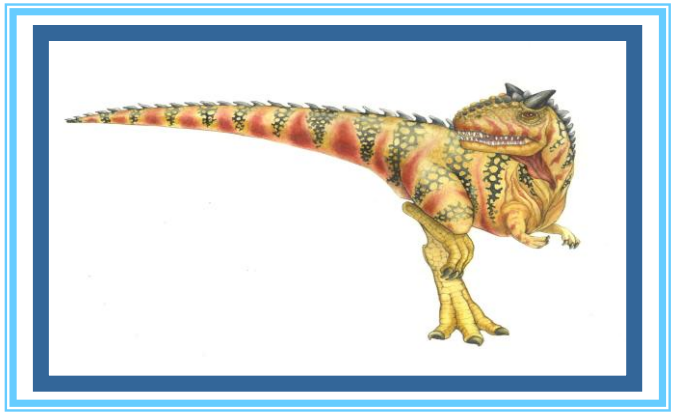
Chapter 5: CPU Scheduling



Outline

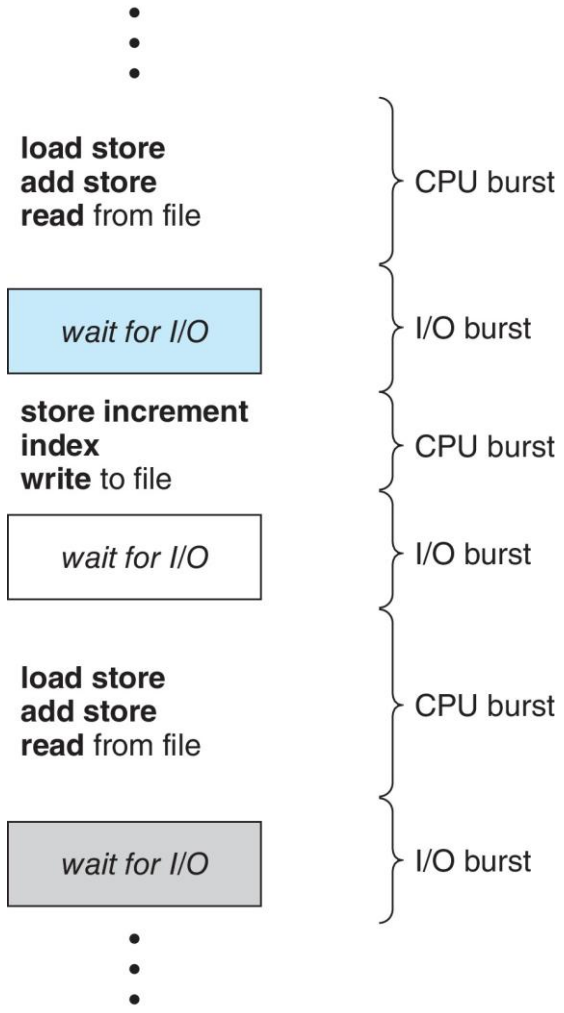
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Operating Systems Examples

Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe the scheduling algorithms used in the Windows and Linux

Basic Concepts

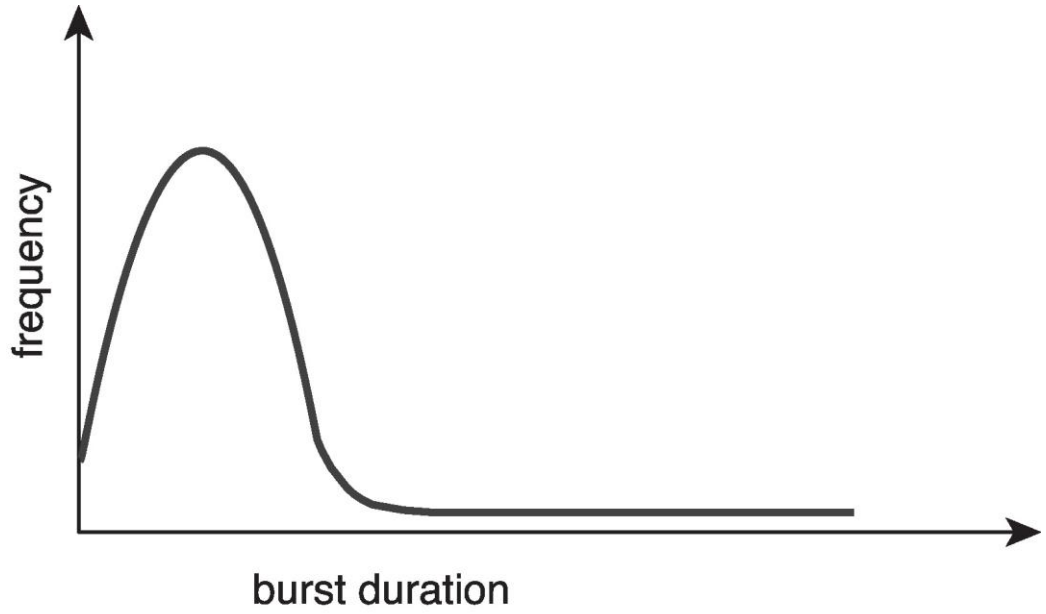
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts



Preemptive and Nonpreemptive Scheduling

- The **CPU scheduler** selects from among the processes in ready queue, and allocates 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
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.

Preemptive and Nonpreemptive Scheduling

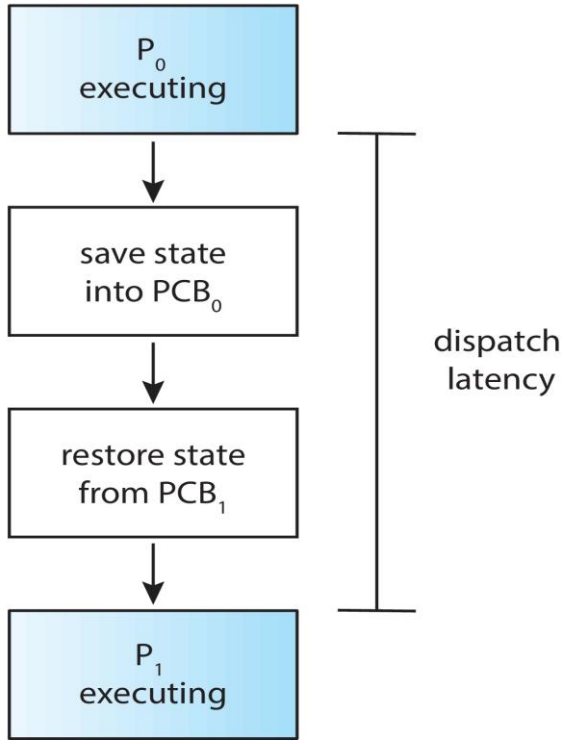
- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**.
- Otherwise, it is **preemptive**.
- Under Nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
- This issue will be explored in detail in Chapter 6.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU 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** – number of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced.

