Scheduling

Bojan Nokovic

Based on: "Operating Systems Concepts", 10th Edition Silberschatz Et al.

Feb. 2021

Why scheduling?

How to make multiprogramming computer systems more productive?

Limited resources

Limited time

OS makes two related kinds of decisions about resources:

- Allocation: who get what.
- Scheduling: how long can they keep it?

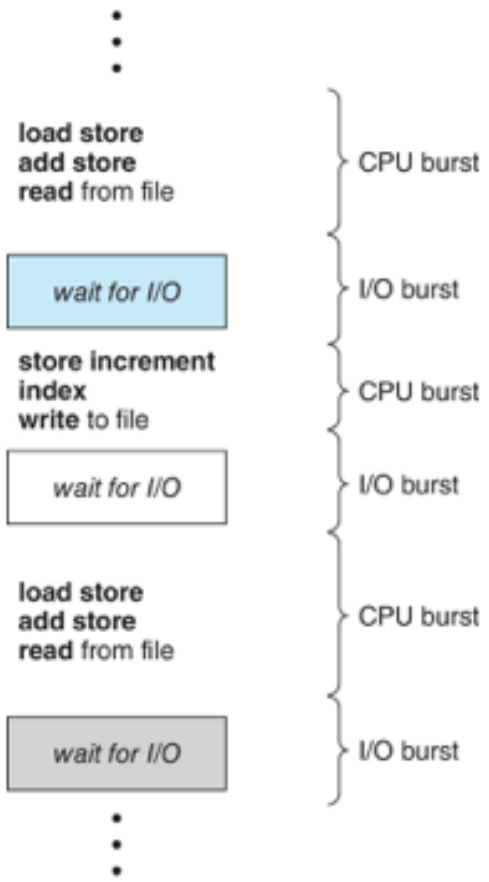
Basic Concept

A process is executed until it must wait for for the completion of some I/O request.

On a multicore system, this concept is extended to all processing cores on the system.

Process execution consists of a cycle of CPU execution and I/O wait.

CPU burst followed by I/O burst.



Types of Scheduling

Long-term scheduling - The decision to add to the pool of processes to be executed.

Medium-term scheduling - The decision to add to the number of processes that are partially or fully in main memory. Part of the swapping function.

Short-term scheduling - The decision as to which available process will be executed by the processor.

I/O scheduling - The decision to which process's pending I/O request shell be handled by an available I/O device.

CPU Scheduler

The CPU scheduler selects from among the processes in ready queue, and allocates the CPU to one of them.

Queue may be ordered in various ways: FIFO queue, a priority queue, a tree, or simply an unordered linked list.

CPU scheduling decisions may take place when a process

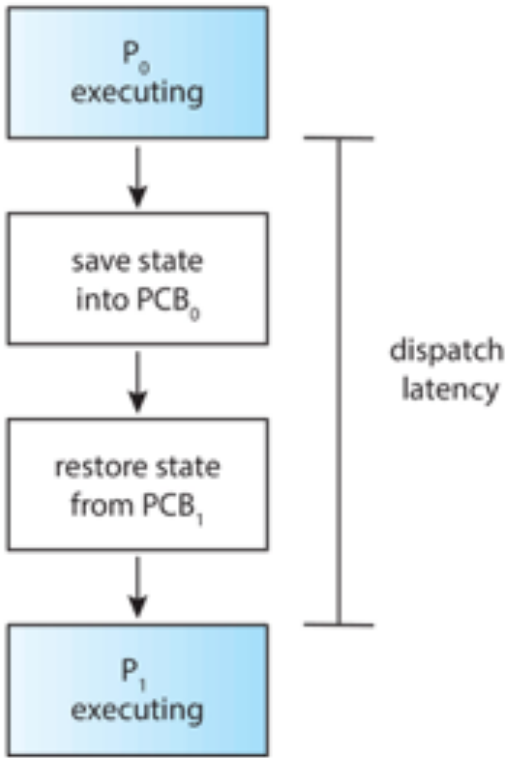
1. Switches from running to waiting state (i.e. as the result of an I/O request)
2. Switches from running to ready state (i.e. when an interrupt occurs)
3. Switches from waiting to ready (i.e. at completion of I/O)
4. Terminates

1 and 4 nonpreemptive or cooperative, otherwise preemptive.

Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler.

- 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

Max CPU utilization - keep the CPU as busy as possible

Max Throughput - # of processes that complete their execution per time unit

Min Turnaround time - amount of time to execute a particular process

Min Waiting time - amount of time a process has been waiting in the ready queue

Min Response time - amount of time it takes from when a request was submitted until the first response is produced.

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: $(0 + 24 + 27)/3 = 17$

First-Come, First-Served (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: $(6 + 3 + 0)/3 = 3$

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

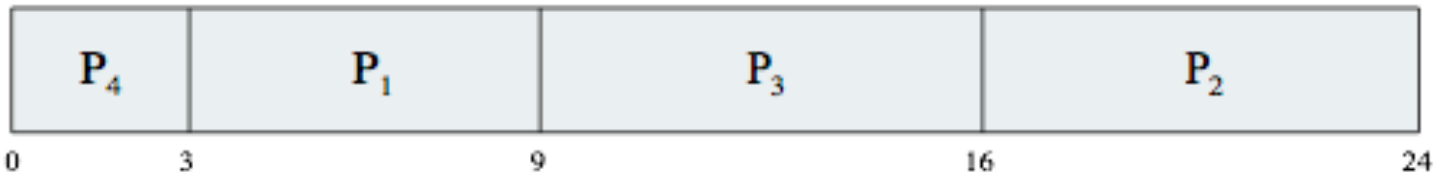
SJF is optimal - gives minimum average waiting time for a given set of processes

- The difficulty is knowing the length of the next CPU request

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

Determining Length of Next CPU Burst

We expect that the next CPU burst will be similar in length to the previous ones.

