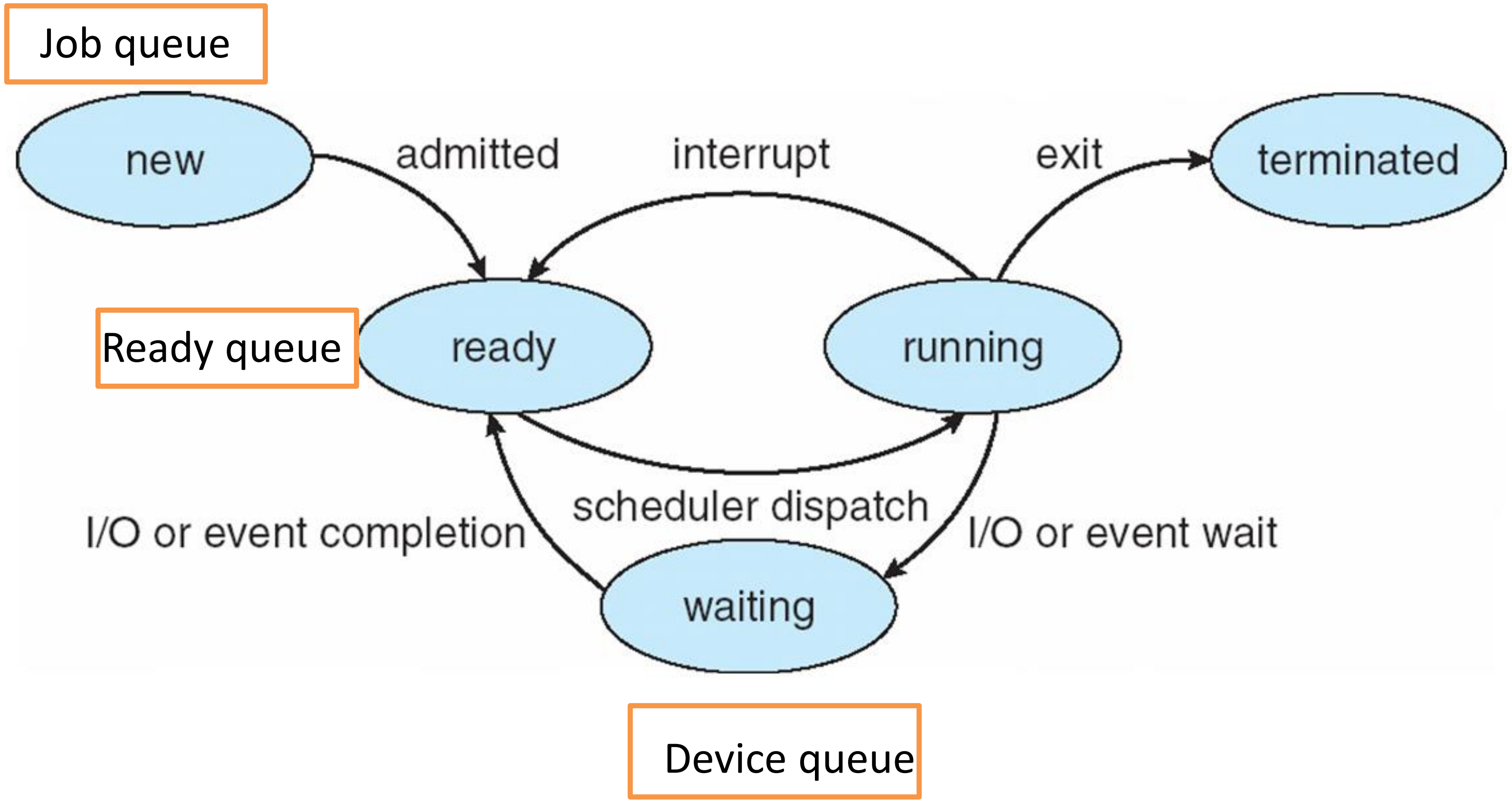


- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does not do useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

Scheduling queues

- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **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

