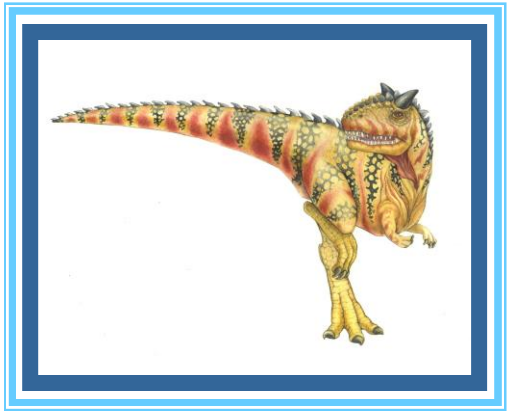
Chapter 5: CPU Scheduling



Chapter 5: CPU Scheduling

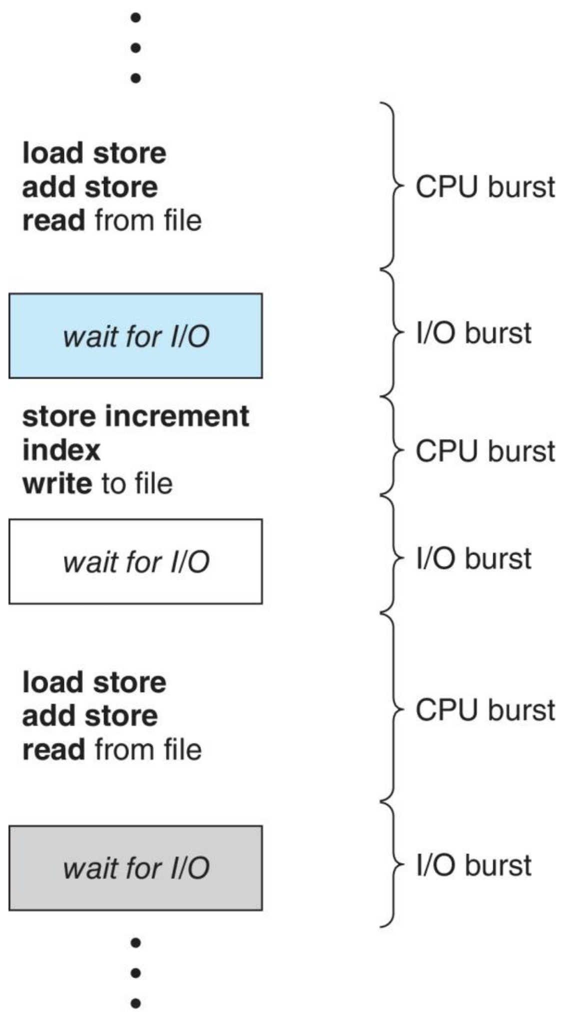
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Apply modeling and simulations to evaluate CPU scheduling algorithms

Basic Concepts

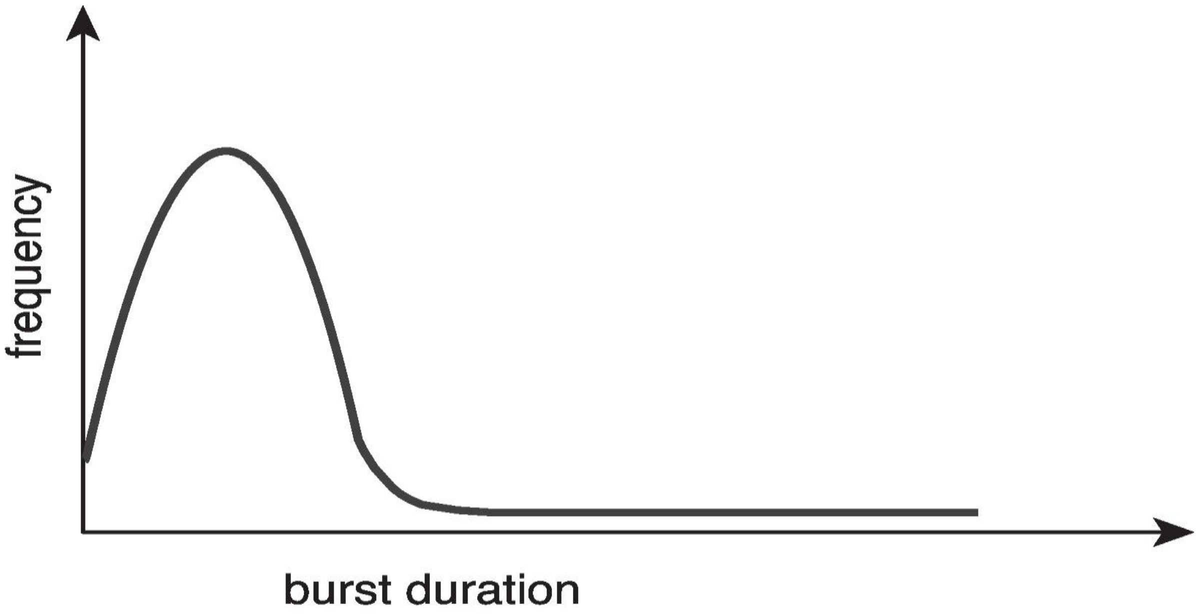
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of primary concern



Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts

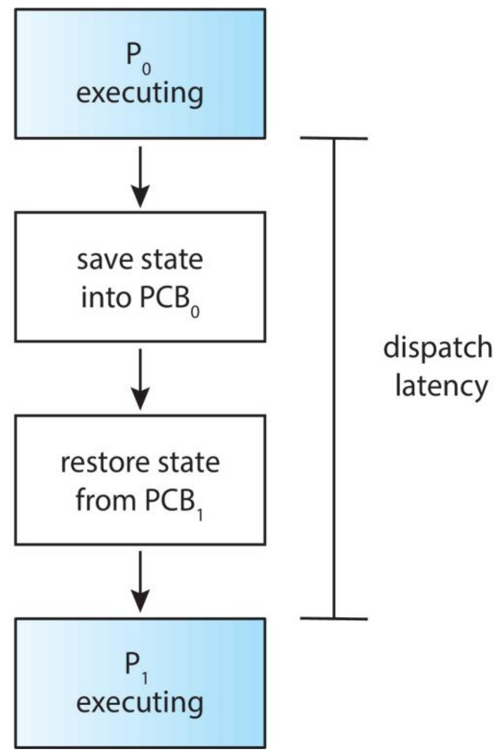


CPU Scheduler

- The CPU scheduler selects from among the processes in ready queue, and allocates the a CPU core to one of them
  - Queue may be ordered in various ways
- 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
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

