Module 2

Process Scheduling

Contents

- Introduction to threads
- Multithreading Models
- Basic Concepts of CPU Scheduling
- Scheduling Algorithms: FCFS, SJF, RR, Priority,
- Multilevel Queue, Multilevel Feedback Queue.

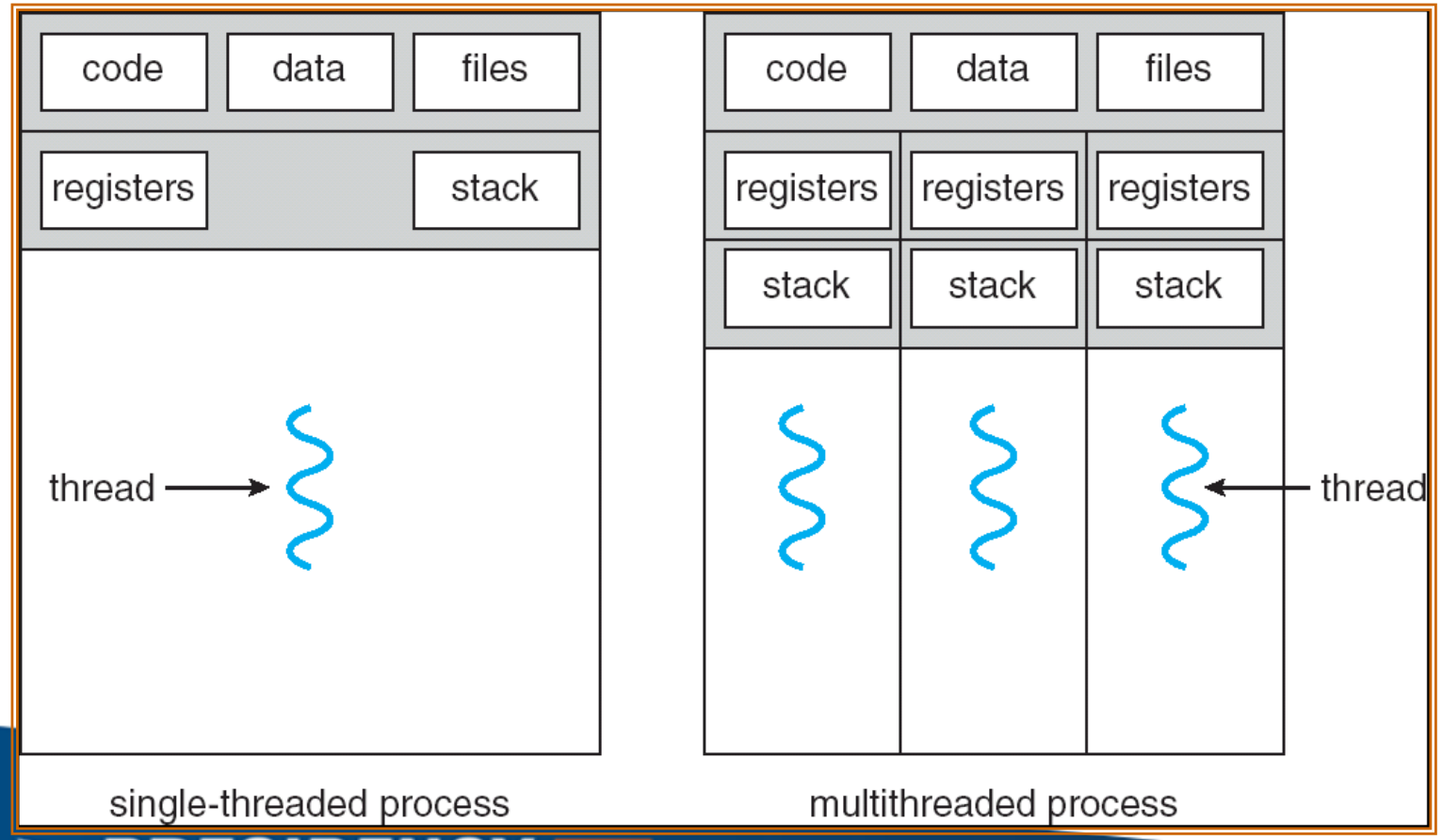
S1 Introduction to OS

Threads

Threads

- Called a light weight process
  - A thread is a basic unit of ***CPU utilization***
  - Multi-threaded processes share data for greater efficiency

Single and Multithreaded Processes



Four Primary Benefits of Threading

- **Responsiveness**
  - Interactive applications can respond to user while other threads are blocked on I/O or performing lengthy computations
- **Resource Sharing**
  - Threads share memory, code, and data, which enables several threads of activity within the same address space
- **Economy/Efficiency**
  - Resource allocation is costly. Sharing reduces overhead and reduces the cost of context switching during CPU scheduling
- **Utilization of Multiprocessor Architectures**
  - Can leverage multi-CPU architectures for genuine parallelism

User-level threads

- Threads can be provided at the **user** or **kernel level**
- User level: kernel knows nothing about threads
- Implemented in a library by somebody without touching the kernel
- User library handles
  - Thread creation
  - Thread deletion
  - Thread scheduling
- Benefits:
  - Faster creation and scheduling
  - Drawbacks:
  - One thread blocking during I/O blocks **all** threads in process (even ready-to-run threads)

User-level threads (cont'd)

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel-level threads

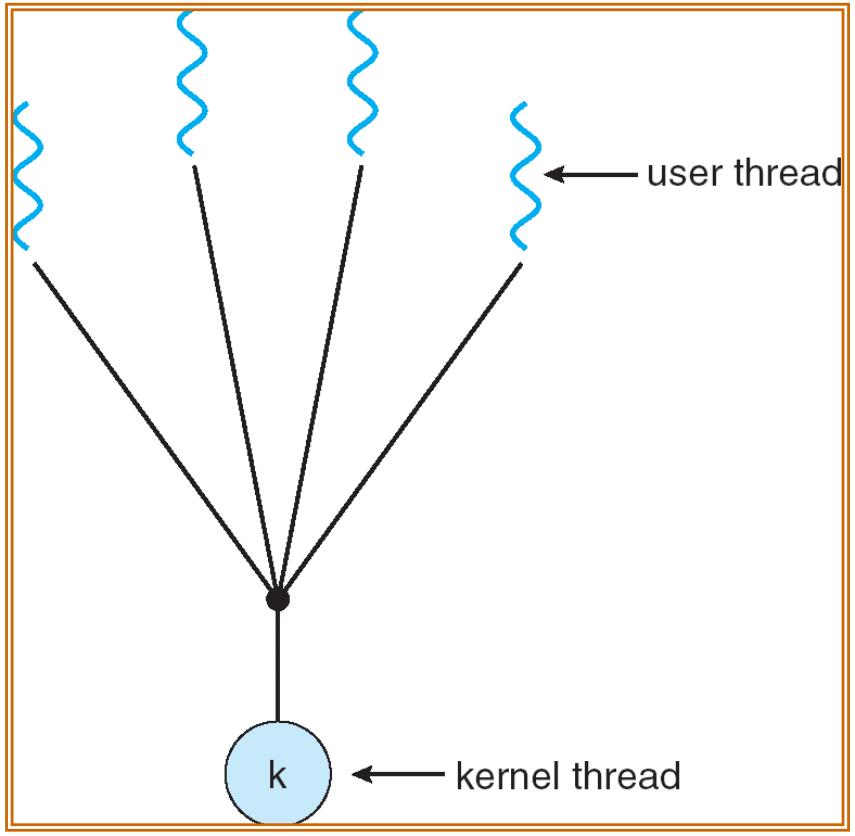
- Kernel knows about threads
- Kernel handles thread creation, deletion, scheduling
- Benefits:
  - Kernel can schedule another thread if current one does blocking I/O
  - Kernel can schedule multiple threads on different CPUs on SMP multiprocessor
- Drawbacks:
  - Slower to schedule, create, delete than user-level
- Most modern OS support kernel-level threads
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