Scheduling queues

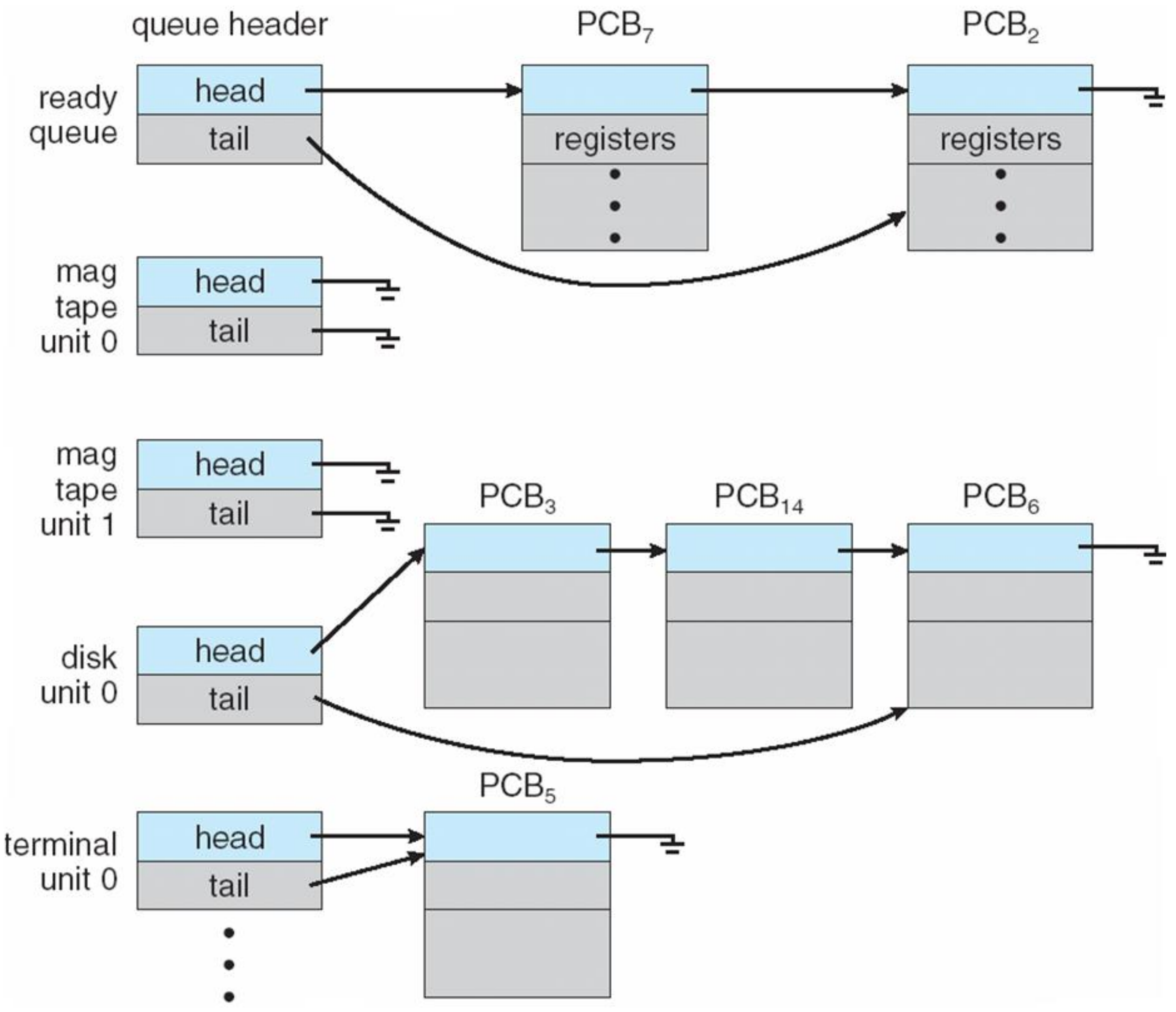


Ready Queue And Various I/O Device Queues

Queues are linked list of PCB's

Device queue

Many processes are waiting for disk



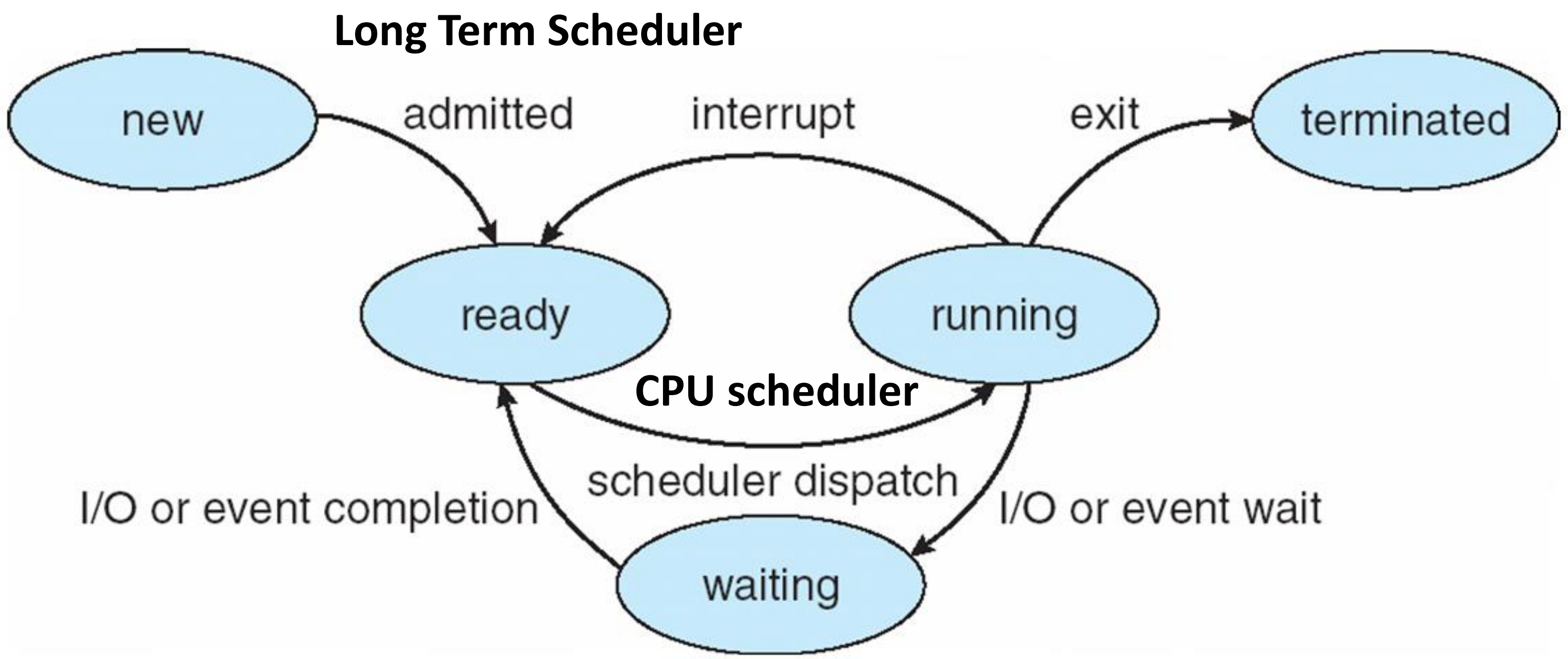
Process Scheduling

- We have various queues
- Single processor system
  - Only one CPU=> only one running process
- Selection of one process from a group of processes
  - **Process scheduling**

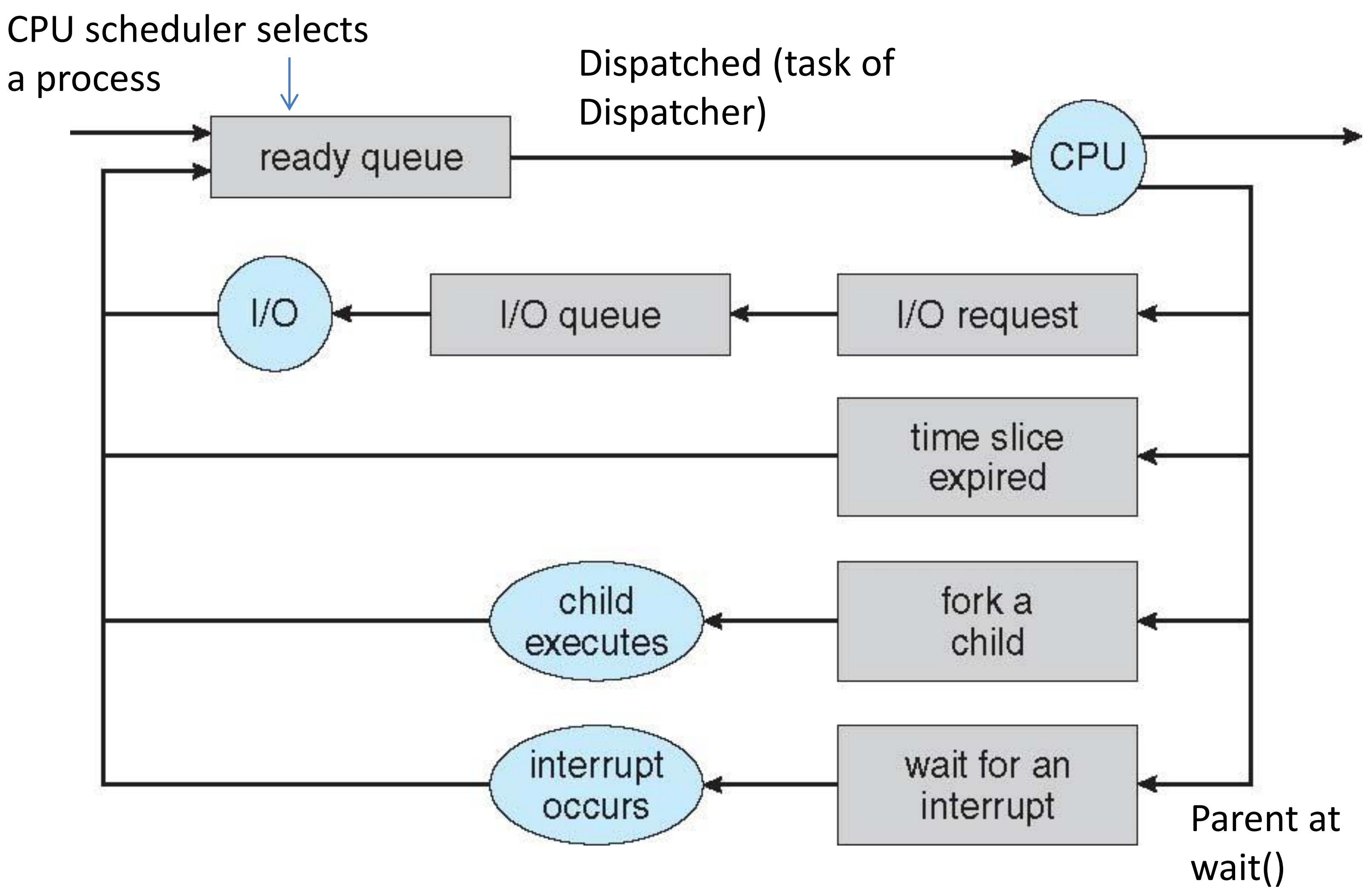
Process Scheduling

- Scheduler
  - Selects a process from a set of processes
- Two kinds of schedulers

1. Long term schedulers, job scheduler
   - A large number of processes are submitted (more than memory capacity)
   - Stored in disk
   - Long term scheduler selects process from job pool and loads in memory
2. Short term scheduler, CPU scheduler
   - Selects one process among the processes in the memory (ready queue)
   - Allocates to CPU


Long Term Scheduler
new
admitted
interrupt
exit
terminated
ready
running
CPU scheduler
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Representation of Process Scheduling



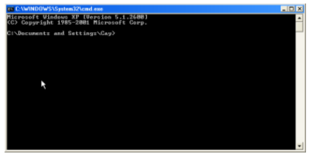
Dispatcher

- 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

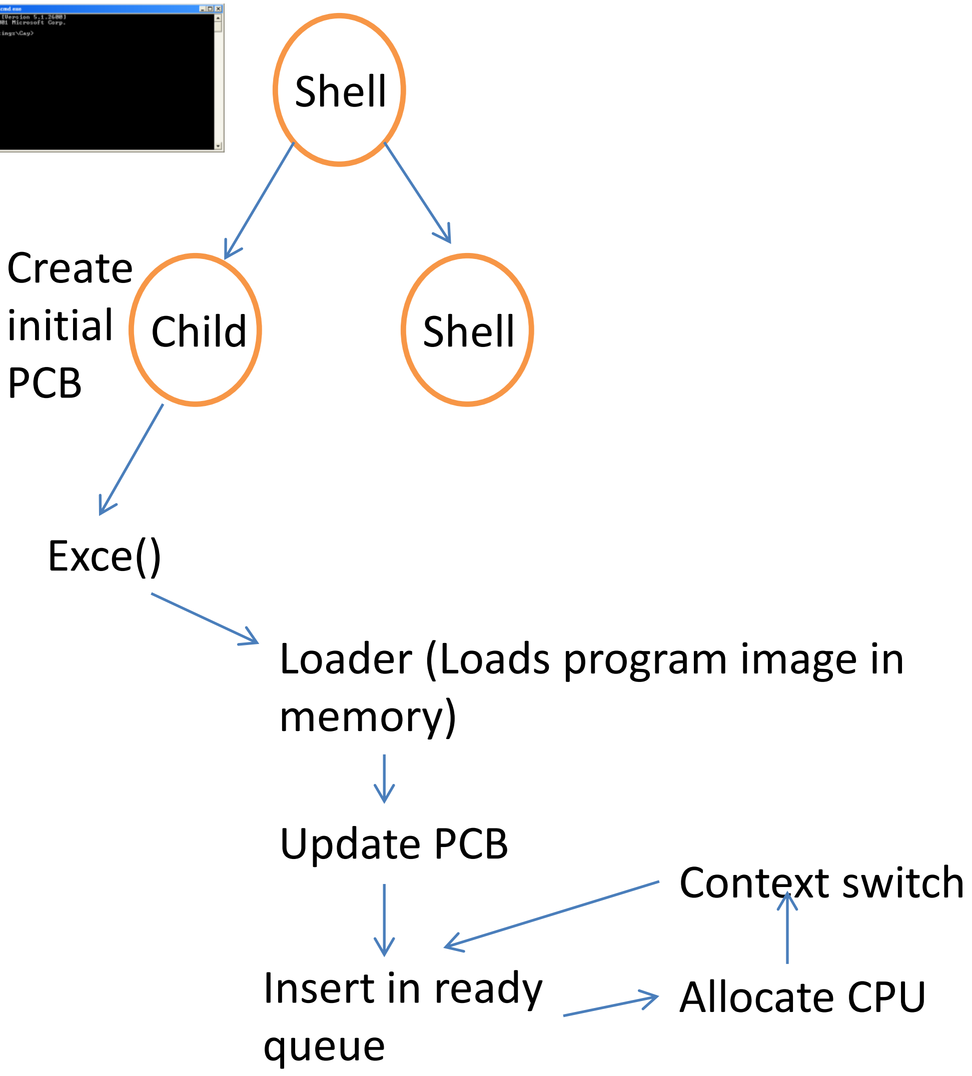
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Creation of PCB

a.out



```
C:\WINDOWS\System32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Gay>
```