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

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes or jobs completed per time unit (second)
  - Efficient use of the resources (CPU, memory, disk, etc.)
  - Minimize the overhead (for instance, context switching)
- Waiting time – amount of time a process waiting in the ready queue
- Turnaround time – amount of time to execute a particular process, usually measured by the CPU burst time plus waiting time
  - For single CPU burst, turnaround time = waiting time + CPU burst time
- Response time – amount of time it takes from when a request was submitted until the first response is produced
  - Time to echo a keystroke in editor
  - Time to compile a program
- Fairness
  - Resources such as CPU are utilized in some “fair” manner

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

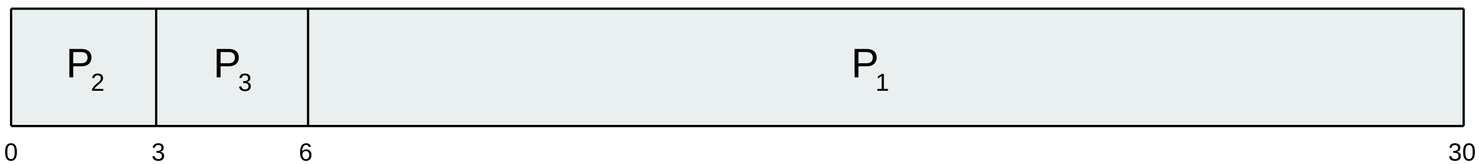
- In earlier systems, FCFS means that one program is scheduled to run until completion (including all I/O)
- Now this means a process finishes its current CPU burst time

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
- *Convoy effect* - short process stuck behind long process, potentially bad for short jobs!
  - Consider one CPU-bound and many I/O-bound processes
  - Waiting in banks: depositing a check, stuck behind new account opening

Round Robin (RR)

- 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.
- Given n processes, 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, or the process blocks upon completing its current CPU burst time ($< q$)
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ interleaved, but q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

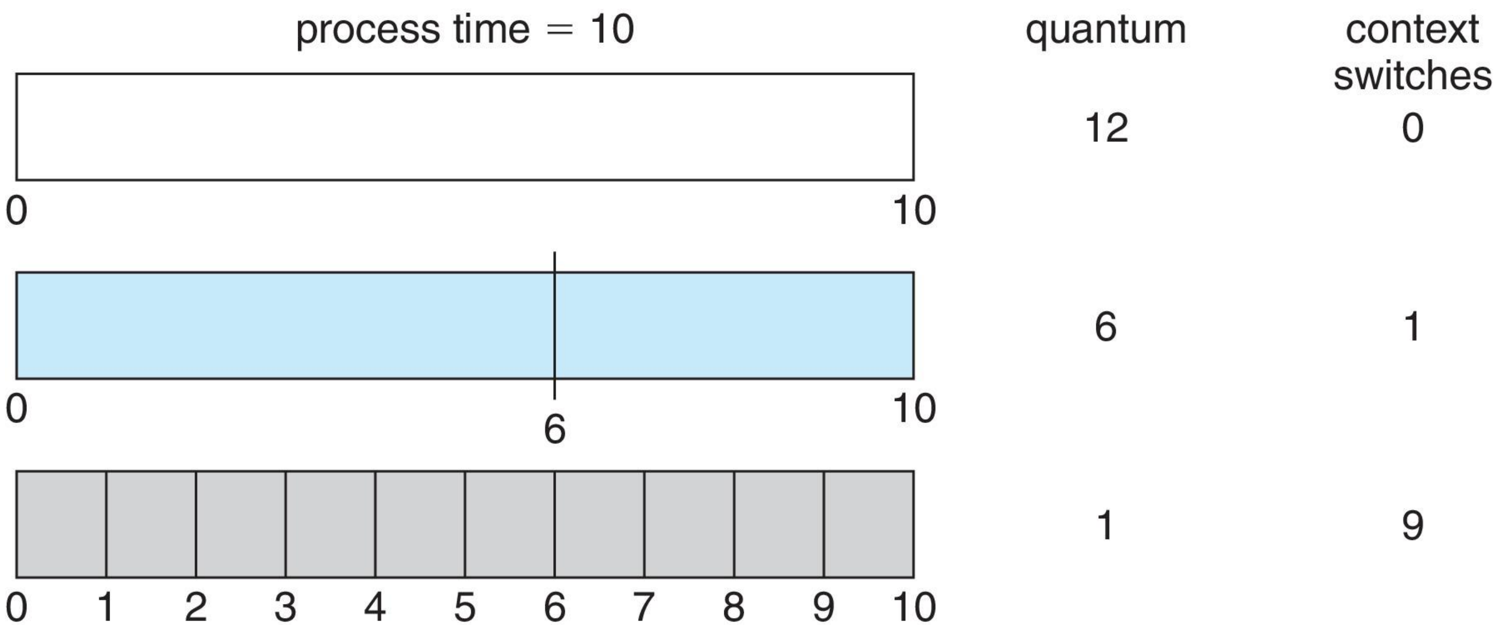
- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

- Waiting time for P1=6, P2=4,P3=7
- Average waiting time (6+4+7)/3 = 5.67
- **Response time** for P1=4, P2=7, P3=10, average = 7
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec
- Better for short job (fair), context switching adds up for long jobs

Time Quantum and Context Switch Time



Comparisons b/t FCFS and RR

Assuming zero-cost context-switching time, is RR always better than FCFS?

- Simple example: 10 jobs starting at the same time, each taking 100s of CPU time; RR scheduler quantum of 1s;

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

- Average job completion time is much worse under RR!
  - Bad when all jobs have the same length

Earlier Example w/ Different Quantum

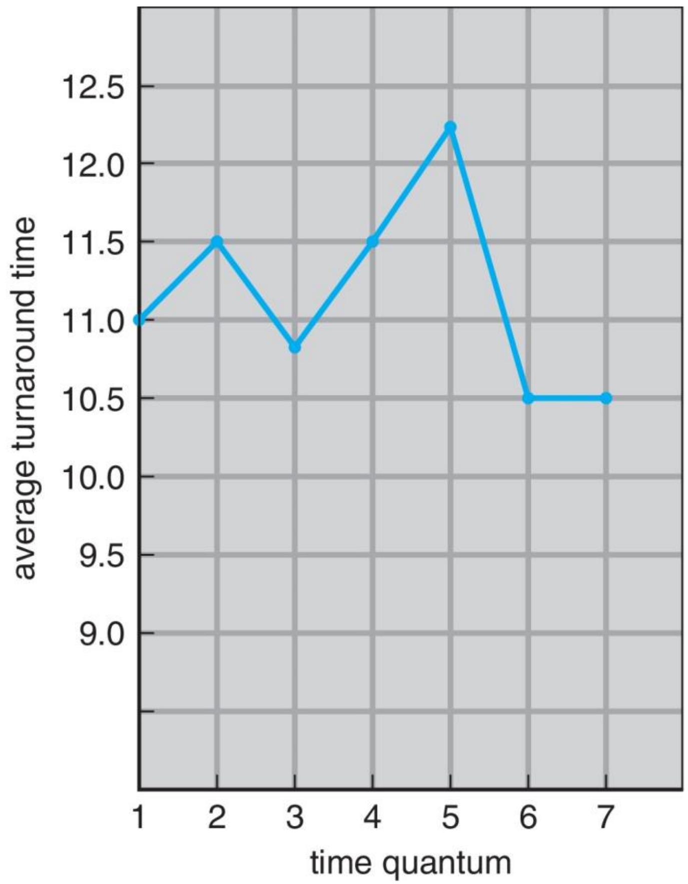
- Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- The average turnaround time does not necessarily improves as the time quantum size increases
- In general, the average turnaround time can be improved if most processes finish their current CPU bursts within a single quantum
- The time quantum can not be too big, in which RR degenerates to an FCFS policy
- **A rule of thumb**: 80% of CPU bursts should be shorter than the time quantum q

