Bilkent University
Department of Computer Engineering
CS342 Operating Systems

CPU Scheduling

Last Update: Mar 17, 2023

Objectives and Outline

Outline

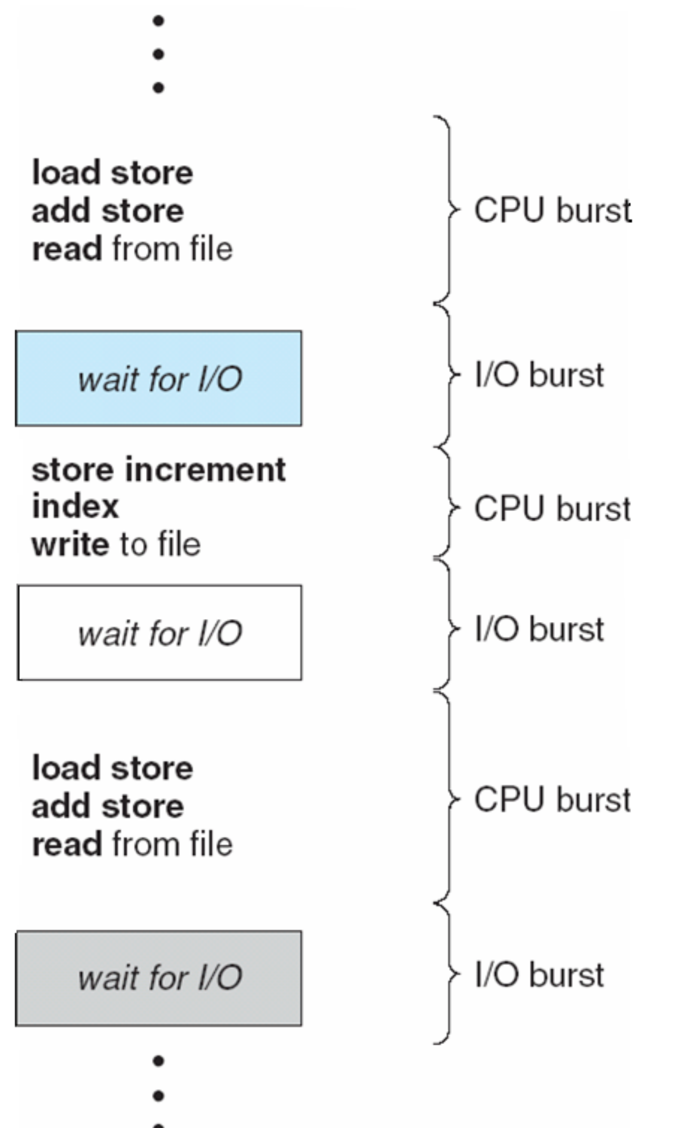
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objective

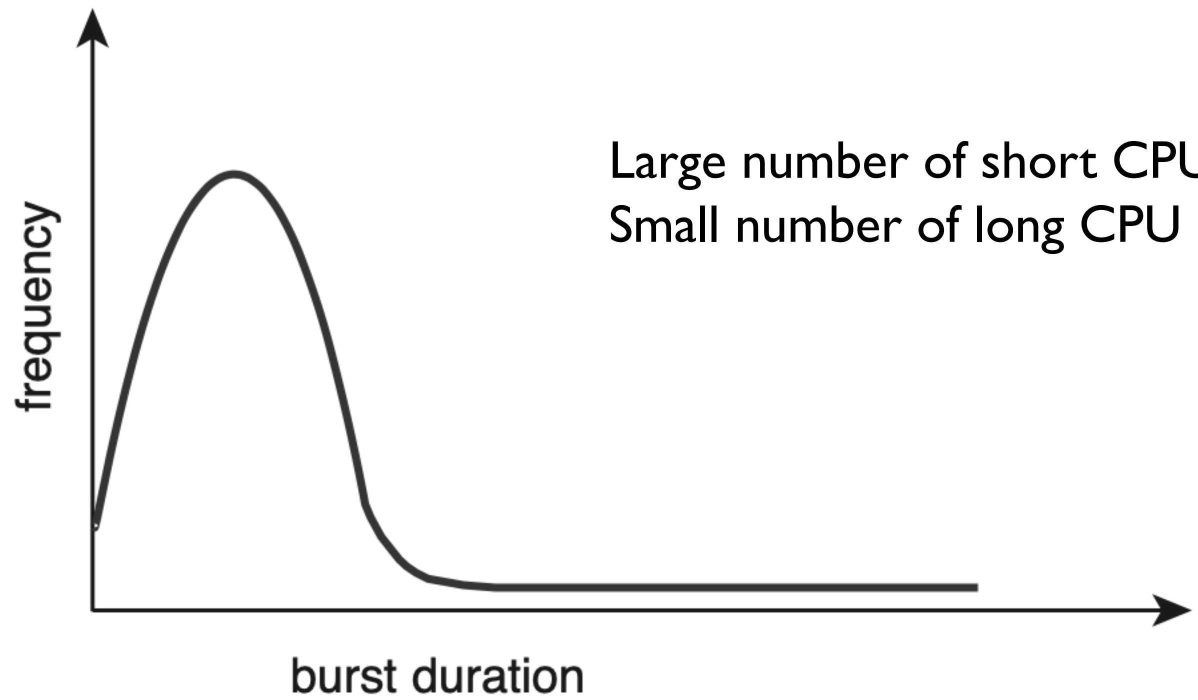
- To introduce CPU scheduling, which is the basis for multi-programmed (multi-tasking) operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming.
- Process execution usually consists of a cycle of CPU execution and I/O wait.
- CPU–I/O Burst Cycle repeated.



Histogram of CPU-burst Times



Large number of short CPU bursts
Small number of long CPU bursts

CPU burst distribution

CPU Scheduler

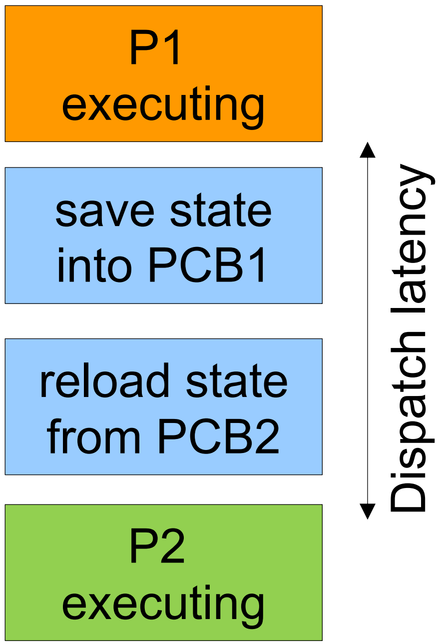
- Selects from among the processes (threads) in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (e.g., system call)
  2. Switches from running to ready state (upon interrupts)
  3. Switches from waiting to ready (e.g., I/O completed)
  4. Terminates
- Scheduling under 1 and 4 is nonpreemptive
- All other scheduling is preemptive

CPU Scheduler: kernel design

- Virtually all modern operating systems are preemptive.
  - A process is not allowed to run as long as it wishes.
- A preemptive system is more responsive and suitable for real-time tasks.
- But preemptive scheduling may cause race conditions among processes/threads sharing data.
    - A context switch can happen at any moment (not only when waiting for I/O)
    - A process may be in middle of updating a shared data.
- Preemption effects kernel design.
  - A process running in kernel mode (as a result of a system call or interrupt) may be in middle of updating a shared kernel structure.
  - Kernel should use locks, etc., to avoid race conditions.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
  - switching context from one process to the other
  - 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.



Linux tools

- Assume you have a process started with pid x.
- Type at command shell:
  cat /proc/x/status
- At the bottom of the output you will see the number of context switches that the process experiences. Like the following:

```
voluntary_ctxt_switches: 9
nonvoluntary_ctxt_switches: 18
```

- The vmstat command can give statistics about context switches as well.

Scheduling Criteria

Many criteria exist for comparing scheduling algorithms.

- 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

  Turn. Time = Finish time (completion time) – Start time (arrival time)

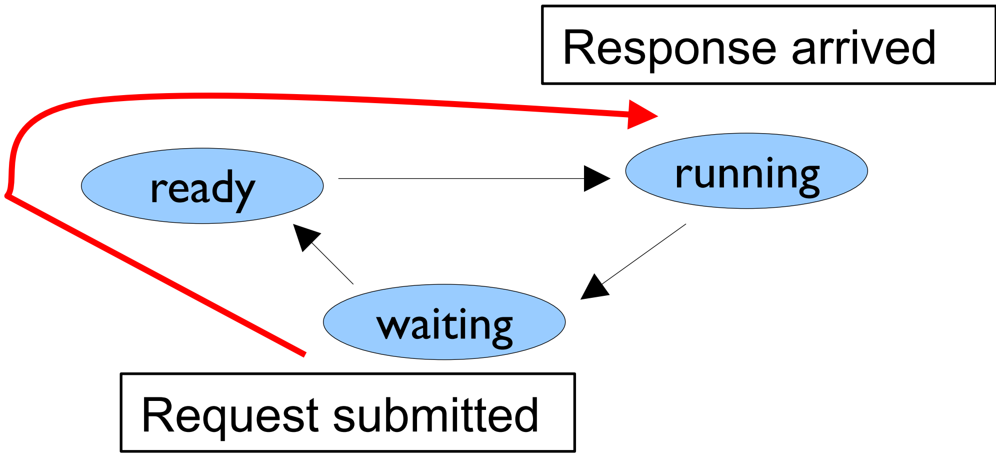
- Waiting time: *total* amount of time a process has waited in the ready queue (in ready state)

- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time

Scheduling Criteria

- Response time: amount of time it takes from when a request was submitted until the first response is produced (for interactive systems)
  - In other words, elapsed time between the arrival to ready queue and the first execution in the CPU.

Important for Interactive Processes (interacting with a user)



We want to minimize response time

Scheduling Algorithms

Scheduling algorithms

- We will describe several algorithms.
- We will illustrate their operation and calculate metrics.
  - Several processes are considered.
- An accurate illustration should consider many CPU and I/O bursts for each process.
- For simplicity, for each process, a single CPU burst is considered.
- Basic metric used: waiting time
- Algorithms are discussed assuming system has one core (CPU).
  - Later, we will discuss scheduling issues in multicore (multiprocessor) systems.

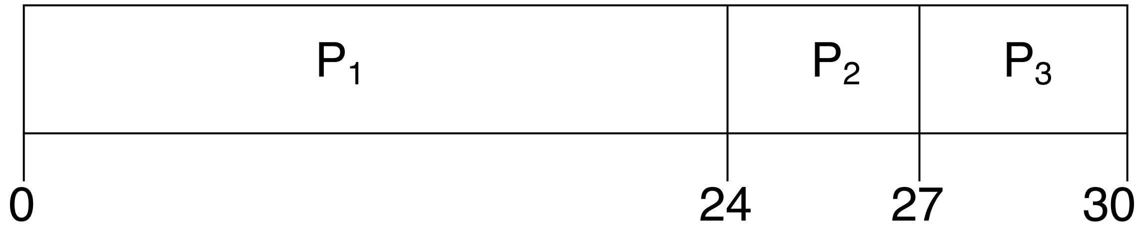
First-Come, First-Served (FCFS) Scheduling

- Simple scheduling algorithm.
- Process that requests the CPU first, is allocated the CPU first.
- FIFO queue can be used in its implementation.
- The PCB of the process that becomes ready is added to the tail of the queue.
- When a running process finishes its CPU burst (by terminating or by doing a I/O call and going into waiting state), it is removed from the CPU and ready queue. Then the head of the queue is allocated the CPU.
- Code of FCFS is simple.
- Time to wait till next execution can be quite long.