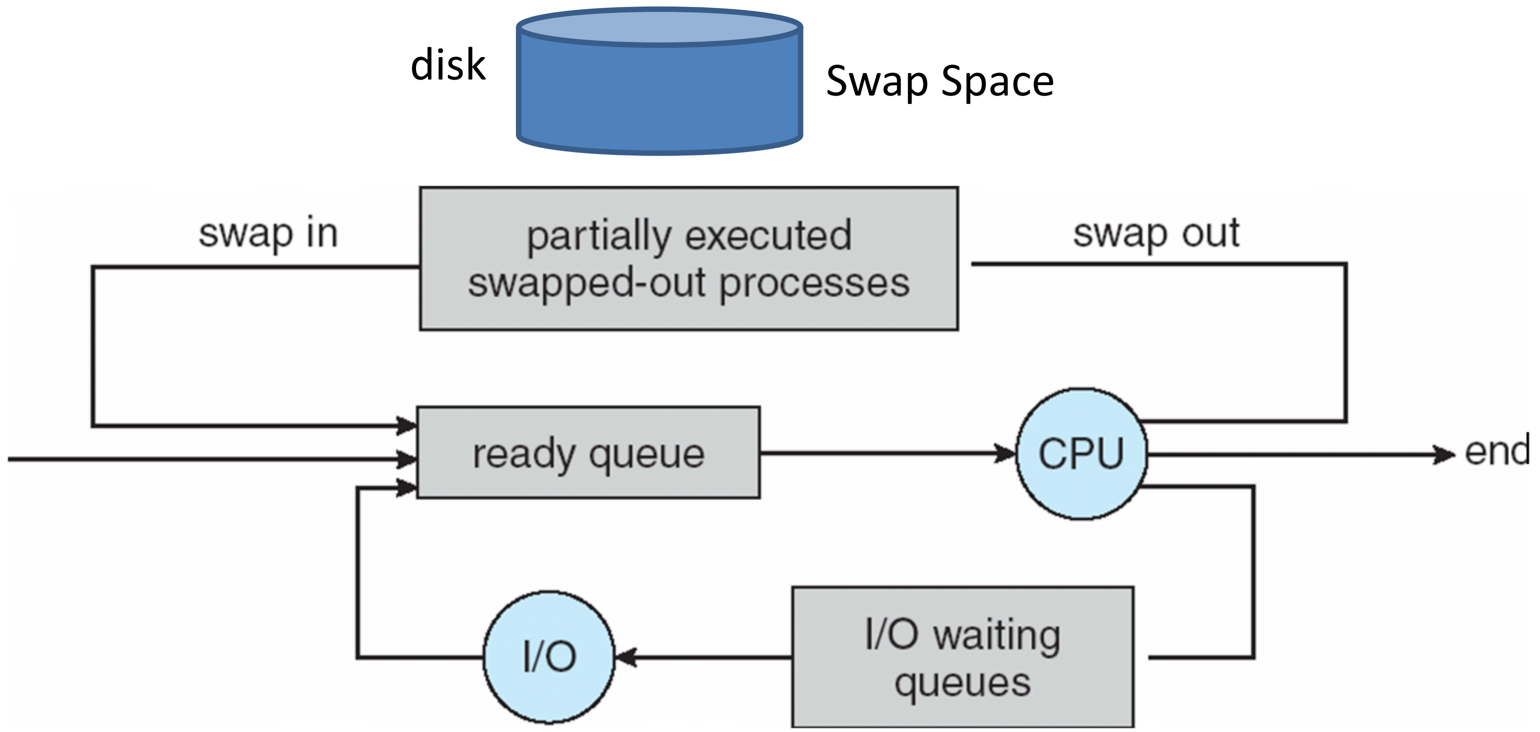
Schedulers

- **Scheduler**
  - Selects a process from a set
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system

Schedulers: frequency of execution

- Short-term scheduler is invoked **very frequently** (milliseconds) $\Rightarrow$ (must be fast)
  - After a I/O request/ Interrupt
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - Ready queue empty
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
    - Devices unused
- Long term scheduler ensures good process mix of I/O and CPU bound processes.

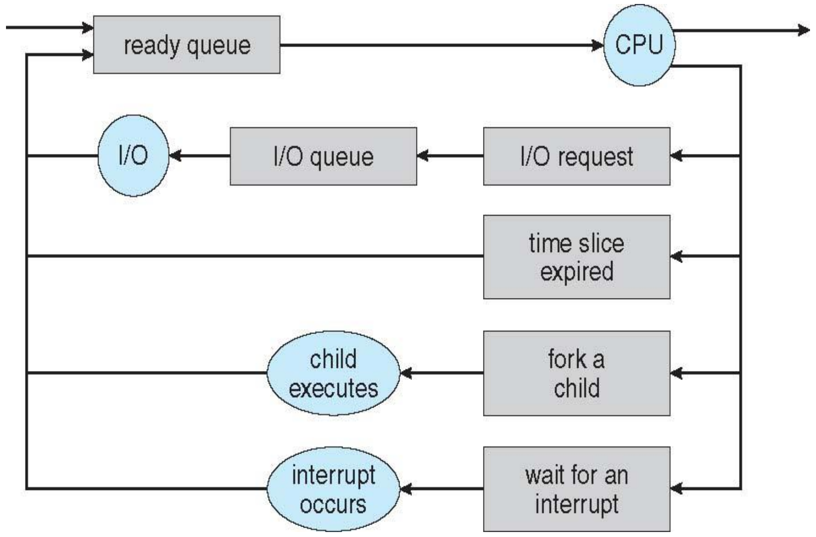
Addition of Medium Term Scheduling



Swapper

ISR for context switch

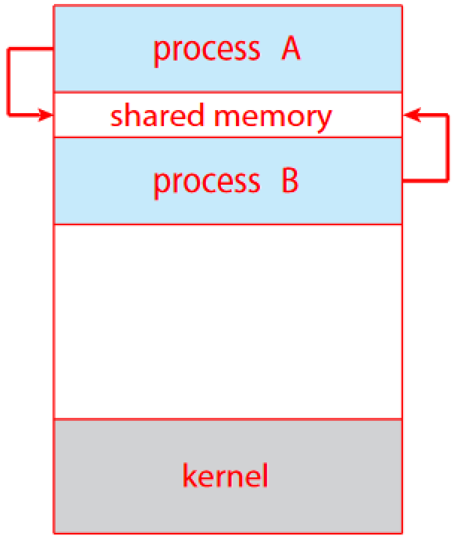
```
Current <- PCB of current process
Context_switch()
{
            Disable interrupt;
            switch to kernel mode
            Save_PCB(current);
            Insert(ready_queue, current);
            next=CPU_Scheduler(ready_queue);
            remove(ready_queue, next);
            Dispatcher(next);
            switch to user mode;
            Enable Interrupt;

}
Dispatcher(next)
{
            Load_PCB(next); [update PC]


}
```



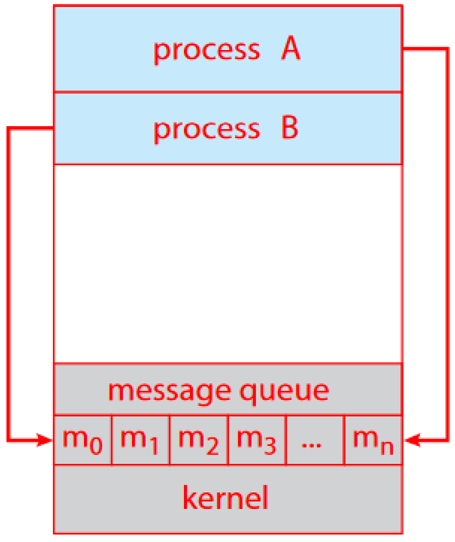
Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data— that is, send data to and receive data from each other.
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Interprocess Communication



(a)



(b)

In the **shared-memory model**, a **region of memory** that is shared by the cooperating processes is established.
Processes can then exchange information by reading and writing data to the shared region.