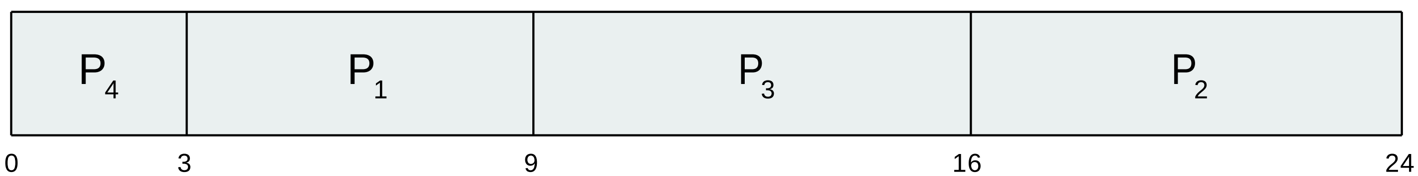
Shortest-Job-First (SJF) Scheduling

- What if we knew the future
- Associate with each process the length of its next CPU burst
  - To schedule the process with the shortest CPU burst
- SJF is optimal – yields the minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user
  - The basic idea is to get the short jobs out of system sooner
  - Big effect on short jobs, only small effect on long jobs
  - This can be applied to whole program or current CPU burst

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
- The (best) FCFS can perform the same if arrival order is the same

Determining Length of Next CPU Burst

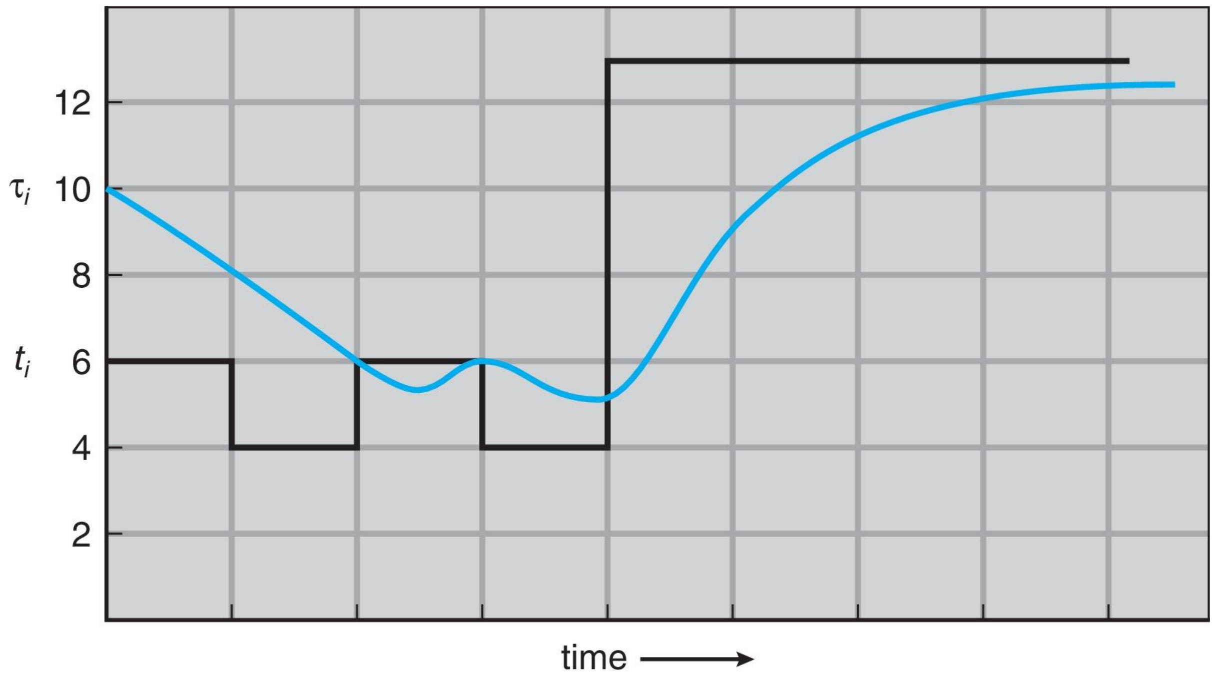
- Can estimate the length based on the past behavior
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, and exponential averaging algorithm

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

- Commonly, α set to ½
- Preemptive version called shortest-remaining-time-first（SRTF）

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

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

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

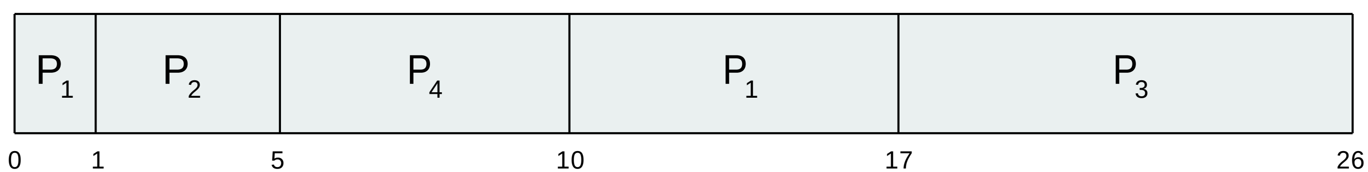
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor, thus its effect is diminishing exponentially fast

Example of Shortest-remaining-time-first

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

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

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec
- Now scheduling needs to be considered with each arrival

Comparison of SJF/SRTF and FCFS

- SJF/SRTF are the best we can do at minimizing the average waiting time (or the average turnaround time)
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - SRTF is always at least as good as SJF

- SJF/SRTF performs the same as FCFS if all processes have the same CPU burst times
- SJF/SRTF can lead to starvation

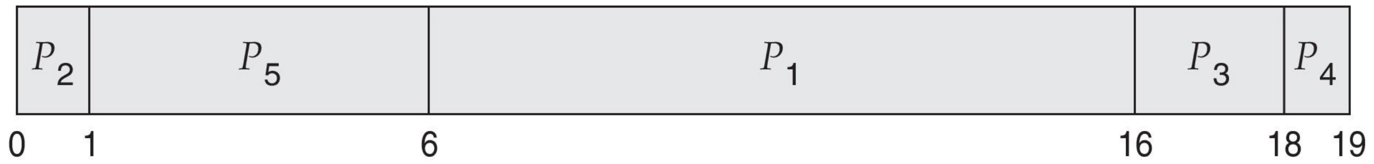
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority), it can be
  - Preemptive（upon new arrival from higher priority)
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- 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