S2 Multithreading Models

Multithreading Models

- Threading Model – defines the relationship between user-level threads and kernel-level threads
- Many-to-One
  - Many user threads mapped to a single kernel thread
- One-to-One
  - Each user thread mapped to a unique kernel thread
- Many-to-Many
  - Many user threads multiplexed among a potentially smaller set of underlying kernel threads

Many-to-One Model

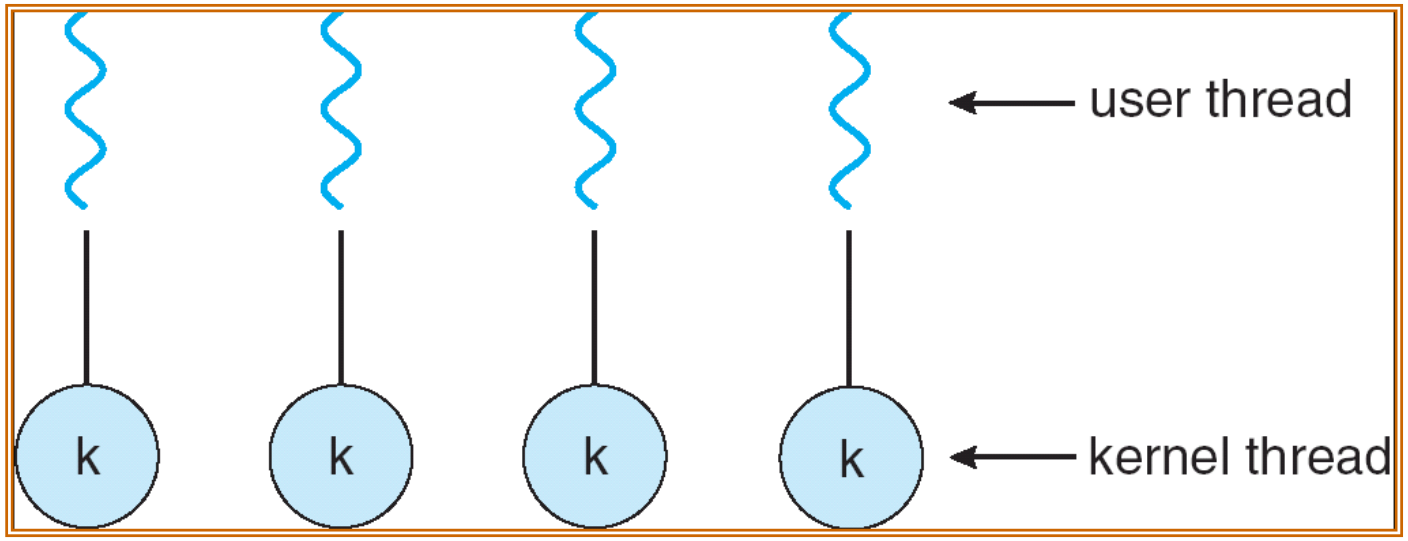


Many user-level threads mapped to single kernel thread

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
- Pros (Efficiency)
  - Threads managed by threading library (in user space)
  - Don’t need to block for system calls during management
- Cons
  - Entire process will block if ANY thread makes a blocking system call (say, for I/O)
  - Can’t leverage multi-processing architectures, because only one user-level thread can access the kernel at a time

One-to-one Model

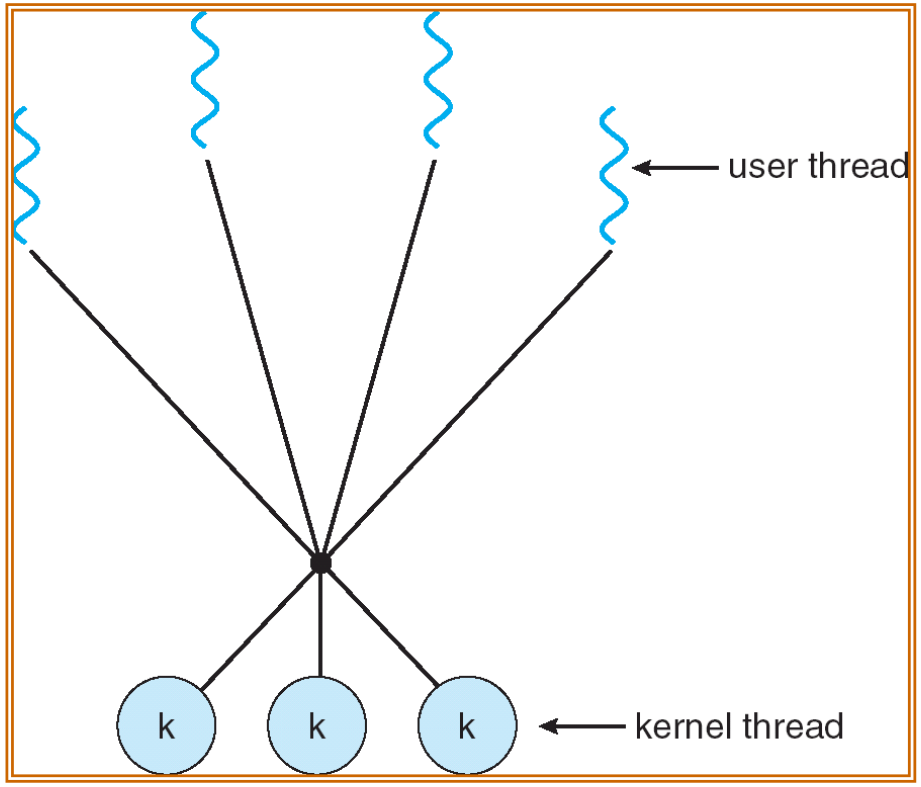


Each user-level thread maps to a unique kernel thread

One-to-One

- Each user-level thread maps to a unique kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
- Pros
  - Potential for greater concurrency – one thread can execute while another is blocked
  - Allows multiple threads to run in parallel on multi-processors
- Cons
  - Creating user thread requires creating a unique kernel thread
  - Overhead of kernel-thread creation can burden performance
  - Number of user threads bounded by number of kernel threads

Many-to-Many Model



Many user-level threads mapped to single kernel thread

Many-to-Many Model

- Many user level threads are mapped to many kernel threads
- Allows OS to create a sufficient number of kernel threads
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package
- Advantages over Many-to-One and One-to-One models:
  - Developers can create as many user threads as necessary
  - Corresponding kernel threads can execute in parallel
  - Other threads can execute while another thread blocks