In the **message-passing model**, communication takes place by means of messages exchanged between the cooperating processes
(Kernel involvement, slow)

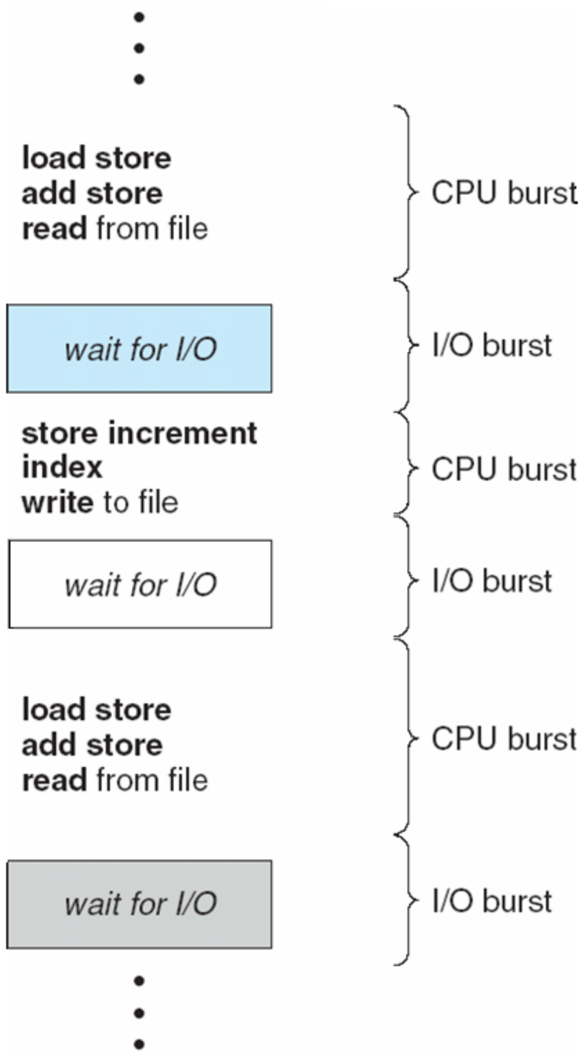
CPU Scheduling

- Describe various CPU-scheduling algorithms
- Evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

Basic Concepts

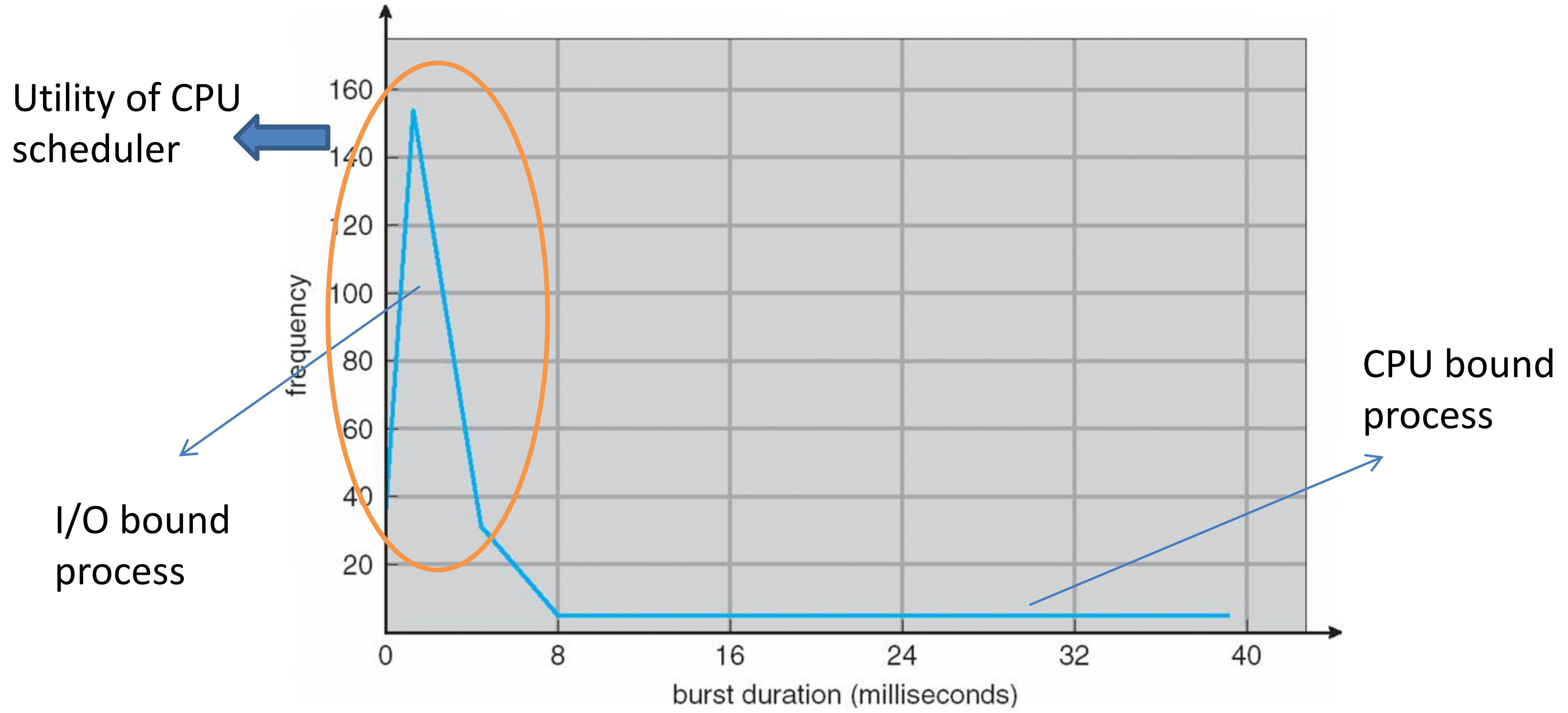
- Maximum CPU utilization obtained with multiprogramming
  - Several processes in memory (ready queue)
  - When one process requests I/O, some other process gets the CPU
  - Select (schedule) a process and allocate CPU

Observed properties of Processes



- CPU–I/O Burst Cycle
- Process execution consists of a *cycle* of CPU execution and I/O wait
- Study the duration of CPU bursts

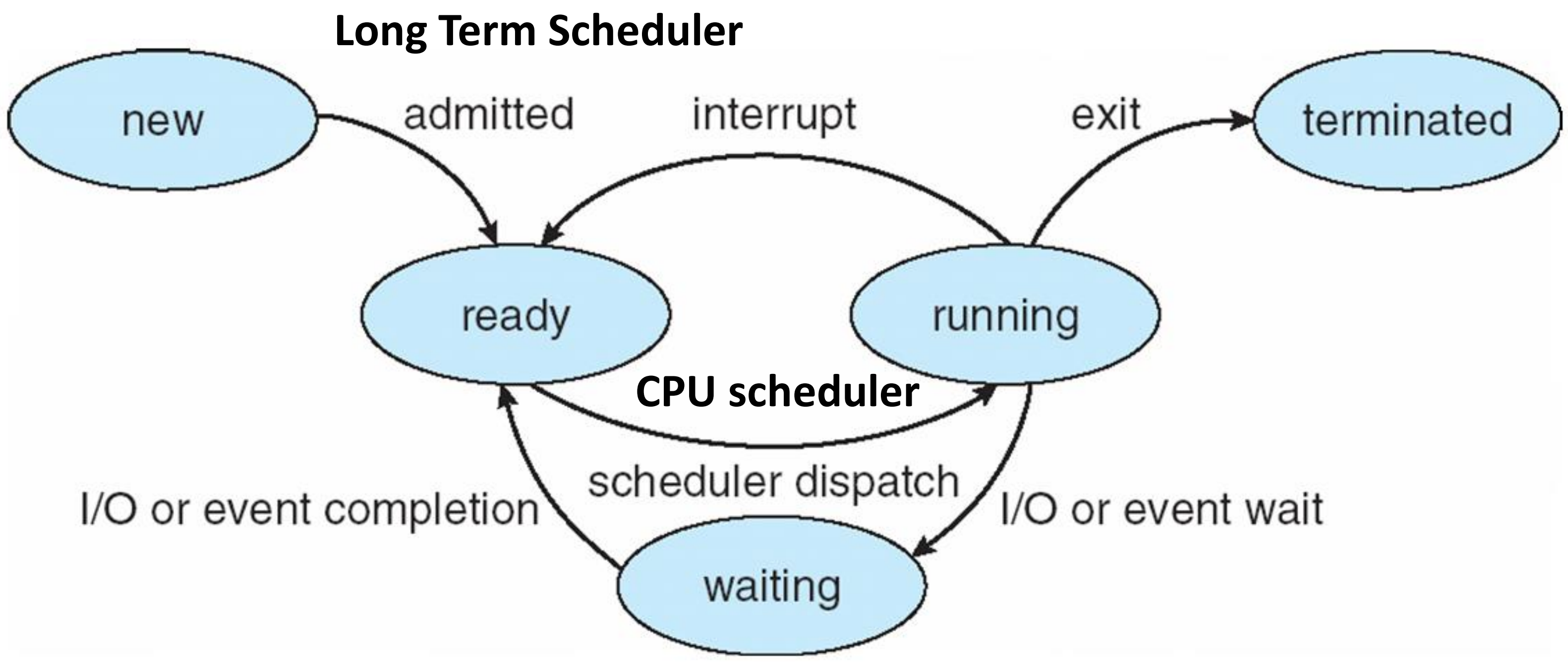
Histogram of CPU-burst Times



Large number of short CPU bursts and small number of long CPU bursts

Preemptive and non preemptive

- Selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways (not necessarily FIFO)
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**


Long Term Scheduler
new
admitted
interrupt
exit
terminated
ready
running
CPU scheduler
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Preemptive scheduling

Preemptive scheduling

Results in cooperative processes

Issues:

– Consider access to shared data
  - Process synchronization

– Consider preemption while in kernel mode
  - Updating the ready or device queue
  - Preempted and running a “ps -el”

Race condition

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, not output

Scheduling Algorithm Optimization Criteria

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

- Mostly optimize the average
- Sometimes optimize the minimum or maximum value
  - Minimize max response time

- For interactive system, variance is important
  - E.g. response time
- System must behave in predictable way

Scheduling algorithms

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling
- Round Robin (RR)

First-Come, First-Served (FCFS) Scheduling

- Process that requests CPU first, is allocated the CPU first
- Ready queue=>FIFO queue
- Non preemptive
- Simple to implement