Scheduling Algorithm Optimization Criteria

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

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- At t=0, suppose that the processes arrive in the order: P_1 , P_2 , P_3

The Gantt Chart for the schedule is:

- **Turn-around Time** =Completion Time – Arrival Time
  - Turn-around Time for P_1=24-0=**24**; P_2=27-0=**27**; P_3=30-0=**30**
  - Average Turn-around time = (24 + 27 + 30)/3 =36
- **Waiting Time** = Turnaround Time – Burst Time
  - Waiting time for P_1 = 24-24=**0**; P_2 = 27-3=**24**; P_3 = 30-3=**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
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

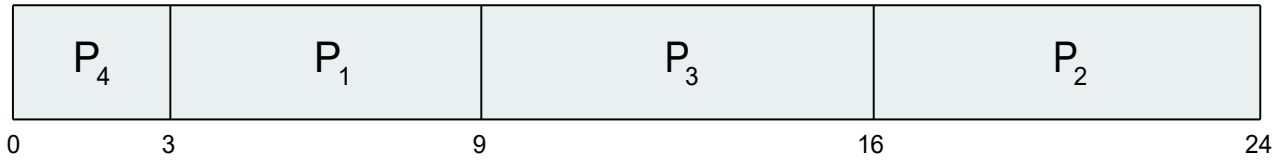
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the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

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

All processes are arrived at time t=0

- SJF scheduling chart



- **Turn-around Time** =Completion Time – Arrival Time
  - Turn-around Time for P_1=9-0=**9**; P_2=24-0=**24**; P_3=16-0=**16** ; P_4=3-0=**3**
  - Average Turn-around time = (9 + 24 + 16 + 3)/4 =13
- **Waiting Time** = Turnaround Time – Burst Time
  - Waiting Time for P_1=9-6=**9**; P_2=24-8=**16**; P_3=16-7=**9** ; P_4=3-3=**0**
  - Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

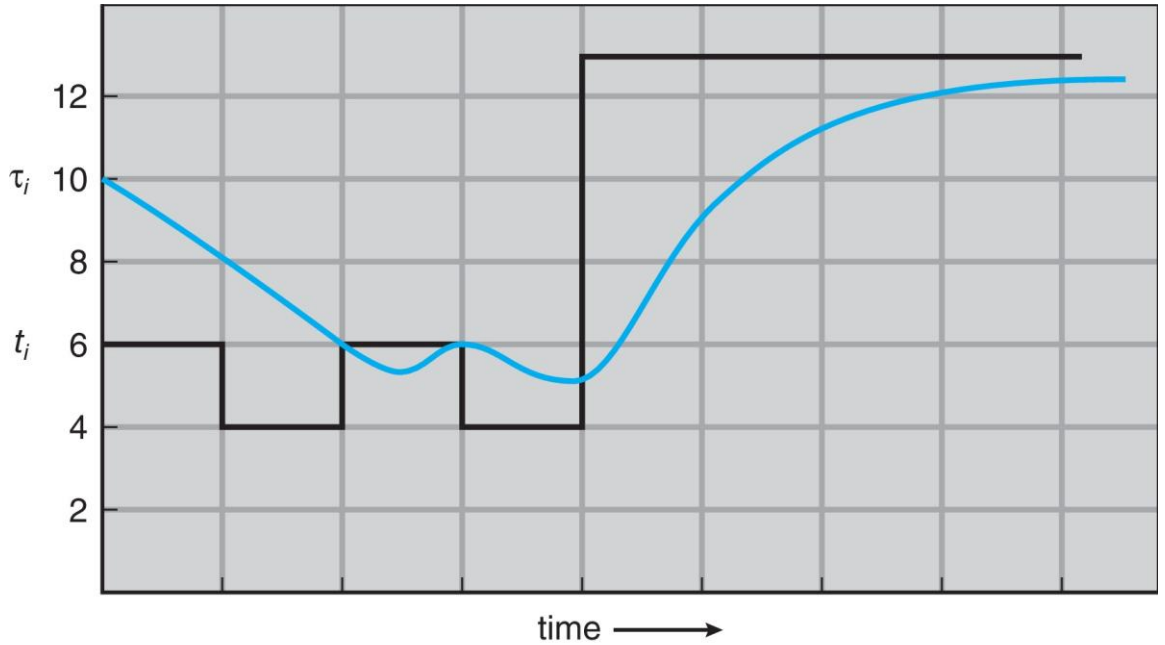
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

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

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

- Commonly, α set to ½

Prediction 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	...

Shortest Remaining Time First Scheduling

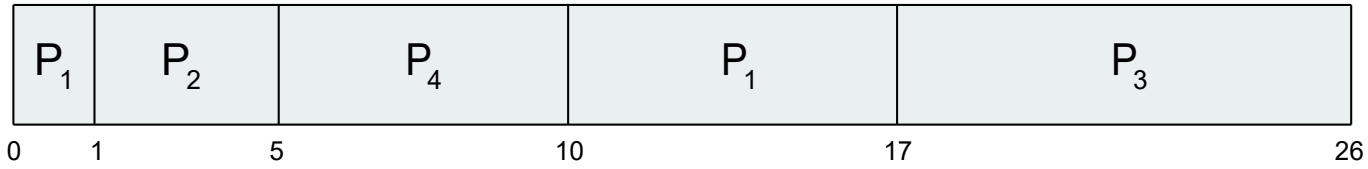
- Preemptive version of SJF
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm.
- Is SRT more “optimal” than SJF in terms of the minimum average waiting time for a given set of processes?

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



- **Turn-around Time** =Completion Time – Arrival Time
  - Turn-around Time for P_1=17-0=**17**; P_2=5-1=**4**; P_3=26-2=**24** ; P_4=10-3=**7**
  - Average Turn-around time = (17 + 4 + 24 + 7)/3 =12
- **Waiting Time** = Turnaround Time – Burst Time
  - Waiting Time for P_1=17-8=**9**; P_2=4-4=**0**; P_3=24-9=**15** ; P_4=7-5=**2**
  - Average waiting time = (9 + 0 + 15 + 2)/4 = 26/4 = 6.5

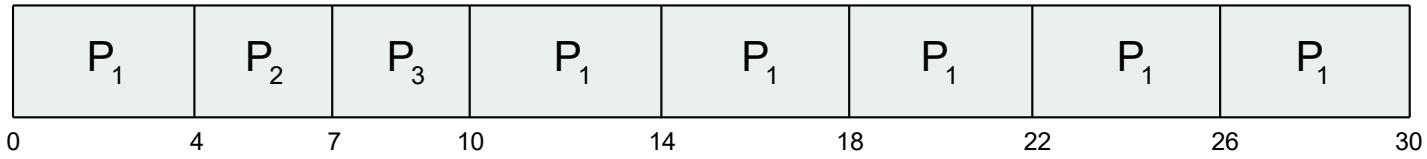
Round Robin (RR)

- 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.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO (FCFS)
  - q small $\Rightarrow$ RR
- Note that q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