First-Come, First-Served (FCFS) Scheduling

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

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



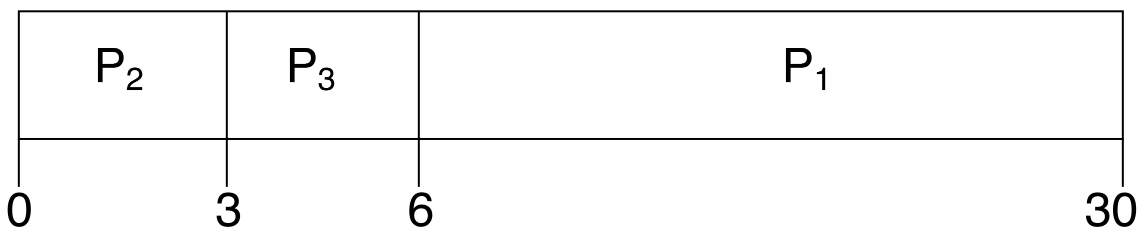
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 =$ 17 ms

FCFS Scheduling (Cont)

Suppose that the same processes arrive at time 0 in the following 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: $(6 + 0 + 3)/3 = 3$ ms
- Much better than previous case

FCFS Scheduling (Cont)

- The average waiting time in FCFS is usually not minimal.
- The average waiting time can be vary substantially if CPU burst times of processes vary greatly.
- FCFS is nonpreemptive. A process is allowed to run until it finishes its CPU burst (which will happen when process requests an I/O operation or terminates).
- Causes convoy effect: short processes behind long process
  - Results in low CPU and device utilization.
  - Short CPU burst processes are I/O bound.
  - Long CPU burst processes are CPU-bound.
- Not good for interactive systems.

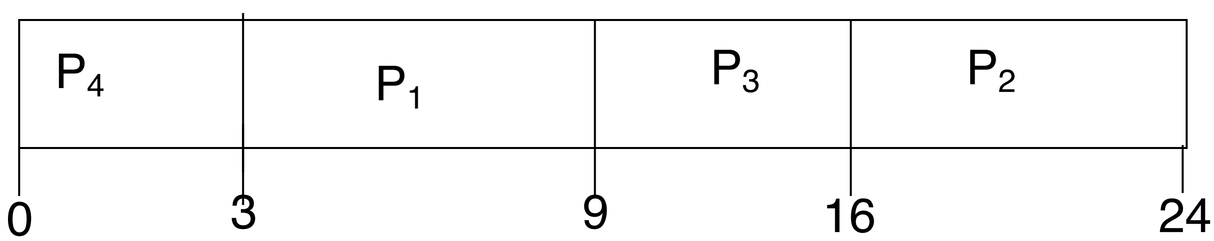
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time.
- When CPU becomes available, the process that has the smallest next CPU burst is allocated the CPU.
    - A better name would be *shortest-next-cpu-burst* algorithm.
        - Since scheduling depends on next cpu burst of a process, rather than its total length.
- If there is a tie (two processes having the same next shortest CPU burst), the tie can be broken by applying FCFS.

Example of SJF

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	0.0	8
P_3	0.0	7
P_4	0.0	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7 ms

Shortest-Job-First (SJF) Scheduling

- SJF is optimal for waiting time: gives minimum average waiting time for a given set of processes.
- Moving a short process before a long process (exchanging these processes in the ready list) decreases the waiting time of the short process more than it increases the waiting time of the long process.
  - Therefore the average waiting will decrease with each such exchange.
- The difficulty in implementing SJF is knowing the length of the next CPU burst for each process.
- But we can approximate the SJF scheduling.
  - Predict the length of the next CPU burst.
  - We expect next CPU burst to be similar in length to the previous CPU burst (or bursts).

Determining Length of Next CPU Burst

- We can predict (estimate) the length of the next CPU burst.
- Prediction can utilize the the lengths of the previous CPU bursts of a process.
- We can use exponential averaging of previous burst lengths as the prediction method.
- We can keep a running exponential average:
    - Updated after each CPU execution (CPU burst) completed.
- We need to do this for each process.
    - When the current CPU burst finishes (for a process), we need to update the prediction (running exp avg) (of that process).
- When scheduling is needed, we select the process that has smallest predicted value (smallest running exp avg).

Predicting the Length of next CPU Burst: Exponential Averaging

- Let t_n denotes the length of the n^{th} CPU burst.
  - This is the actual length (known after burst finishes)
- Let τ_{n+1} denote the predicted value for the next CPU burst (i.e., the new avg).
- Assume the first CPU burst is Burst_0 and its length is t_0.
- τ_n denotes the previous estimate (i.e., the old avg) (before t_n)
- Define α (a weight factor) to be: $0 < \alpha < 1$.
- Then we define the new avg:

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

Exponential Averaging

- We have CPU bursts as: Burst_0 Burst_1, Burst_2, …, Burst_n, Burst_{n+1}.
- The actual lengths of these bursts are denoted by: t_0, t_1, t_2, t_3, …., t_n, t_{n+1}. Let τ_0 be initial estimate (i.e., estimate for Burst_0) and let it be a constant value like $10\ ms$. Then

$$\tau_1 = \alpha t_0 + (1-\alpha)\tau_0$$

- If we expand the formula, we get:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + \cdots$$
$$+(1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^n \alpha t_0 + (1-\alpha)^{n+1} \tau_0$$

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

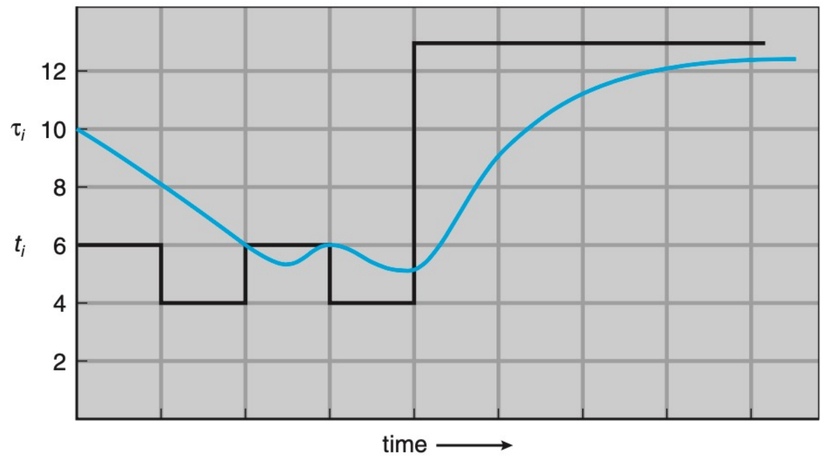
Exponential Averaging

- If $\alpha = 0$:
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

- If $\alpha = 1$:
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts.

- Usually we have α between 0 and 1, for example 0.5.

Exponential Averaging



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

An exponential average with $\alpha = 1/2$ and $\tau = 10$ ms.

Example

- τ_0 = 10 ms. Measured CPU bursts: t_0 = 8ms, t_1 =16ms, t_2 =20ms, t_3 =10ms
- Assume α = ½. Then we have:
  - τ_1 = ½ x 8 + ½ x 10 = 9
  - τ_2 = ½ x 16 + ½ x 9 = 12.5
  - τ_3 = ½ x 20 + ½ x 12.5 = 16.25
  - τ_4 = ½ x 10 + ½ x 16.25 = 13.125.
- The next CPU burst length is estimated to be 13.125 ms. After that burst has completed, it is measured as t_4.
- We update τ for a process, when its burst has completed.
- Hence we will maintain up-to-date τ value for each process. We will select the process with minimum τ.

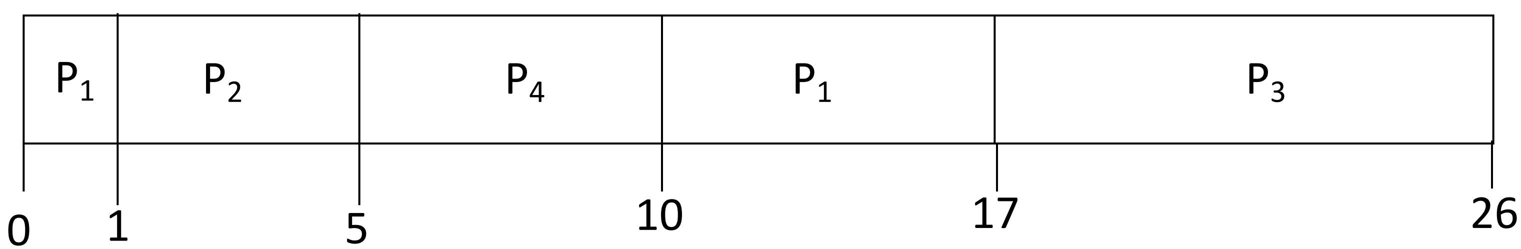
Shortest Remaining Time First (SRTF)

- Preemptive version of SJF.
- While a job A is running, if a new job B comes whose length is shorter than the remaining time of job A, then B preempts A and B runs.

Shortest Remaining Time First (SRTF)

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	1.0	4
P_3	2.0	9
P_4	3.0	5

- SRJF scheduling chart



- Average waiting time = (9 + 0 + 2 + 15) / 4 = 6.5 ms

Example

- Assume we have the following processes. Find out the finish time, waiting time and turnaround time of each process for the following scheduling algorithms: FCFS, SJF, SRTF.

Process	Arv time	CPU Burst
A	0	30
B	5	20
C	10	12
D	15	10

Example

FCFS: Processes will run in the order they arrive.
The following is the finish, turnaround, waiting time of each process.

	Arv	Burst	Finish	Turnaround	Waiting
A	0	30	30	30	0
B	5	20	50	45	25
C	10	12	62	52	40
D	15	10	72	57	47

Example

SJF: running order will be: A(30) D(10) C(12) B(20)

	Arv	Burst	Finish	Turnaround	Waiting
A	0	30	30	30	0
B	5	20	72	67	47
C	10	12	52	42	30
D	15	10	40	25	15

Example

SRTF: running order will be: A(5) B(5) C(12) D(10) B(15) A(25)

	Arv	Burst	Finish	Turnaround	Waiting
A	0	30	72	72	42
B	5	20	47	42	22
C	10	12	22	12	0
D	15	10	32	17	7

Round Robin (RR) Algorithm

- Each process is allowed to run a small unit of time (called *time quantum* or *time slice*) in CPU.
- Time quantum (time slice) is usually a value between 10 and 100 ms.
- Then another process is run.
- That means a process is not allowed to execute in CPU as long as it wishes (i.e., until its CPU burst completes).
  - The running process can be forcefully taken from CPU (preempted) if its time quantum (time slice) finishes, even though the CPU burst of the process did not finish.

Round Robin (RR) Algorithm

- Can be implemented by using a queue
  - The ready processes are put in a queue.
- The head of the queue is selected to run next (when scheduling needed).
- A process that has run and finished its time quantum is put to the tail of the queue. Preemptive scheduling.
- A new arriving process (a new process or a process completing an I/O wait) is added to the tail of the queue.
- A process that finishes its CPU burst will be no longer be in the queue (has terminated or is waiting for something).
- When a process finishes its CPU burst before its time quantum expires, again another process is scheduled immediately.