Performance evaluation

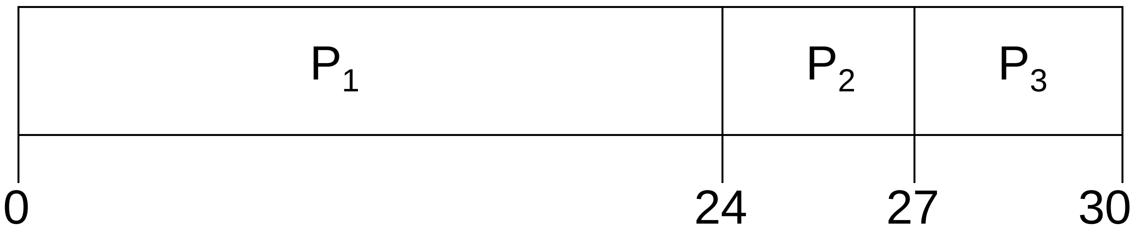
- Ideally many processes with several CPU and I/O bursts

- Here we consider only one CPU burst per process

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



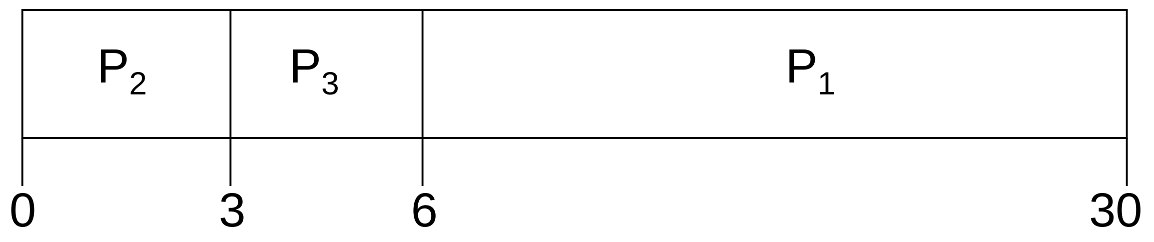
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- Average waiting time under FCFS heavily depends on process arrival time and burst time
- **Convoy effect** - short process behind long process
  – Consider one CPU-bound and many I/O-bound processes

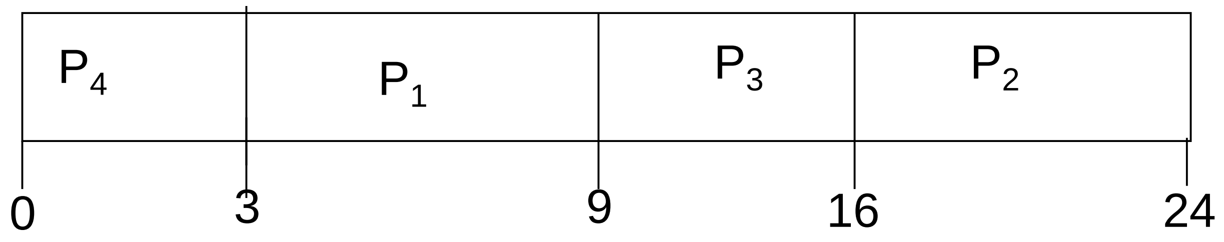
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Allocate CPU to a process with the smallest next CPU burst.
  - Not on the total CPU time
- Tie=>FCFS

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

Avg waiting time for FCFS?

SJF

- SJF is optimal – gives minimum average waiting time for a given set of processes (Proof: home work!)
- The difficulty is knowing the length of the next CPU request
- Useful for Long term scheduler
  - Batch system
  - Could ask the user to estimate
  - Too low value may result in “time-limit-exceeded error”

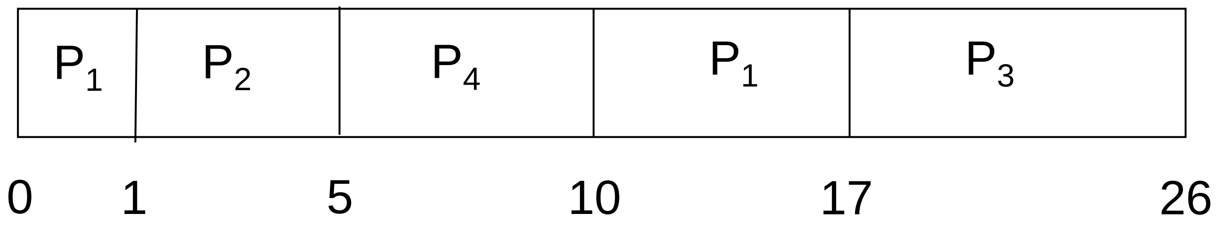
Preemptive version

Shortest-remaining-time-first

- Preemptive version called **shortest-remaining-time-first**
- 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 msec

Avg waiting time for non preemptive?

Determining Length of Next CPU Burst

- Estimation of the CPU burst length – should be similar to the previous burst
  - Then pick process with shortest predicted next CPU burst
- Estimation can be done by using the length of previous CPU bursts, using time series analysis

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

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

Boundary cases α=0, 1

- Commonly, α set to ½

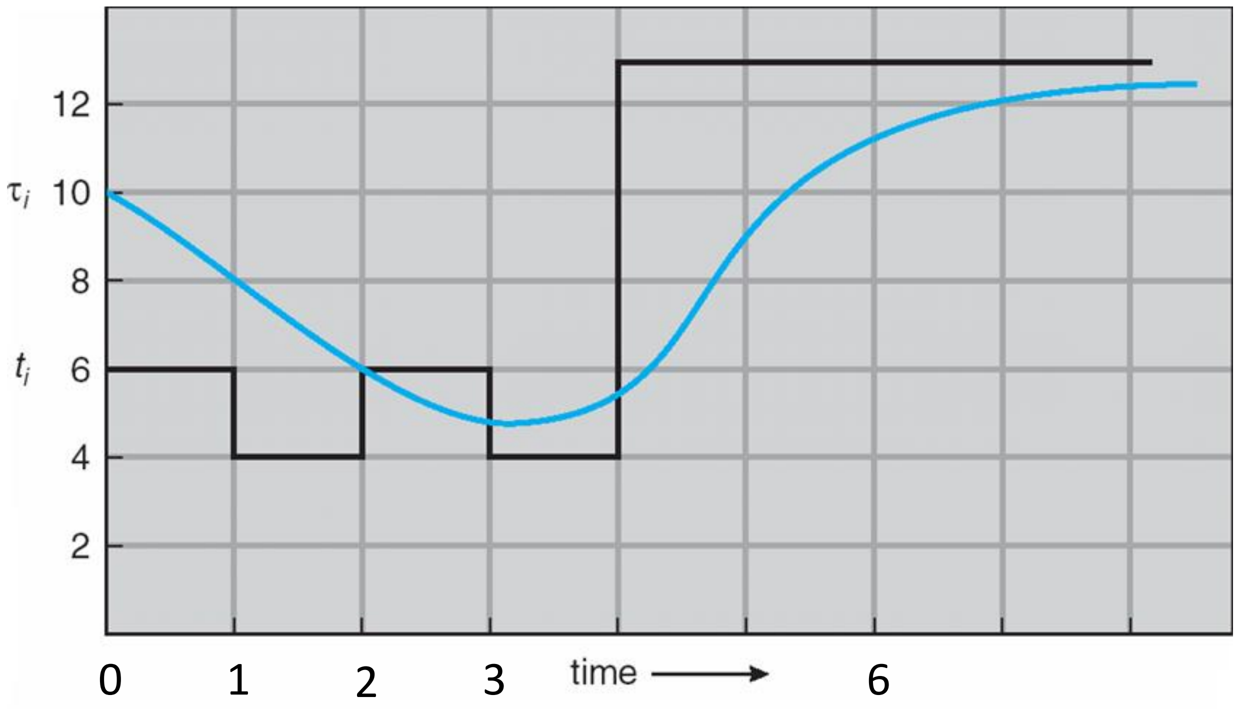
Examples of Exponential Averaging

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

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

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

Prediction of the Length of the Next CPU Burst



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

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)
- Set priority value
  - Internal (time limit, memory req., ratio of I/O Vs CPU burst)
  - External (importance, fund etc)

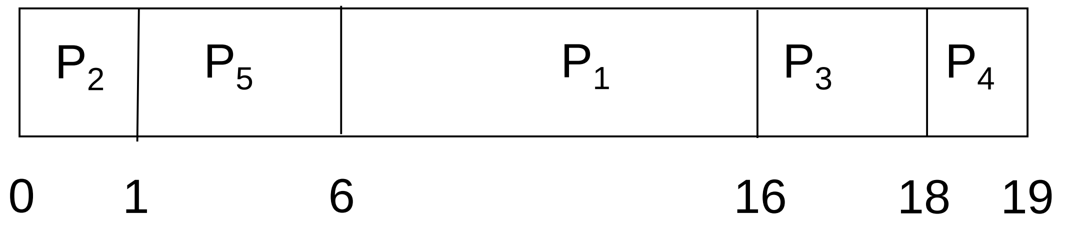
nice

- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Two types
  - Preemptive
  - Nonpreemptive
- 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 msec

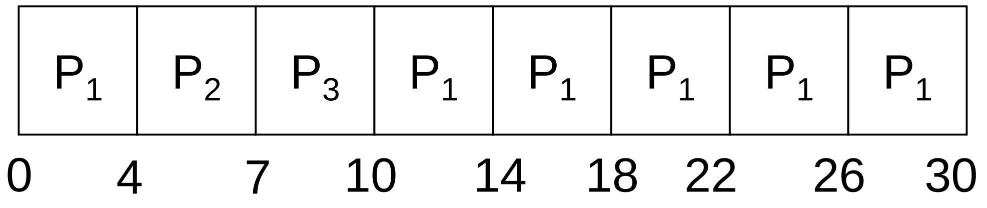
Round Robin (RR)