Self Learning – Threading Issues

Threading Issues

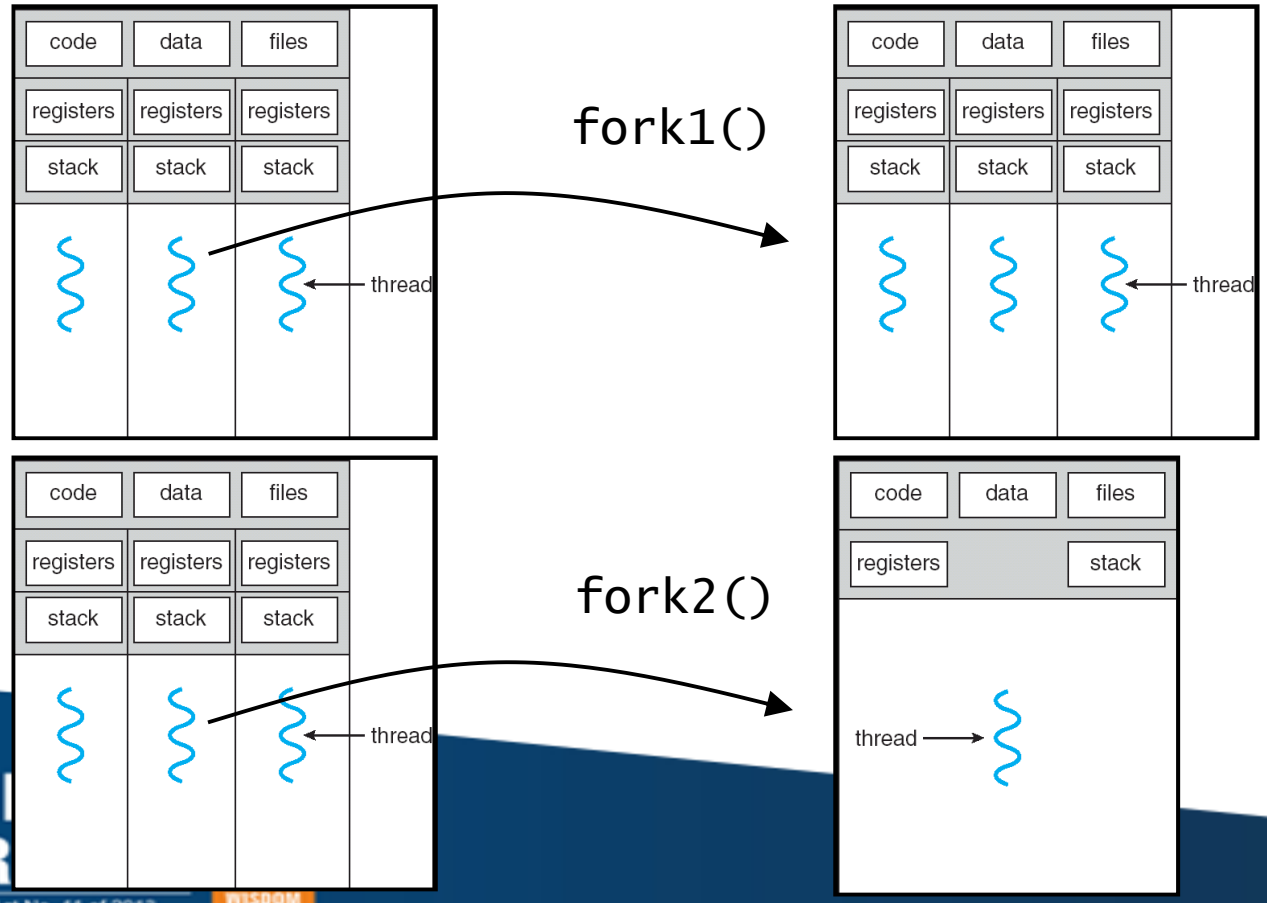
- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- Some UNIX systems provide both versions of **fork()**
- Which version should one use?
  - If **exec()** is called immediately after **fork()**
    - Only the calling thread needs to be duplicated
    - Why? Because **exec()** replaces the whole process anyway
  - If **exec()** is not called after **fork()**
    - Then typically duplicate all threads

Semantics of `fork()` and `exec()`

- Does `fork()` duplicate only the calling thread or all threads?
- Some UNIX systems have two versions of `fork()`
- `exec()` usually replaces all threads with new program



Thread Cancellation

- Cancellations = terminating a thread before it has finished
  - Example: Multi-threaded database search
  - Example: Clicking “stop” while loading a webpage

- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately (INTERRUPT)
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled (POLLING)

- Which approach to use?
  - Deferred cancellation is safer
  - Asynchronous cancellation can be easier/faster

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Generated Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
    - Example: Division by zero
  - Deliver the signal to every thread in the process
    - Example: Control-C should kill entire process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals to the process

Thread Pools

- Waiting space for threads like buffer, create threads
- Motivating example: a web server running on an SMP machine
- To handle each connection:
  1. Create a new process to handle it
     - too slow, inefficient
  2. Create a new thread to handle it
- Option 2 better but still has some problems:
  - Some overhead for thread creation/deletion
  - Thread will only be used for this connection
  - Unbounded number of threads might crash web server
- Better solution: use a thread pool of (usually) fixed size

Thread Pools (cont'd)

- Threads in pool sit idle
- Request comes in:
  - Wake up a thread in pool
  - Assign it the request
  - When it completes, return it to pool
  - If no threads in pool available, wait
- Advantages:
  - Usually slightly faster to wake up an existing thread than create a new one
  - Allows the number of threads in the application to be limited by the size of the pool
  - More sophisticated systems dynamically adjust pool size

Thread-specific Data

- Allows each thread to have its own copy of data
- Useful for implementing protection
  - For example, user connects to bank's database server
  - Server process responds, has access to all accounts
  - Multiple people may be accessing their accounts at the same time
  - Thread assigned to manipulate or view user's bank account
  - Using thread-specific data limits (unintentional or erroneous) access by other threads

Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

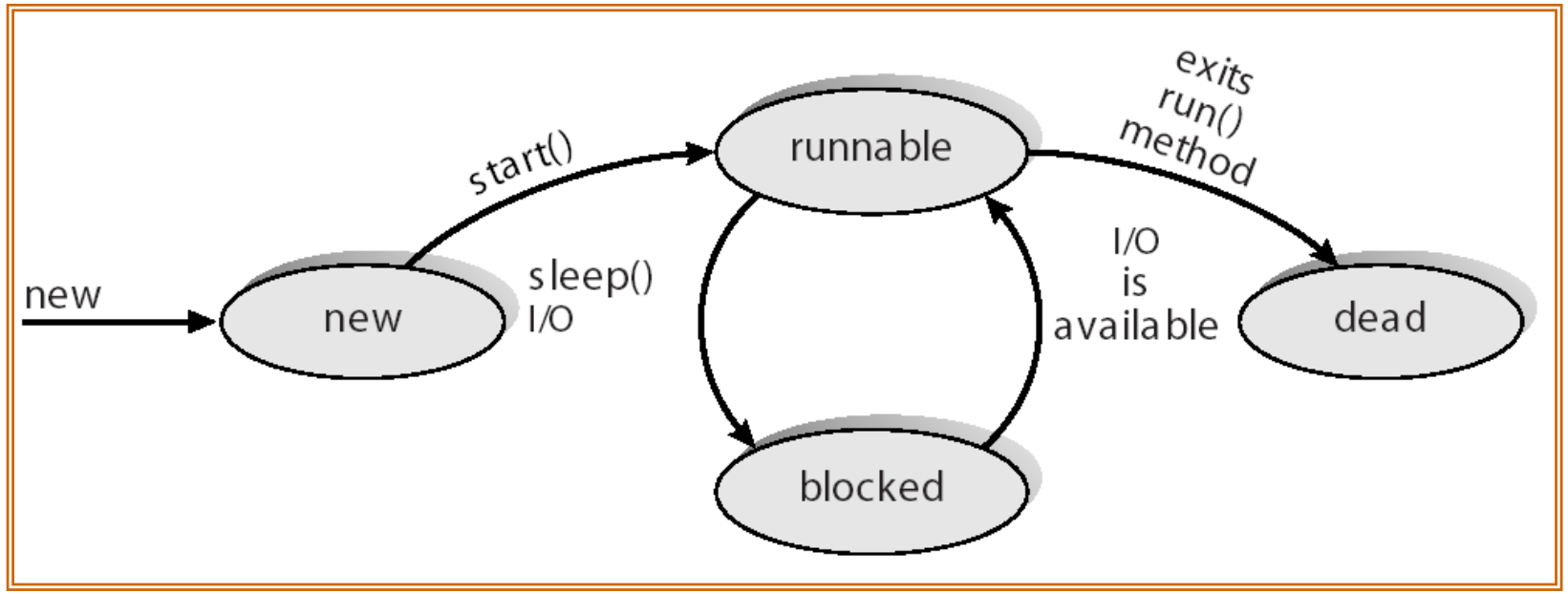
Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Java Thread States