Round Robin (RR) Algorithm

- 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 (response time)

- Performance
  - q large => behaves like FIFO
  - q small => good CPU sharing, good response time

  q must be large with respect to context switch time, otherwise overhead is too high.

Example of RR with time quantum = 4 time units

Process	Burst Time
P_1	24
P_2	3
P_3	3

All processes arrived at time 0.

- 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 times:
P_1: 10-3 = 6 ms
P_2: 4-0 = 4 ms
P_3: 7-0 = 7 ms
Avg = 17/3 = 5.66 ms

- Typically, RR has higher average turnaround time than SJF and FCFS, but is has better response time.

Example

Process	Arrival Time (ms)	CPU Burst Length (ms)
A	0	40
B	15	25
C	25	30
D	35	45
E	55	25

Finish time of each process?

a) Round Robin $q = 30$

b) Round Robin $q = 10$

Example

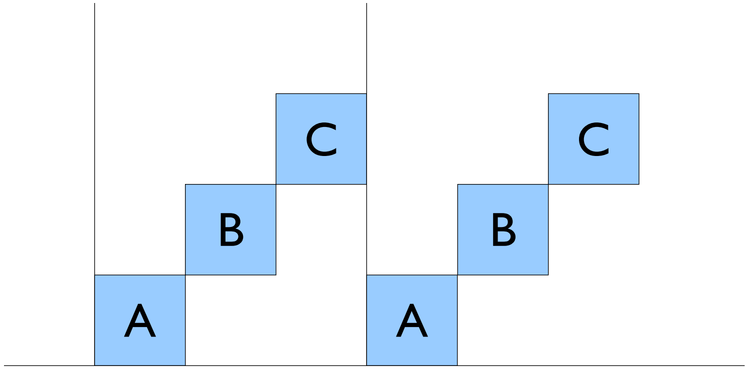
Solution

	RR q=30	RR q=10
A	95	80
B	55	105
C	85	125
D	165	165
E	150	150

RR vs FCFS

- Round Robin is good for fast response (small response time), but not for small turnaround time.

Assume 3 jobs all arrived at time 0. Each has a CPU burst = 10 ms



RR q=5 ms

Turnaround times

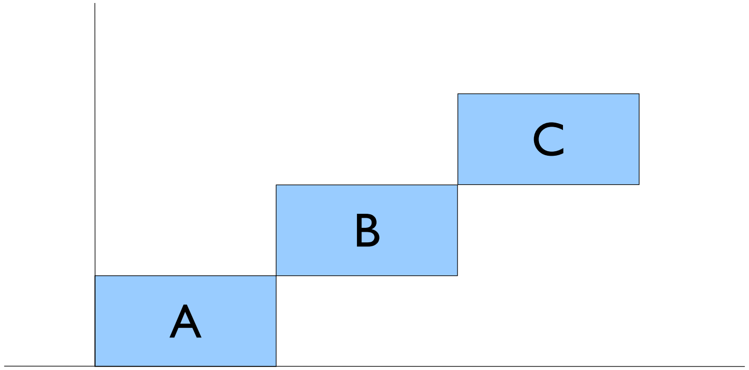
A: 20 ms

B: 25 ms

C: 30 ms

Avg TT = 25 ms

Worst response time = 10 ms



FCFS

Turnaround times

A: 10 ms

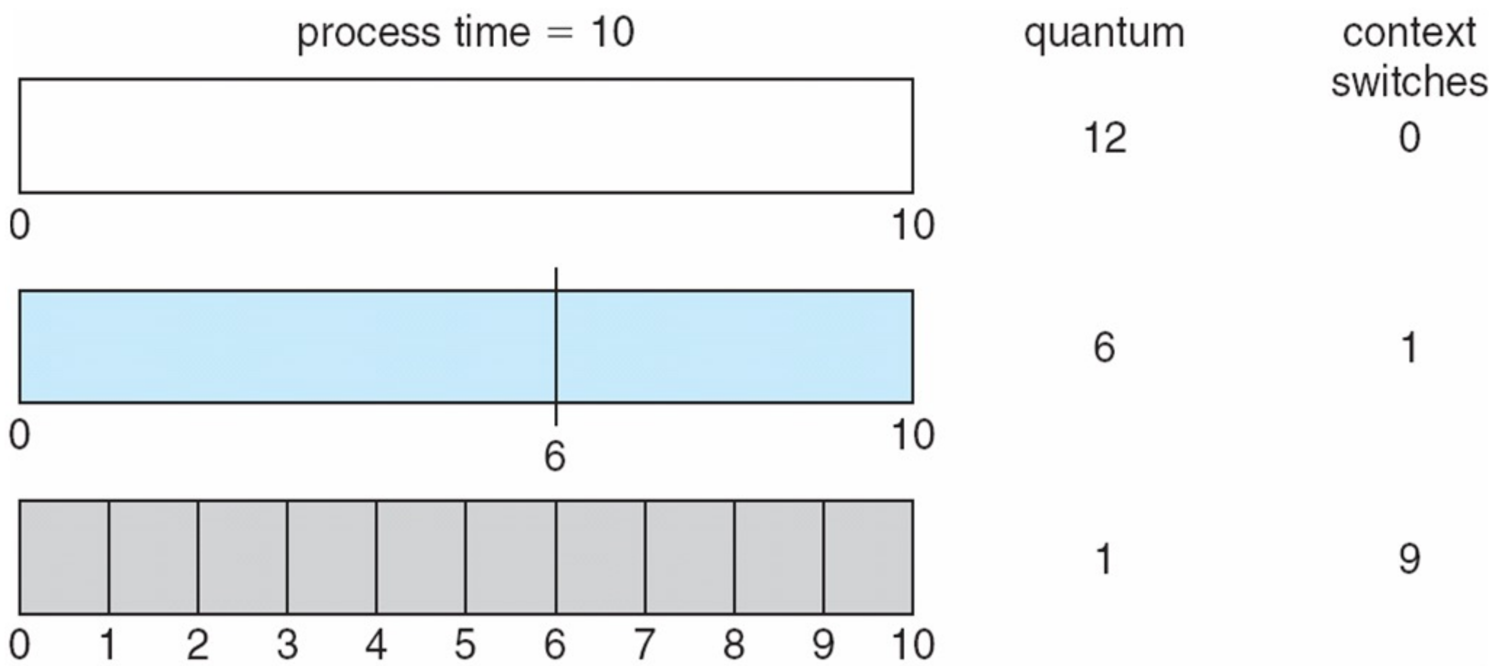
B: 20 ms

C: 30 ms

Avg TT = 20 ms

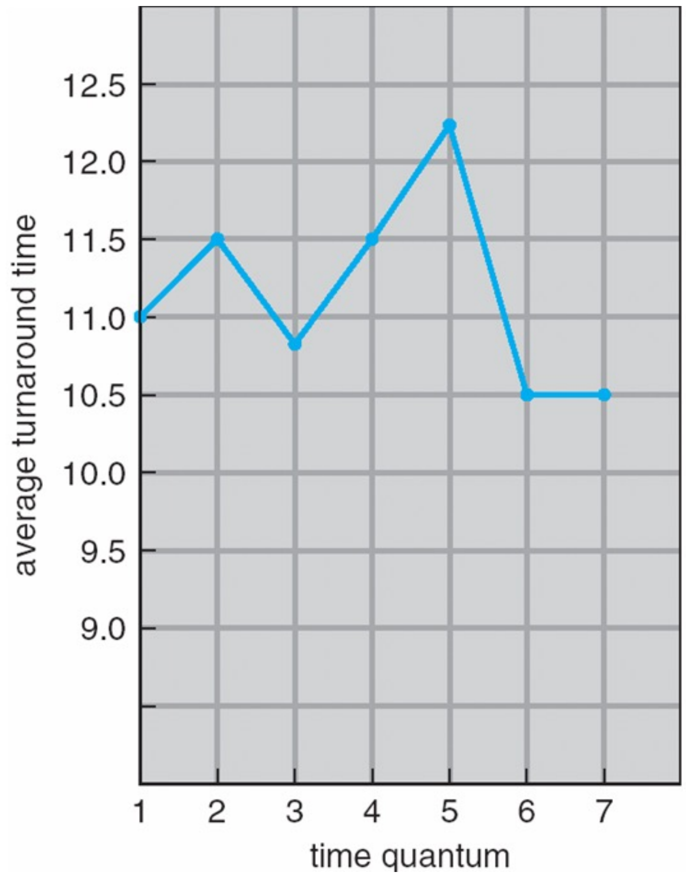
Worst response time = 20 ms

Time Quantum and Context Switches



If time quantum (q) is small, a process experiences more context switches during a CPU burst execution. Hence we will have more overhead.

Average turnaround time varies with time quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

When time quantum (q) gets larger, average turnaround time gets smaller. Exceptions can occur for particular values of burst lengths and time quantum.

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority.
  - Equal priority processes are scheduled in FCFS order.
- Convention: the *smaller* the priority number, the *higher* the priority, or vice versa.
- Has two versions:
  - Preemptive (higher priority process preempts the running one)
  - Non-preemptive
- SJF is a priority scheduling (non-preemptive) where priority is the *predicted next CPU burst time*.
- SRTF is preemptive priority scheduling.

Example

	Arv	CPU burst	Priority
A	0	20	3
B	5	15	2
C	10	20	0
D	25	15	1
E	30	20	1

Nonpreemptive priority scheduling:

AAAACCCCDDDEEEEBBB

assuming each letter is 5 time units

Finish times: A: 20, B: 90, C: 40, D: 55, E: 75

Preemptive priority scheduling:

ABCCCCDDDEEEEBBAAA

Finish times: A: 90, B: 75, C:30, D: 45, E: 65

In case of a tie, we can choose the one that is early in the queue (FCFS).

Priority Scheduling

- Problem: Starvation (indefinite blocking).
  - Low priority processes may never have a chance to execute.
- Solution: Aging.
  - As time progresses, increase the priority of the process.

Priority Scheduling and Round Robin together.

- Priority scheduling can be combined with RR.
  - Highest priority process is run first.
  - If there are multiple processes with the same priority, they can be served with RR. We have RR queue used for each priority.

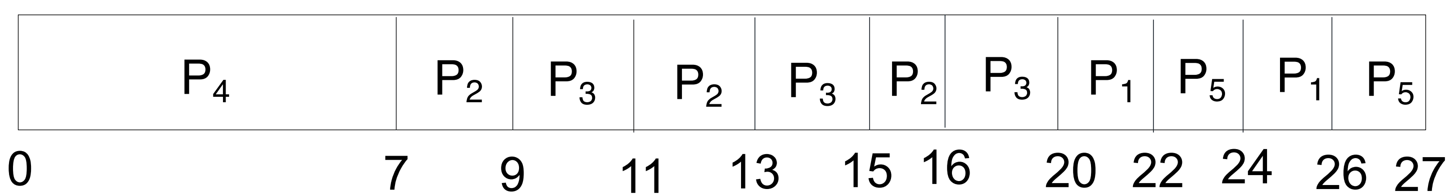
All bursts arrived at time 0.

Time quantum q = 2

P1 4 (3)
P2 5 (2)
P3 8 (2)
P4 7 (1)
P5 3 (3)

(priority)

burst time



In this example: smaller priority number indicates higher priority.