- Designed for time sharing system
- 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.
- Implementation
  - Ready queue as FIFO queue
  - CPU scheduler picks the first process from the ready queue
  - Sets the timer for 1 time quantum
  - Invokes despatcher
- If CPU burst time < quantum
  - Process releases CPU
- Else Interrupt
  - Context switch
  - Add the process at the tail of the ready queue
  - Select the front process of the ready queue and allocate CPU

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Avg waiting time = ((10-4)+4+7)/3=5.66

Round Robin (RR)

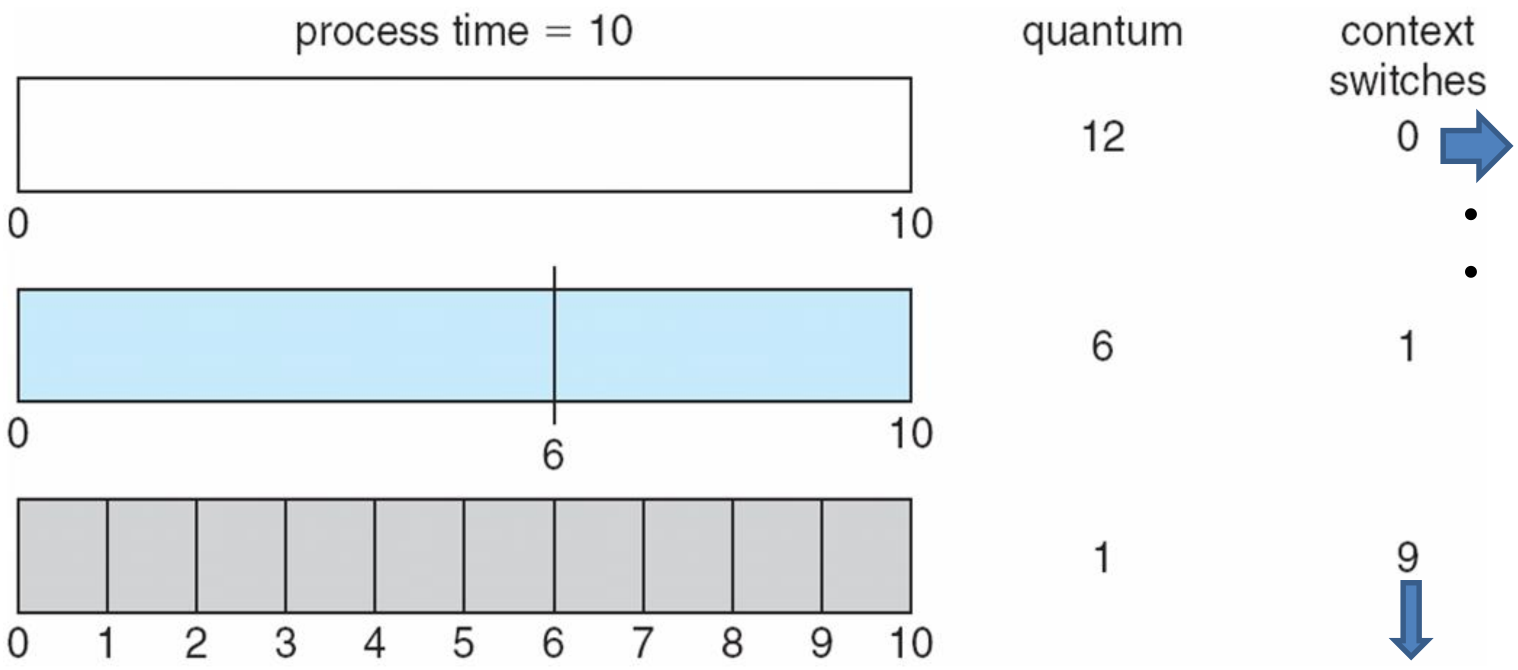
- Each process has a time quantum *T* allotted to it
- Dispatcher starts process P_0, loads a external counter (timer) with counts to count down from *T* to 0
- When the timer expires, the CPU is interrupted
- The context switch ISR gets invoked
- The context switch saves the context of P_0
  - PCB of P_0 tells where to save
- The scheduler selects P_1 from ready queue
  - The PCB of P_1 tells where the old state, if any, is saved
- The dispatcher loads the context of P_1
- The dispatcher reloads the counter (timer) with *T*
- The ISR returns, restarting P_1 (since P_1's PC is now loaded as part of the new context loaded)
- P_1 starts running

Round Robin (RR)

- 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.
- Timer interrupts every quantum to schedule next process
- Performance depends on time quantum q
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ Processor sharing (n processes has own CPU running at 1/n speed)

Effect of Time Quantum and Context Switch Time

Performance of RR scheduling



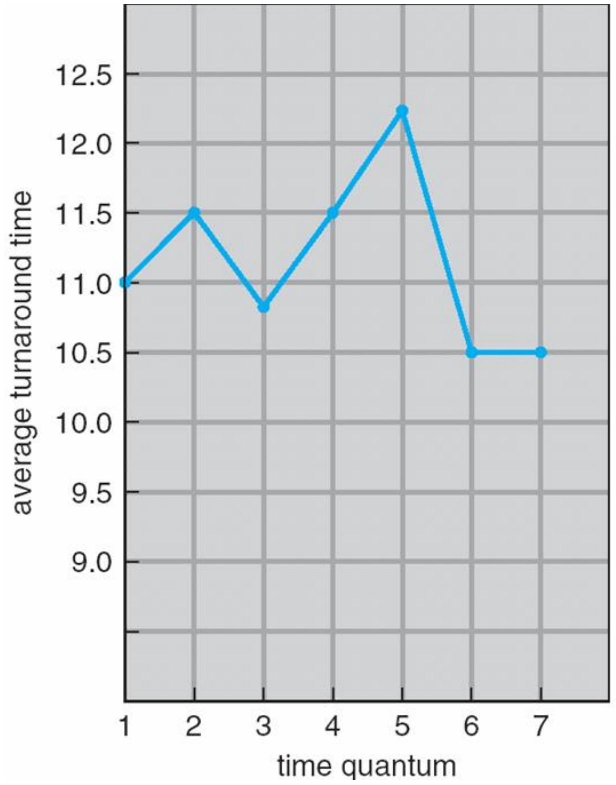
- No overhead
- However, poor response time

- Too much overhead!
- Slowing the execution time

- *q* must be large with respect to context switch, otherwise overhead is too high
- q usually 10ms to 100ms, context switch < 10 microsec

Effect on Turnaround Time

- TT depends on the time quantum and CPU burst time
    - Better if most processes complete there next CPU burst in a single q



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- Large q=> processes in ready queue suffer
- Small q=> Completion will take more time

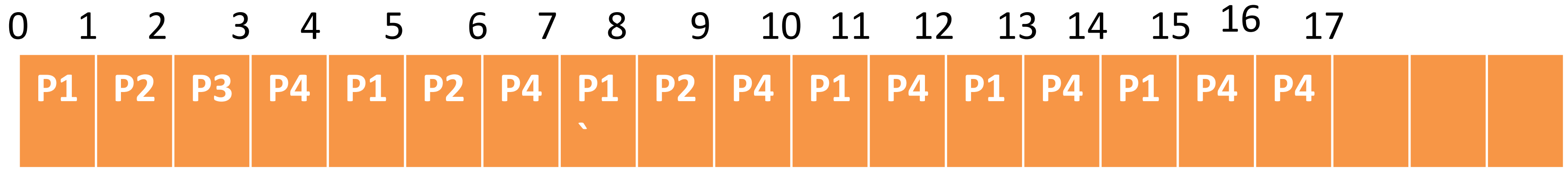
80% of CPU bursts should be shorter than q

Response time

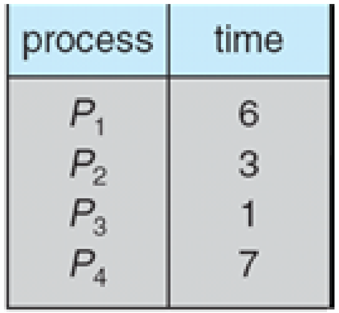
Typically, higher average turnaround than SJF, but better *response time*