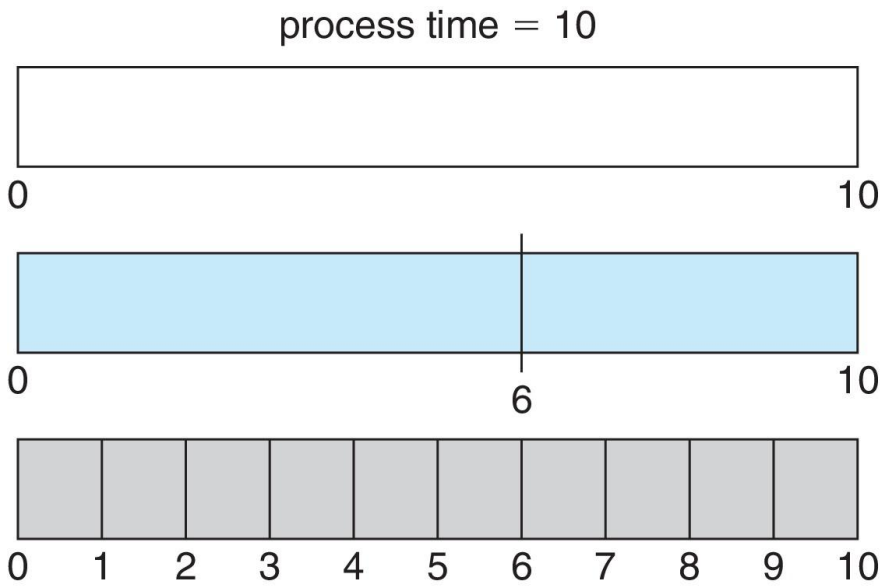
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



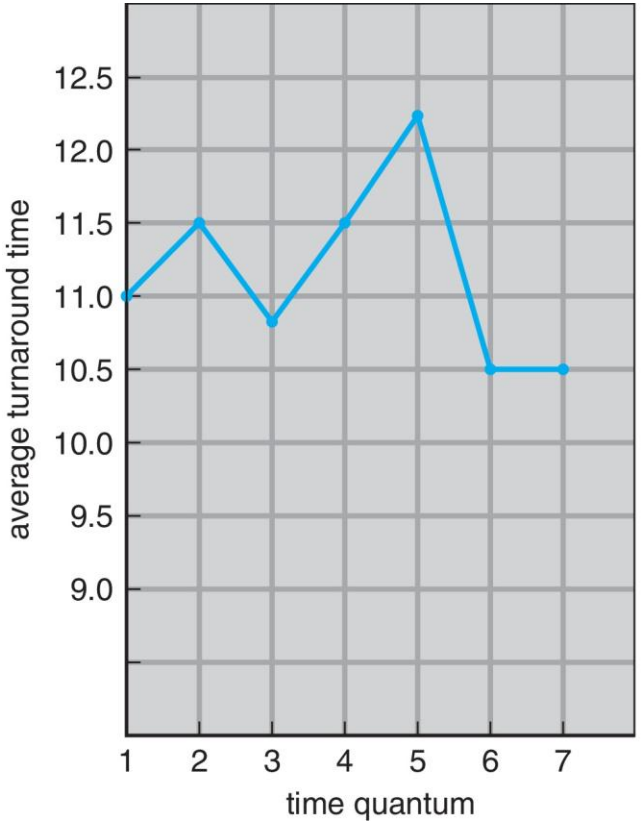
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Rule of thumb:
80% of CPU bursts should be shorter than q

Priority Scheduling

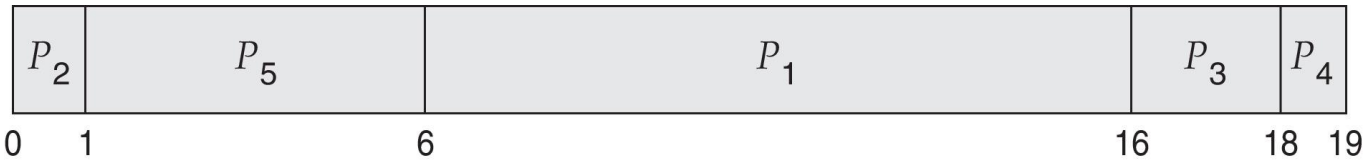
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡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

All processes are arrived at time t=0

- Priority scheduling Gantt Chart



- **Turn-around Time** =Completion Time – Arrival Time
  - Turn-around Time for P_1=16-0=**16**; P_2=1-0=**1**; P_3=18-0=**18** ; P_4=19-0=**19** ; P_5=6-0=**6**
  - Average Turn-around time = (16 + 1 + 18 + 19 + 6)/5 =12
- **Waiting Time** = Turnaround Time – Burst Time
  - Waiting Time for P_1=16-10=**6**; P_2=1-1=**0**; P_3=18-2=**16** ; P_4=19-1=**18** ; P_5=6-5=**1**
  - Average waiting time = (6 + 0 + 16 + 18 + 1)/5 = 41/5 = 8.2

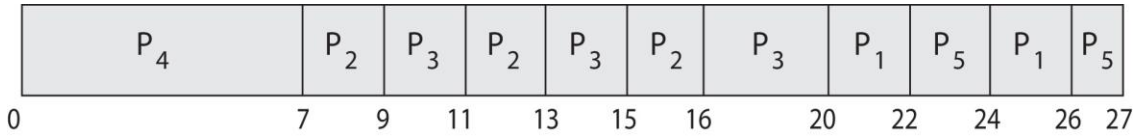
Priority Scheduling w/ Round-Robin

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

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

All processes are arrived at time t=0

Gantt Chart with time quantum = 2



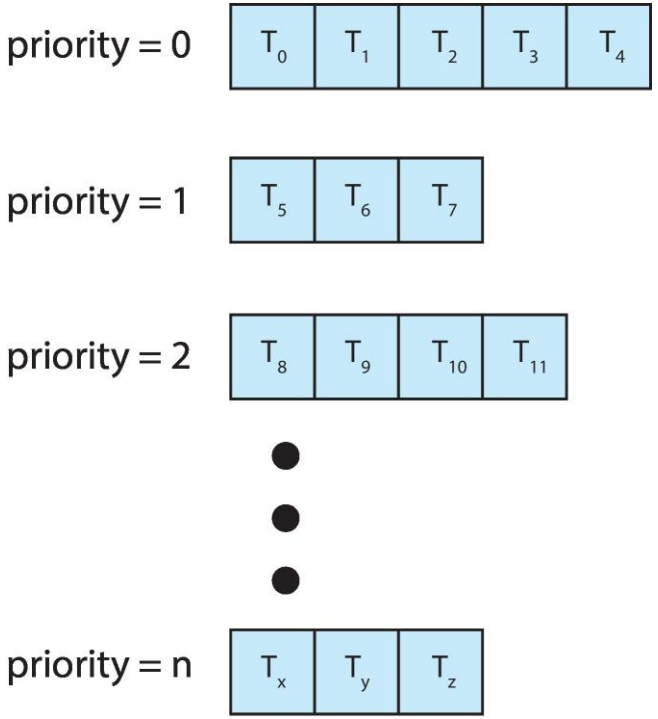
- **Turn-around Time** =Completion Time – Arrival Time
  - Turn-around Time for P_1=26-0=**26**; P_2=16-0=**16**; P_3=20-0=**20** ; P_4=7-0=**7** ; P_5=27-0=**27**
  - Average Turn-around time = (26 + 16 + 20 + 7 + 27)/5 =19,2
- **Waiting Time** = Turnaround Time – Burst Time
  - Waiting Time for P_1=26-4=**22**; P_2=16-5=**11**; P_3=20-8=**12** ; P_4=7-7=**0** ; P_5=27-3=**24**
  - Average waiting time = (22 + 11 + 12 + 0 + 24)/5 = 69/5 = 13,8

