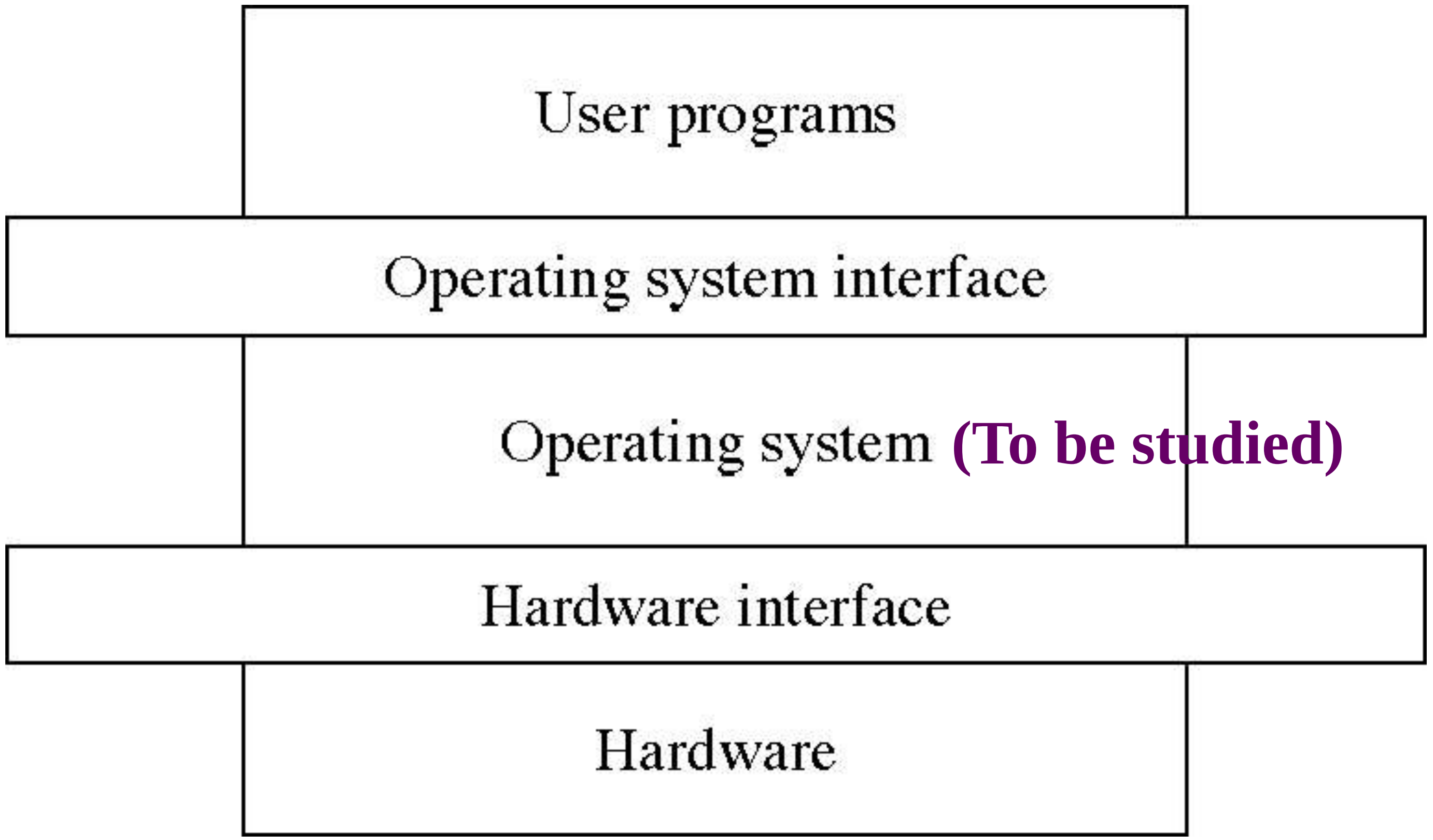
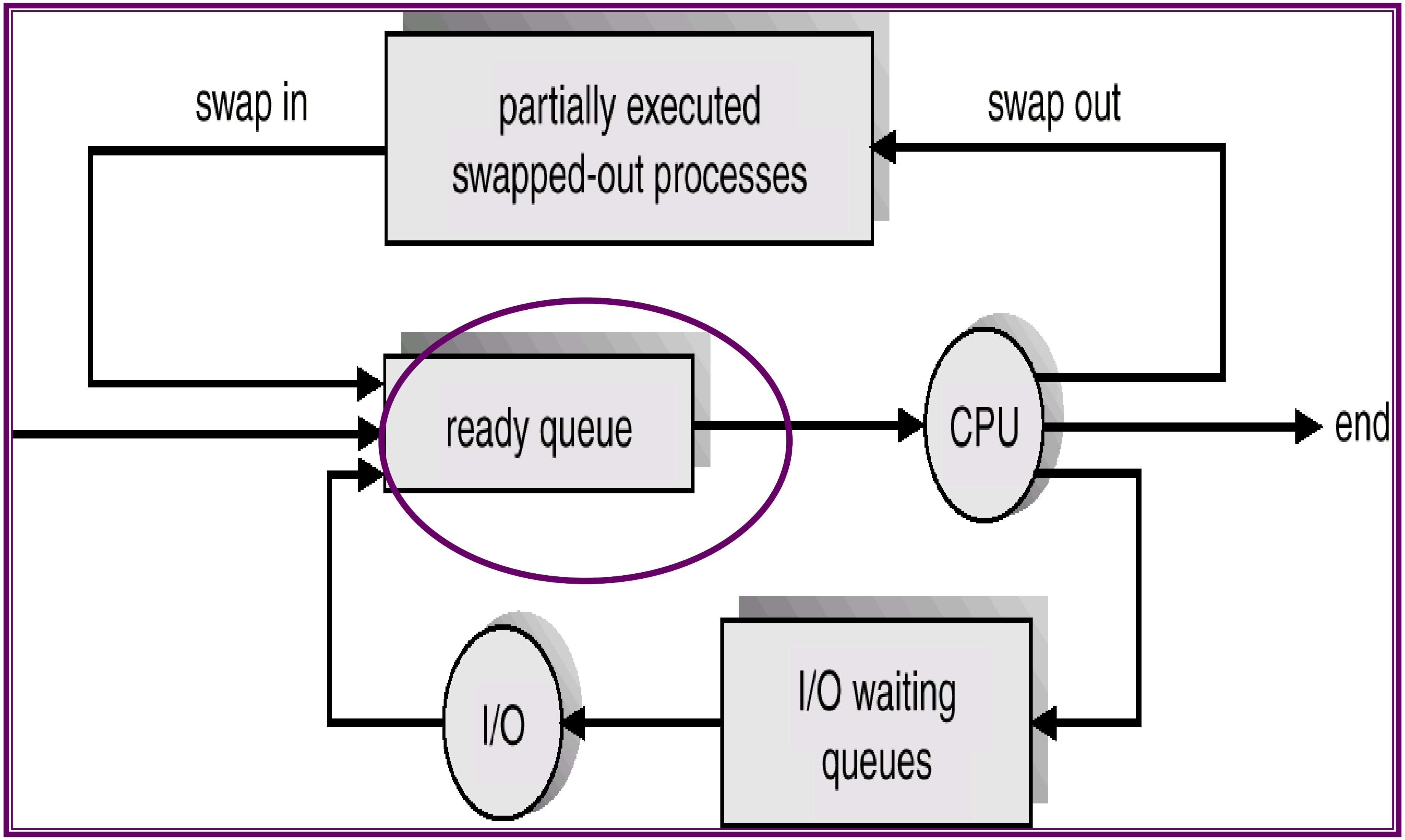

User programs
Operating system interface
Operating system (To be studied)
Hardware interface
Hardware

Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real-Time Scheduling
- Algorithm Evaluation
- Process Scheduling Models

Scheduling

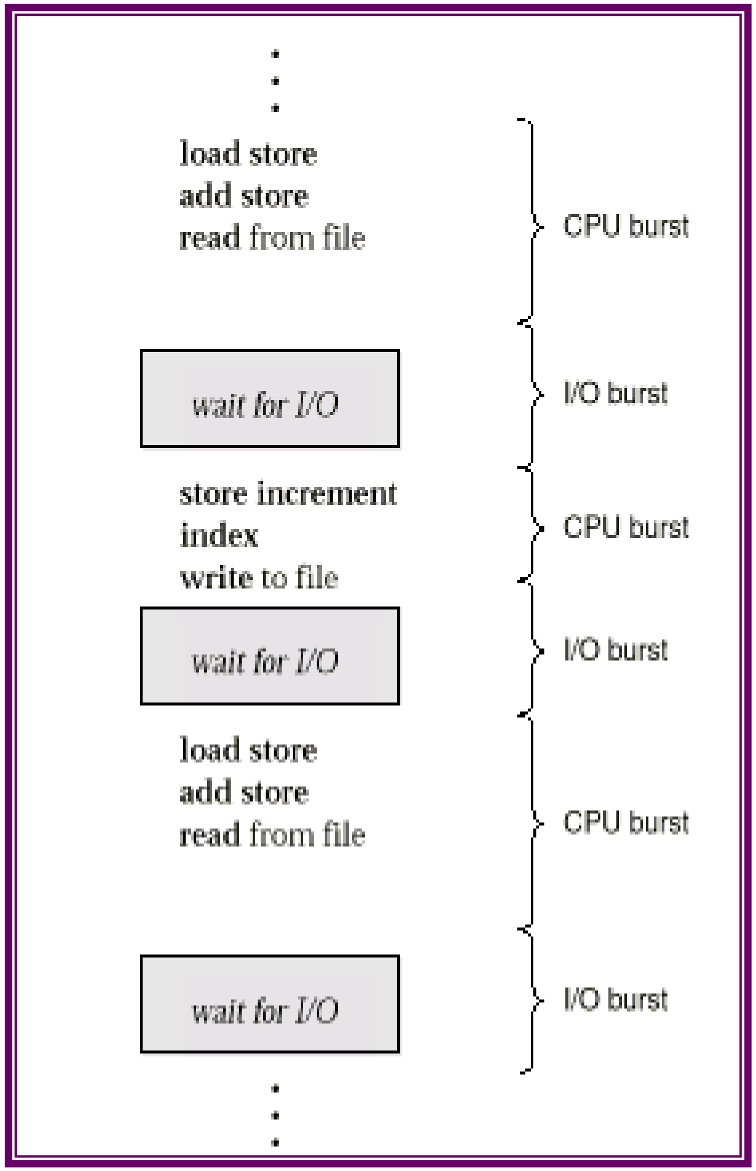

swap in
partially executed swapped-out processes
swap out
ready queue
CPU
end
I/O
I/O waiting queues