Turnaround Time

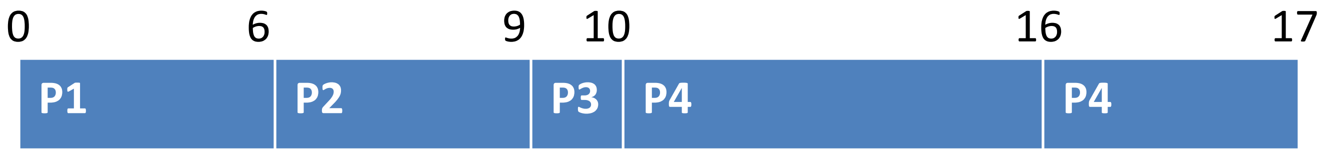
q=1



Avg Turnaround time=
(15+9+3+17)/4=11



process	time
P_1	6
P_2	3
P_3	1
P_4	7



q=6

(6+9+10+17)/4=10.5

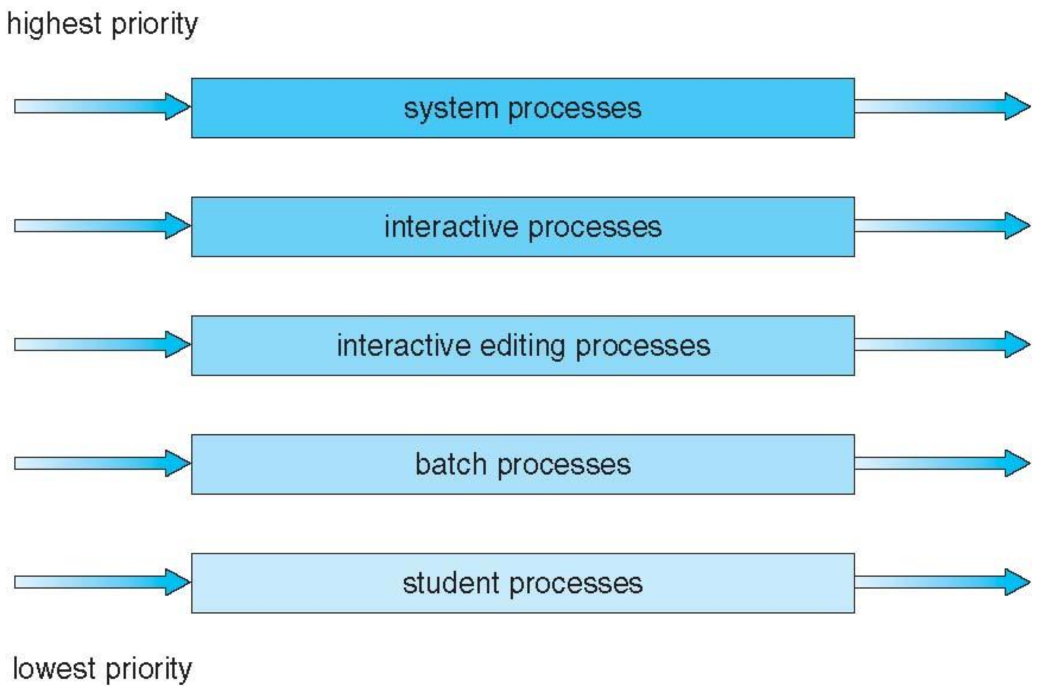
Process classification

- Foreground process
  - Interactive
  - Frequent I/O request
  - Requires low response time
- Background Process
  - Less interactive
  - Like batch process
  - Allows high response time
- Can use different scheduling algorithms for two types of processes ?

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently assigned in a given queue
  - Based on process type, priority, memory req.
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background).
  - Possibility of starvation.

Multilevel Queue Scheduling



- No process in batch queue could run unless upper queues are empty
- If new process enters
  - Preempt

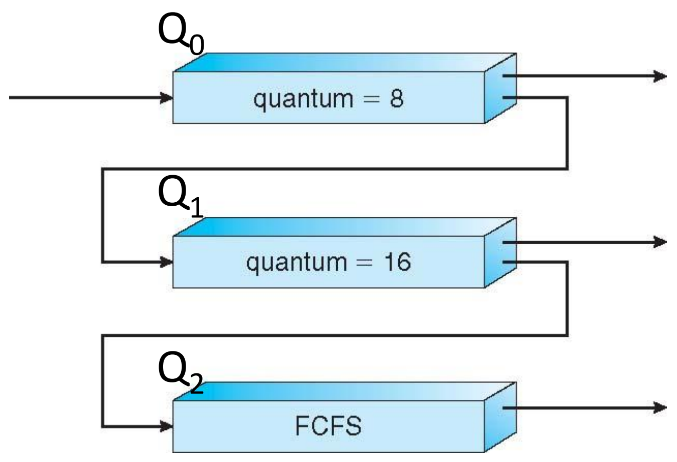
Another possibility

- 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
- 20% to background in FCFS

Multilevel Feedback Queue

- So a process is permanently assigned a queue when they enter in the system
  - They do not move
- **Flexibility!**
  - Multilevel-feedback-queue scheduling
- A process can move between the various queues;
- Separate processes based of the CPU bursts
  - Process using too much CPU time can be moved to lower priority
  - Interactive process => Higher priority
- Move process from low to high priority
  - Implement aging

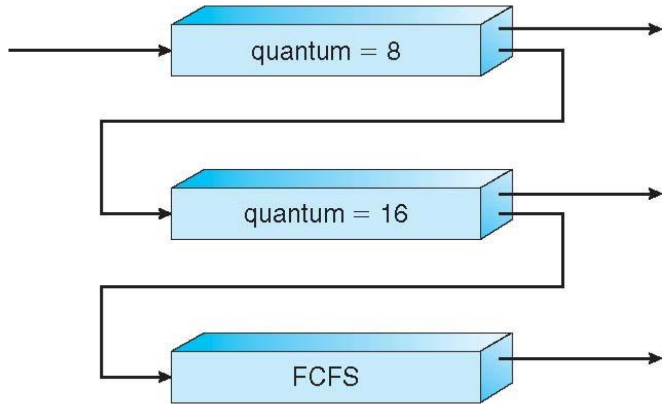
Example of Multilevel Feedback Queue



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

- Scheduling
  - A new job enters queue Q_0
    - 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 receives 16 milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Multilevel Feedback Queues



- Highest Priority to processes
  CPU burst time <8 ms
- Then processes >8 and <24

- 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

Multiple-Processor Scheduling

- If multiple CPUs are available, multiple processes may run in parallel

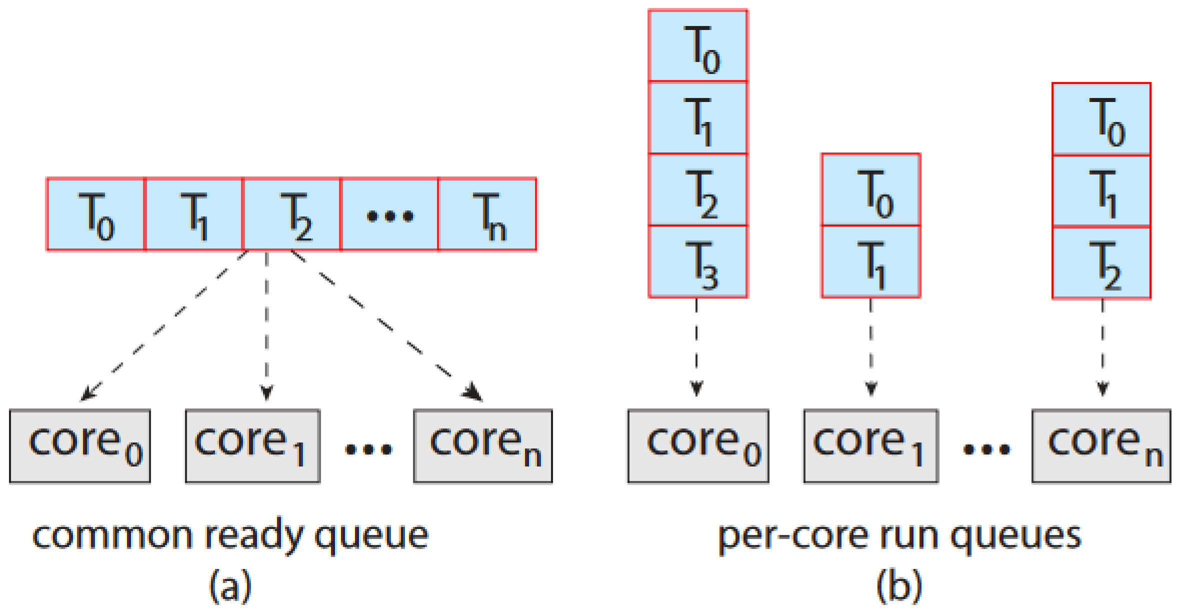
- However **scheduling** issues become correspondingly more **complex**.
- Many possibilities have been tried
- As we saw with CPU scheduling with a single-core CPU
  - there is no one best solution

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** –
  - Master server
  - only **one processor** accesses the **system data** structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling,
- all processes in **common ready queue**, or each has its **own private queue** of ready processes

- Scheduler for each processor **examine the ready queue**
  - select a process to run.

Multiple-Processor Scheduling



- We have a possible **race condition**
- **Locking** to protect the common ready queue from this race condition.
- Accessing the shared queue would likely be a **performance bottleneck**

- Permits each processor to schedule process from its **private ready queue**
- **Does not suffer** from the possible **performance** problems
- Most common approach on systems supporting SMP.
- **Load balancing**

Multi core processors

- SMP systems have allowed several processes to run **in parallel** by providing **multiple physical processors**.

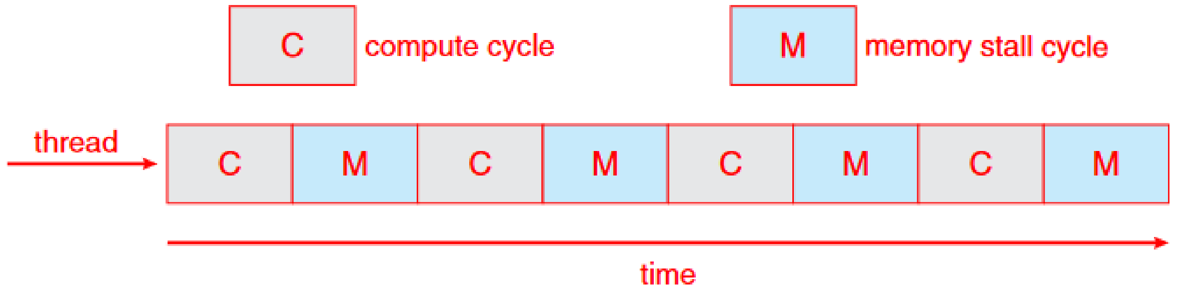
- Recently, **multiple computing cores** on the **same physical chip**, resulting in a **multicore processor**.