S3 Basic Concepts of CPU Scheduling

Motivation

- CPU scheduling is the basis of multiprogrammed OSs
  - By switching the OS among processes the OS can make the computer more productive
  - Essentially, scheduling is used to determine which ready to run task will execute next
  - There are a number of scheduling policies:
    - First come, first served
    - Shortest job first
    - Priority based
    - Round robin
    - Have to decide which algorithm to select for a given system

Terminology

- On OSs that support them, it is kernel level threads (not processes) that are actually being scheduled by the OS
  - Terms *process scheduling* and *thread scheduling* are often used interchangeably
  - ***Process scheduling*** will refer to general scheduling concepts
  - ***Thread scheduling*** will refer to thread specific ideas and implementations

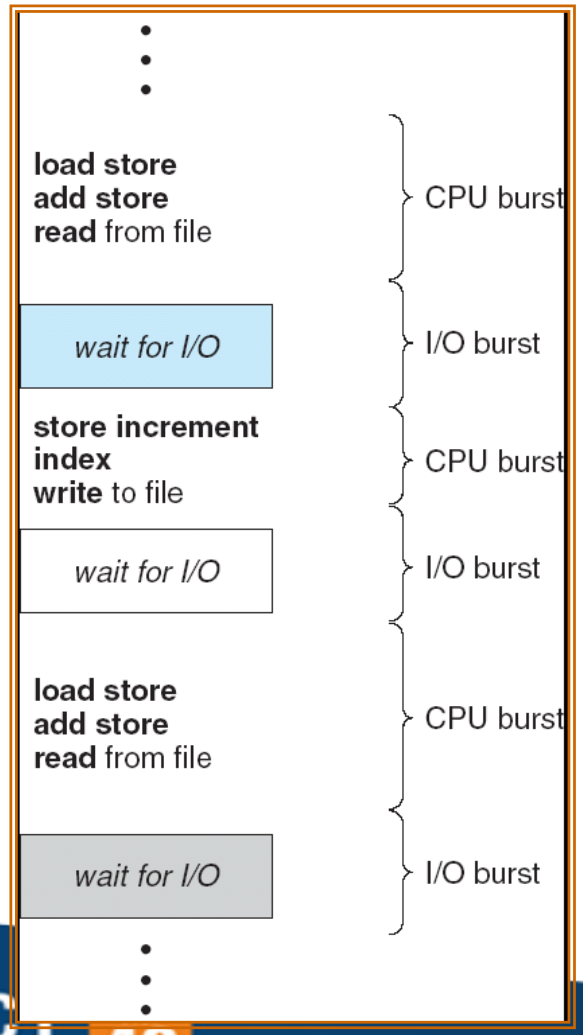
Basic Concepts

- In a single processor system only one process can run at a time
  - Others must wait until the CPU is free and they are scheduled
- Objective of multiprogramming is to have some process running at all times to maximize CPU utilization
  - Several processes are kept in memory
  - When one process has to wait, the OS takes the CPU away from that process and gives it to another process
  - Every time one process has to wait, another takes over the CPU
  - **Scheduling** determines order in which available processes are allocated the CPU

CPU/IO Burst Cycle

- Success of CPU scheduling depends on an observed property of processes
- Process execution consists of a *cycle* of CPU execution and I/O wait → **CPU-IO burst cycle**
  1. Process execution begins with a CPU burst
  2. Followed by an I/O burst
  3. And so on …
  4. Final CPU burst ends with a system request to terminate execution

Alternating Sequence of CPU And I/O Bursts



CPU Scheduler

- Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed
- **CPU scheduler (or short term scheduler)** selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- Ready queue is not necessarily a First In First Out (FIFO) queue
  - May be, or may be a:
  - Priority queue or an unordered list
- Conceptually all the processes in the ready queue are lined up waiting for a chance to run on the CPU
  - Records in the queue are generally the PCB of the processes

Non preemptive Scheduling

- CPU scheduling decisions may take place when a process:
  1. **Switches from running to waiting state (e.g. waiting for the termination of a child process)**
  2. Switches from running to ready state (e.g. when an interrupt occurs)
  3. Switches from waiting to ready (e.g. at completion of I/O)
  4. **Terminates**
- If scheduling takes place only under cases 1 and 4, the scheduling scheme is nonpreemptive
- In nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU, either by terminating or by switching to the waiting state
- All other scheduling is *preemptive*

Preemptive Scheduling

- In current OSs, preemption is the norm
- Preemptive scheduling incurs a cost associated with access to shared data
  - Need mechanisms to coordinate access to shared data → ensure that data remains in a consistent state
- Traditionally, processes running on a single processor system are nonpreemptive when they are in kernel mode
  - Wait for the system call to complete before doing a context switch

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

Scheduling Criteria (1)

- Choice of a different scheduling algorithm may favor one class of processes over another
- Many possible criteria for comparing scheduling algorithms
- The characteristics chosen make a difference in which algorithm is judged to be the best

Scheduling Criteria (2)

Criteria include:

- 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. Interval from the time of submission of a process to the time of 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** final output (for interactive system)

Optimization Criteria

CPU utilization
- Maximize

Throughput
- Maximize

Turnaround time
- Minimize

Waiting time
- Minimize

Response time
- Minimize

S4 Scheduling Algorithms: FCFS

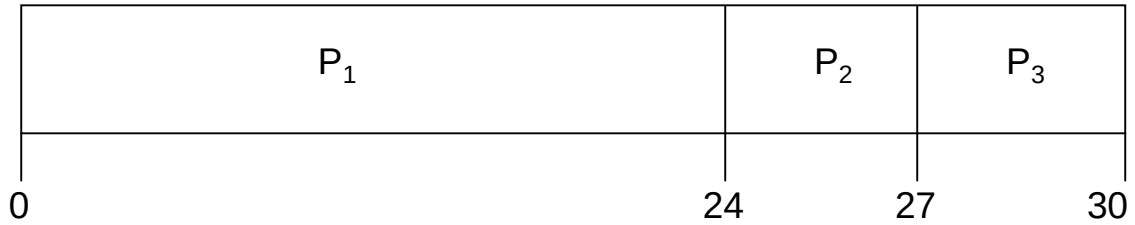
First-Come, First-Served (FCFS) Scheduling

- **Simplest CPU scheduling algorithm**
- **Process that requested the CPU first is allocated the CPU first**
- **Implementation managed by a First In First Out (FIFO) queue**
  - When a process enters the ready queue its PCB is linked onto the tail of the queue
  - When the CPU is free, it is allocated to the process at the head of the queue and the running process is removed from the queue
- **Average waiting time under FCFS policy is often quite long**
- **Nonpreemptive algorithm → once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU**