Multilevel Queue

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

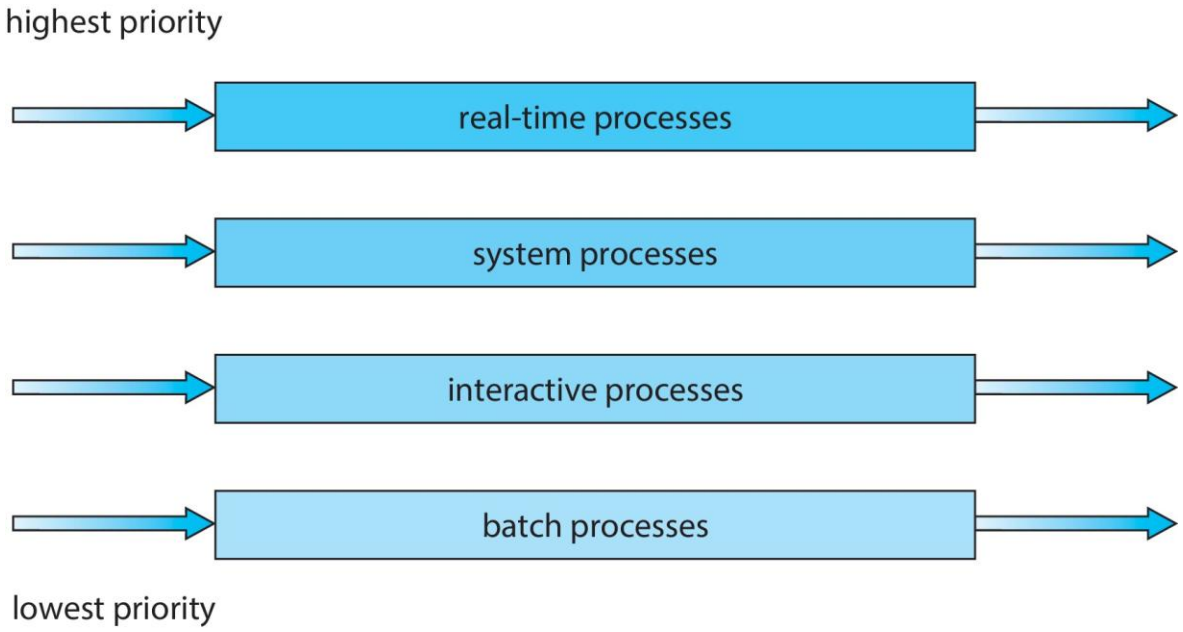
Multilevel Queue

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



Multilevel Queue

- Prioritization based upon process type

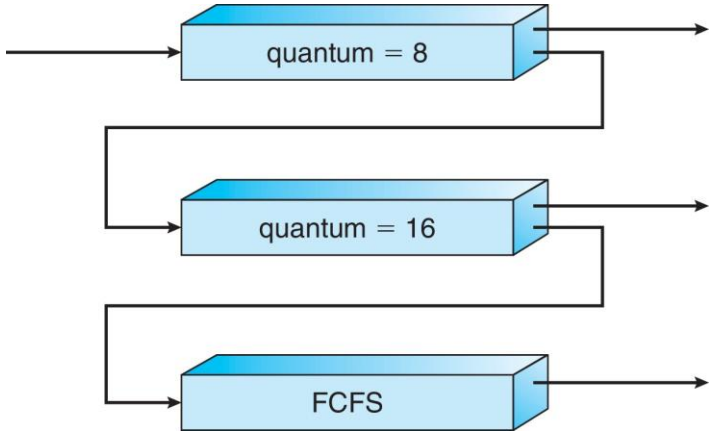


Multilevel Feedback Queue

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

Example of Multilevel Feedback Queue

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



Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **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

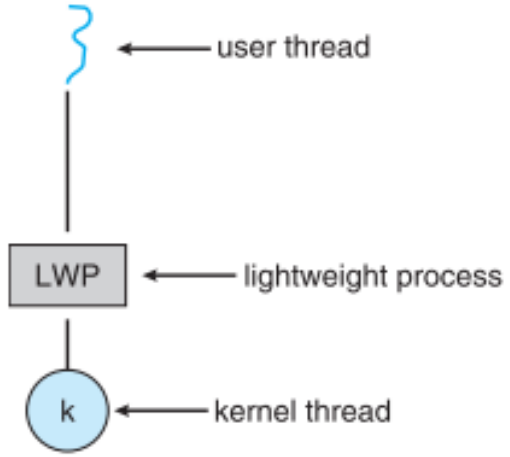


Figure 4.13 Lightweight process (LWP).

Pthread Scheduling

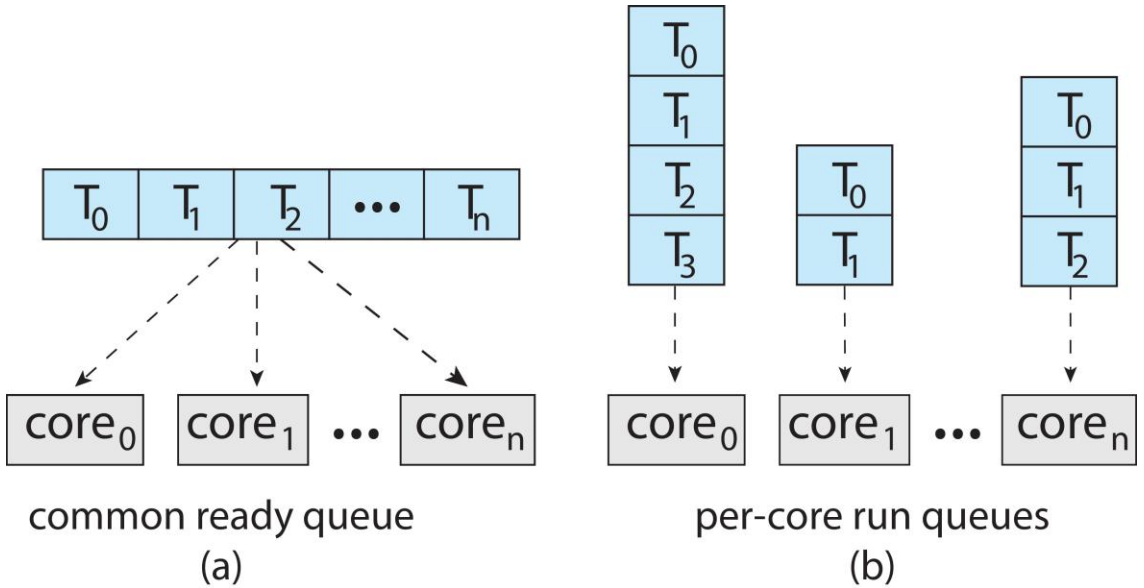
- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocessors may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