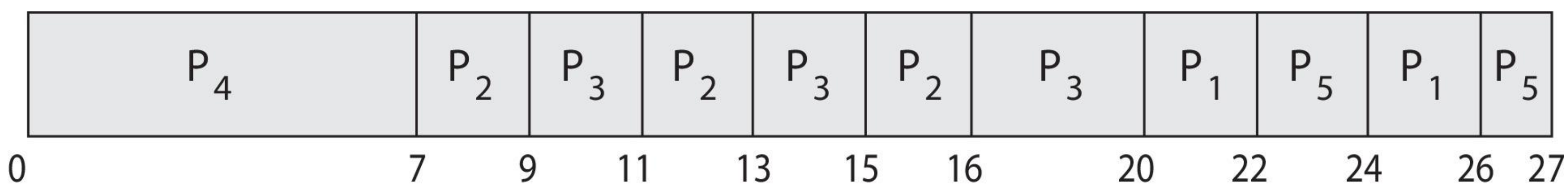


- Average waiting time = 8.2 msec

Priority Scheduling w/ Round-Robin

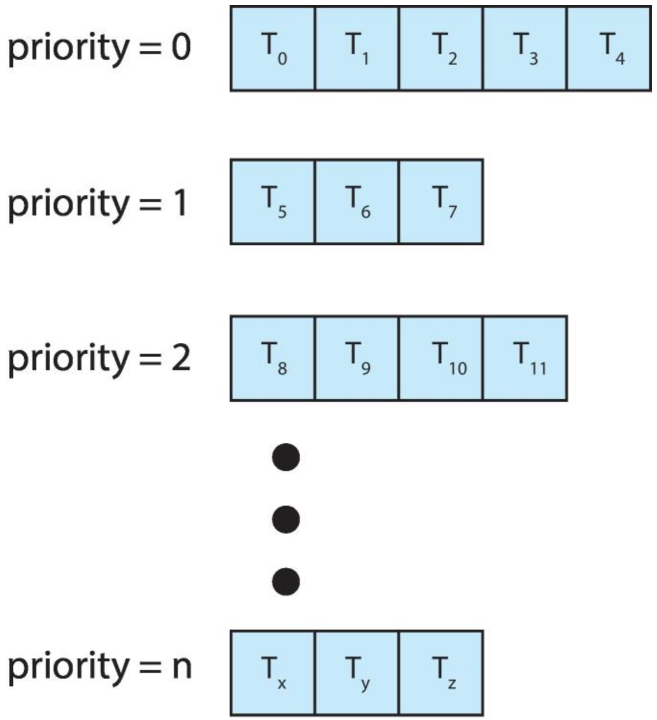
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with 2 ms time quantum



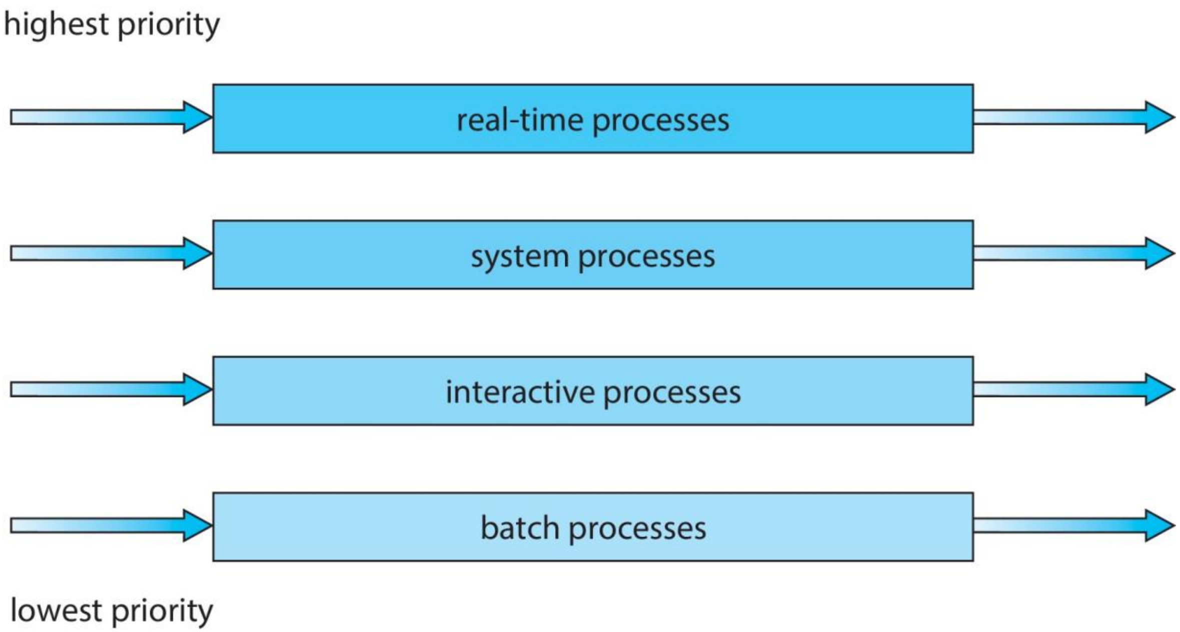
Multilevel Queue

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



Multilevel Queue

- Prioritization based upon process type
- Each queue gets certain amount of CPU time (60%, 20%, 10%, 10%)

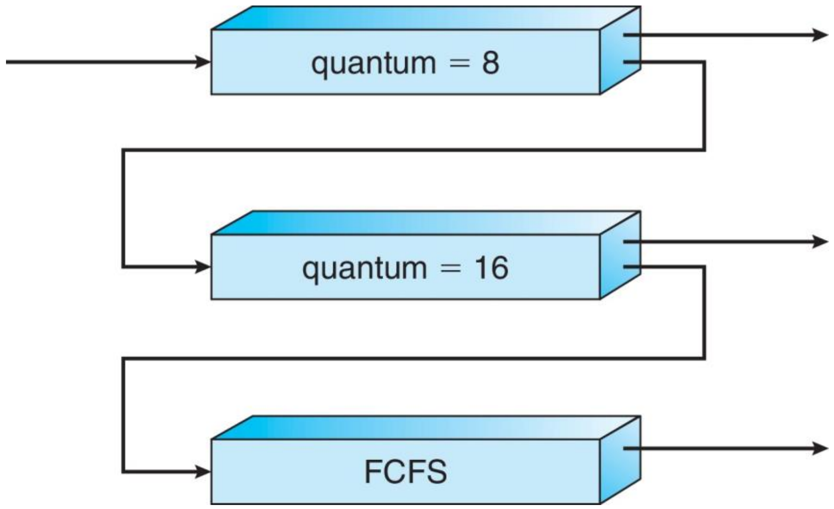


Multilevel Feedback Queue (MLFQ)