Basic Concepts

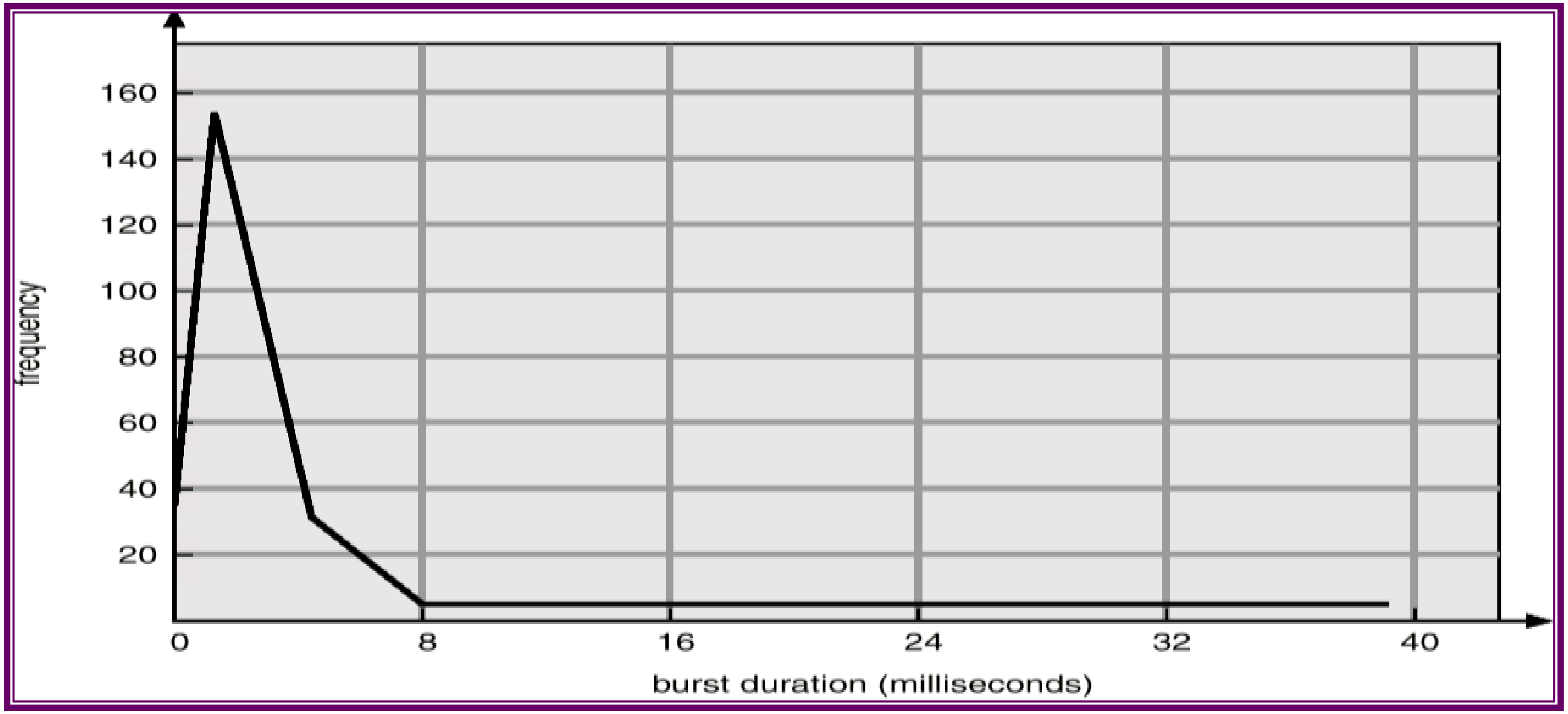
- CPU scheduling is the basis of multi-programmed OS.
- In the uni-processor system only one process runs at a time. Other processes must wait until CPU is free and can be rescheduled.
- Idea: the process is executed until it must wait for completion of I/O request.
- When one process has to wait, OS takes CPU away from that process and gives the CPU to another process.
- All the computer resources are schedules before use.
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.
- CPU burst distribution

Observed property

- Process execution consists of a cycle of execution and I/O wait.
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.
- Process execution starts with CPU-burst. That is followed by I/O burst.
- The execution ends with CPU burst.
- CPU burst distribution



Histogram of CPU-burst Times



Many short CPU bursts and a few long CPU bursts.

CPU and I/O bound processes

- An I/O bound program would have many short CPU bursts.
- A CPU bound program would have long CPU bursts.

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- Short-term scheduler
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (due to I/O).
  2. Switches from running to ready state (due to interrupt)
  3. Switches from waiting to ready. (Completion of I/O)
  4. Terminates.
- Scheduling under 1 and 4 is *non-preemptive*.
  - Once a 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.
    - WINDOWS 3.1 and MAC
- All other scheduling is *preemptive.*
  - The process can be preempted and another process can be run.
    - UNIX
    - Difficult to implement.
    - (Refer the book for the issues)

Designing Preemptive Scheduling based OS is difficult

- It incurs cost to access shared data
- Affects the design of OS
  - ☞Interrupts occcur at arbitrary time, changing of data modified by kernel
- Section of code modified by interrupts should be guarded
- But, all modern OSs are preemptive scheduling due to performance advantages as compared to non- preemtive 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.

Scheduling Criteria

- Scheduling algorithm favors one process to another.
- User oriented criteria
  - **Turnaround time** – amount of time to execute a particular process
    - The interval from the time of submission of a process to the time of completion.
    - The sum of periods spent in waiting to get into memory, waiting in the ready queue, executing on CPU and doing I/O
  - **Response time** – amount of time it takes from when a request was submitted until the first response is produced, **not** FINAL/LAST OUTPUT (for time-sharing environment)
  - **Deadlines:** When the process completion deadlines can be specified, the scheduling discipline should subordinate their goals to that of maximizing the percentage of deadlines met.

Scheduling Criteria

- System oriented
  - **Throughput** – # of processes that complete their execution per time unit
  - **CPU utilization** – keep the CPU as busy as possible; percentage of time the processor is busy.
    - It may range from 0 to 100 percent.
    - In a real system, it should range from 40 percent (lightly loaded) to 90 percent (heavily loaded)
  - **Waiting time:** It is the sum of the periods spent waiting in the ready queue.
- **System oriented: other**
  - **Fairness:** In the absence of guidance from user or other system supplied guidance, processes should be treated the same, and no process should suffer starvation.
  - **Enforcing priorities**: When processes are assigned priorities, the scheduling policy should favor higher priority processes.
  - **Balancing resources:** The scheduling policy should keep the resources of the system busy. Processes that underutilize the stressed resources should be favored which involves long term scheduling.

Optimization Criteria

- Maximize
  - CPU utilization
  - Throughput
- Minimize
  - turnaround time
  - waiting time
  - response time
    - (Some suggest variance of response time)
  - **Power consumption (Mobile)**

Scheduling Algorithms

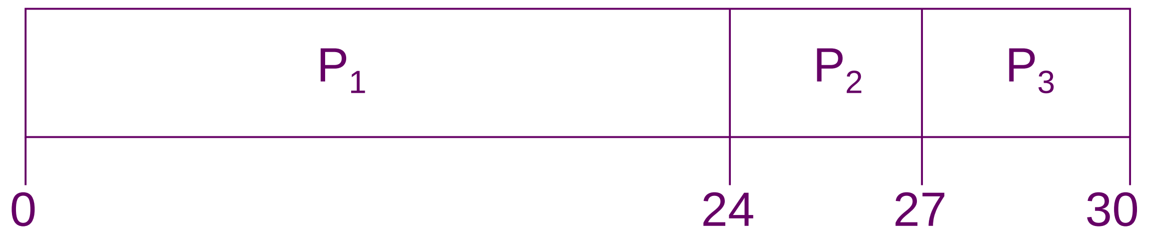
- FIFO
- SJF
- Priority
- Round Robin
- Multilevel
- Multilevel feedback

First-Come, First-Served (FCFS) Scheduling

- The process that requests CPU first is allocated the CPU first.

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:



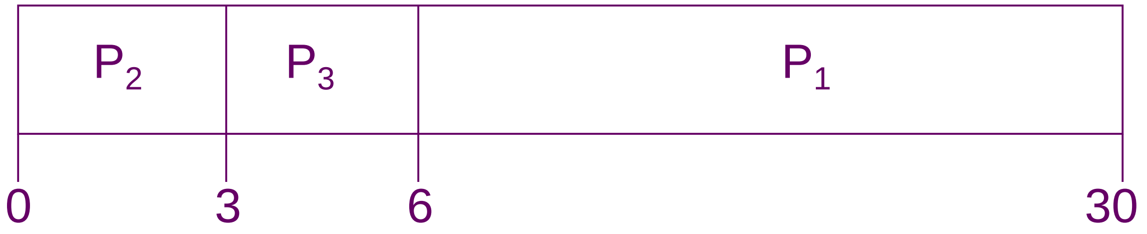
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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.
- *Dynamic environment*
  - *Convoy effect:* short process behind long process

FCFS Scheduling (Cont.)

- Scenarios
  - One CPU bound process and many I/O bound process
  - CPU bund process will hold the CPU for a long time
  - All I/O processes are in ready queue
    - The I/O devices are idle
  - This is a convoy effect
    - Several processes wait for one big process.
- FCFS algorithm is non-preemptive
- Positive Points
  - code is simple to write and understand
- Negative points
  - It is not good for timesharing systems
  - Average waiting time may be too long.
  - Convoy effect results in lower CPU and device utilization.
  - Penalizes short processes; penalizes I/O bound processes
  - Response time may be high, especially there is a large variance in process execution times.

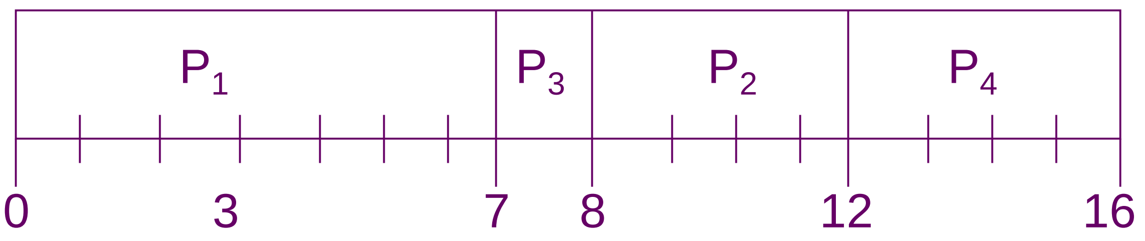
Shortest-Job-First (SJF) Scheduling

- When CPU is available it is assigned to the process that has the smallest next CPU burst.
- If the bursts are equal, FCFS is used to break the tie.
- Associate with each process the length of its **next CPU burst**. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - **Non-preemptive** – once CPU is given to the process it cannot be preempted until it 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 Shortest-Remaining-Time-First (SRTF).
- **SJF is optimal** – gives minimum average waiting time for a given set of processes.