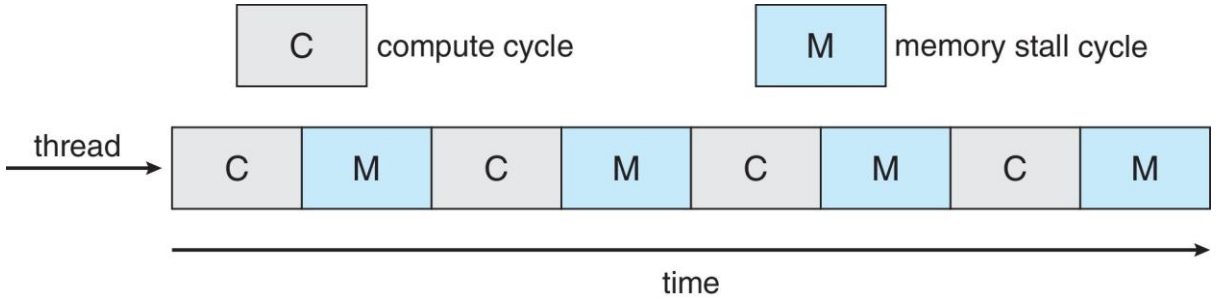
Multiple-Processor Scheduling

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



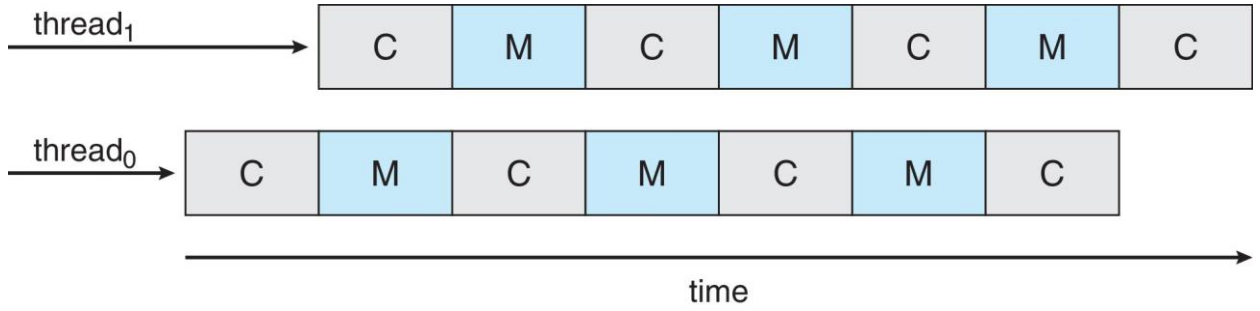
Multicore Processors

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



Multithreaded Multicore System

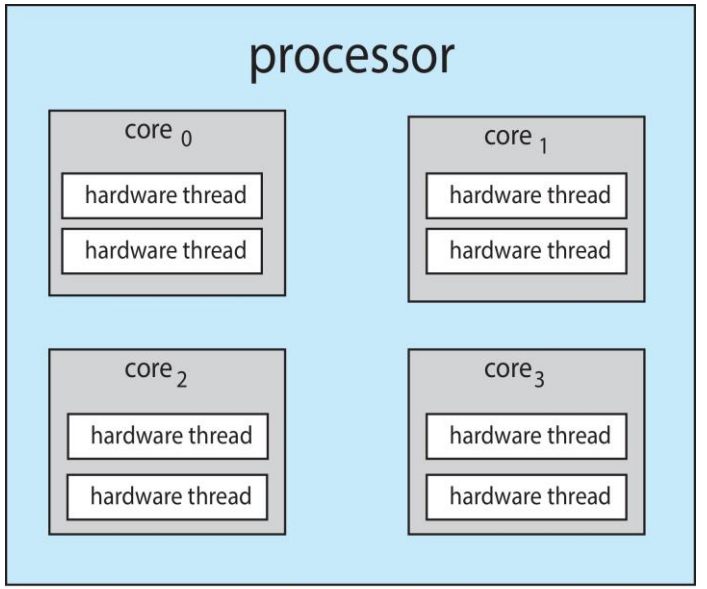
- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure

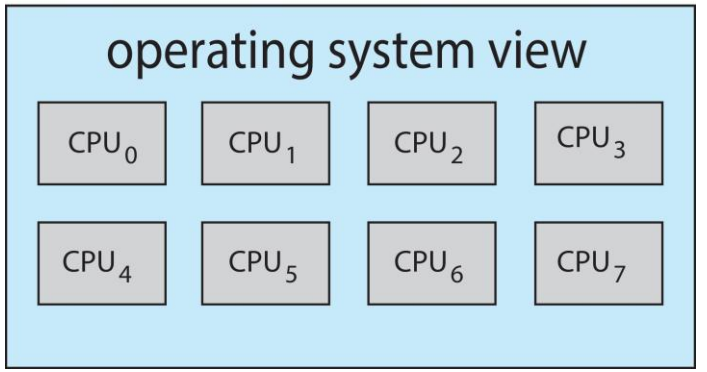


Multithreaded Multicore System

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

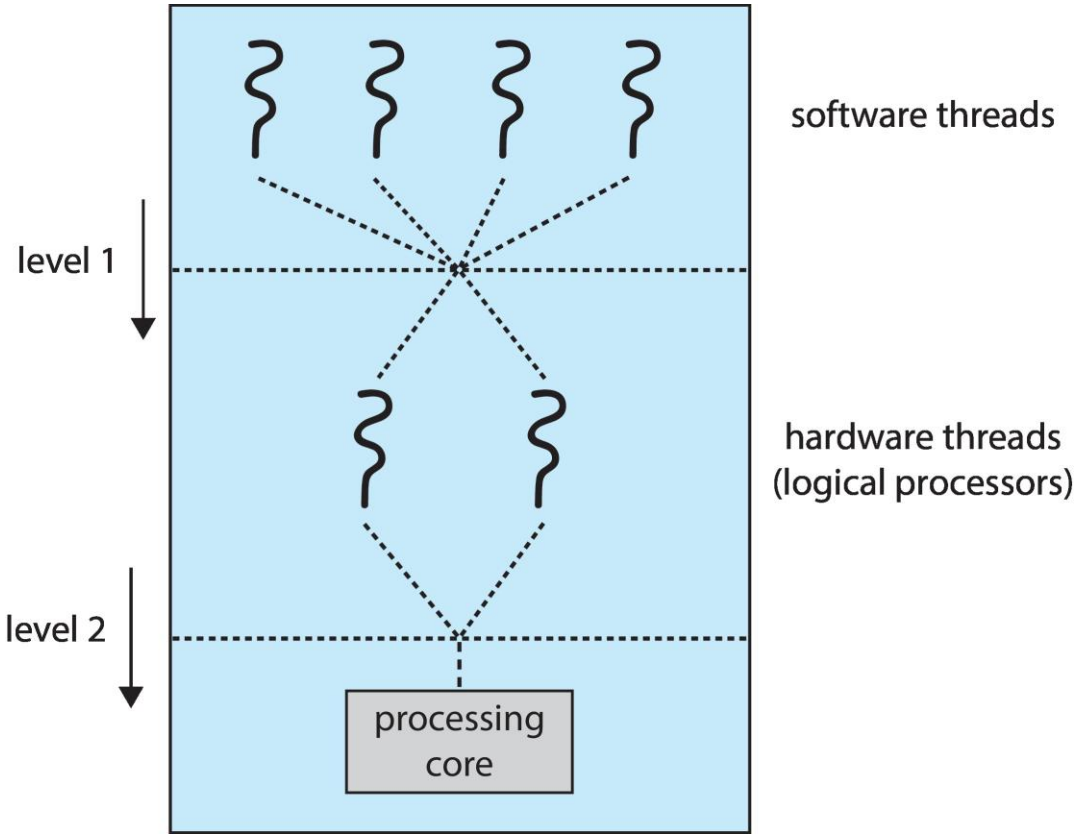
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.