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

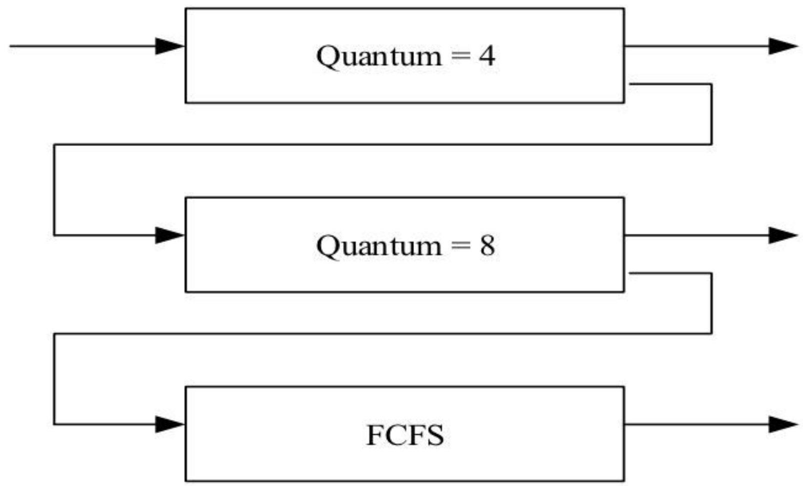
- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



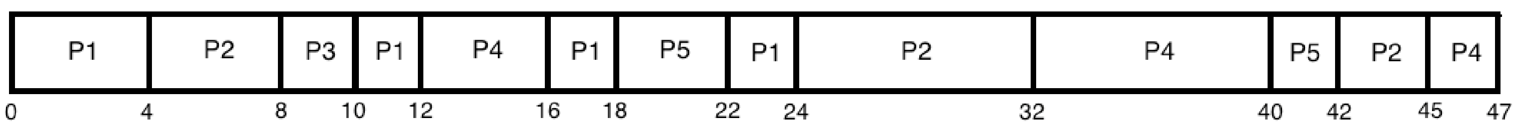
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete,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

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top

MLFQ Example



Process	Arrival Time (ms)	Burst Time (ms)
P1	0	10
P2	2	15
P3	5	2
P4	12	14
P5	18	6

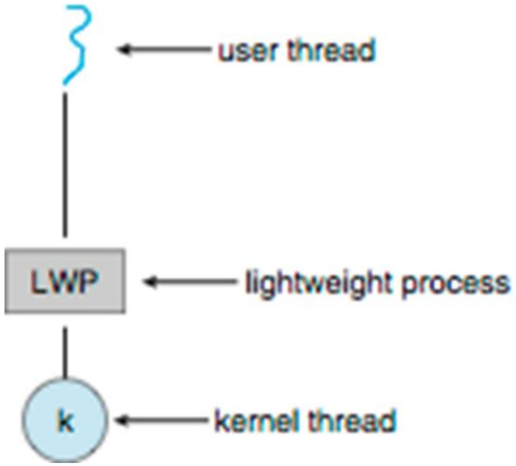


Multilevel Feedback Queue (MLFQ)

- MLFQ is commonly used in many systems such as BSD Unix, Solaris, Window NT and subsequent Window operating systems
- MLFQ has several distinctive advantages:
  - It does not require prior knowledge on CPU burst time
  - It handles interactive jobs by delivering similar performance as that of SJF or SRTF
  - It is also “fair” by making progress on CPU-bound jobs
- The possible starvation problem can be handled by reshuffling the jobs to different queues periodically
  - After some period, move all jobs to the top queue

Thread Scheduling

- When the OS supports threads, the kernel-level threads are the ones being scheduled, not processes, User-level threads are managed by a thread library instead
- The OS uses an intermediate data structure between user and kernel threads, referred as a lightweight process (LWP)
  - It appears to be a virtual processor on which user threads are scheduled to "run"
  - Each LWP attached to kernel thread (one-to-one)



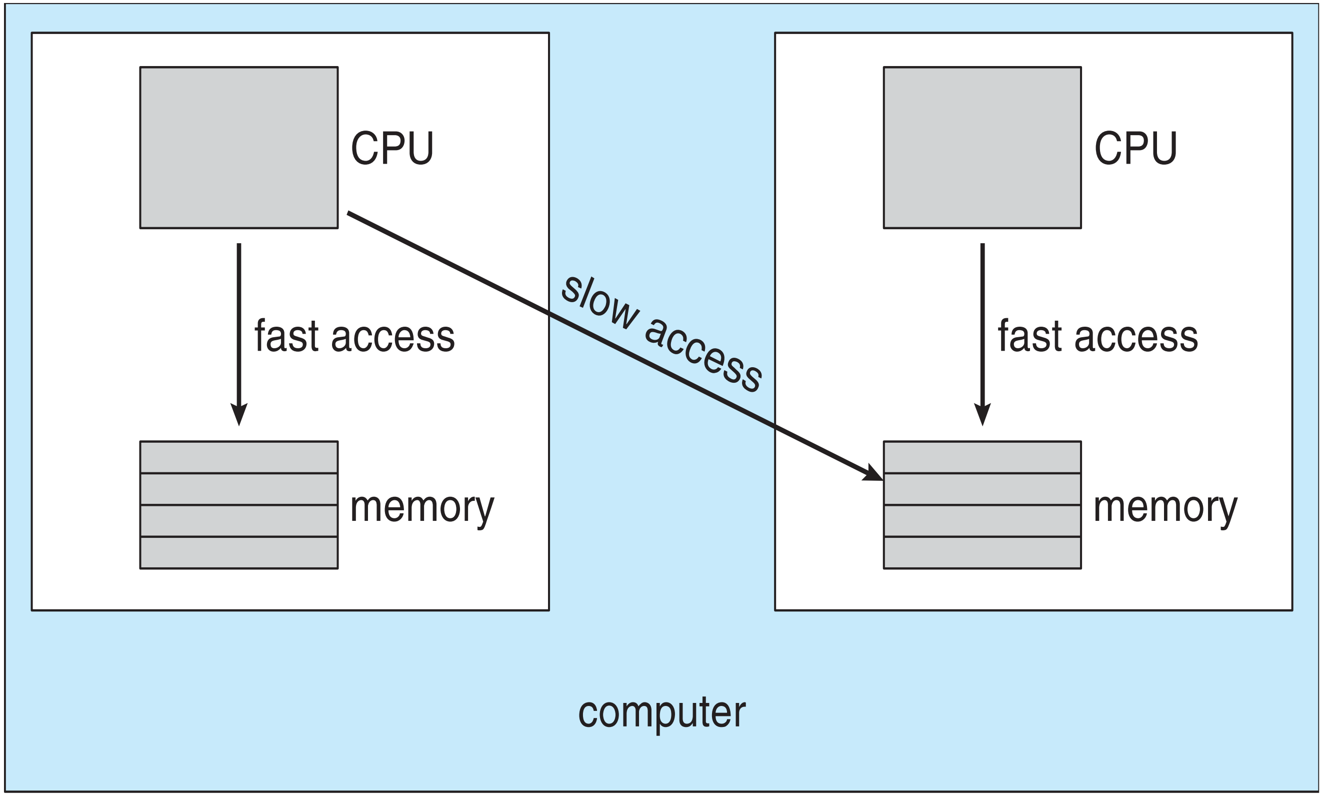
- Under many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP/ This is known as process-contention scope (PCS) since scheduling competition takes place among the threads belonging to the same process, typically done via priority set by programmers
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in the system – CPU scheduling
- Systems using one-to-one mapping model, such as Windows, Linux, and Solaris, schedule threads using only SCS

Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available – load sharing
- Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing. The other processors execute only user code
- Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - All modern OS support SMP, including Window, Linux, and Mac OS X
- Processor affinity – process has affinity for processor on which it is currently running,
  - When a thread has been running on one processor, the cache content of that processor stores the memory accesses by that thread. We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
  - There is a high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another
  - Soft affinity – the OS attempt to keep a process running on the same processor, not guaranteeing it
  - Hard affinity – allow a process to specify a subset of processors on which it may run

NUMA and CPU Scheduling

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

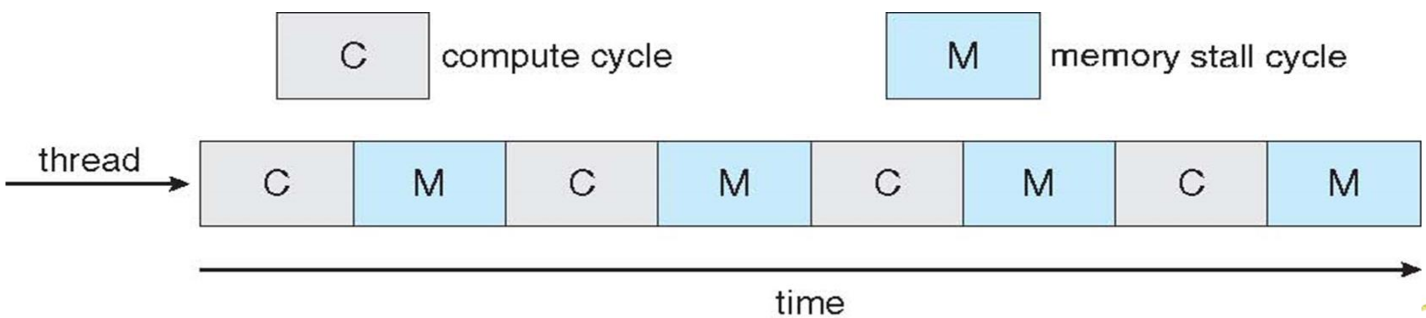


Multiple-Processor Scheduling – Load Balancing

- On SMP systems, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
- Push migration – a specific task periodically checks the load on each processor, and if it finds an imbalance, pushes task from overloaded CPU to idle or less-busy CPUs
- Pull migration – idle processors pulls waiting task from a busy processor
- Push and pull migration need not to be mutually exclusive and are in fact often both implemented in parallel on load-balancing systems
- Load balancing often counteracts the benefits of processor affinity

Multicore Processors

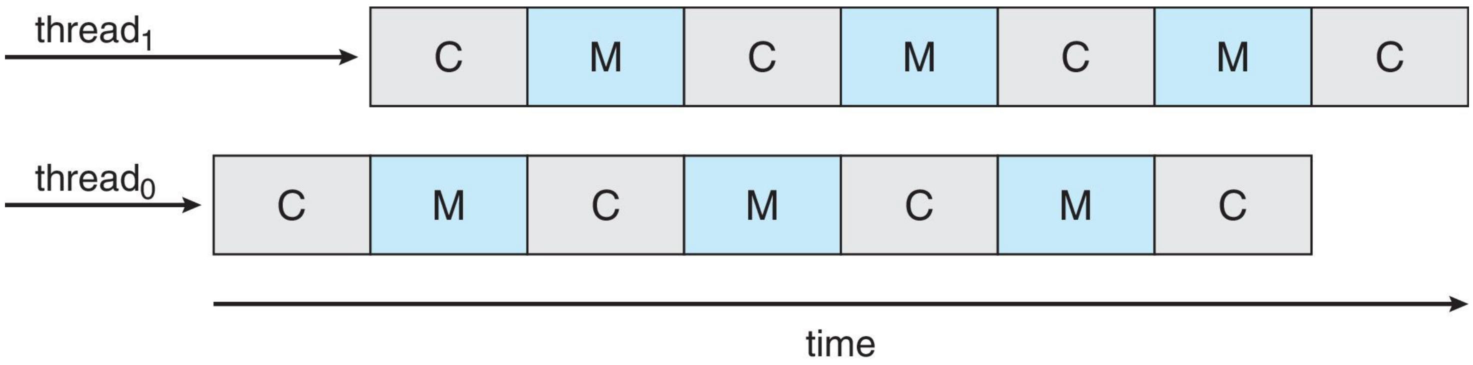
- Recent trend to place multiple processor cores on same physical chip
  - Faster and consumes less power
- Multiple threads per core also growing
  - Memory stall: a situation when a processor accesses memory, it spends a significant amount of time waiting for the data to become available, due to various reasons such as cache miss
- The scheduling can takes advantage of memory stall to make progress on another thread while memory retrieve happens
  - If one thread stalls while waiting for memory, the core can switch to another thread. This becomes a dual-thread processor core, or resembles two logical processors
  - A dual-threaded, dual-core system presents four logical processors to the operating system
  - UltraSPARC T3 CPU has 16 cores per chip and 8 hardware threads per core, from OS perspective, this appear to be 128 logical processors