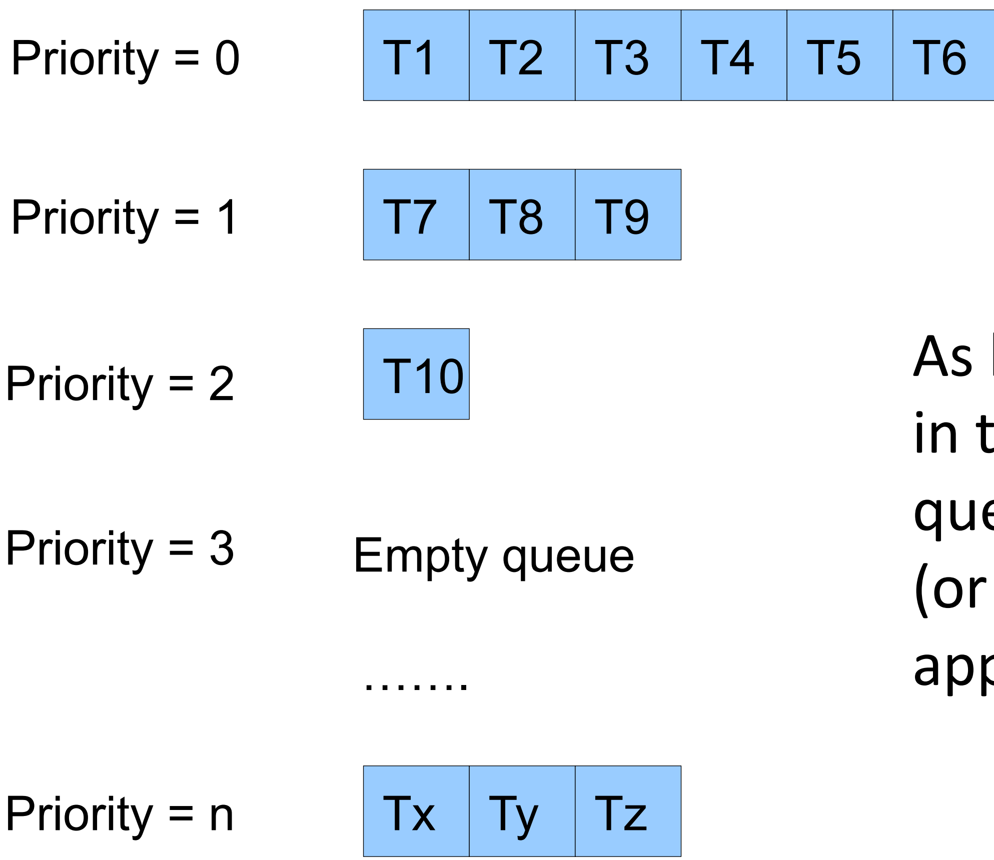
Multilevel Queue Scheduling

- Ready queue can be partitioned into separate (multiple) queues.
- For example: We may have 2 queues
    - foreground (interactive – interacting with a user), and
    - background (non-interactive) processes.
  - Foreground processes have higher priority (to have small response time, since they are interactive)
  - Each queue has its own scheduling algorithm.
    - foreground – RR (we can not wait a process too long)
    - background – FCFS (response time is not important)

Multilevel Queue Scheduling
Scheduling between queues

- Scheduling must be done between the queues. We can have the following alternatives:
  - Fixed (strict) priority scheduling. Serve foreground processes first. If no foreground process, then start serving background processes.
    - Possibility of starvation.
  - Time slicing. Each queue gets a certain amount of CPU time, which it can schedule among its processes. For example:
    - 80% of time serve foreground processes with RR.
    - 20% of time serve serve background processes with FCFS.

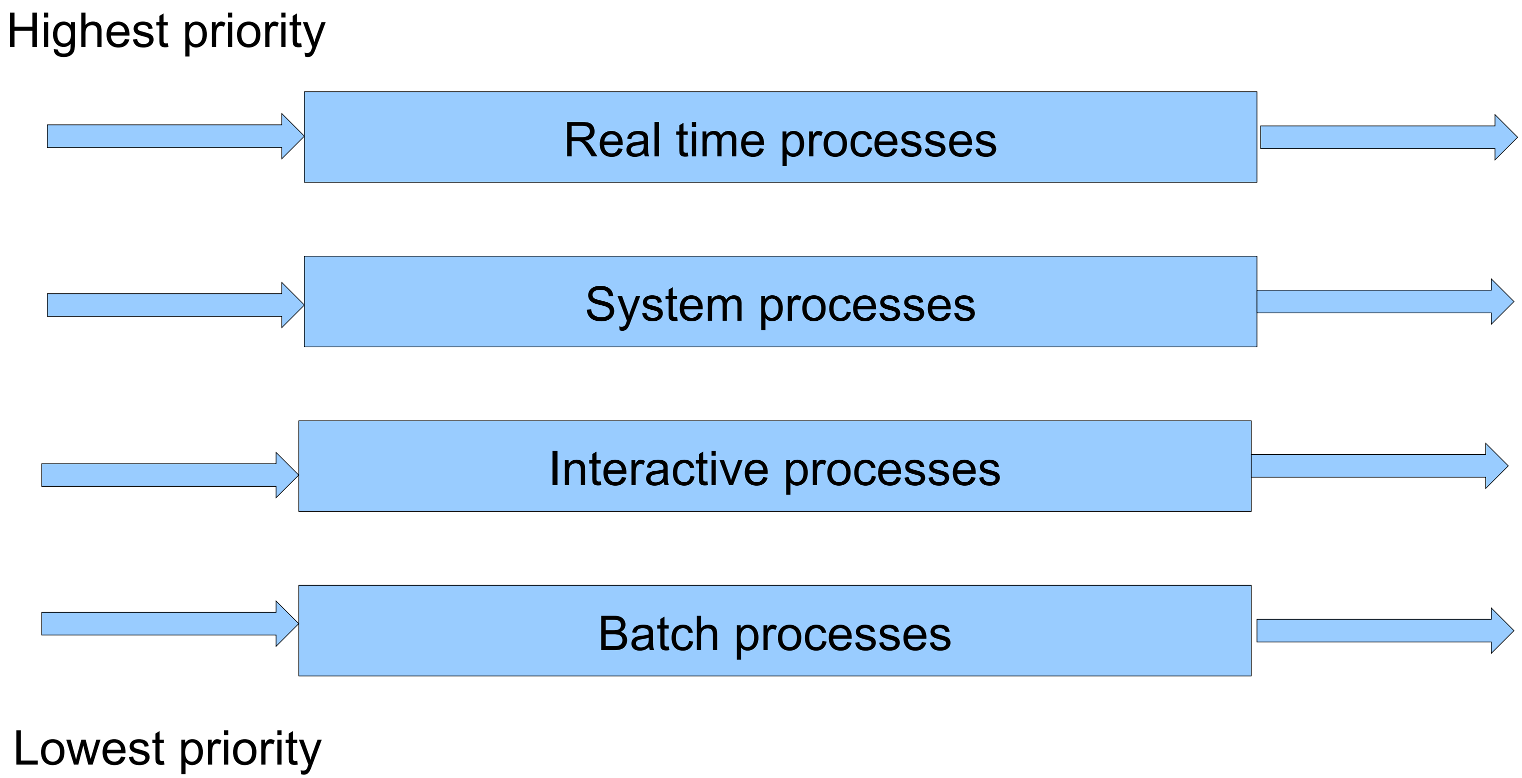
Multilevel Queue Scheduling



Can be served with RR (if burst bigger than *q*, preempted and put to the tail)

As long as there are tasks in the highest priority queue, they are served. (or time slicing can be applied).

Multilevel Queue Scheduling



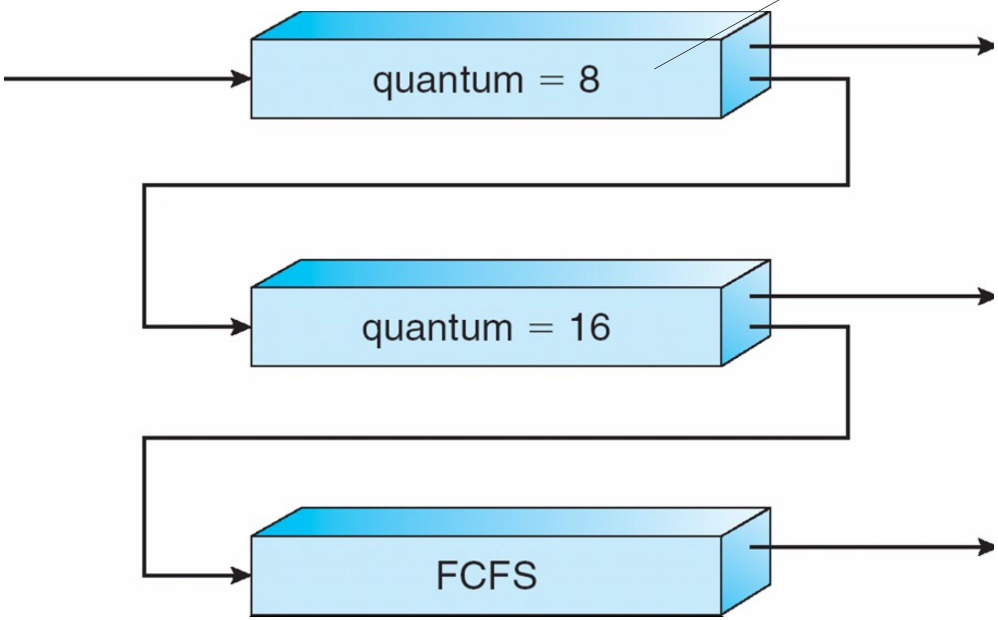
Multilevel Feedback Queue (MLFQ) scheduling

- A general scheduling algorithm.
- A process can move between the various queues. Aging can be implemented this way.
- A specific Multilevel-Feedback-Queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithm for each queue.
  - method to decide when to upgrade a process.
  - method to decide when to downgrade a process.
  - method to decide which queue an arriving process will enter.

Example of Multilevel Feedback Queue

- For example, we can have the parameters as follows.
- Three queues:
  - Q_0 – RR with q = 8 ms
  - Q_1 – RR time q = 16 ms
  - Q_2 – FCFS
- Scheduling rules:
  - A new job enters queue Q_0.
  - A job in Q_0 has 8 ms time quantum. If it does not finish its burst in 8 ms, it is moved to queue Q_1 (seems it is non-interactive)
  - At job in Q_1 16 ms time quantum. If it does not finish its burst in 16 ms, it is moved to queue Q_2 (seems it is really non-interactive)
  - At Q_2, jobs are served with FCFS.

Multilevel Feedback Queues



If a process here finishes its burst before 8 ms, it stays in this queue. i.e., put back to this queue when it becomes ready again, e.g., returns from I/O.

FCFS queue is served if there is no process in Q_0 and Q_1.

To prevent starvation, a process that waits too ling in a lower-priority queue, may gradually be moved to a higher priority queue.

Thread Scheduling

Thread Scheduling Issues

- Thread API may give some amount of control to the application programmers about how threads will be scheduled.
- Some systems implementing M:1 (many-to-one) and/or M:M models, may allow specifying at which level new threads will be scheduled:
  - at user-level (by tread library), or
  - at kernel-level (by kernel).