Multithreaded Multicore System

- Two levels of scheduling:
  1. The operating system deciding which software thread to run on a logical CPU
  2. How each core decides which hardware thread to run on the physical core.



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed

Two approaches to load balancing:

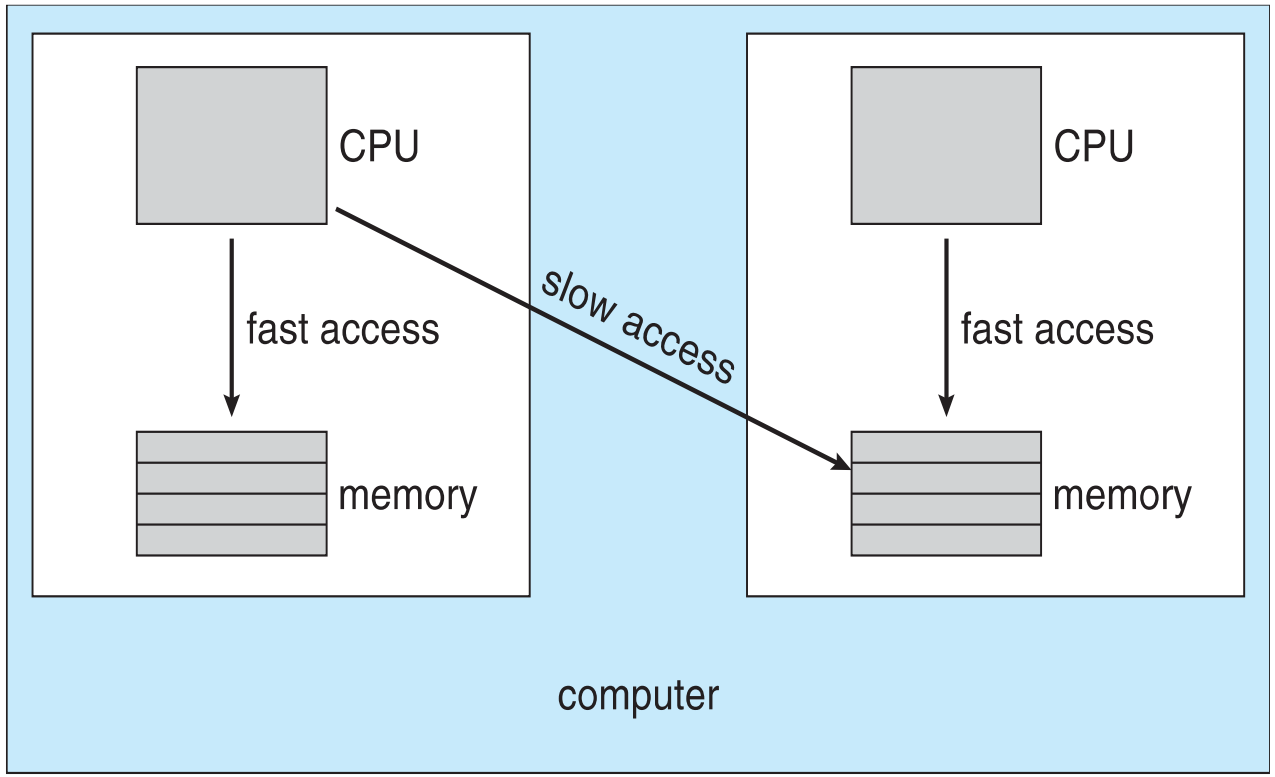
1. **Push migration** – a specific task periodically checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
2. **Pull migration** – idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the **cache** contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- **Load balancing may affect processor affinity** as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

NUMA and CPU Scheduling

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

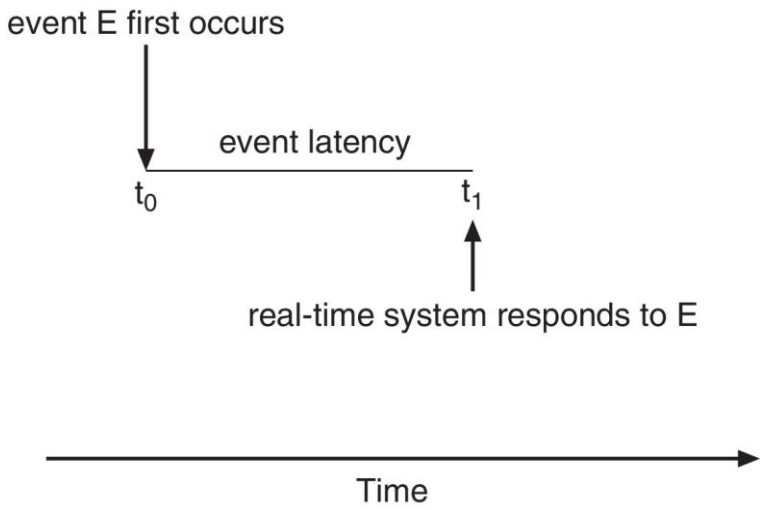


Real-Time CPU Scheduling

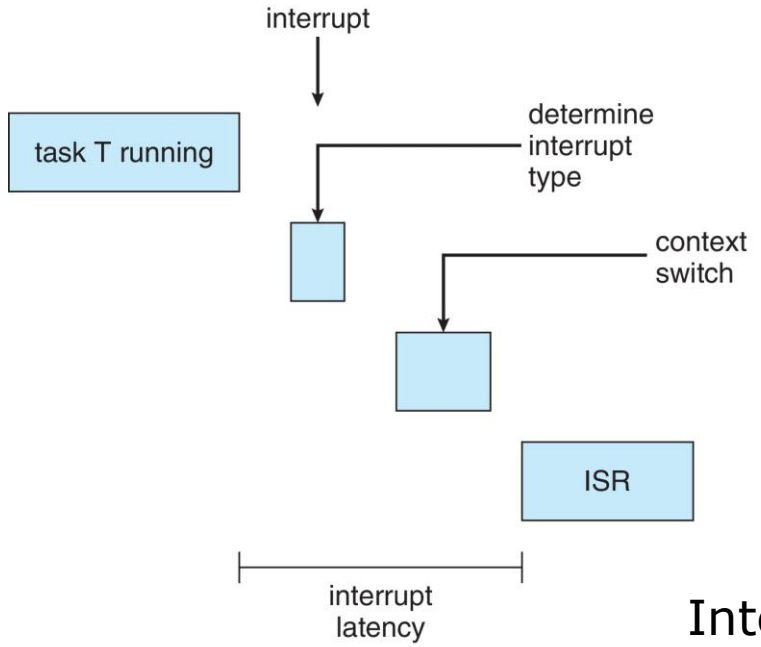
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems –** task must be serviced by its deadline

Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another



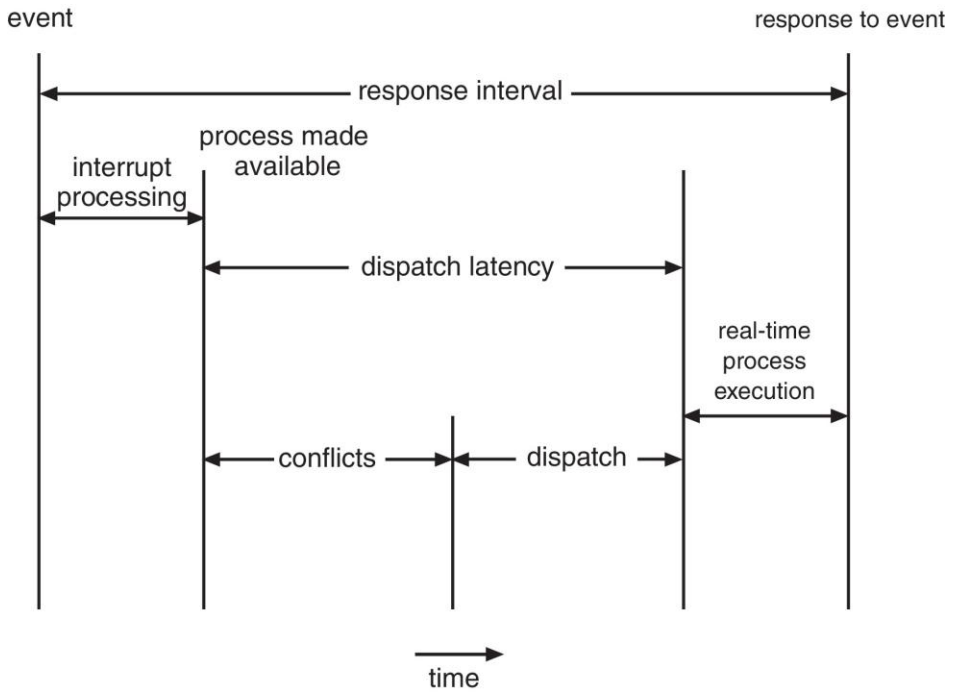
Event latency



Interrupt latency

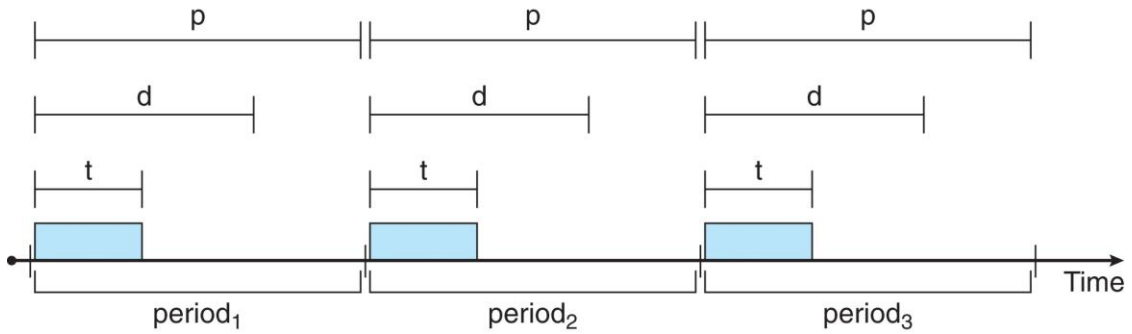
Dispatch Latency

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes



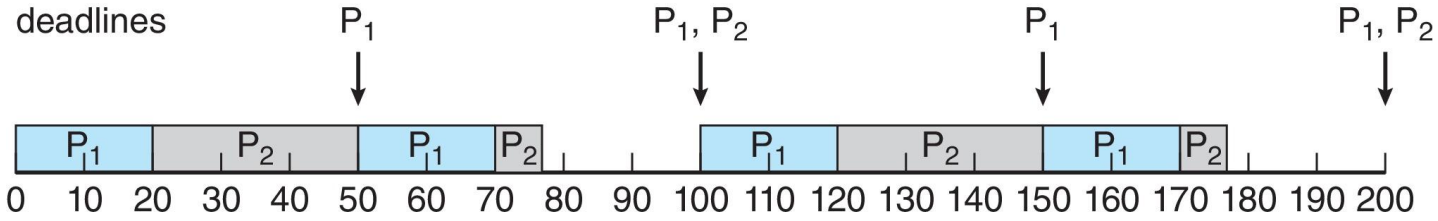
Priority-based Scheduling

- 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*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



Rate Monotonic Scheduling

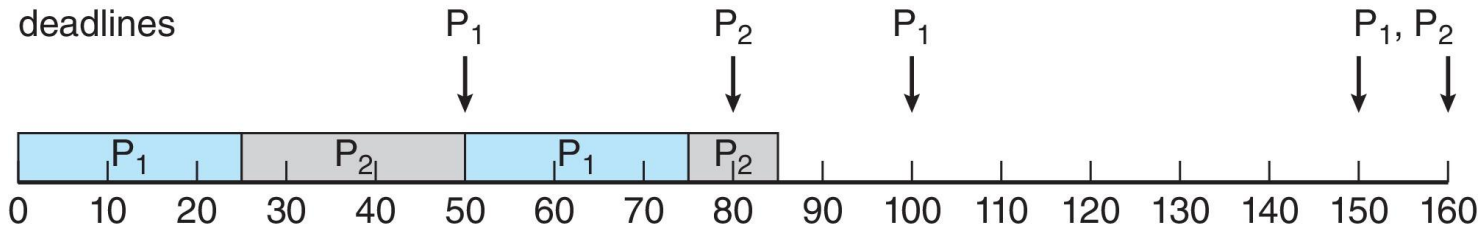
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



The periods & time for:
P1 = 50 , cpu-burst t1=20
P2 = 100, cpu-burst t2=35

Missed Deadlines with Rate Monotonic Scheduling

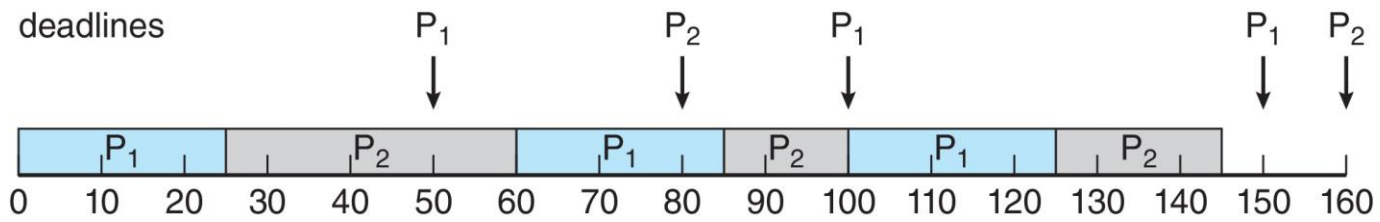
- Process P_2 misses finishing its deadline at time 80
- Figure



The periods & time for:
P1 = 50, cpu-burst t1=25
P2 = 80, cpu-burst t2=35

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
- Figure



The periods & time for:
P1 = 50, cpu-burst t1=25
P2 = 80, cpu-burst t2=35

Proportional Share Scheduling

- *T* shares are allocated among all processes in the system
- An application receives *N* shares of time where N < T
- This ensures each application will receive ***N*** / *T* of the total processor time

- Example:
  - Total of T = 100 shares, to be divided to three processes
    - A = 50 shares → 50% of total processor time
    - B = 15 shares → 15% of total processor time
    - C = 20 shares → 20% of total processor time
    - Total allocation = 85 shares of the total 100 shares
  - If a new process D requested 30 shares, the admission controller would deny D entry into the system.