First-Come, First-Served (FCFS) Scheduling

Process	CPU Burst Time (ms)
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive (at time 0) in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



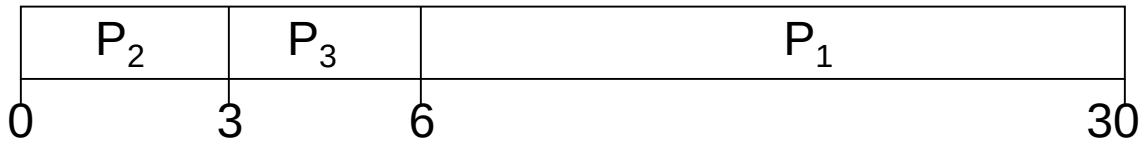
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

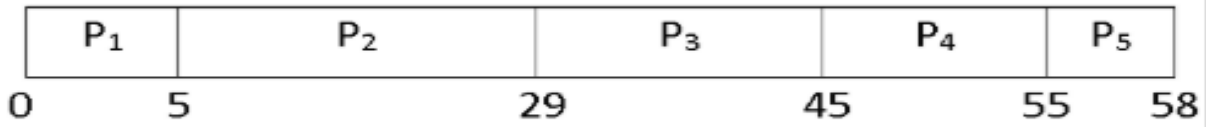


- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
  - Average waiting time varies substantially if the process's CPU burst times vary greatly
- *Convoy effect* →short process behind long process
  - I/O bound processes waiting in the ready queue until the CPU bound process is done

FCFS contd

Process	Burst Time(ms)
P_1	5
P_2	24
P_3	16
P_4	10
P_5	3

Now we calculate the average waiting time, average turnaround time and throughput.



- **Average Waiting Time**
- First of all, we have to calculate the waiting time of each process.
  Waiting Time = Starting Time - Arrival Time
  Waiting time of
  P1 = 0
  P2 = 5 - 0 = 5 ms
  P3 = 29 - 0 = 29 ms
  P4 = 45 - 0 = 45 ms
  P5 = 55 - 0 = 55 ms
  Average Waiting Time = Waiting Time of all Processes / Total Number of Process
  Therefore, average waiting time = (0 + 5 + 29 + 45 + 55) / 5 = 25 ms
- **Average Turnaround Time**
- *Turnaround Time = Waiting time in the ready queue + executing time + waiting time in waiting-queue for I/O*

- Turnaround time of
  P1 = 0 + 5 + 0 = 5ms
  P2 = 5 + 24 + 0 = 29ms
  P3 = 29 + 16 + 0 = 45ms
  P4 = 45 + 10 + 0 = 55ms
  P5 = 55 + 3 + 0 = 58ms
  Total Turnaround Time = (5 + 29 + 45 + 55 + 58)ms = 192ms
  Average Turnaround Time = (Total Turnaround Time / Total Number of Process) = (192 / 5)ms = 38.4ms

- **Throughput**
- Here, we have a total of five processes. Process P1, P2, P3, P4, and P5 takes 5ms, 24ms, 16ms, 10ms, and 3ms to execute respectively.
  Throughput = (5 + 24 +16 + 10 +3) / 5 = 11.6ms
  It means one process executes in every 11.6 ms.

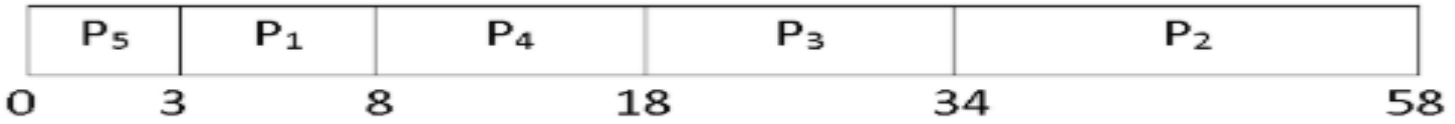
S5 Scheduling Algorithms: SJF and SRTF

Shortest-Job-First (SJF) Scheduling (1)

- Associate with each process the **length of its next CPU burst.** Use these lengths to schedule the process with the shortest time → CPU is assigned to the process that has the smallest next CPU burst
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length **less than remaining time of current executing process**, preempt. This scheme is know as the S**hortest-Remaining-Time-First (SRTF)**

SJF (Shortest Job First) Scheduling

Process	Burst Time(ms)
P_1	5
P_2	24
P_3	16
P_4	10
P_5	3



Average Waiting Time

We will apply the same formula to find average waiting time in this problem. Here arrival time is common to all processes(i.e., zero).

Waiting Time for

P1 = 3 - 0 = 3ms

P2 = 34 - 0 = 34ms

P3 = 18 - 0 = 18ms

P4 = 8 - 0 = 8ms

P5 = 0ms

Now, Average Waiting Time = (3 + 34 + 18 + 8 + 0) / 5 = 12.6ms

Average Turnaround Time

According to the SJF Gantt chart and the turnaround time formulae,

Turnaround Time of

P1 = 3 + 5 = 8ms

P2 = 34 + 24 = 58ms

P3 = 18 + 16 = 34ms

P4 = 8 + 10 = 18ms

P5 = 0 + 3 = 3ms

Therefore, Average Turnaround Time = (8 + 58 + 34 + 18 + 3) / 5 = 24.2ms

Throughput

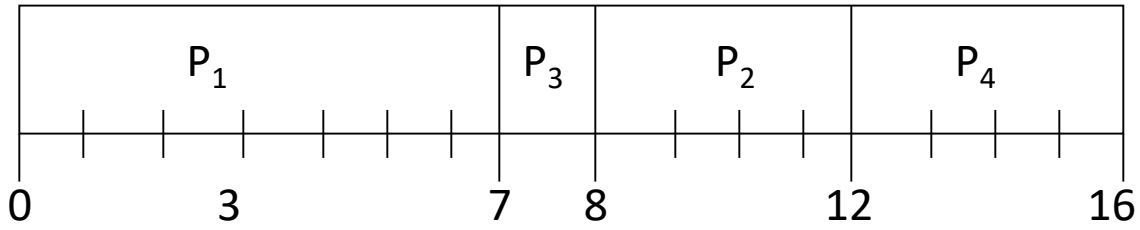
Here, we have a total of five processes. Process P1, P2, P3, P4, and P5 takes 5ms, 24ms, 16ms, 10ms, and 3ms to execute respectively.

Therefore, Throughput will be same as above problem i.e., 11.6ms for each process.