Example of Non-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 (non-preemptive)

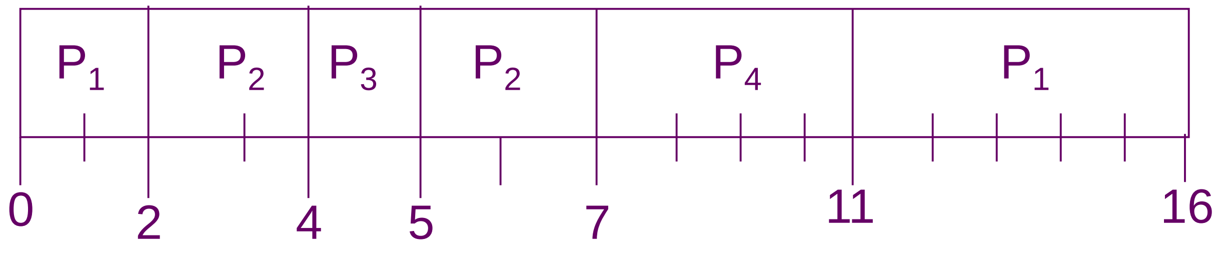


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

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

Determining Length of Next CPU Burst

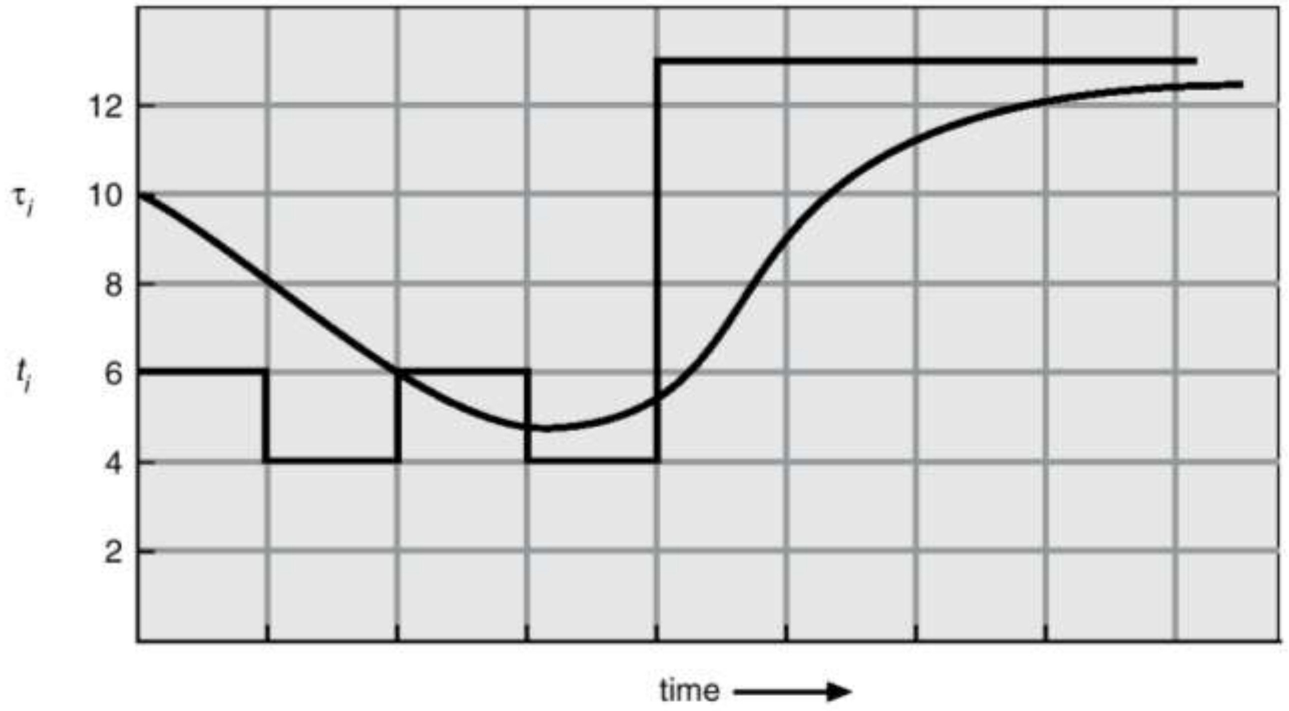
- Difficulty: Knowing the next CPU burst of each process.
- Solution: approximation.
- Simplest approximation: running average of each “burst” for each process.
  - $\tau_{n+1} = 1/n \sum t_i$ (i=1 to n)
  - Where, t_i processor execution time for the i’th instance of this process (processor execution time for batch job, processor burst time for interactive job)
  - τ_i; predicted value for the ith instance
  - $\tau 1$;predicted value for the first instance (not calculated)
- To avoid recalculating the entire summation each time we can rewrite $\tau_{n+1} = 1/n\ t_n + (n-1)/n\ \tau_n$
- The above formula gives equal weight to each instance
- Normally most recent references likely reflect future behavior
- Common technique is exponential averaging.

Exponential averaging

1. $t_n =$ actual lenght 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 :

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

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	. . .

Examples of Exponential Averaging

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

- $\alpha = 0$
  - ☞ $\tau_{n+1} = \tau_n$; Recent history does not count.
  - ☞ If $\alpha \rightarrow 0$; greater weight is given to past values.
  - ☞ If $\alpha \rightarrow 1$; greater weight is given to recent values.
- $\alpha = 1$
  - ☞ $\tau_{n+1} = t_n$
  - ☞ Only the actual last CPU burst counts.
- If we expand the formula, we get:

  $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots$
  $+ (1 - \alpha)^j\, \alpha\, t_{n-i} + \ldots$
  $+ (1 - \alpha)^{n+1}\, \tau_0$
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.
- If $\alpha = 0.8$, then $\tau_{n+1} = 0.8\, t_n + 0.16\, t_{n-1} + 0.032\; t_{n-2} + \ldots$

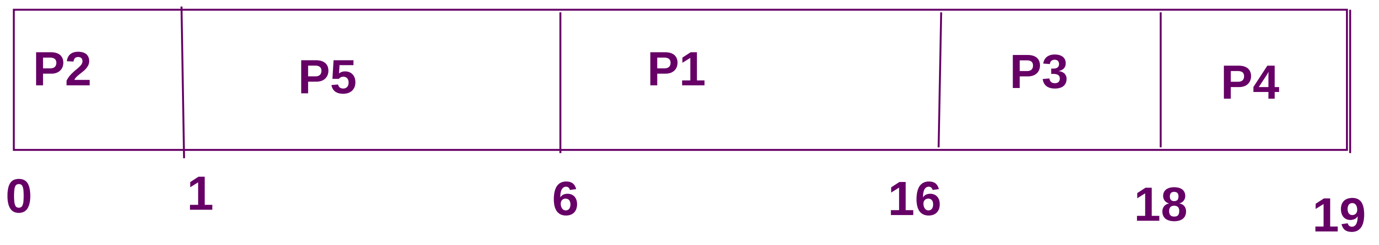
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
- Equal priority process is scheduled in FCFS order.
- SJF is a priority scheduling where priority is the predicted next CPU burst time.

Example: Priority scheduling

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

- Priority
- Average waiting time = 8.2 msec



Priority scheduling…

- Priority scheduling can be either **preemptive or non-preemptive.**
- **A preemptive priority scheduling algorithm** will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- **A non-preemptive priority** scheduling algorithm will simply put the new process at the head of the ready queue.

Priority scheduling: Problem

- **Problem ≡ Starvation** – low priority processes may never execute.
- In an heavily loaded system, a stream of high priority processes can prevent low priority process from ever getting the CPU.
- **Solution ≡ Aging** – as time progresses increase the priority of the process.
  - ☞Ex: if priorities range from 0 (low)-127 (high), every 15 minutes we can increment the priority of the waiting process.

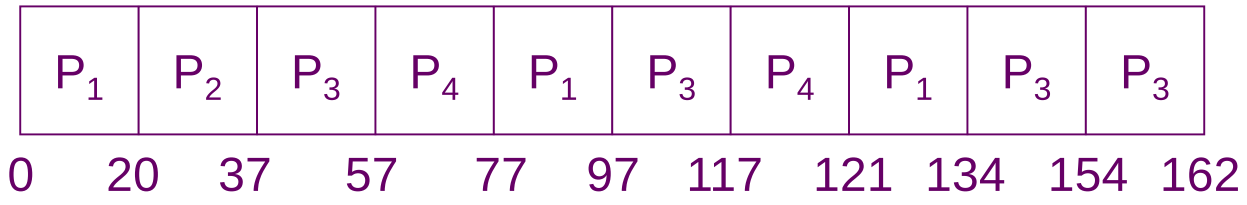
Round Robin (RR)

- 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.
- The ready queue is treated as a circular queue.
  - ☞ If the CPU burst is less than 1 time quantum, the process leaves the system.
  - ☞ If the CPU burst is greater than the time quantum, interrupt occurs and context switch will be executed.
- 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.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

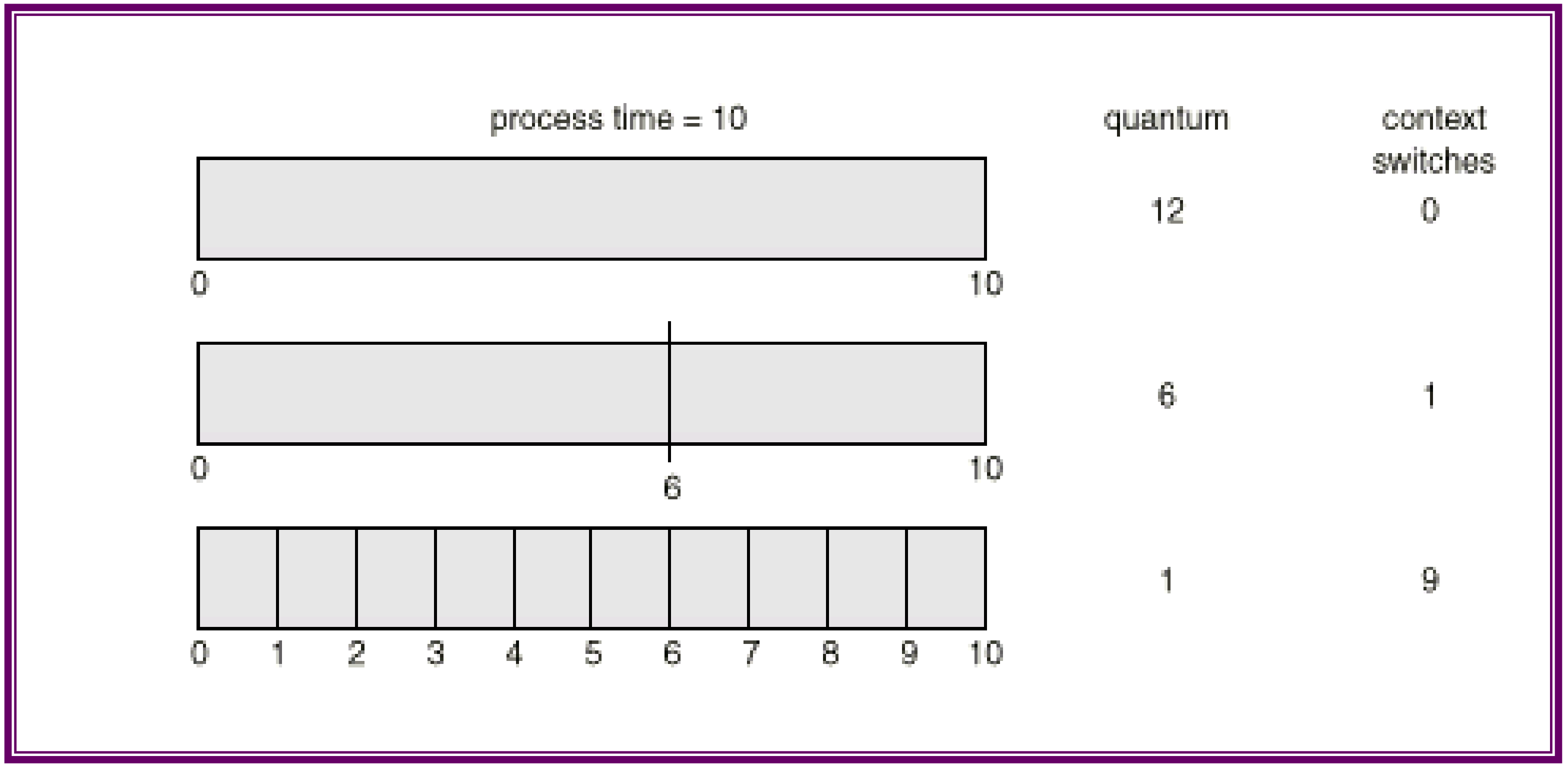
- The Gantt chart is:



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

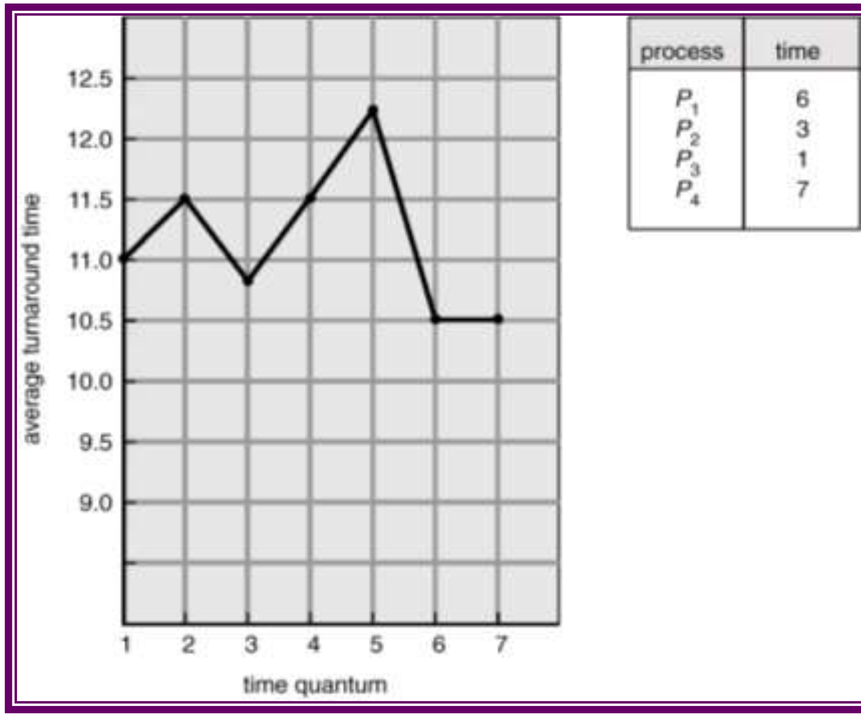
Round Robin Performance

- Performance depends on the size of the time quantum.
- If the time quantum is large, RR policy is the same as the FCFS policy.
- If the time quantum is too small, there will be more number of context switches.



RR performance..

- If the context switch time is 10 % of time quantum, then 10 % of CPU time will be spent on the context switch.
- However, if we increase the time quantum the average turnaround time may not be improved.
- In general the average turnaround time will be improved if process finishes next CPU burst within single time quantum.
- Performance Summary:
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ more number of context switches.
  - *q* must be large with respect to context switch, otherwise overhead is too high.
- **Rule of thumb: 80 % of the CPU bursts should be shorter than the time quantum.**

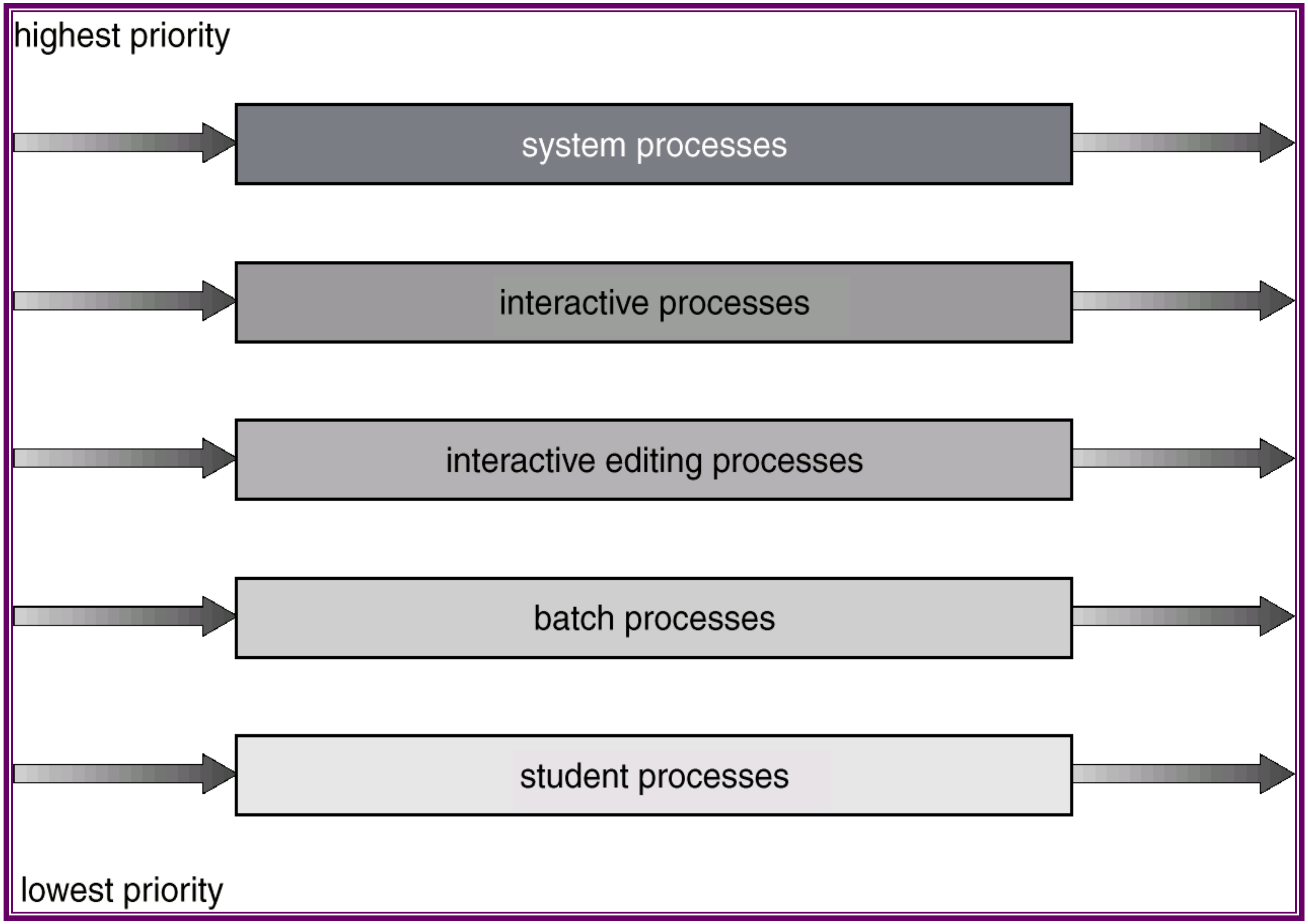


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into several separate queues: Example: foreground (interactive); background (batch)
  - Foreground processes may have priority over background processes.
- 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.
  - 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 Queue Scheduling



Multilevel Queue

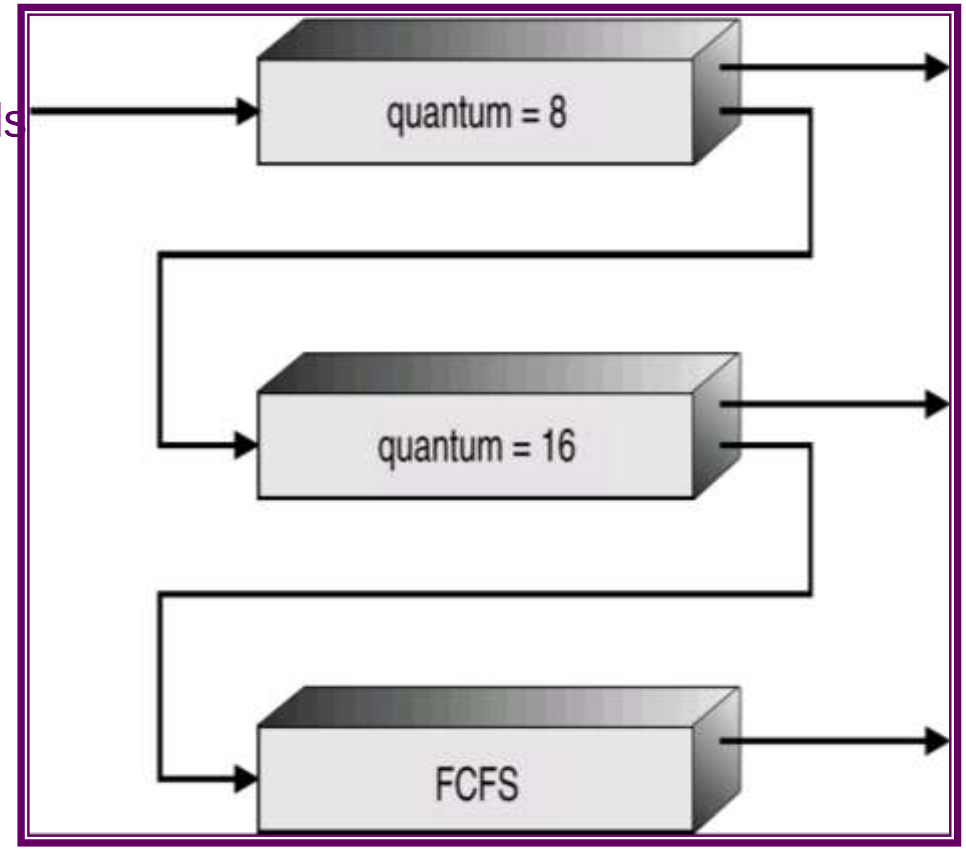
- Each queue has absolute priority over lower-priority queues.
  - No process in the batch queue can run unless the above queues are empty.
  - If the interactive process enters while the batch process is running, the batch process would be preempted.
  - Solaris 2 uses a form of this algorithm.
- Another policy is time slice:
  - Each queue gets certain portion of CPU time.
    - The foreground queue may receive 80% of CPU time, and background queue receives 20% of CPU time.
- Aging: A process can move between the various queues; aging can be implemented this way.