Thread Scheduling Issues

- Scheduling at user level means process-contention scope (PCS), since scheduling competition is happening within the process
  - In Pthreads: PTHREAD_SCOPE_PROCESS option
- Scheduling at kernel level means system-contention scope (SCS), since scheduling competition is happening within the kernel with other threads seen by the kernel
  - In Pthreads: PTHREAD_SCOPE_SYSTEM option
- Since Linux uses 1:1 model, it schedules with SCS.
- In Linux, we can also specify the scheduling class/policy. This effects the scheduling algorithm used by the kernel.
  - FIFO: FCFS scheduling
  - RT: real time scheduling
  - OTHER: schdeluling algorithm for ordinary processes (default)

Example: Pthread Scheduling API

```
int main(int argc, char *argv[])
{
        int i; pthread t tid[5]; pthread attr t attr;

        pthread attr init(&attr); // get the default attributes
        pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
        /* set the scheduling policy - FIFO, RT, or OTHER */
        pthread_attr_setschedpolicy(&attr, SCHED_OTHER);

        for (i = 0; i < 5; i++) // crete five threads
                pthread create(&tid[i],&attr,runner,NULL);
        for (i = 0; i < 5; i++)
                pthread join(tid[i], NULL);
}
void *runner(void *param)
{
        printf("I am a thread\n");
        pthread_exit(0);
}
```

Multiprocessor Scheduling

Multiple-Processor Scheduling

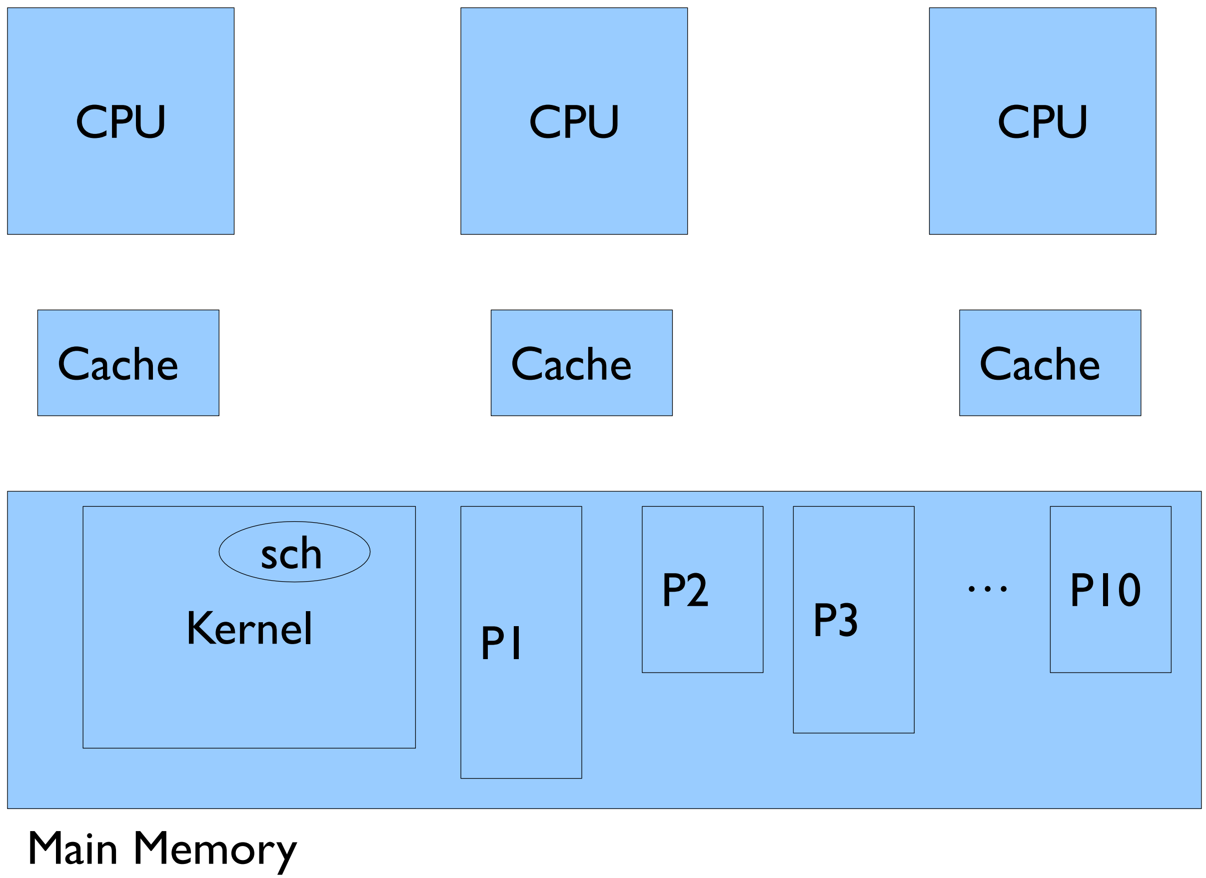
- CPU scheduling more complex when multiple CPUs are available in a computer.
- Algorithms we have seen can be applied. Additionally, there are more issues to consider.
- Different multiprocessor architectures:
  - Multicore CPUs
  - Multithreaded cores (Hyperthreading – HW threads)
  - NUMA systems (non-uniform memory access)
  - Heterogenous multiprocessing (CPUs are different)

Multiprocessor scheduling

- Asymmetric multiprocessing:
  - Only one processor accesses the kernel data structures, alleviating the need for data sharing.

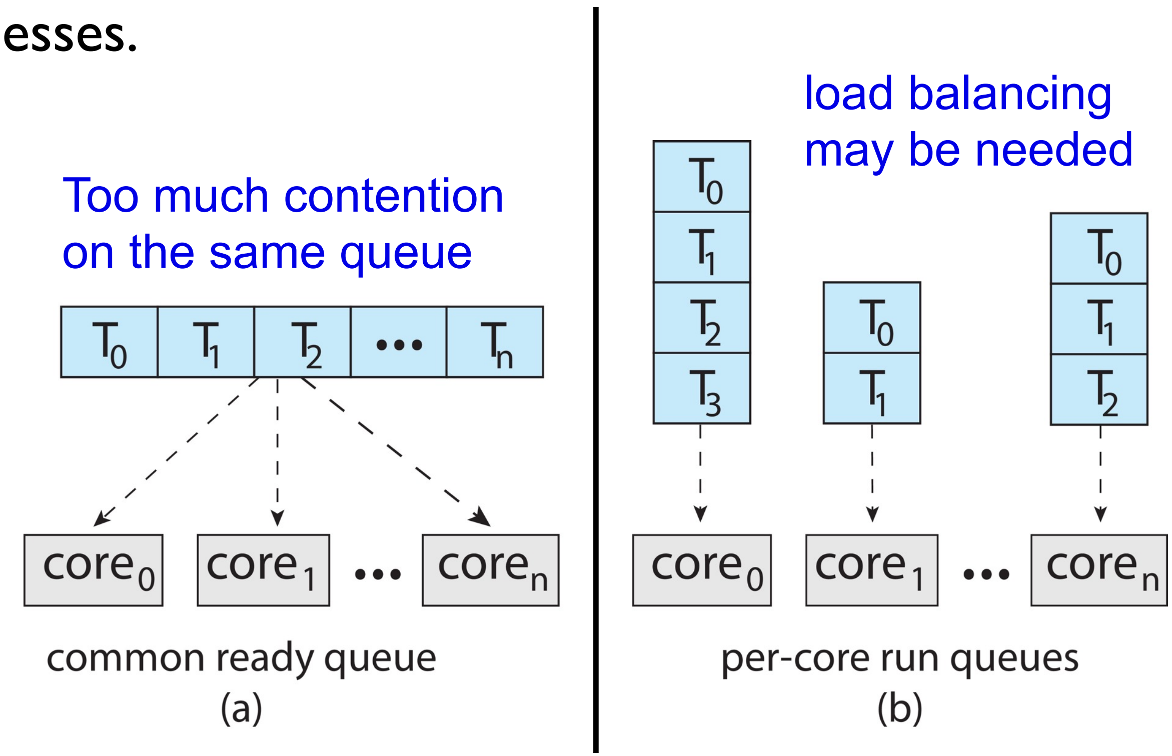
- Symmetric multiprocessing (SMP):
  - Each processor is self-scheduling.
  - Kernel can run on any processor when a hardware or software interrupt occurs.
  - Common in our desktop and laptop computers.

Multiple-Processor Scheduling: SMP



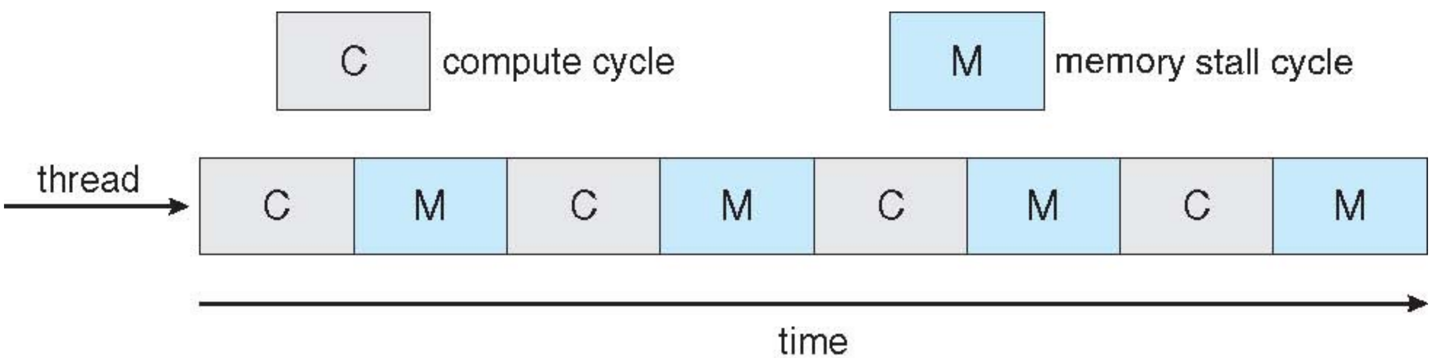
SMP

- We have two alternatives for enqueuing ready processes (threads):
  - a) All processes can be put into a common ready queue
  - b) Each processor can have has its own private queue of ready processes.



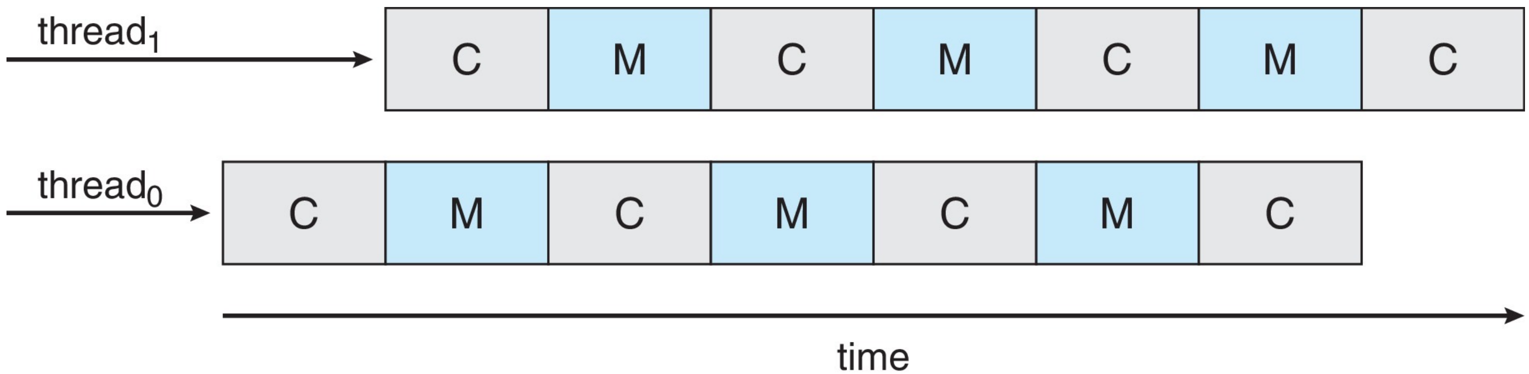
Multicore Processors

- Recent trend is to place multiple processor cores (CPUs) on same physical chip (die)
- Faster and consumes less power
- Multiple (hw) threads per core also growing (hardware threading)
  - Takes advantage of *memory stall* to make progress on another thread while memory retrieve is happening.