- **Each core** maintains its **architectural state** and thus appears to the operating system to be a separate **logical CPU**

- SMP systems that use multicore processors are **faster and consume less power**
  - than systems in which each CPU has its own physical chip

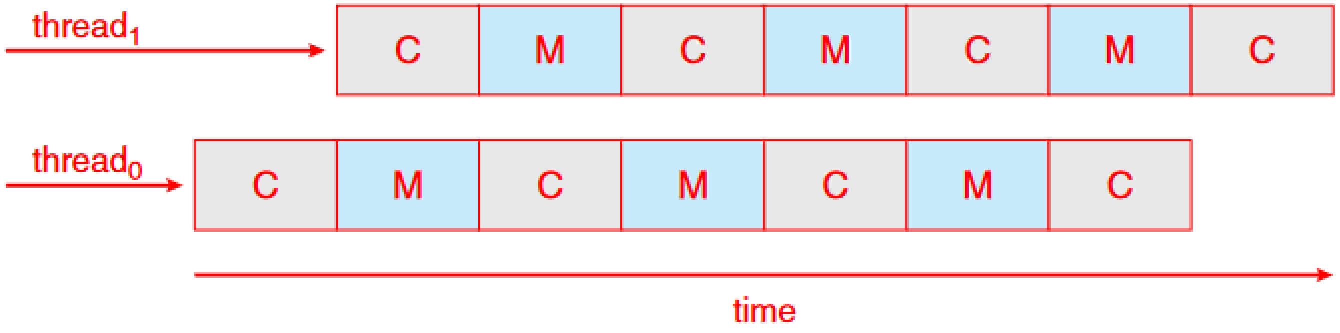
Challenge: Memory stall

- When a processor **accesses memory**, it spends a significant amount of **time waiting** for the data to become available.
- This situation, known as a **memory stall**, occurs primarily because
  - modern processors operate at much faster speeds than memory.
  - For cache miss

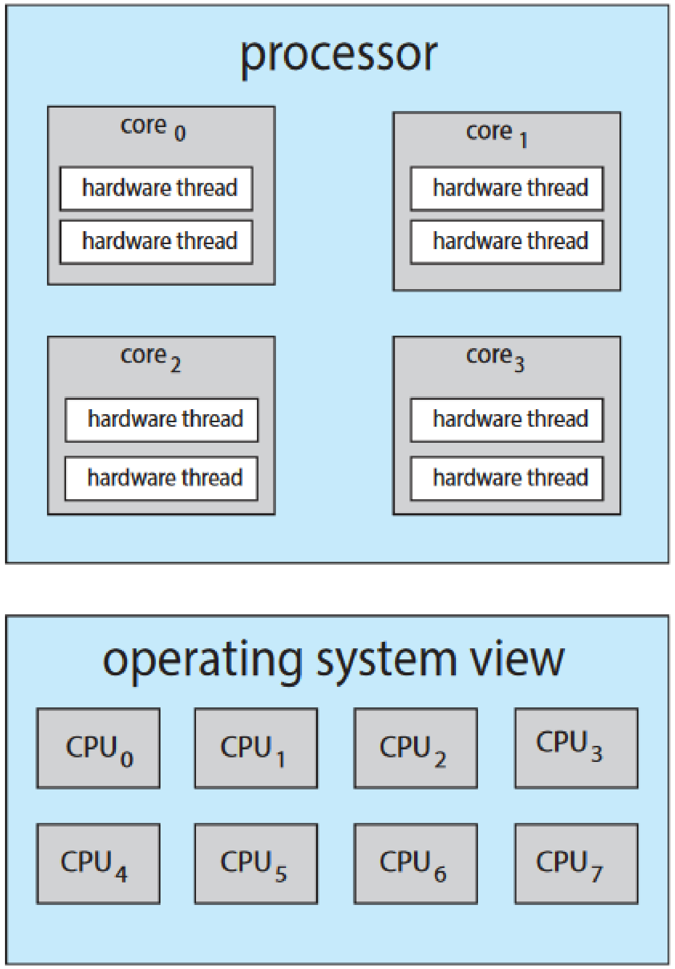


Solution: Hardware threads

- Recent hardware designs have implemented **multithreaded processing cores** in which two (or more) hardware threads are assigned to **each core**.
- That way, if **one hardware thread stalls** while waiting for memory, the core can **switch** to another thread.



Hyper-threading



- **Each hardware thread** maintains its **architectural state**, such as **instruction pointer and register set**,
- Thus appears as a **logical CPU** that is available to run a **software process**.

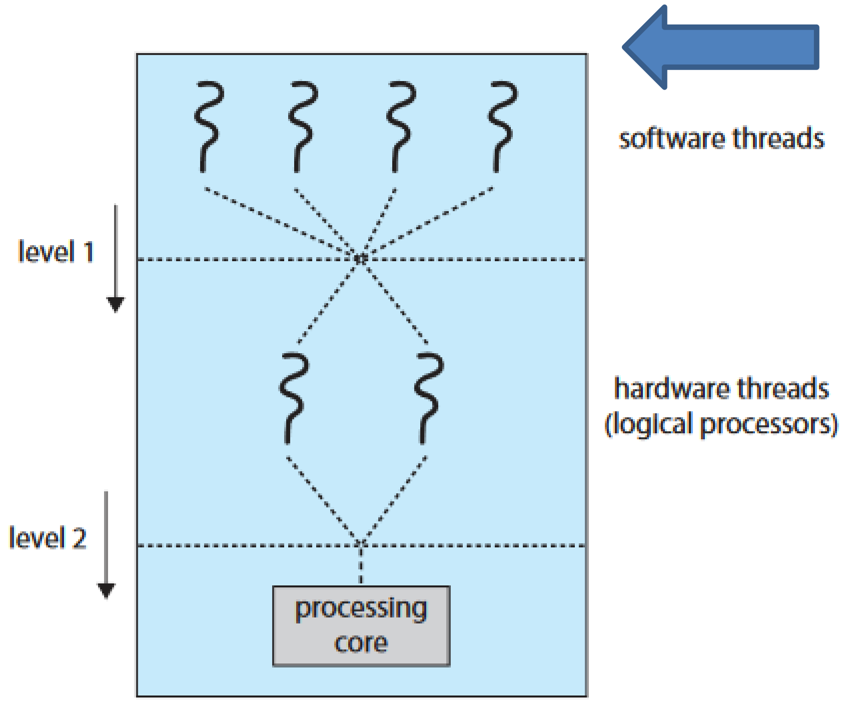
Contemporary Intel processors—such as the **Intel i7—support two threads per core**,

Oracle Sparc M7 processor supports **eight threads per core**, with eight cores per processor, thus providing the operating system with 64 logical CPUs

Dual scheduling

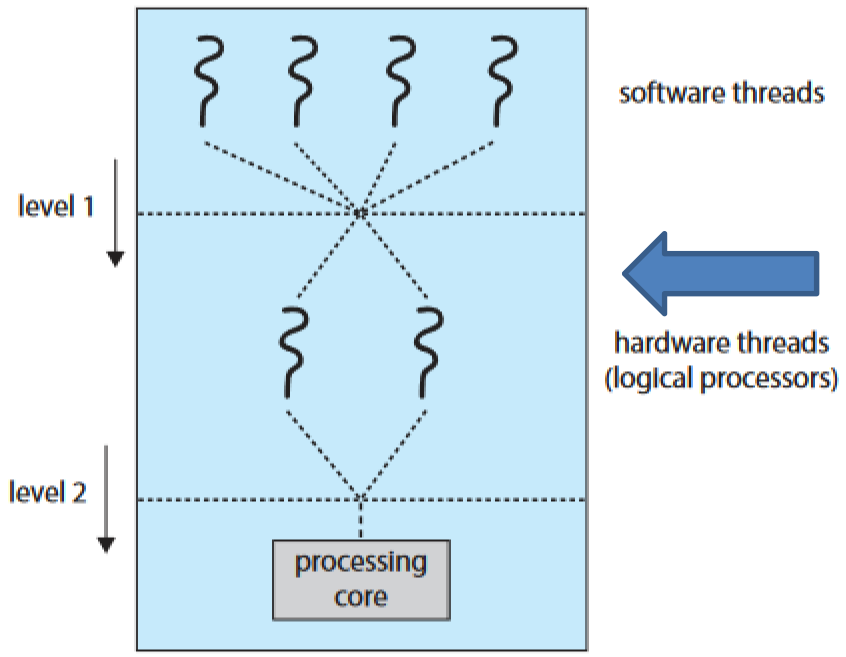
- It is important to note that
- a **processing core** can only execute **one hardware thread** at a time.
  - **resources of the physical core** (such as caches and pipelines) must be shared among its **hardware threads**
- Multithreaded, multicore processor actually requires **two different levels of scheduling**

Dual scheduling



- The **scheduling decisions** that must be made by the operating system as it **chooses which software process** to run on each **hardware thread** (logical CPU).

- For this level of scheduling, the operating system may choose **any scheduling algorithm**



- A second level of scheduling specifies **which hardware thread to run on a core**.
- One approach is to use a **simple round-robin algorithm** to schedule a hardware thread to the processing core.
- This is the approach adopted by the UltraSPARC T3.
- Another approach is used by the Intel Itanium
- Assigned to each hardware thread is a dynamic urgency value ranging from 0 to 7