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time (ms)
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

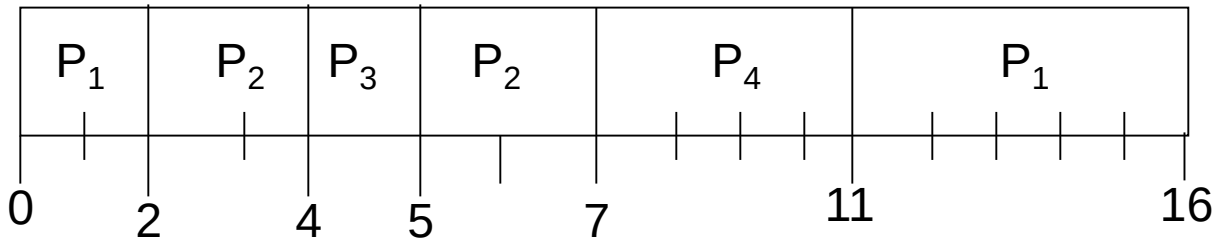


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

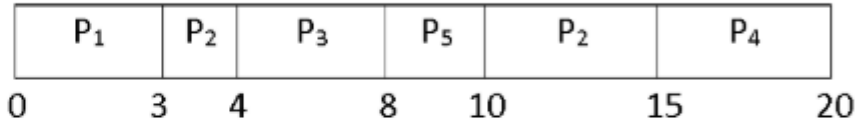


- Average waiting time = (9 + 1 + 0 +2)/4 = 3

SRTF

Shortest Remaining Time First(SRTF)

Process	Burst Time(CPU)	Arrival Time(ms)
P_1	3	0
P_2	6	2
P_3	4	4
P_4	5	6
P_5	2	8



Shortest-Job-First (SJF) Scheduling(2)

- SJF is provably optimal: – gives minimum average waiting time for a given set of processes
  - Moving a short process before a long process decreases the waiting time for the short process more than it increases the waiting time for the long process
  - Therefore average waiting time decreases
- Difficulty lies in knowing the length of the next CPU request
  - In long term scheduling (e.g. batch system) can use the length of the process time limit that a user specifies when submitting the job → SJF frequently used in long term scheduling
  - SJF cannot be implemented at the level of short term CPU scheduling because we do not know the length of the next CPU burst

S6 Scheduling Algorithms: PS

Priority Scheduling (1)

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (in this text, smallest integer ≡ highest priority)
  - Equal priority processes are scheduled using FCFS
  - Priority scheduling can be:
    - preemptive: preempt CPU if priority of newly arrived process is higher than priority of currently running process
    - nonpreemptive: put the new highest priority process at the head of the ready queue
- SJF is a priority scheduling where priority is the inverse of the predicted next CPU burst time (largest CPU burst → lowest priority)

Priority Scheduling (2)

- Problem ≡ Starvation – low priority processes is indefinitely blocked and may never execute
  - In a heavily loaded system a steady stream of higher priority processes can prevent a lower priority process from ever getting the CPU
- Solution ≡ Aging – as time progresses gradually increase the priority of processes that have been waiting for a long time

Priority

Priority Scheduling Example

Process	CPU Burst Time	Priority
P_1	6	2
P_2	12	4
P_3	1	5
P_4	3	1
P_5	4	3

Gantt Chart

P_4	P_1	P_5	P_2	P_3

Average Waiting Time

First of all, we have to find out the waiting time of each process.

Waiting Time of process

P1 = 3ms

P2 = 13ms

P3 = 25ms

P4 = 0ms

P5 = 9ms

Therefore, Average Waiting Time = (3 + 13 + 25 + 0 + 9) / 5 = 10ms

Average Turnaround Time

First finding Turnaround Time of each process.

Turnaround Time of process

P1 = (3 + 6) = 9ms

P2 = (13 + 12) = 25ms

P3 = (25 + 1) = 26ms

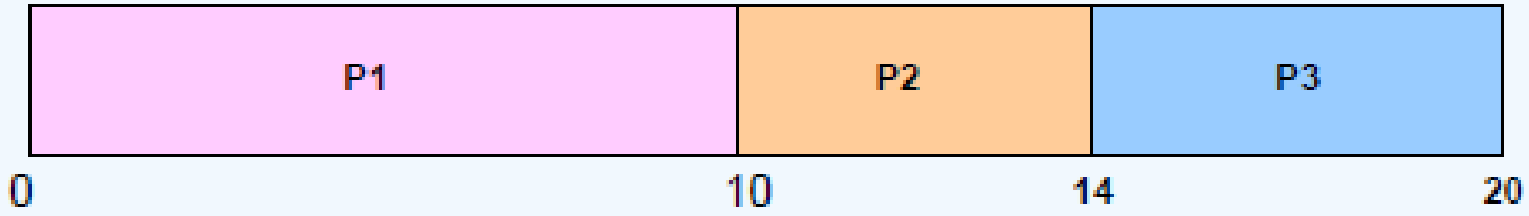
P4 = (0 + 3) = 3ms

P5 = (9 + 4) = 13ms

Therefore, Average Turnaround Time = (9 + 25 + 26 + 3 + 13) / 5 = 15.2ms

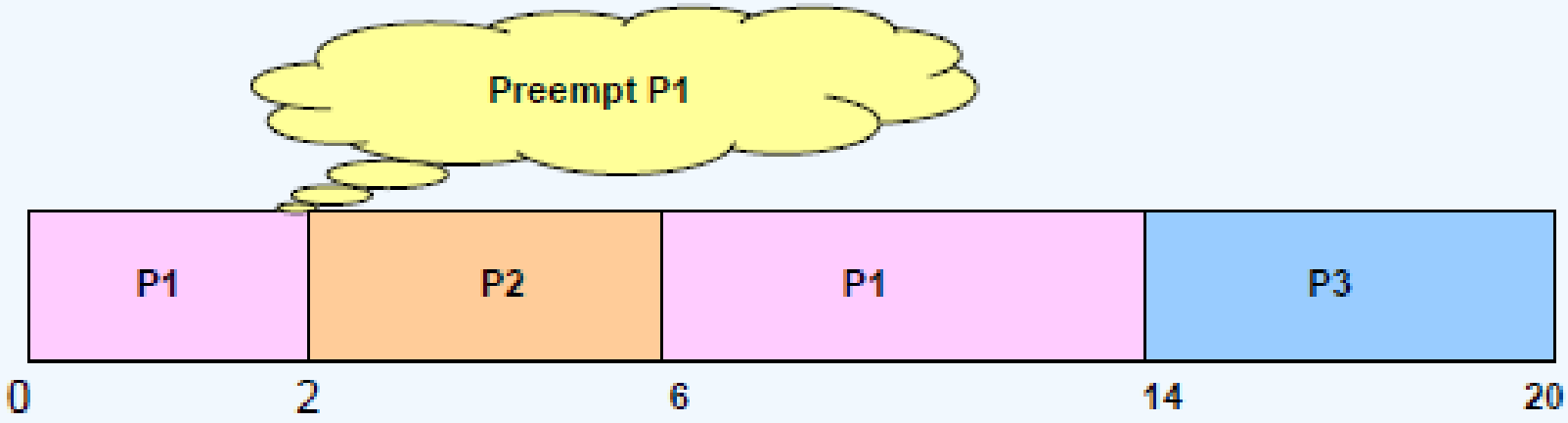
Priority Scheduling
(Non Preemptive method)

Process	Execution Time	Priority	Arrival time in msec
P1	10	2	0
P2	4	1	2
P3	6	3	0



Priority Scheduling (Preemptive method)

Process	Execution Time	Priority	Arrival time in msec
P1	10	2	0
P2	4	1	2
P3	6	3	0



S7 Scheduling Algorithms: RR

Round Robin (RR)

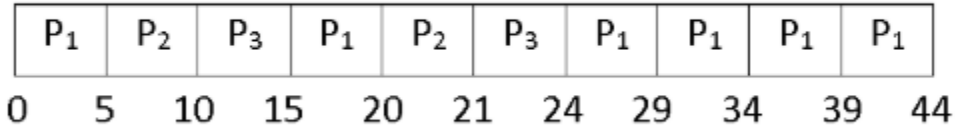
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1) x *q* time units until its next quantum.
- Performance depends on size of q
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ RR approach is called processor sharing and creates appearance that each of the *n* processes has its own processor running at speed 1/n
  - q must be large with respect to context switch, otherwise overhead is too high

Round Robin

time quantum 5ms

Process	CPU Burst Time
P_1	30
P_2	6
P_3	8

Gantt Chart



Average Waiting Time

For finding Average Waiting Time, we have to find out the waiting time of each process.