Multithreaded Multicore System

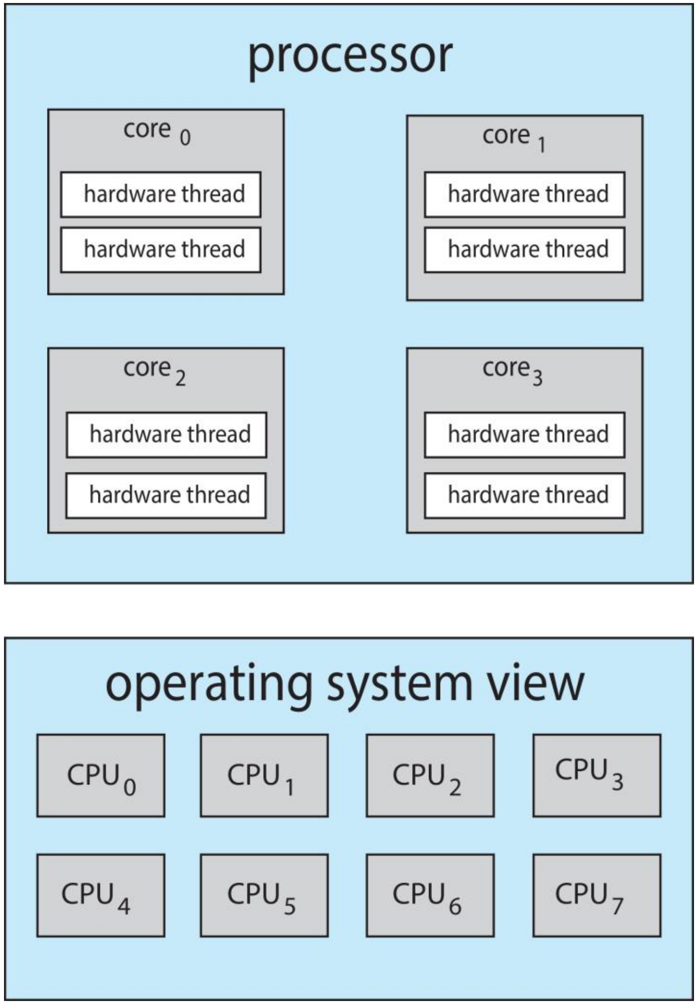
Each core has > 1 hardware threads.

If one thread has a memory stall, switch to another thread!



Multithreaded Multicore System

- Chip-multithreading (CMT) assigns each core multiple hardware threads. (Intel refers to this as hyperthreading.)
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

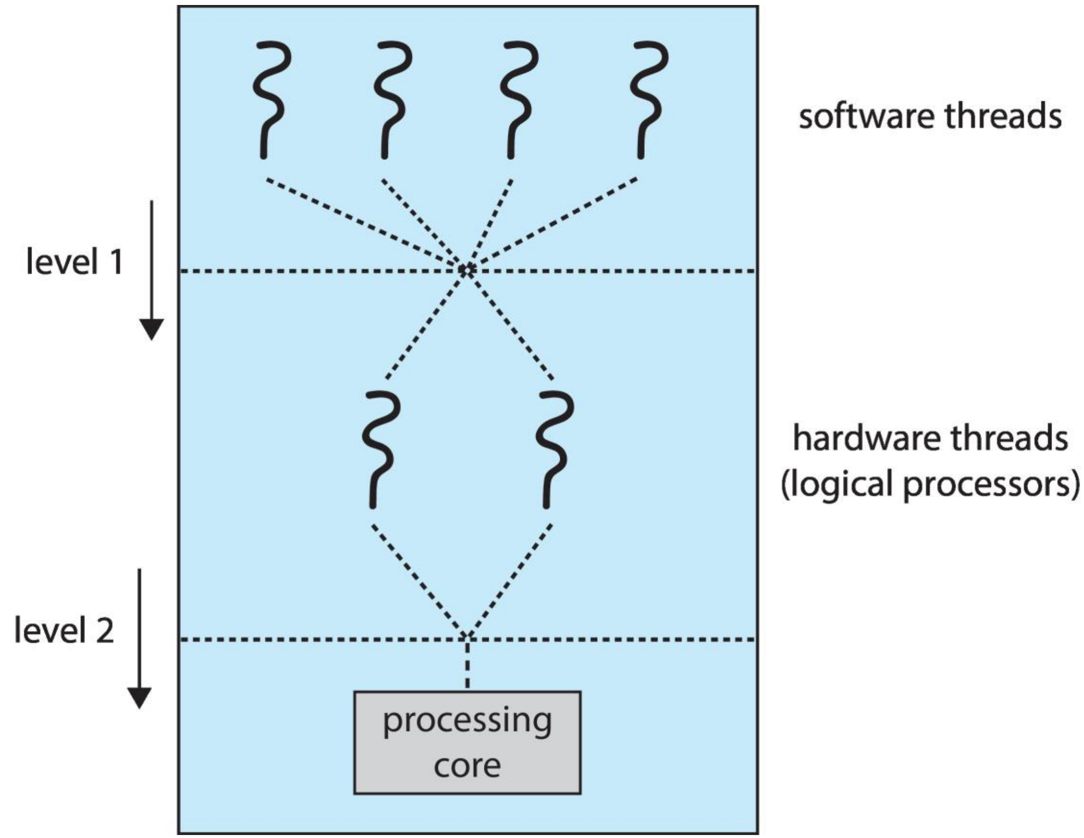


Multithreaded Multicore System

- Two levels of scheduling:

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

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



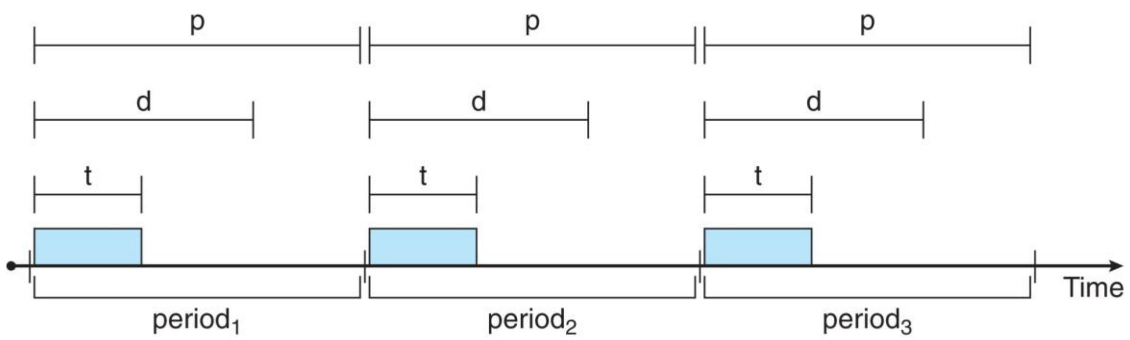
Real-Time CPU Scheduling

- This requires performance guarantee – predictability

- Hard real-time systems – task must be serviced by its deadline
  - Attempt to meet all deadlines
  - EDF - Earlier Deadline First scheduling
- Soft real-time systems – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
  - Attempt to meet deadlines with high probability
  - Minimize the miss ratio or maximize completion ratio