Multithreaded Multicore System

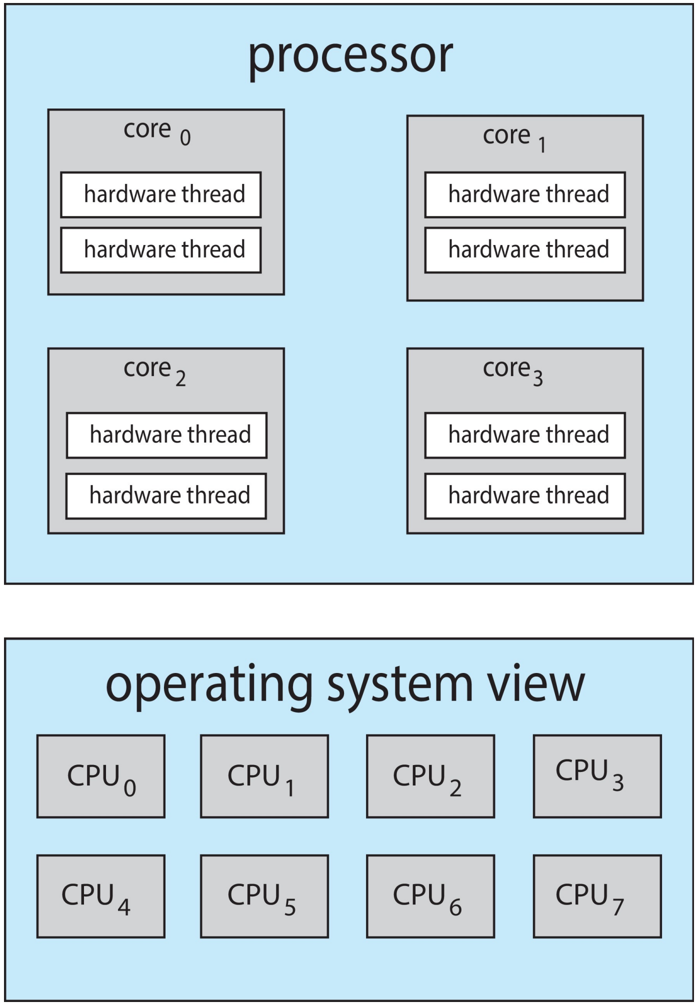
- Each core has more than 1 hardware threads (for example, 2).
- If one thread has a memory stall, hw switches to another thread. This is done at hardware level.



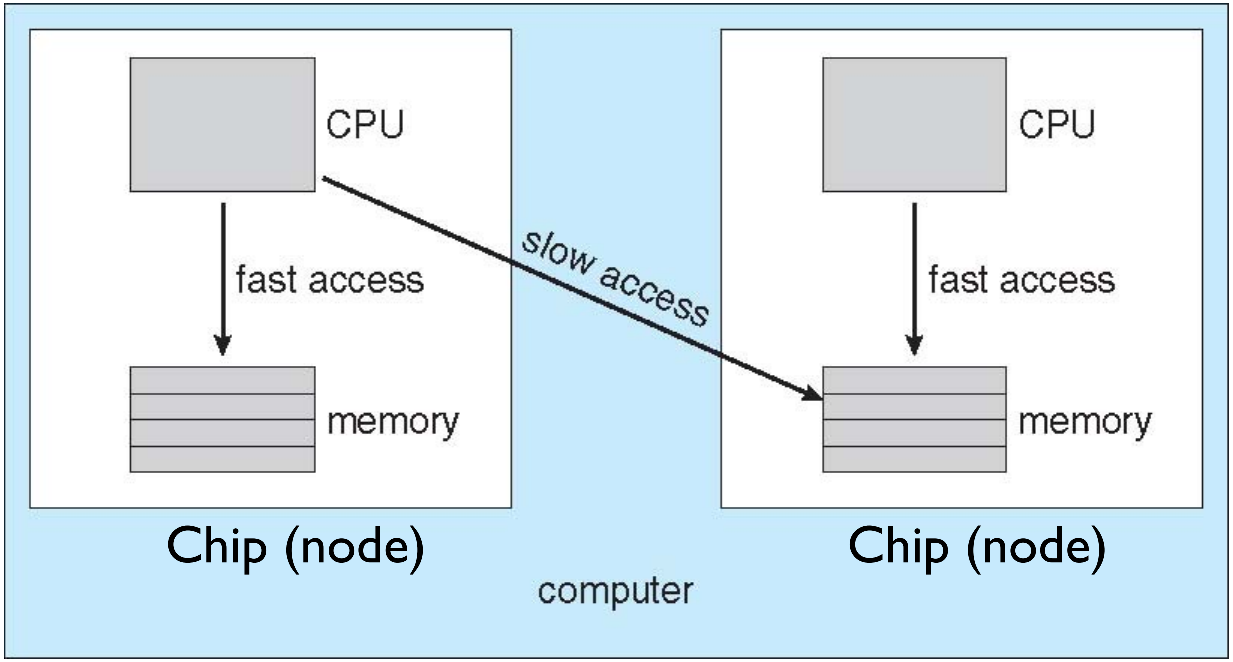
Appears to OS as two CPUs

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.



NUMA and CPU Scheduling



NUMA: non-uniform memory access. Multiple nodes (chips) in a computer. Each node has CPU and Memory. Memory in a chip is accessible to other CPUs on other chips, but with more delay.

Processor Affinity

- A thread (process) has *affinity* to the CPU it is running (due to per CPU cache).
- If run in the same CPU, no need to re-populate the cache.
- Soft affinity: system tries to schedule to the same CPU
- Hard affinity: system always schedules to the same CPU

- Linux has `sched_setaffinity()` system call for hard affinity.

Load Balancing

- If load is unbalanced (in a NUMA system):
  - Load balancing among the cores on the same chip can be performed.
  - Load balancing between different NUMA processors (chips) is not desirable.
    - Cache needs to be populated
    - Access to memory in other chip is slower.

Heterogenous Multiprocessing (HMP)

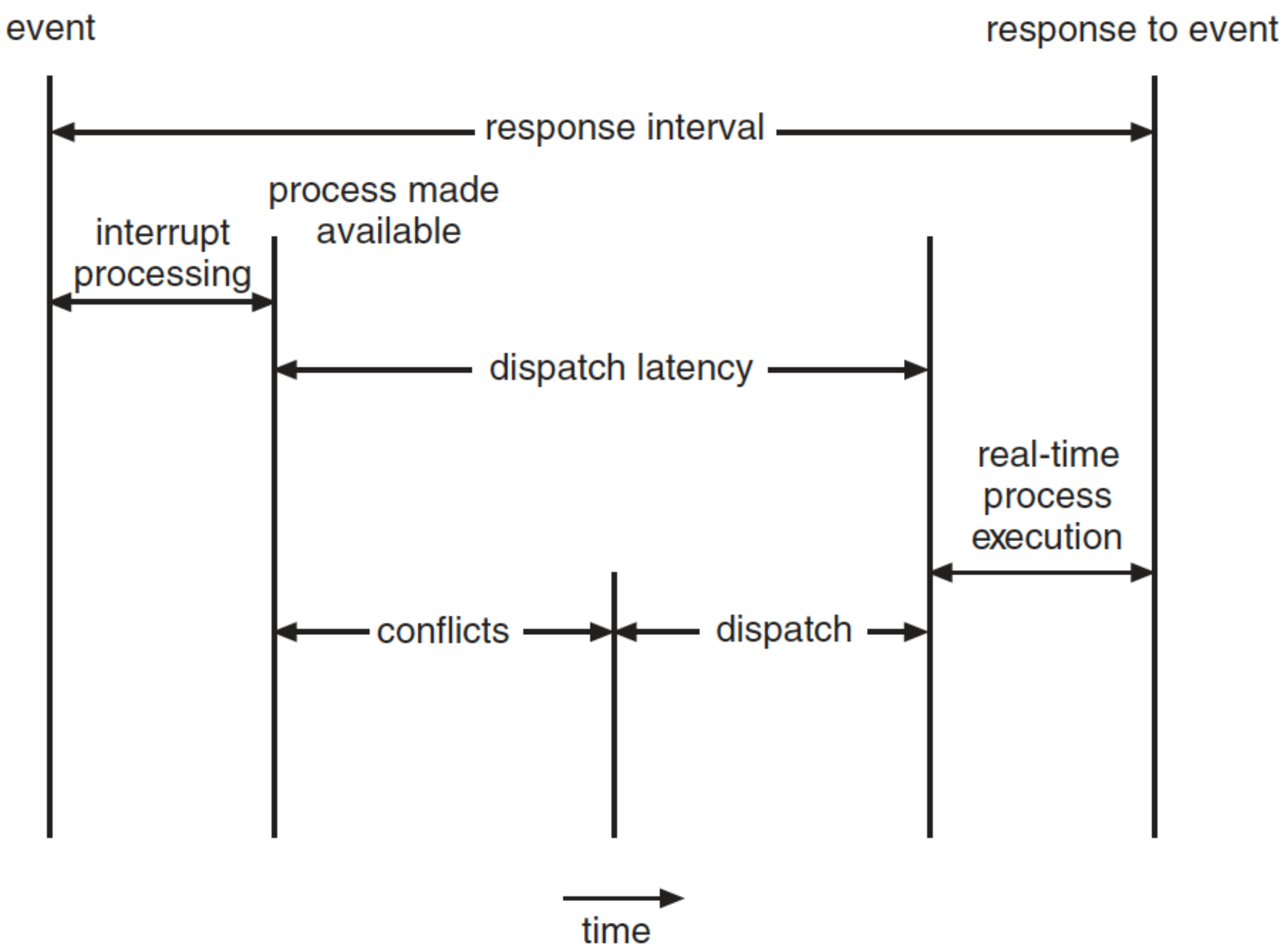
- In mobile devices, there are
  - Little cores (slow CPUs) (consuming less power)
  - Big cores (fast CPUs) (consuming more power)
- Tasks are assigned to cores based on their processing needs.
- Tasks that need high performance are scheduling to big cores. But they should not run too long.
- Tasks in little cores can run too long.