Multilevel Feedback Queue Scheduling

- Process moving is allowed
- The basic idea is to separate processes with different CPU-burst characteristics.
  - If a process uses too much CPU time, it will be demoted to lower priority queue.
  - If a process waits too long, it will be promoted to higher priority queue.
    - Aging prevents starvation

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – 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.
- The process that arrives in Q_0 will preempt a process in Q_2.
- Highest priority is given to the process of having 8 msec CPU burst.



Multilevel Feedback Queue Scheduling

- 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
- Multi-level is a general scheduling algorithm
- It can be configured to match a specific system under design.
  - It requires some means of selecting the values for all the parameters.

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 **process-contention scope (PCS)** since scheduling competition is within the process
  - 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 Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Multiple-Processor Scheduling

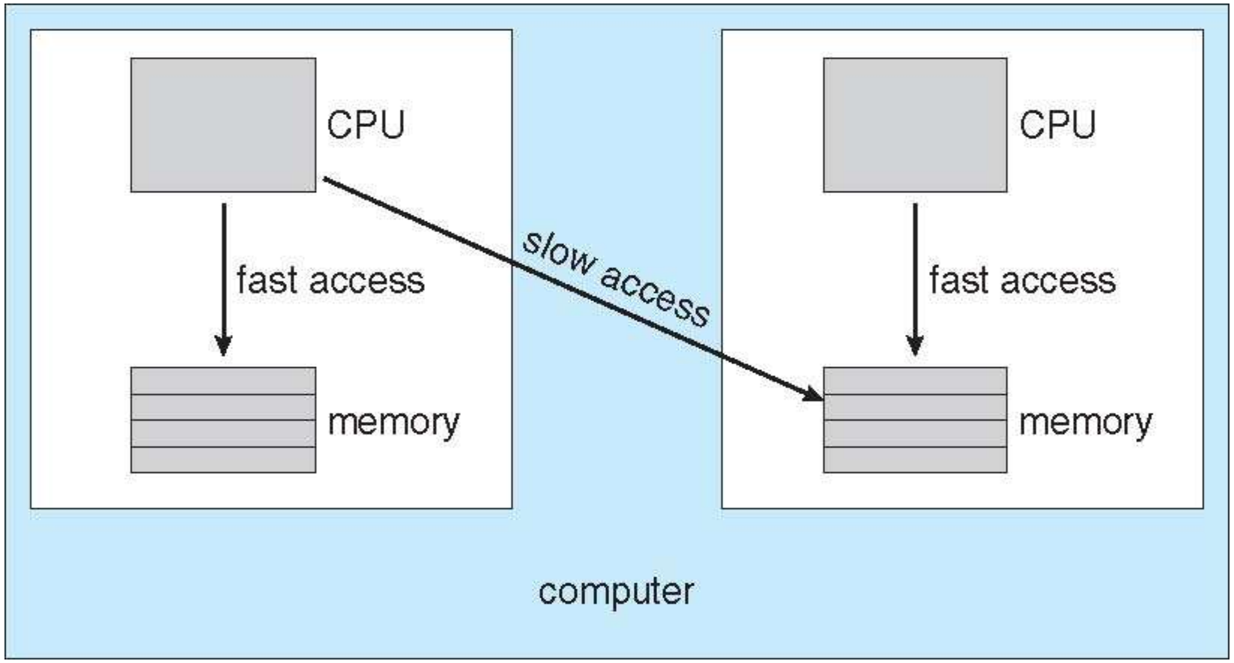
- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
  - The processors are identical
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
    - Master server,
 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

Multiple-Processor Scheduling: Issues

- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity: Efforts will be made to run the process on the same CPU, but not guaranteed.**
  - **hard affinity: Process do not migrate among the processors.**
  - Variations including **processor sets**
    - **Processor set is assigned to a process. It can run on any processor.**

NUMA and CPU Scheduling

NUMA: Non-Uniform Memory Access : CPU has faster access to some parts of main memory than to other parts. It occurs in the systems with combined CPU and memory boards.



Note that memory-placement algorithms can also consider affinity

Multiprocessor Scheduling: Load Balancing

- Load balancing attempts to keep the workload evenly distributed across all the processors in an SMP system.
- Push migration and pull migration
  - Push migration
    - A specific process periodically checks the load on each processor and evenly distributes the processes.
  - Pull migration
    - Idle processor pulls a waiting task from a busy processor.

Multiprocessor Scheduling: Load Balancing

- Load balancing attempts to keep the workload evenly distributed across all the processors in an SMP system.
- Push migration and pull migration
  - Push migration
    - A specific process periodically checks the load on each processor and evenly distributes the processes.
  - Pull migration
    - Idle processor pulls a waiting task from a busy processor.

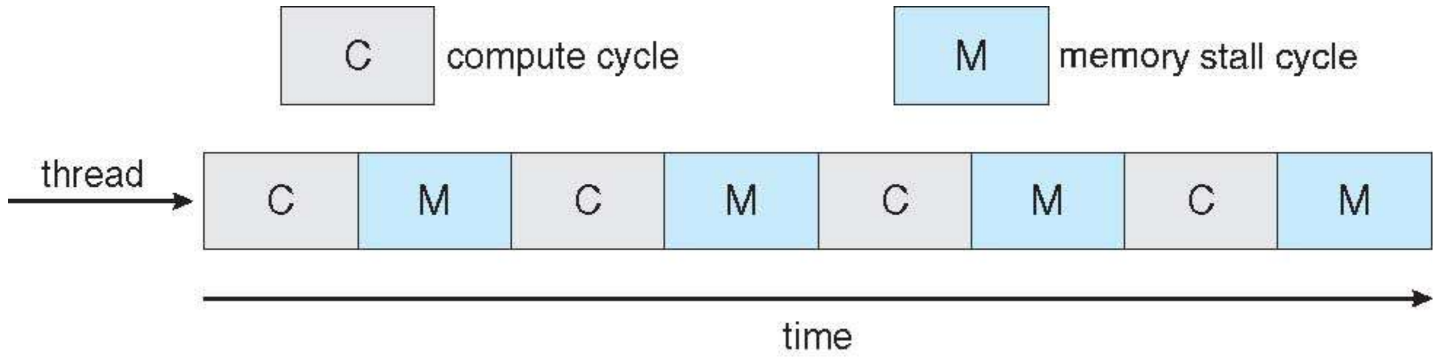
Multiprocessor Scheduling: Load Balancing

- Load balancing attempts to keep the workload evenly distributed across all the processors in an SMP system.
- Push migration and pull migration
  - Push migration
    - A specific process periodically checks the load on each processor and evenly distributes the processes.
  - Pull migration
    - Idle processor pulls a waiting task from a busy processor.

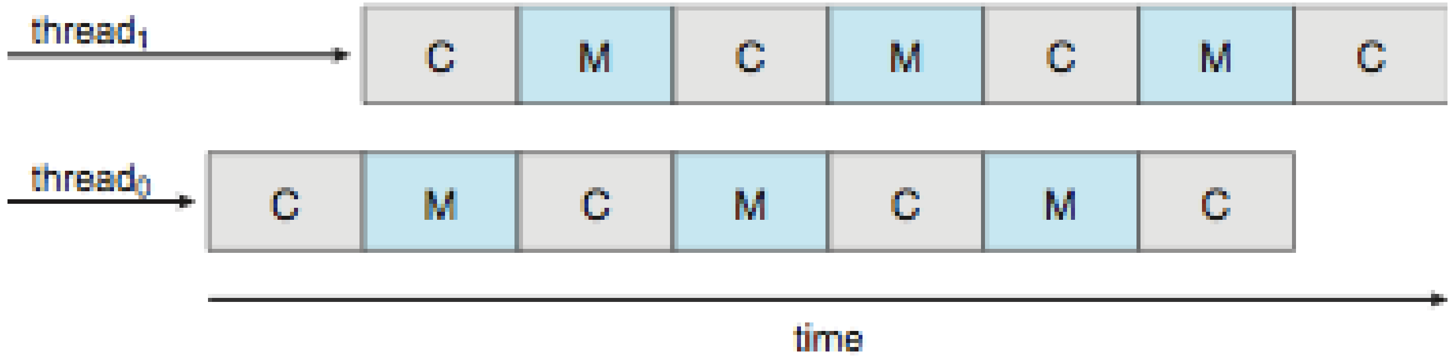
Multicore Processors

- Recent trend is to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Memory stall: When a processor accesses main memory, it spends significant amount of time in waiting.
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



- Two threads are associated with one core



- One core may have many logical processors
- A dual core may have four logical processors

Out-of-order execution

- **out-of-order execution** (or more formally **dynamic execution**) is a paradigm used in most high-performance central processing units to make use of instruction cycles that would otherwise be wasted.
- In this paradigm, a processor executes instructions in an order governed by the availability of input data and execution units, rather than by their original order in a program.
- In doing so, the processor can avoid being idle while waiting for the preceding instruction to complete and can, in the meantime, process the next instructions that are able to run immediately and independently.
- Reference: https://en.wikipedia.org/wiki/Out-of-order_execution

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)

- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests

- Can undo good scheduling algorithm efforts of guests

Algorithm Evaluation

- How to select CPU scheduling algorithm ?
- First problem is selection off criteria
  - CPU utilization, response time, or throughput
- To select an algorithm we have to select relative importance of these measures.
  - Maximize CPU utilization under the constraint that maximum response time is one second.
  - Maximize throughput such that turnaround time is linearly proportional to total execution time.
- After selecting the criteria various algorithms have to be evaluated.
- The following methods are followed for evaluation
  - Analytical modeling
    - Deterministic modeling
    - Queuing models
  - Simulation
  - Implementation