Waiting Time of

P1 = 0 + (15 - 5) + (24 - 20) = 14ms

P2 = 5 + (20 - 10) = 15ms

P3 = 10 + (21 - 15) = 16ms

Therefore, Average Waiting Time = (14 + 15 + 16) / 3 = 15ms

Average Turnaround Time

Same concept for finding the Turnaround Time.

Turnaround Time of

P1 = 14 + 30 = 44ms

P2 = 15 + 6 = 21ms

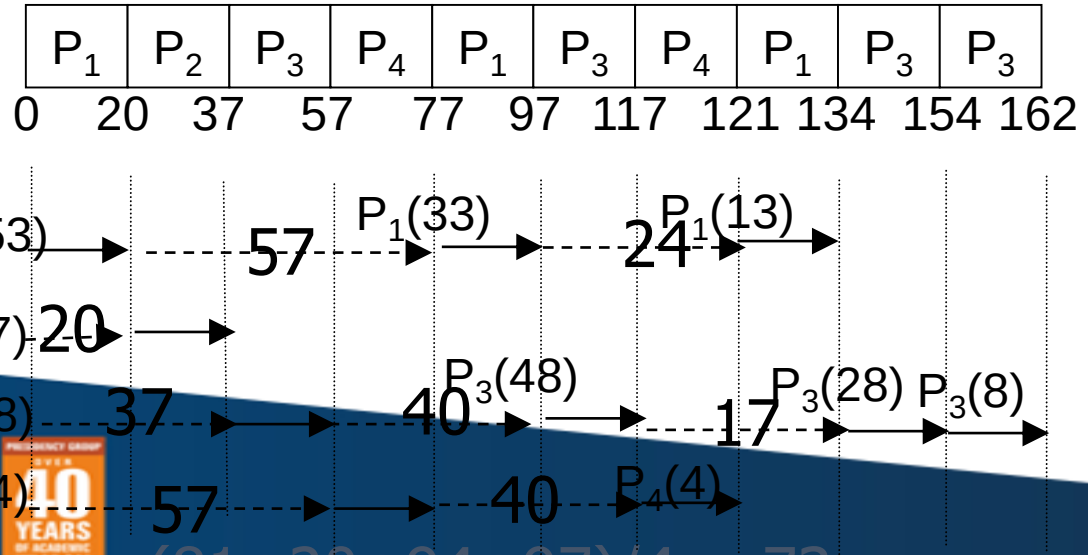
P3 = 16 + 8 = 24ms

Therefore, Average Turnaround Time = (44 + 21 + 24) / 3 = 29.66ms

Round Robin Scheduling

- Each process is given CPU time in turn, (i.e. time quantum: usually 10-100 milliseconds), and thus waits no longer than (n – 1) * time quantum
- time quantum = 20

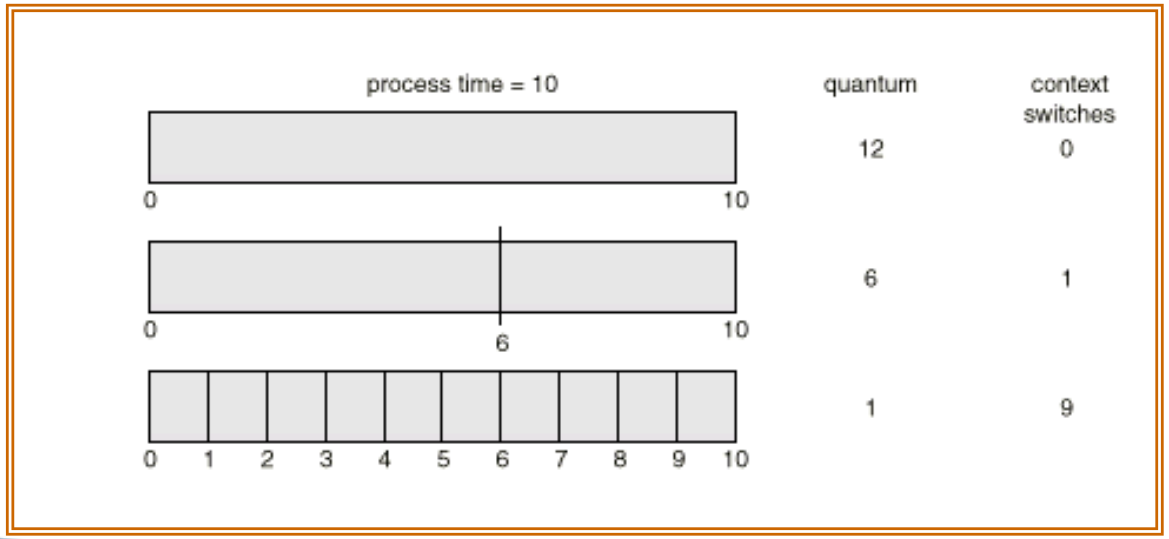
Process	Burst Time	Wait Time
P_1	53	57 +24 = 81
P_2	17	20
P_3	68	37 + 40 + 17= 94
P_4	24	57 + 40 = 97



Average wait time = (81+20+94+97)/4 = 73

Round Robin Scheduling

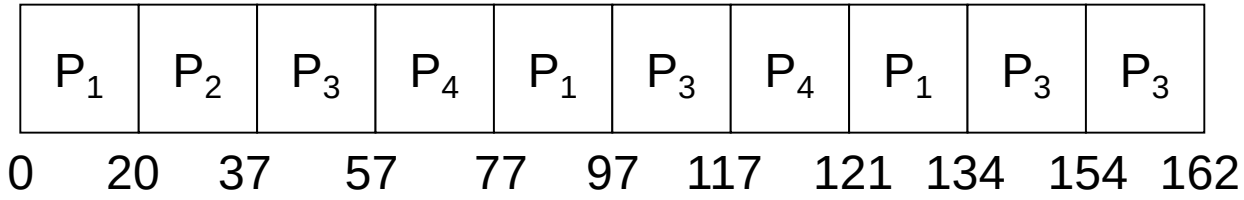
- Typically, higher average turnaround than SJF, but better *response*.
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.



Example of RR with Time Quantum = 20

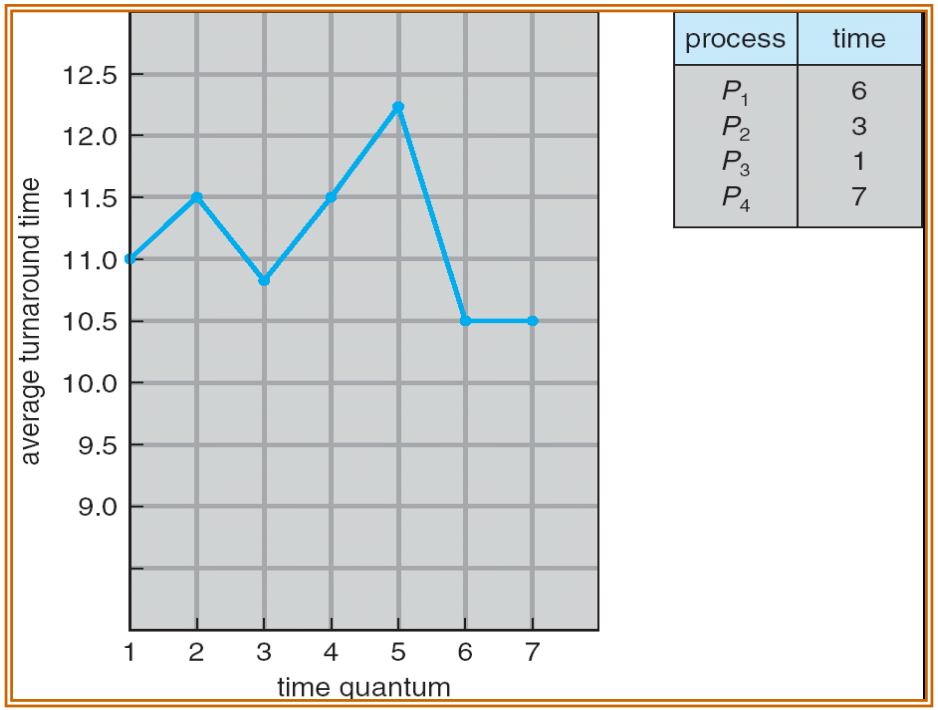
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