Real-Time CPU Scheduling

Real time systems

- Real-time processes (tasks) running in a system
  - There are hard and soft real-time systems
- Commonly, RT systems are event-driven systems
  - When an event occurs: need to respond (run a task) as quickly as possible
  - Minimizing latency is important
- Latency
  - Interrupt latency: time from interrupt till running the respective ISR, which marks the related task available (ready)
  - Dispatch latency: time between task becomes ready until it gets running

Latency


event
response to event
response interval
process made available
interrupt processing
dispatch latency
real-time process execution
conflicts
dispatch
time

Scheduling

- Priority based scheduling
  - Priority based algorithm used
  - Preemption may be provided as well
  - Soft real-time systems apply this.
  - No hard guarantees
  - Example: video player in a computer
- Hard real-time guarantees
  - There are deadlines for processes (tasks) to execute
  - Processes may be periodic or not
  - Admission control required

Hard guarantees

- Periodic tasks:
  - Has period (p), i.e., has rate = $\frac{1}{p}$
  - Requires some certain amount of cpu-time (t) periodically.
  - Has deadline (d)
  - We must have: $0 \leq t \leq d \leq p$
  - Example scheduling algorithm: Rate monotonic scheduling
- Non periodic tasks:
  - Example scheduling algorithm: Earliest deadline first (EDF)

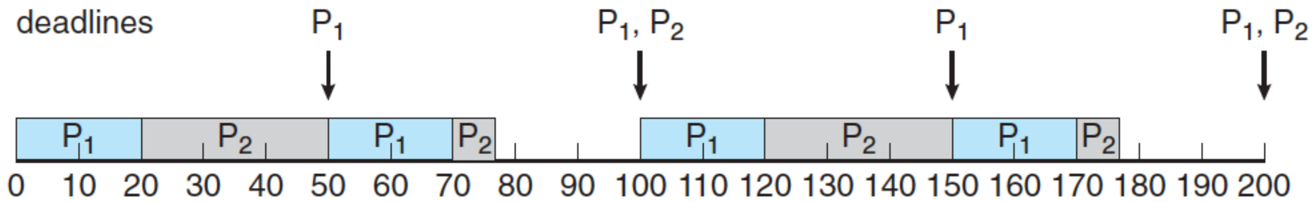
Rate monotonic scheduling

- Static priority used
  - Priority inversely proportional with period.
- Preemptive: high-priority task preempts low-priority task
- CPU burst time (processing time) equal in all periods.
- CPU burst must be executed before the next period.
  - *Implicit Deadline = beginning of next period.*
- Can underutilize CPU.
- Is optimal for static priority case.
  - That means if any static-priority scheduling algorithm can meet all the deadlines, then the rate-monotonic algorithm can too.

Rate monotonic scheduling

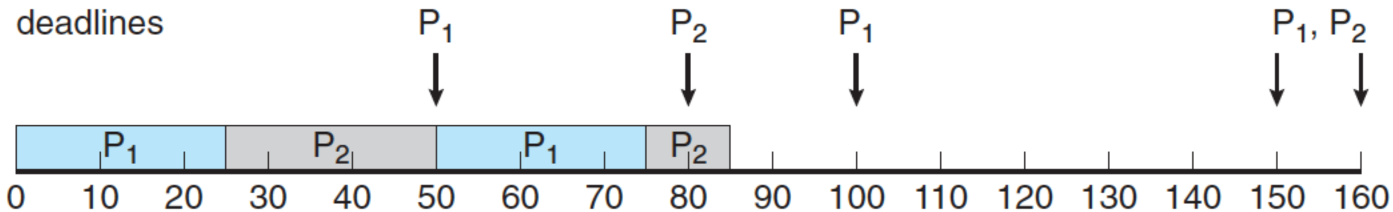
Example: P1: cputime:20; period 50
P2: cputime:35; period100

20/50 + 35/100 = 75 / 100 less than 1



Example: P1: cputime:25; period 50
P2: cputime:35; period 80

25/50 + 35/80 = 375/400 less than 1



deadline missed

Rate monotonic scheduling

- A feasible schedule (not missing deadlines) for N processes is guaranteed to exist if combined CPU utilization of N processes is less than a bound (worst-case CPU utilization)

$$\text{Worst-case CPU utilization} = N(2^{1/N}-1)$$

Combined utilization <= Worst-case utilization

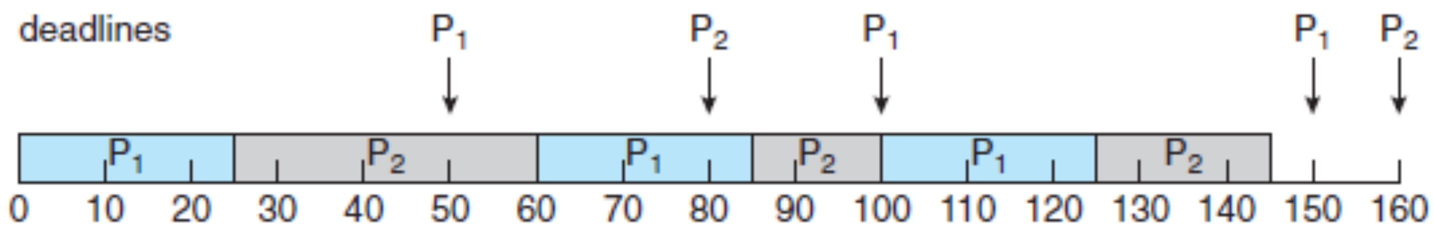
- With $N = 2$, worst-case utilization is 83%. That means:
  - When combined utilization (for $N = 2$) is 75%, we are guaranteed to schedule 2 periodic processes.
  - When combined utilization (for $N = 2$) is above 83%, we are not guaranteed to schedule 2 periodic processes.
- As N approaches infinity, utilization bound is 69%.

Earliest Deadline First (EDF)

- Dynamically assigns priorities depending on deadlines.
  - When a process becomes runnable (ready), it announces its deadline.
  - The process that has the earliest deadline is scheduled.
- Tasks do not need to be periodic
  - CPU bursts may be different.
- Theoretically optimal: can meet deadlines and can utilize CPU 100%.

Earliest Deadline First (EDF)

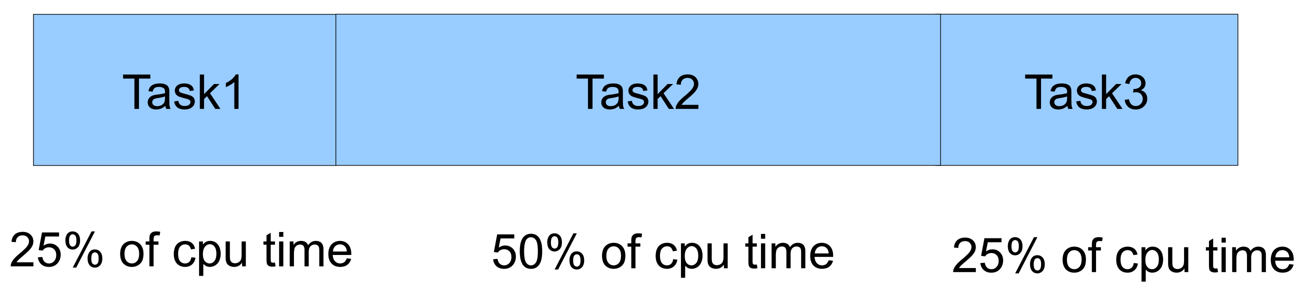
- Example: Two periodic processes (tasks):
  P1: cputime: 25; period 50
  P2: cputime: 35; period 80



Proportional Share Scheduling

- Allocate T shares among all applications (processes)
- If an application has N shares, it can get N/T of CPU time.
- System ensures that each process gets CPU time proportional to its shares.
- Admission control policy required.

Assume T=200 (total 200 shares)
Task 1 has 50 shares
Task 2 has 100 shares
Task 3 has 50 shares



Operating System Examples

- Linux scheduling

Linux Scheduling: CFS Algorithm

- To schedule ordinary processes (not real-time), Linux uses an algorithm called CFS: Completely Fair Scheduling.
- Linux has two scheduling classes at the moment. 1) real-time, and 2) time-sharing.
- Each class uses a fixed priority range (set of priority values that can be assigned to processes internally)
  - 1) real-time class (0-99).
  - 2) time-sharing (default scheduling) class (100-140). CFS algorithm is used. Our ordinary processes are in this class.

Linux Scheduling: CFS Algorithm

- Fair share scheduling. If all processes are of equal priority, they can get the same share of the cpu time (approximately).
- Efficient and scalable. Selects a task to run next very quickly.
- Processes have priorities. 40 different priority values (called nice values).

$$-20, -19, \dots, -1, 0, 1, 2, \dots, 18, 19$$

-20 is highest priority.

  - Can be set using the nice command. Default value is 0.
- CFS assigns a proportion of CPU time based on nice value indirectly.

CFS

- CFS uses virtual runtime (vruntime) for each process (granularity: nanoseconds)
- vruntime indicates how long a process has used the cpu so far (virtually).
- When scheduling decision
- n is to be made, task with the smallest vruntime is picked to run next (*since it could not use the cpu as much as others so far*).
- If a process with nice value 0 runs x ms in cpu (actual time), its vruntime is increased by $x\ ms$.
- The vruntime is advanced more (than x) for a lower priority (higher nice value) process, and less (than x) for a higher priority process, when such a process really runs x ms in cpu.

CFS

- What is the time slice for a process?
  - Time slice is set dynamically.
- CFS has a parameter called targeted latency (or, sched_latency).
  - Typical sched_latency is 48 ms.
- CFS uses sched_latency to determine how long (i.e., timeslice) a process will run in cpu before considering a context switch.
  - If there are n ready processes of equal priority, then the timeslice of each will be $48/n$. Over the latency period, each process will have a chance to run.
  - For example: if there 4 processes of equal priority, each will get 12 ms timeslice, i.e., each process is allowed to run 12 ms, then a context switch is considered.

CFS

- If too many processes (say 100), timeslice will be too small, and overhead will be too high.
- Therefore there is minimum timeslice value. It is called min_granularity.
  - It can be 6 ms, for example. Then no process will run less than 6 ms, even though there are many processes.
  - For example: if 10 processes, 48/12 = 4.8, but timeslice will be 6. That means it will take 60 ms (instead of 48) to run these 10 processes; but that is fine.
- Timer can tick every 1 ms. Then CFS checks if current task has reached its time slice. If not reached, process continues, otherwise context switch happens.

CFS

- CFS considers process priority as well (nice value).
- CFS maps nice values to weights using the following table.
- Default nice value is 0. smaller value ==> higher priority
- Higher priority process has more weight.

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

CFS

- A process k gets its cpu share (timeslice) depending on its weight as:

$$timeslice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} \cdot sched\_latency$$

- Example: assume we just have 2 ready processes, A and B, with nice values as -5 and 0.
  - Then weight(A)= 3121, and weight(B) is 1024.
  - Then A's timeslice will be $\cong \frac{3}{4} \times sched\_latency$= 36 ms.
  - B's time slice will be $\cong$ 12 ms.