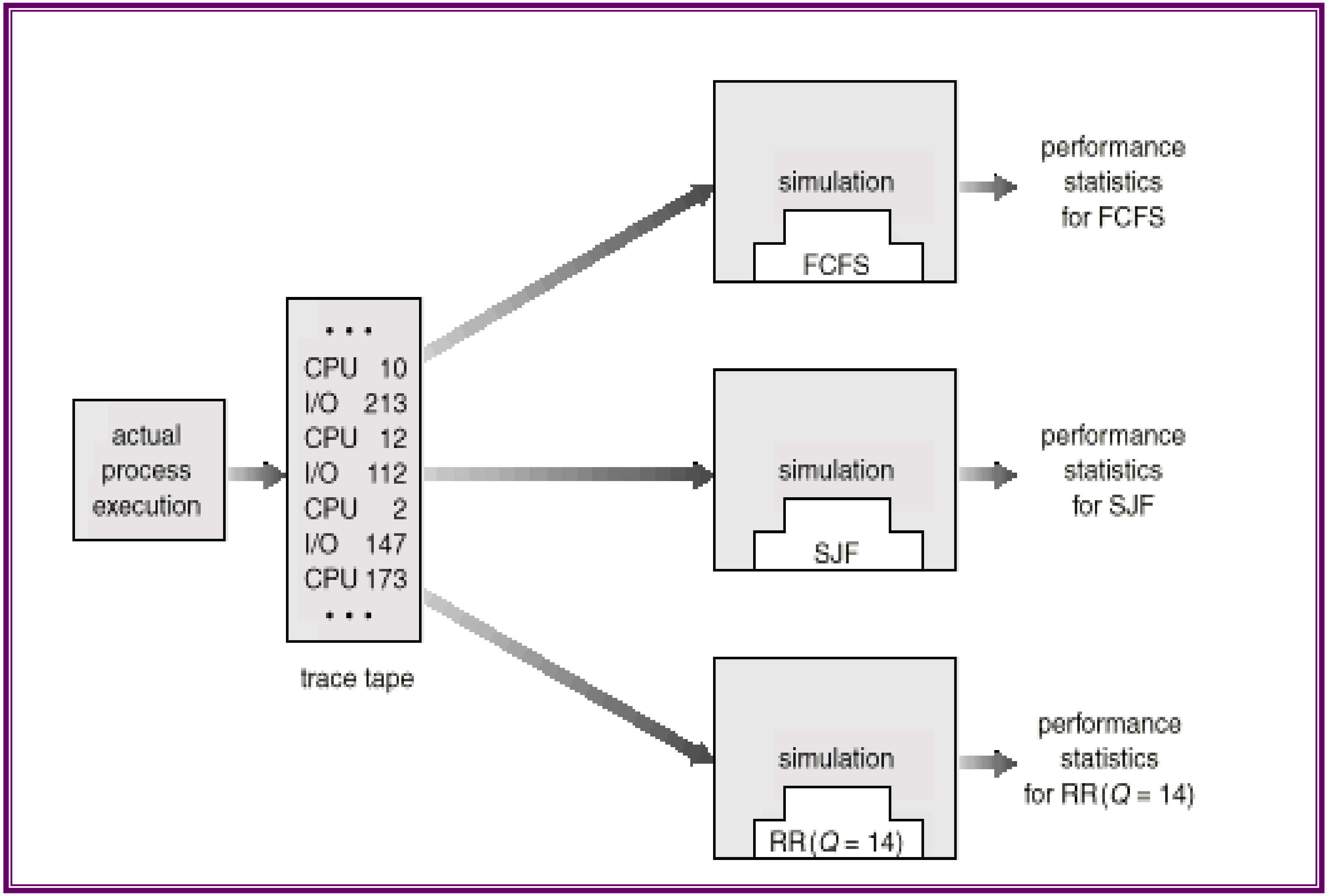
Analytical modeling

- Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.
- One type of analytic evaluation is deterministic modeling.
  - It takes a predetermined workload and defines the performance of each algorithm for that workload.
  - + point: Fast
  - - point: specific to the load
- Queuing models (Reference: https://en.wikipedia.org/wiki/Queueing_theory)
  - No predetermined workload which is very difficult to get.
  - However, distributions of CPU and I/O busts can be approximated.
  - Result is a probability of next CPU burst.
  - Variation of every variable is approximated to one distribution.
  - This area of study is called queuing network analysis.
  - Little's law: $n=\lambda*W$, where $n=$ number of processes in the queue, λ is arrival rate, and w is average waiting time. (When a system is in steady state).
- +ve points: no cost; pencil and paper
- -ve points: too complex equations; only approximation to real system

Simulation

- For accurate evaluation, simulation can be used.
- Involve programming model of the computer system.
- Software data structures represent the major data structures of the system.
- There is a simulation clock which is a variable.
- Whenever clock is changes, system state is modified
  - Activities of the devices, the processes, and the scheduler.
- Random number generator can be used to generate data sets by following appropriate distributions.
  - Processes
  - CPU-burst times
  - Arrivals
  - Departures
- **Trace** can be used which is created by monitoring the real system.
- This area of study is called simulation.

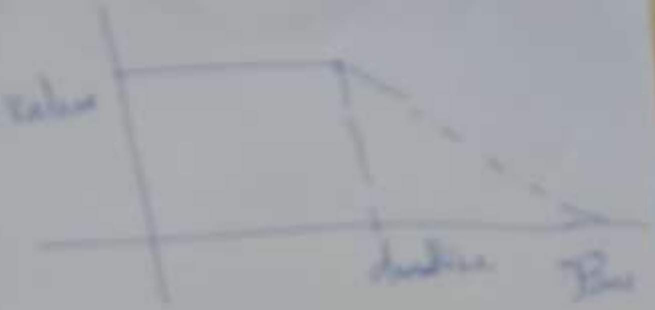
Evaluation of CPU Schedulers by Simulation



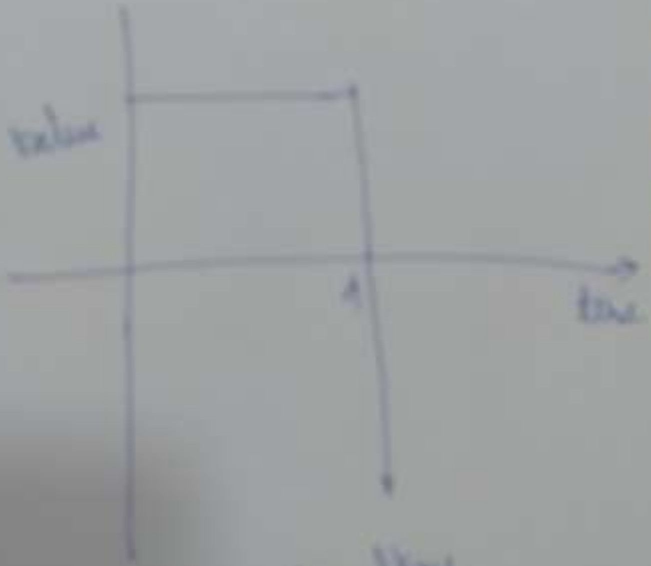
Implementation

- Simulation is also of limited accuracy.
- Correct way is code the algorithm and run in an OS.
- (-ve) Cost of this approach
- (-ve) reaction of users due to constant change

- In reality all the three methods are used.
  - Analytical
  - Simulation
  - Implementation



Soft realtime



Hard realtime

Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time.
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones.

Hard real-time systems

- Process is submitted along with a statement of amount of time in which it needs to complete or I/O.
- The scheduler may admit the process or reject the process as impossible.
- The main concept is resource reservation.
  - The scheduler must know exactly how long each OS function takes to perform.
- It is difficult to realize this guarantee on general purpose OSs.
- Therefore , hard real-time systems are composed of special purpose software running on hardware dedicated to their critical process.

Soft real-time systems

- Computing is less restrictive.
- Requires that critical process should receive priority over lesser priority processes.
- Adding soft real-time functionality may cause problems
  - Unfair allocation of resources may result in longer delays or even starvation for some processes.
- A soft real-time system is nothing but a general purpose OS that can also support multimedia, high-speed interactive graphics and so on.
  - Multimedia and high-speed interactive graphics would not work acceptably in an environment that does not support soft real-time computing.

Soft real-time systems…

- Implementation requires careful design of the scheduler and other related aspects.
- System must have priority scheduling.
- Real time processes must receive highest priority and priority should not degrade over time.
  - Aging is disallowed
- Dispatch latency must be small.
  - Which is very difficult as some system calls are complex.
  - For example, many OSs are forced to wait for a system call to complete to execute context switch.
  - So to keep dispatch latency low, insertion of **preemption points** in system call code is one solution.
  - At preemption point, the process checks whether any high priority process is waiting.
  - Another method is to make entire kernel pre-emptable.

Soft real-time systems…

- If higher priority process needs to read or modify kernel data which is currently accessed by lower priority process, **then higher priority process would be waiting for a lower priority process to finish**.
- This problem can be solved with priority-inheritance protocol
  - All the process which are sharing a resource with high priority processes inherit the high priority until they are done with that resource.
  - After finishing, priority reverts to original value.
- So the conflict phase of dispatch latency has two components
  - Preemption of any process running in the kernel
  - Release by low-priority process resources needed by high-priority resources.
    - In Solaris dispatch latency with preemption is 2 msec and without preemption is 100 msec.

Implementing Real-Time Operating Systems

- In general, real-time operating systems must provide:

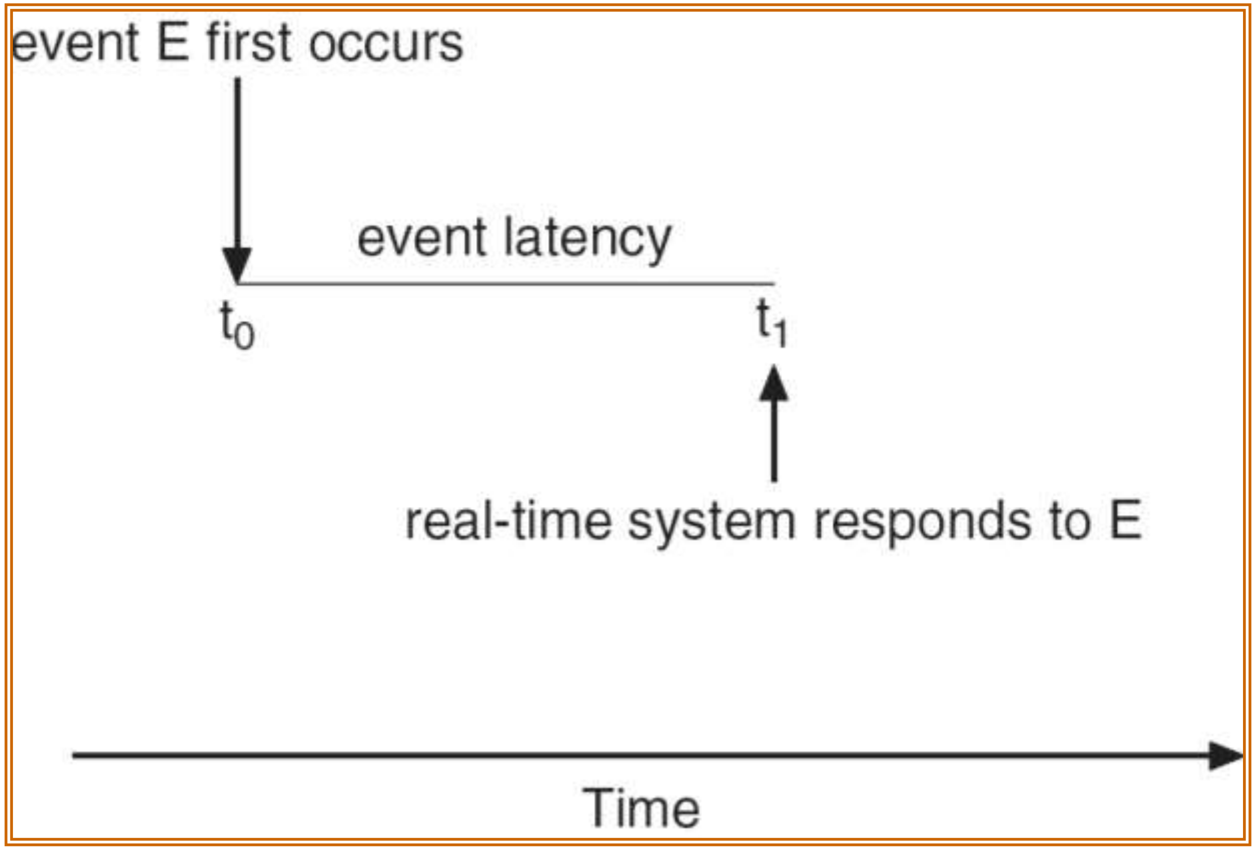
  (1) Preemptive, priority-based scheduling