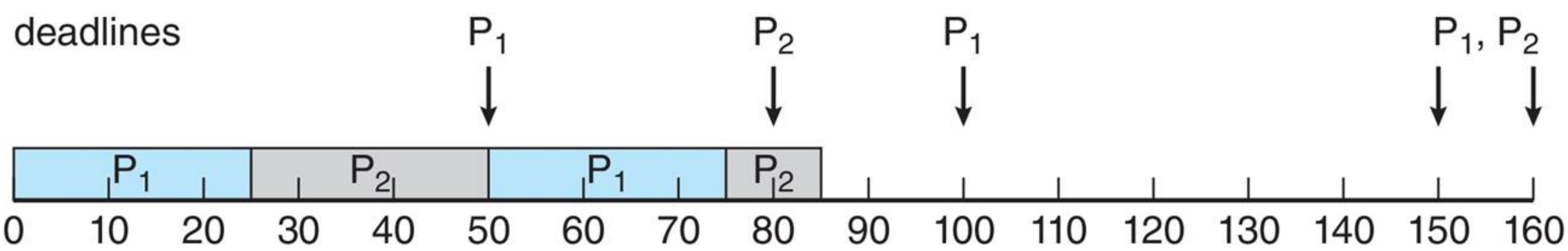
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: periodic ones require CPU at constant intervals
  - Has processing time *t*, deadline *d,* period *p*
  - $0 \leq t \leq d \leq p$
  - Rate of periodic task is $1/p$



Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority;
  - Longer periods = lower priority
- Suppose P_1 has a period of 50 (also deadline), and processing time 25. P_2 has a period of 80 (also deadline), and processing time 35.
  - Since 50 < 80, P_1 is assigned a higher priority than P_2
- Process P2 misses finishing its deadline at time 80

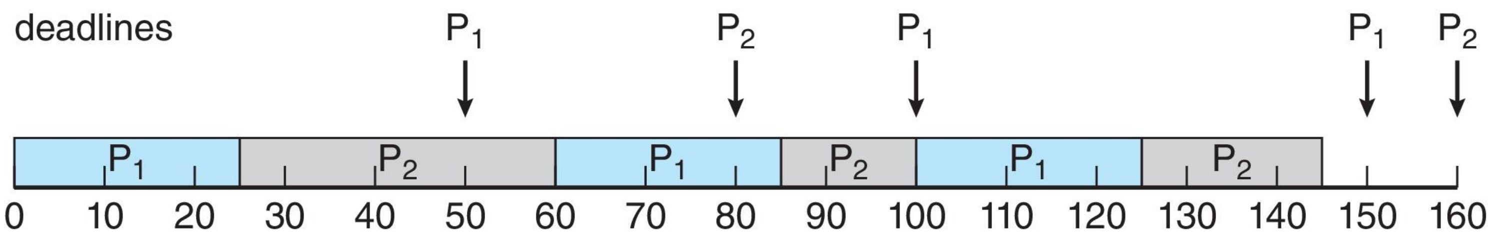


Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;

  the later the deadline, the lower the priority



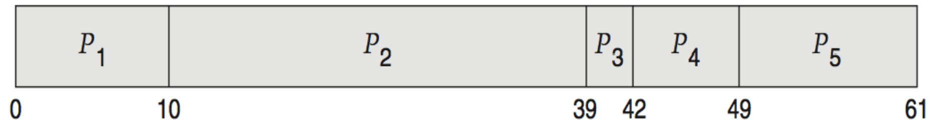
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

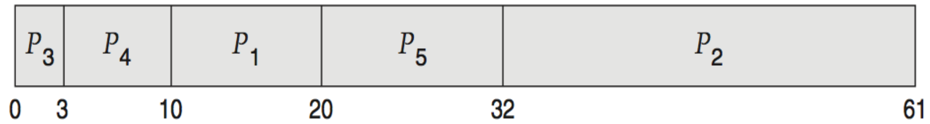
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

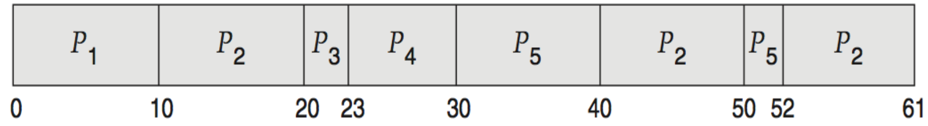
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Queueing Models

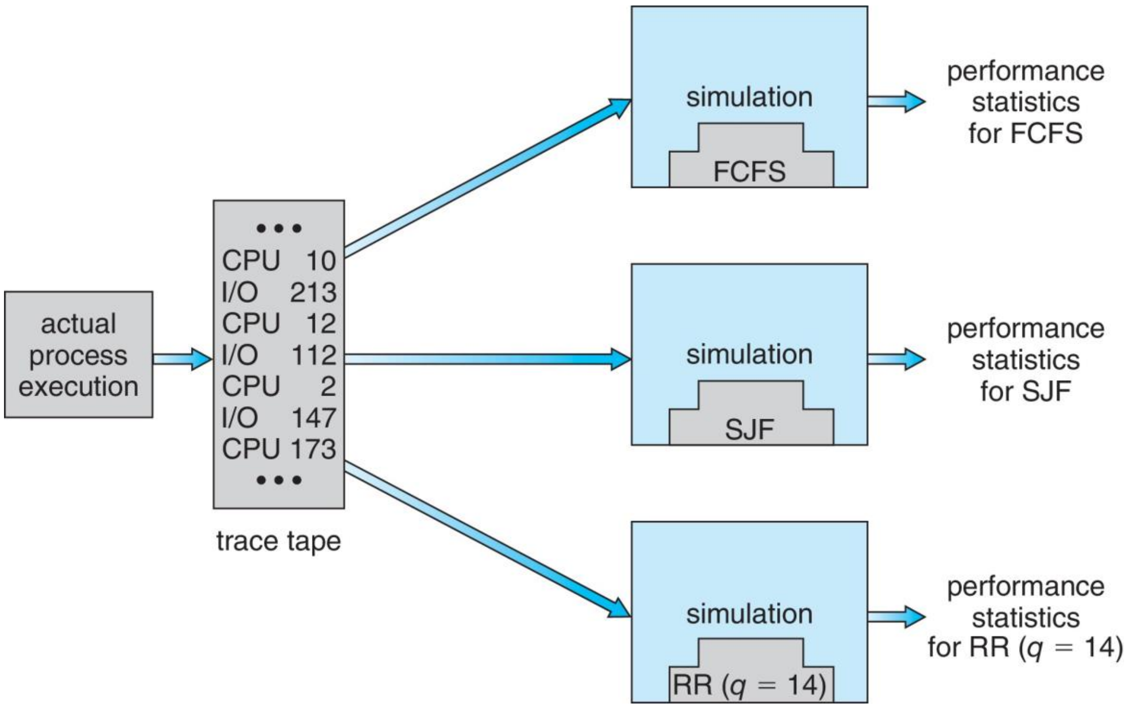
- Mathematical approach for handling stochastic workloads
- Little's Formula
- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

  $n = \lambda \times W$

  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations/Implementations

- Queueing models are restricted, most with no mathematical solution
- Simulations or implementation
  - Build system which allow actual algorithms to run with real data set – more flexible and general



End of Chapter 5