CFS

- A process's vruntime is advanced considering its weight as well.
- When a process k runs t ms (timeslice) in cpu, t is its *actual runtime*. Then, its vruntime is updated as below:

$$vruntime_k = vruntime_k + \frac{weight_0}{weight_k} \cdot actualruntime_k$$

- The vruntime of a higher priority process advances slower.
  - *Then it will have a better chance for quick scheduling.*
- The vruntime of a lower priority process advances faster.
  - *We say, decay for a lower priority process is higher.*

CFS

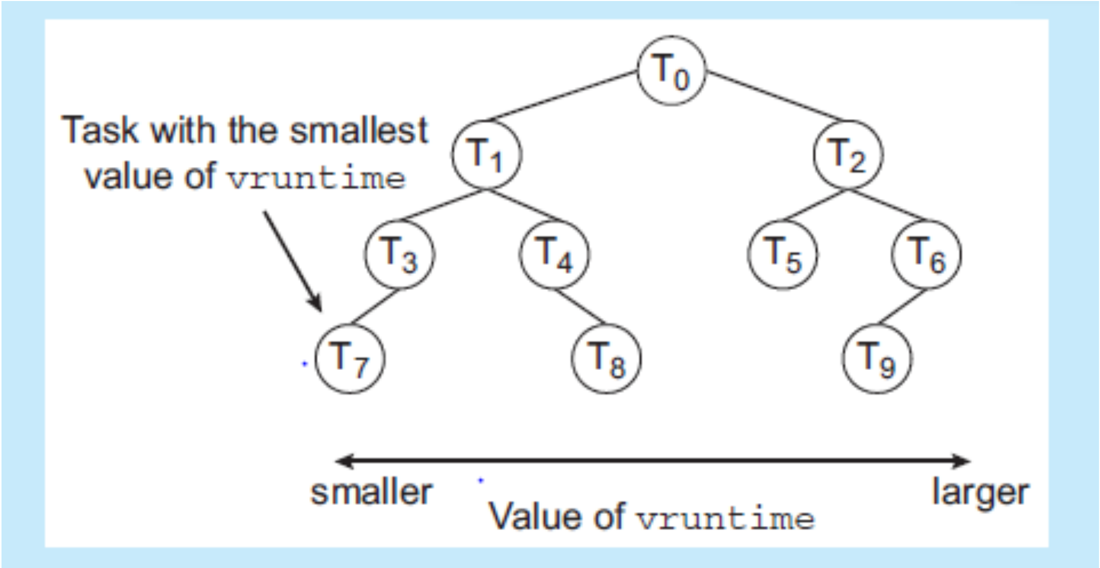
- While executing, a process k may start I/O (system call). Then it will sleep/block in waiting state (no longer in ready state).
- When k returns from waiting (I/O completed), its vruntime can be very behind.
- To prevent monopolizing the cpu by k, such process k's vruntime is set to be equal to the minimum vruntime (of all ready processes at that moment) *minus* some constant, after it returns from I/O.
- In this way process k gets scheduled very quickly but does not monopolize the cpu.
- A process returning from I/O wait (or just created) has its timeslice reset according to its weight.

CFS

- CFS does not have different queues for different priorities.
- CFS keeps all processes in ready state in a red-black tree (a balanced binary search tree) with respect to their vruntimes.
- The process on the left lower side is the one to run next (has the smallest vruntime).
  - Selection: $O(1)$
  - A pointer is kept to the minimum vruntime node.
  - Insertion of a process into tree: $O(logN)$
- No Round Robin applied (no queue structure). Instead CFS just picks the process k with minimum vruntime from the tree and runs it for timeslice(k) ms.

CFS

- Red black tree example. There are 10 runnable processes. T7 is picked up by the scheduler to run next.

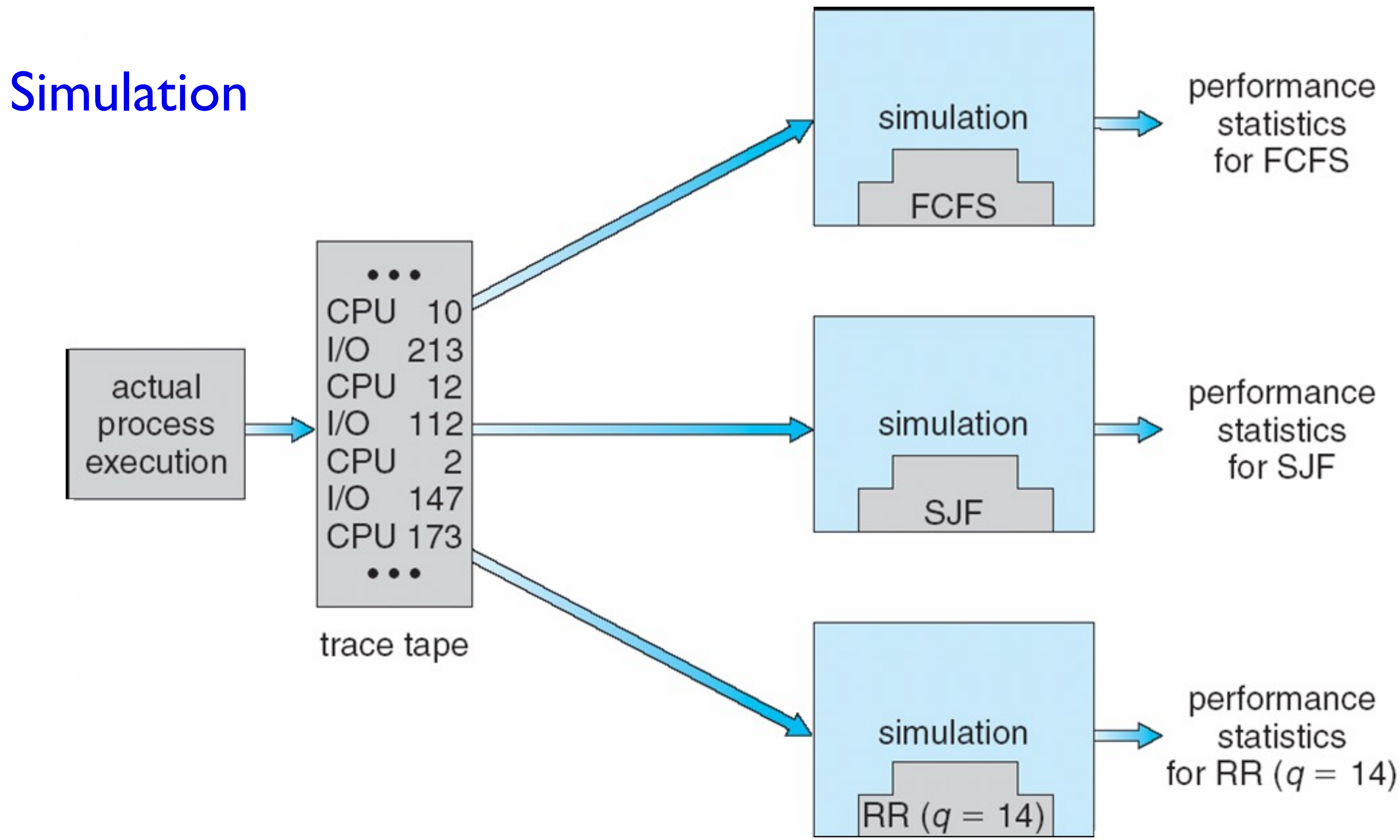


Algorithm Evaluation

Algorithm Evaluation

- Analytic evaluation
  - Deterministic modeling: takes a particular predetermined workload and defines the performance of each algorithm for that workload. Valid for a particular scenario and input.
  - Queuing models.
- Simulation
- Implementation
  - Implement in OS and see how it is performing.

Evaluation of CPU schedulers by Simulation



Queuing Models

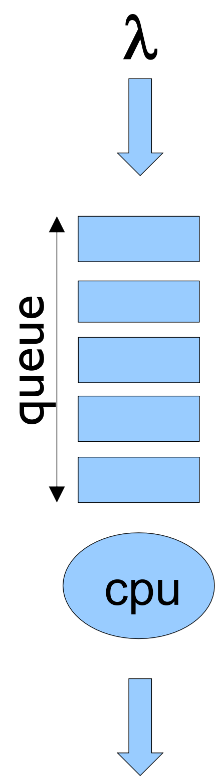
- Average throughput, utilization, and waiting time, queue length can be computed by using queuing models for a system. There is a branch of mathematics called queuing theory.
  - One or more CPUs and one or more ready queues make the system.
- Arrival times of tasks modeled by a distribution (inter-arrival times)
  - *Exponential distribution* is a good approximation for stochastic arrivals (as opposed to deterministic arrivals)
- Service times (required cpu times, i.e., burst lengths) modeled by a distribution.
  - *Exponential distribution* is a good approximation for required CPU times as well.

Queuing Models

- For a queueing system, which does not stay idle when there are tasks in the system, the following formula, called Little's formula holds:

$$N = W \times \lambda$$

  - N: number of tasks (cpu bursts) in the queue (*excluding* the one in the cpu)
  - W: average waiting time in the queue for a task (not including the time spent in CPU, i.e., the service time)
  - λ: arrival rate (number of tasks arriving per second)
- Assuming a stable system (service rate > arrival rate)
- Valid for any arrival and service time distribution.



Queuing Models

- Example: Assume arrival rate is 30 tasks/sec. What is the average queue length (excluding the task executing), if a task waits on the average 200 ms in the queue (not including the cpu execution time)?
- Answer:
  - $N = W \times \lambda$
  - Therefore, $N = 200\, ms \times 30 \frac{tasks}{sec} = 6$. There are 6 tasks in the system, on the average, that are waiting in the queue.

- *Note that if arrivals and service times are deterministic, no queuing and therefore no waiting in the queue will happen. Waiting in the queue happens due to random (stochastic) arrivals, i.e., bursty arrivals.*

References

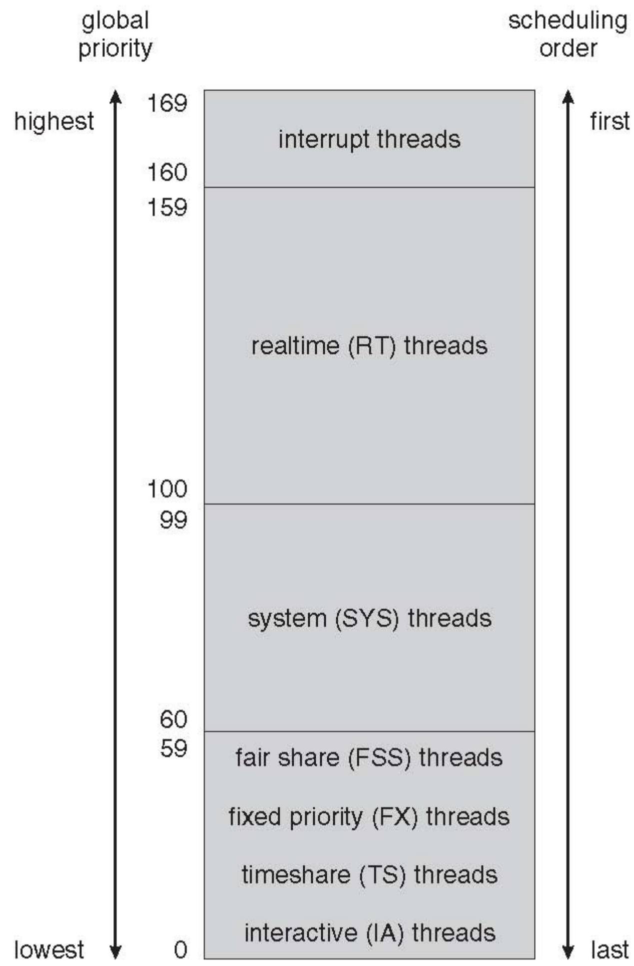
- Operating System Concepts, Silberschatz et al., Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum et al.
- Operating Systems: Principles and Practice, Anderson et al., 2014.
- OSTEP, Arpaci-Dusseau et al.

Additional material (optional)

Solaris Dispatch Table

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 Scheduling



Windows 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