  (2) Preemptive kernels

  (3) Latency must be minimized

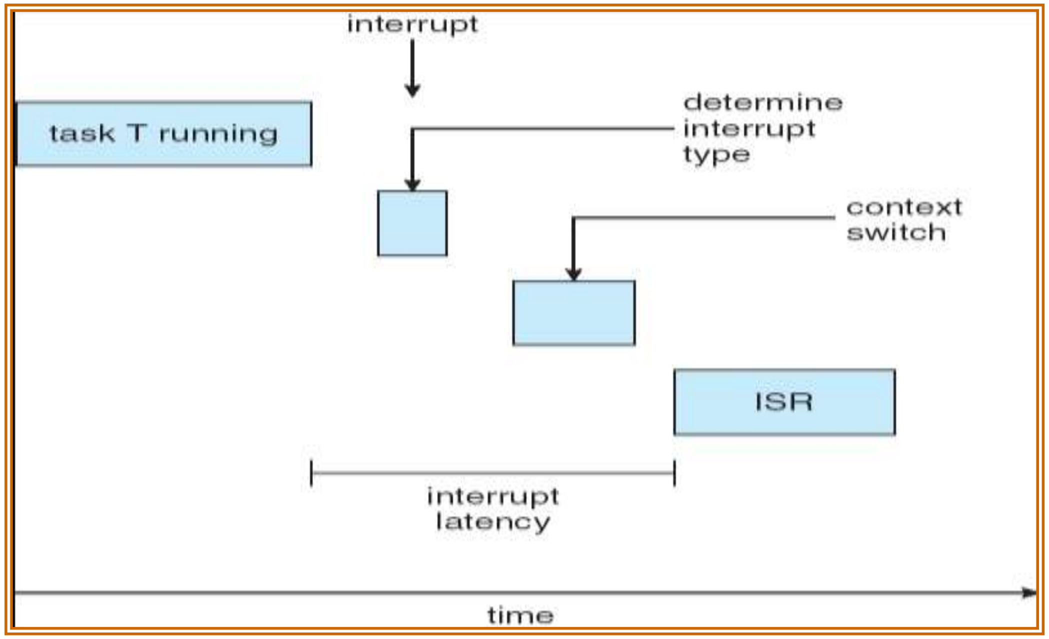
Minimizing Latency

- **Event latency** is the amount of time from when an event occurs to when it is serviced.



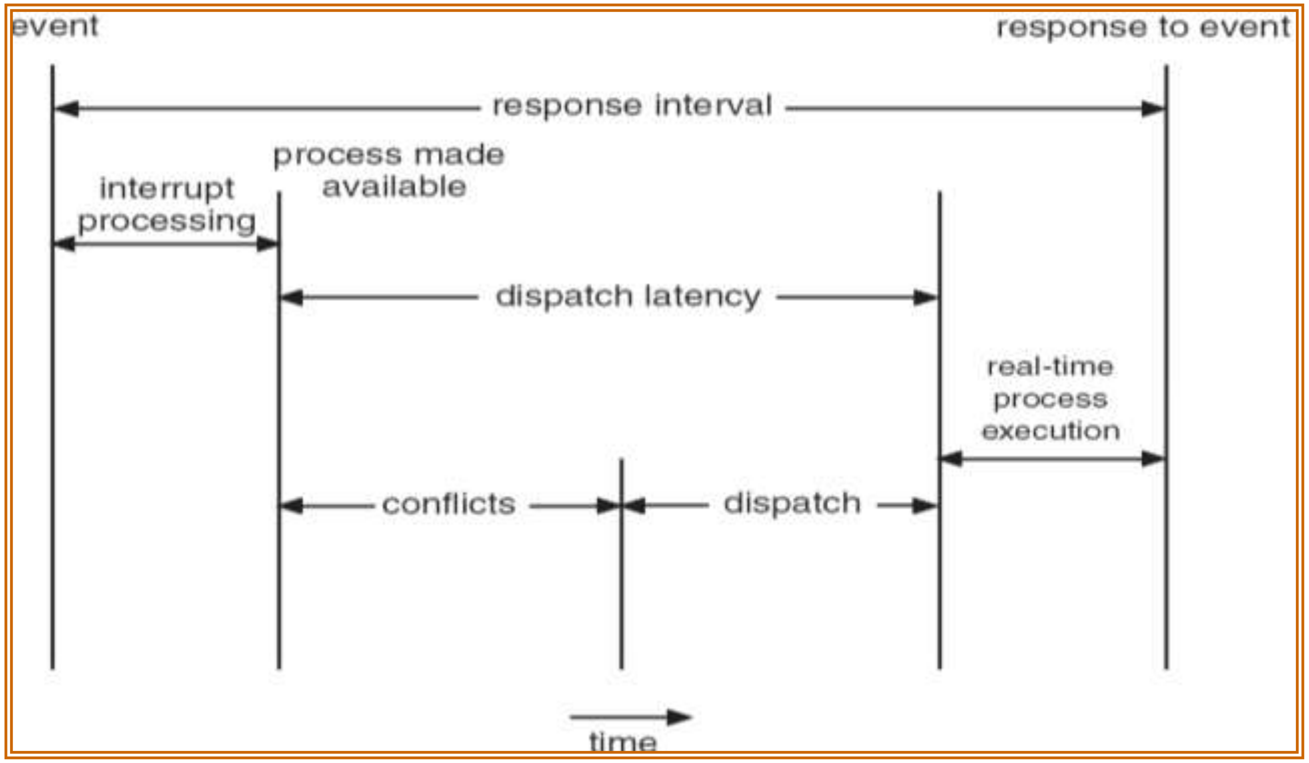
Interrupt Latency

- Interrupt latency is the period of time from when an interrupt arrives at the CPU to when it is serviced.



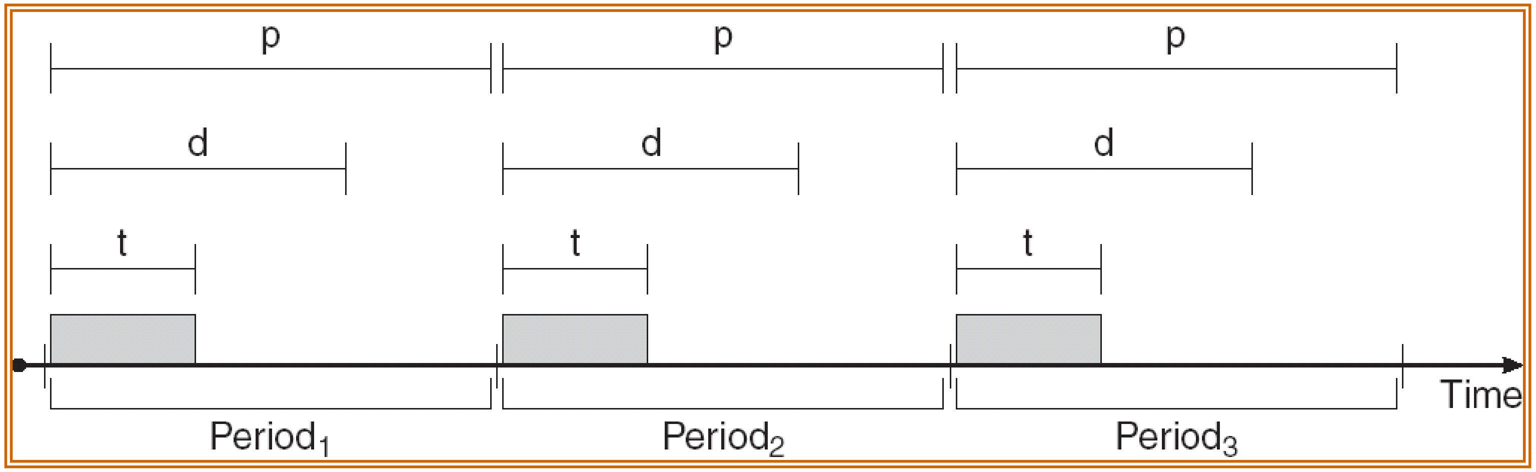
Dispatch Latency

- **Dispatch latency** is the amount of time required for the scheduler to stop one process and start another.



Real-Time CPU Scheduling

- Periodic processes require the CPU at specified intervals (periods)
- ***p*** is the duration of the period
- ***d*** is the deadline by when the process must be serviced
- ***t*** is the processing time



Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.

Earliest Deadline First Scheduling

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority.

Proportional Share Scheduling

- ***T*** shares are allocated among all processes in the system.

- An application receives ***N*** shares where ***N < T***.

- This ensures each application will receive ***N / T*** of the total processor time.

Pthread Scheduling

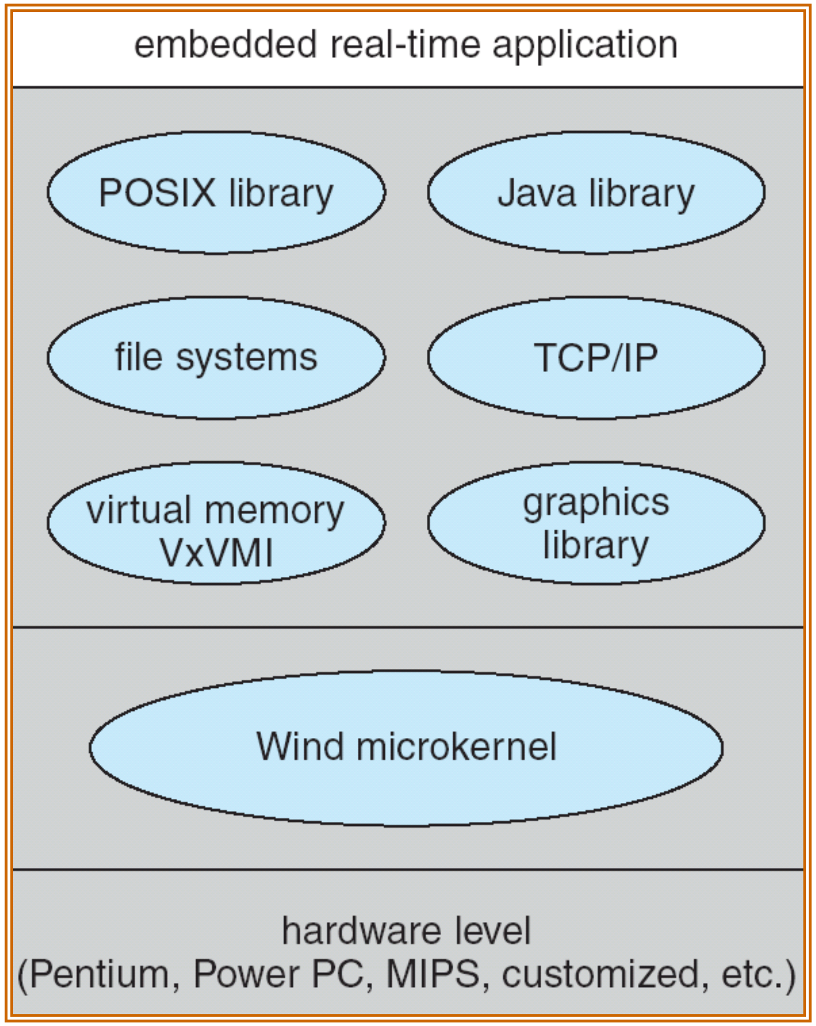
- The Pthread API provides functions for managing real-time threads.