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

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Average turnaround time does not necessarily improve as size of q increases

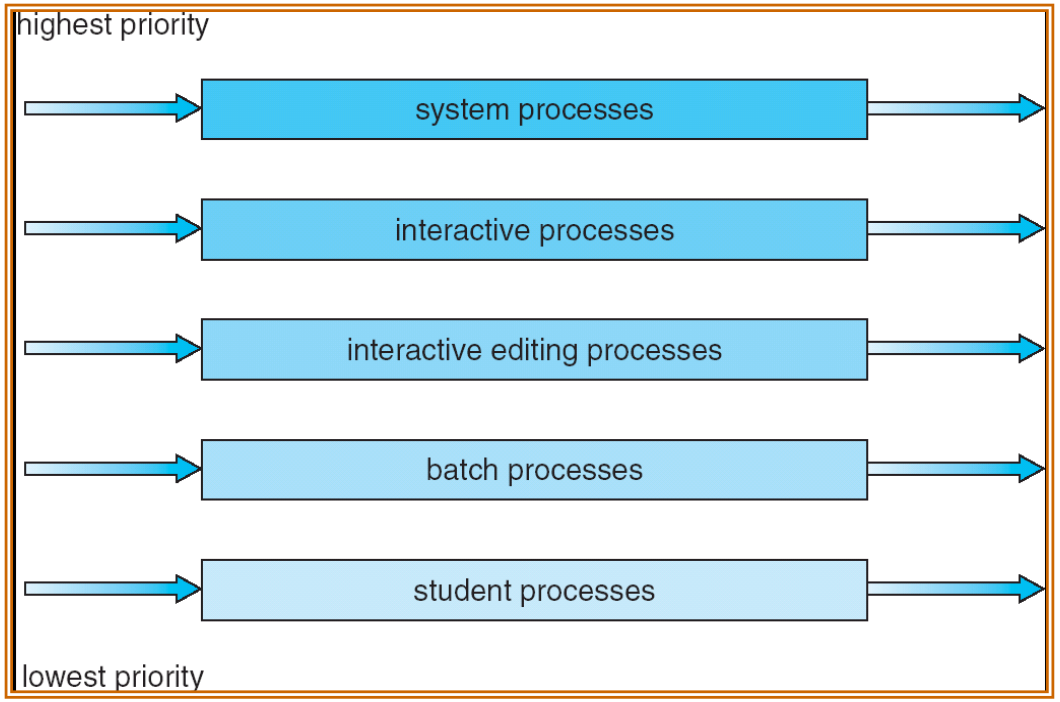
Generally, average turnaround time can be improved if most processes finish their CPU burst in a single time quantum

S8 Multi-level Queue with and without feed-back.

Multilevel Queue

- Ready queue is partitioned into several separate queues:
  - foreground (interactive processes)
  - background (batch processes)
- Processes have different response time requirements, therefore each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Also, scheduling must be done between the queues
  - Commonly implemented as fixed priority preemptive scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Another option: time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes;
    - 80% to foreground in RR
    - 20% to background in FCFS

Multilevel Queue Scheduling



Each queue has absolute priority over lower-priority queues.

No process in batch queue could run unless queues for higher priority processes were empty. If a higher priority process entered queue while batch process was running, it would be preempted.

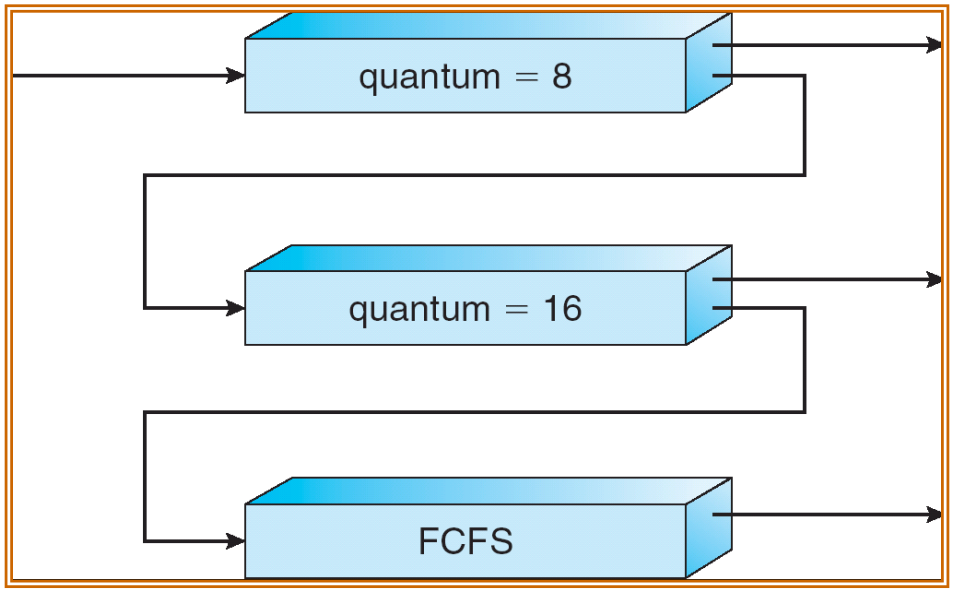
Multilevel Feedback Queue

- When processes are permanently assigned to a queue there is low scheduling overhead but it is inflexible
- In multilevel feedback queues a process can move between the various queues
- Idea: separate processes according to the characteristics of their CPU bursts
  - If a process uses too much CPU time it will be moved to a lower-priority queue
  - If a process waits too long in a lower priority queue it will be moved to a higher priority queue → this prevents starvation

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR with time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - 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.
  - Processes in Q_2 are run on a FCFS basis but only when queues 0 and 1 are empty

Multilevel Feedback Queues (1)



Scheduling algorithm gives highest priority to any process with a CPU burst of 8ms or less.

Processes that need more than 8ms but less than 24ms are also served quickly.

Long processes automatically sink to queue 2

Multilevel Feedback Queue (2)

- A multilevel feedback-queue scheduler is 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 (1)

- CPU scheduling more complex when multiple CPUs are available
- *Load sharing* becomes possible → scheduling becomes more complex
- Considering h*omogeneous processors* only
  - We can use any available processor to run any process in the queue

Multiple-Processor Scheduling (2)

- 2 approaches to CPU scheduling in a MP system

1. *Asymmetric multiprocessing* – all scheduling decisions, I/O processing, and other system activities handled by a single processor → other processors execute user code
   - Only one processor accesses the system data structures, alleviating the need for data sharing
2. Symmetric multiprocessing (SMP) – where each process is self scheduling
   - The scheduler for each processor examines the ready queue and selects a process to execute
   - Load balancing attempts to keep the workload evenly distributed across all the processors
   - Virtually all modern OSs support SMP

Thank You