Operating System Examples

- Linux scheduling
- Windows scheduling

Linux Scheduling in Version 2.6.23 +

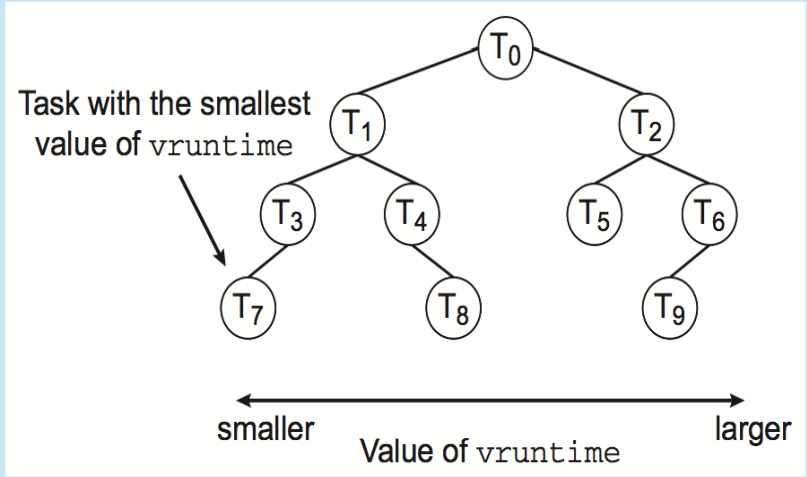
- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Two scheduling classes included, others can be added
    1. Default –using CFS scheduling algorithm
    2. real-time

Completely Fair Scheduler

- The CFS scheduler assigns a proportion of CPU processing time to each task
- This CPU time proportion is calculated based on **nice value** from -20 to +19
  - Lower value is higher priority and has shorter proportion of CPU processing time
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Vruntime initial value

Scenario:
a CPU's run queue (the list of tasks waiting to be run)

Thread A: 100ms
Thread B: 150ms
Thread C: 120ms

Thread A creates a new thread, and Thread D is added.

the vruntime values are:
- Task A: 100ms
- Task B: 150ms
- Task C: 120ms
- Task D: 100ms (initial value)

Vruntime formula:

$$\text{vruntime} = \sum_i \Delta\text{vruntime}_i$$

The `vruntime` increment is calculated using the following formula:

$$\Delta\text{vruntime} = \Delta t \times \frac{\text{load\_weight}_{\text{nice}=0}}{\text{load\_weight}_{\text{task}}}$$

where:

- Δt is the actual time the thread ran since the last update.
- $\text{load\_weight}_{\text{nice}=0}$ is the load weight for nice value 0 (typically 1024).
- $\text{load\_weight}_{\text{task}}$ is the load weight for the task's nice value.

Nice	weight	Nice	weight
-20	88761	0	1024
-19	71755	1	820
-18	58593	2	820
-17	47872	3	687
-16	39063	4	687
-15	31250	5	570
-14	31250	6	570
-13	25000	7	478
-12	25000	8	478
-11	20313	9	390
-10	20313	10	390
-9	16250	11	312
-8	16250	12	312
-7	12500	13	250
-6	12500	14	250
-5	10156	15	203
-4	10156	16	203
-3	8125	17	162
-2	8125	18	162
-1	6250	19	125

The thread D value is the same as the smallest value in the existing runnable processes

Vruntime Calculation

Initial vruntime:

- Task A : Initial vruntime = 100ms, actual runtime= 50, (nice = 0 → weight = 1024)
- Task B: Initial vruntime = 150ms, actual runtime= 100,(nice = 10 → weight = 390)
- Task C: Initial vruntime = 120ms, actual runtime= 150,(nice = -15 → weight = 31250)
- Task D: Initial vruntime = 100ms, actual runtime= 200,(nice = -20 → weight = 88761)

Form

$$\Delta\text{vruntime} = \Delta t \times \frac{\text{load\_weight}_{\text{nice}=0}}{\text{load\_weight}_{\text{task}}} \qquad \text{vruntime} = \sum_i \Delta\text{vruntime}_i$$

Example:

$\Delta\text{Vruntime}_A$ = 50 x 1024/1024 = 50 ms → VruntimeA = 100 + 50 = 150ms

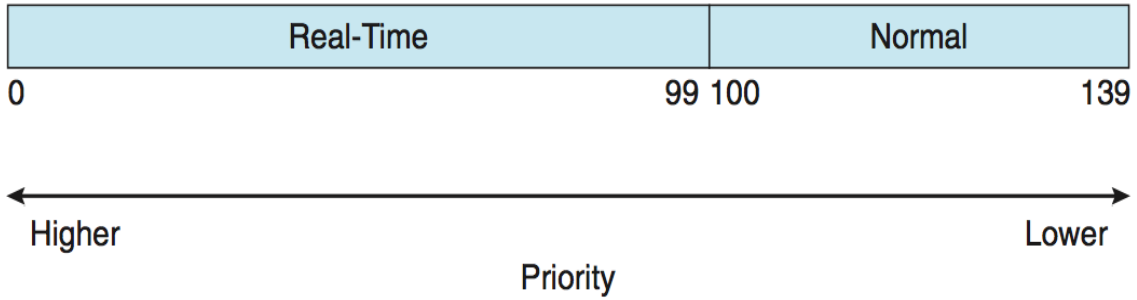
$\Delta\text{Vruntime}_B$ = 100 x 1024/390 = 262,56ms → VruntimeB = 150 + 262,56= 412,56ms

$\Delta\text{Vruntime}_C$ = 150 x 1024/31250 = 4,92 ms → VruntimeC = 120 + 4,92 = 124,92ms

$\Delta\text{Vruntime}_D$ = 200 x 1024/88761 = 2,3ms → VruntimeD = 100+ 2,3= 102,3ms

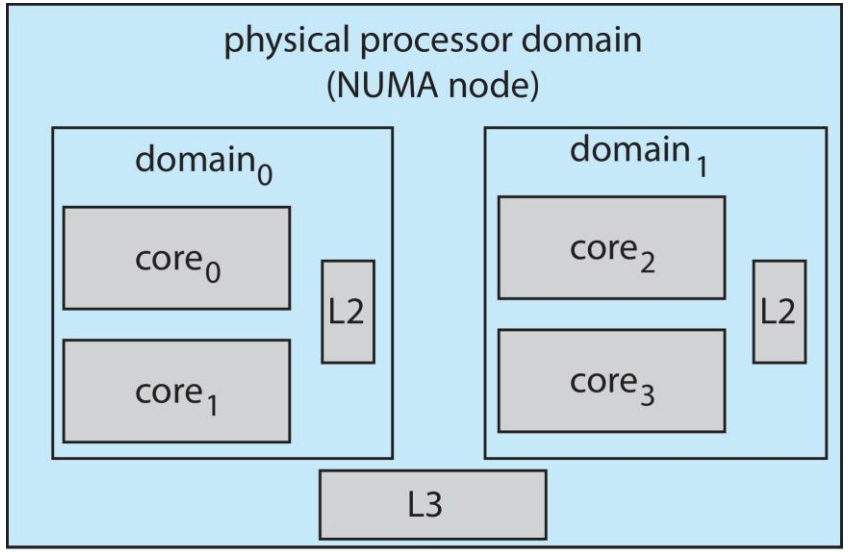
Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e., cache memory.) Goal is to keep threads from migrating between domains.



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

	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 Priorities

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

	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 Priorities

End of Chapter 5