- Pthreads defines two scheduling classes for real-time threads:

  (1) SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority.

  (2) SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority.

VxWorks 5.0


embedded real-time application
POSIX library
Java library
file systems
TCP/IP
virtual memory VxVMI
graphics library
Wind microkernel
hardware level (Pentium, Power PC, MIPS, customized, etc.)

Wind Microkernel https://en.wikipedia.org/wiki/VxWorks

- **VxWorks** is a real-time operating system (RTOS) developed as proprietary software by Wind River Systems, a wholly owned subsidiary of TPG Capital, US.
- First released in 1987, VxWorks is designed for use in embedded systems requiring real-time, deterministic performance and, in many cases,
  - safety and security certification, for industries, such as aerospace and defense, medical devices, industrial equipment, robotics, energy, transportation, network infrastructure, automotive, and consumer electronics.
- The Wind microkernel provides support for the following:

(1) Processes and threads;

(2) preemptive and non-preemptive round-robin scheduling;

(3) manages interrupts (with bounded interrupt and dispatch latency times);

(4) shared memory and message passing interprocess communication facilities.

Solaris 2 scheduling

- Priority based process scheduling
- Four classes of scheduling: Real time, System, Time sharing and Interactive
- Each class includes different priorities and scheduling algorithms.
- A process starts with one LWP and is able to create new LWPs as needed.
- Each LWP inherits the scheduling class and priority of the parent process.
- Default scheduling class is time sharing.
- The scheduling policy for time sharing dynamically alters priorities and assigns time slices of different lengths using a multilevel feedback queue.
- Inverse relationship between priorities and time slices.
- Interactive process receives high priority
- CPU bound process receives low priority.

Solaris 2 scheduling

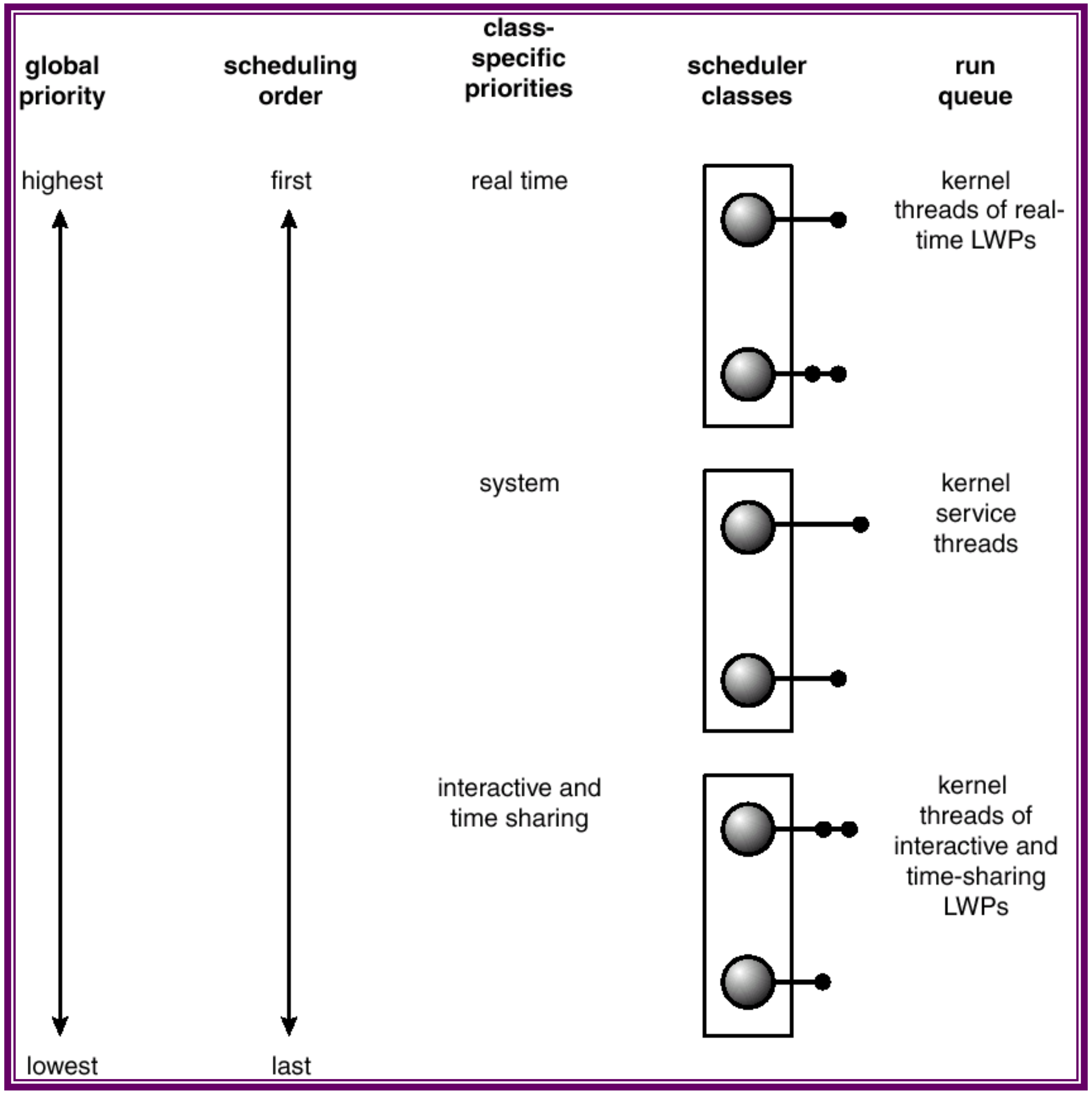
- Solaris 2 uses system class to run kernel processes: scheduler and paging daemon.
- The priority of a system process does not change.
- The scheduling class for system class is not time slice.
  - It runs until it blocks or preempted by a higher priority thread.
- Threads in realtime class are given the highest priority to run among all classes.
  - Allows a guaranteed response time
  - In general few processes belong to real time class.
- The selected thread runs until one of the following occurs.
  - It blocks
  - It uses its time slice (if it is not a system thread)
  - It is preempted by a higher-priority thread.
- If multiple threads have same priority, the scheduler uses round-robin queue

Dispatch Table

- Fields of dispatch table
  - Priority
    - Higher number indicates a higher priority
  - Time quantum
    - Time quantum of associated priority
  - Time quantum expired
    - New priority if it consumes entire time quantum (lowered)
  - Return from Sleep
    - Priority of thread returning from sleep

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris 2 Scheduling


global priority
scheduling order
class-specific priorities
scheduler classes
run queue
highest
first
real time
kernel threads of real-time LWPs
system
kernel service threads
interactive and time sharing
kernel threads of interactive and time-sharing LWPs
lowest
last

WINDOWS 2000/XP scheduling

- WINDOWS 2000 schedules threads using a priority-based, preemptive scheduling algorithm.
- Scheduler ensures that highest priority thread always run.
- Scheduler is called dispatcher.
- A thread will run until
  - Preempted by high priority thread
  - Until it terminates
  - Until its time quantum ends
  - Until it calls a blocking system call.
- The dispatcher uses 32-level priority scheme.
- Priorities are divided into two classes
  - Variable class (1 to 15)
  - Real-time class (16-31)
- Priority 0 is used for memory management.
- The dispatcher uses a queue for each scheduling priority and traverses from highest to lowest.
- If no ready thread is found, the dispatcher executes a special thread called idle thread.

WINDOWS 2000 scheduling

- WINDOWS 2000 identifies several priority classes. These include:
  - REALTIME_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS
  - BELOW_NORMAL_PRIORITY_CLASS
  - IDLE_PRIORITY_CLASS
- Except REALTIME_PRIORITY_CLASS all other are variable class priorities; the priority the thread belong to this class may change.
- Within each priority class there is a relative priority
  - TIME_CRITICAL
  - HIGHEST
  - ABOVE_NORMAL
  - NORMAL
  - BELOW_NORMAL
  - LOWEST
  - IDLE
- The priority of each thread is based upon the priority class it belongs to and the relative priority within the class.
- In addition each thread has a base priority which is equal to normal priority.
- The initial priority is typically the base priority of the class

Windows 2000 Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

WINDOWS 2000 scheduling

- When a thread quantum runs out, the thread is interrupted; if the thread is in variable-priority class its priority is lowered.
- When the variable priority thread is released from wait operation, the dispatcher boosts its priority.
  - Good response time to interactive threads.
  - Keeps I/O devices busy.
  - CPU bound processes use CPU cycles background.
- When a process moves to foreground, the time quantum is increased by some factor---typically by three.

LINUX

- Two separate process scheduling algorithms
  - Time sharing algorithm for fair preemptive scheduling among multiple processes.
  - Other is designed for real-time tasks
- LINUX allows only user processes to preempt
- A process can not be preempted in kernel mode, even real-time process is available to run.
- Every process is assigned with scheduling class, that defines which of the algorithms to apply to the process.
- First class is for time—sharing processes
  - LINUX uses prioritized credit based algorithm
  - Each process posses a certain number of scheduling credits.
  - While choosing, the process with most credits is selected.
  - Every time the interrupt occurs the process loses one credit
  - When its credits reaches zero it is suspended and another process is chosen.

LINUX (earlier than 2.5)

- If no runnable process have any credits, LINUX performs re-crediting operation. It adds credits to every process
  - Credits=(Credits/2)+priority
- This algorithm uses two factors: process history and priority.
- Suspended process will accumulate more credits.
- It also gives high priority to interactive processes
- Linux implements two real-time scheduling classes
  - FCFS and round robin.
- In the FCFS class the process runs until it blocks.
- LINUX real-time scheduling is soft—rather than hard.
- The scheduler guarantees priorities, but kernel never gives guarantees
  - Kernel process can not preempted