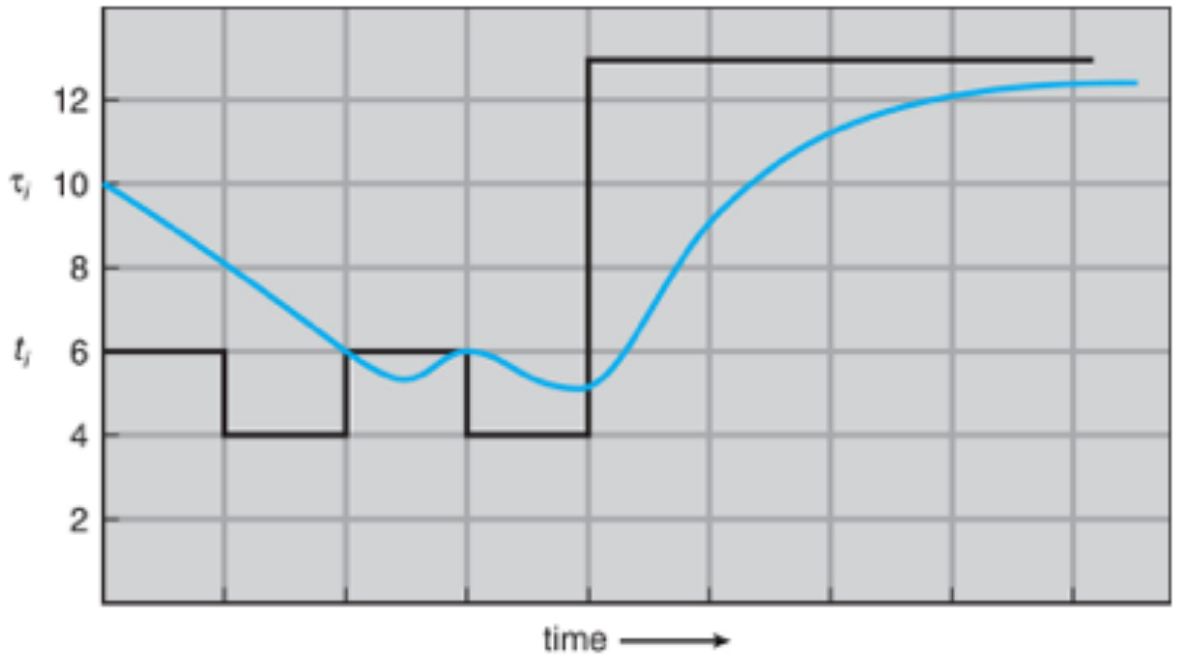
- Pick process with shortest predicted next CPU burst

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 $\frac{1}{2}$ - recent history and past history are equally weighted.

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

$\alpha = 0$

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

$\alpha = 1$

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

If we expand the formula, we get:
$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + (1 - \alpha)^2 \alpha t_{n-2} + .. + (1 - \alpha)^{n+1} \tau_0$

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

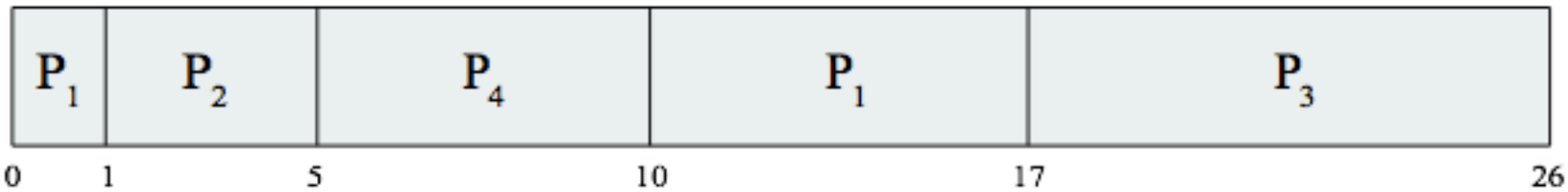
Preemptive version called shortest-remaining-time-first scheduling.

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 = 6.5 msec

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 $\frac{1}{n}$ of the CPU time in chunks of at most q time units at once.

No process waits more than (n-1)q time units.

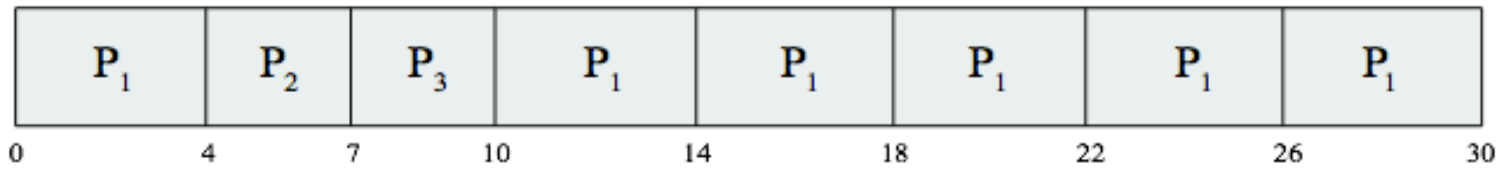
Timer interrupts every quantum to schedule next process.

Example of RR with Time Quantum

Time quantum: 4ms

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:

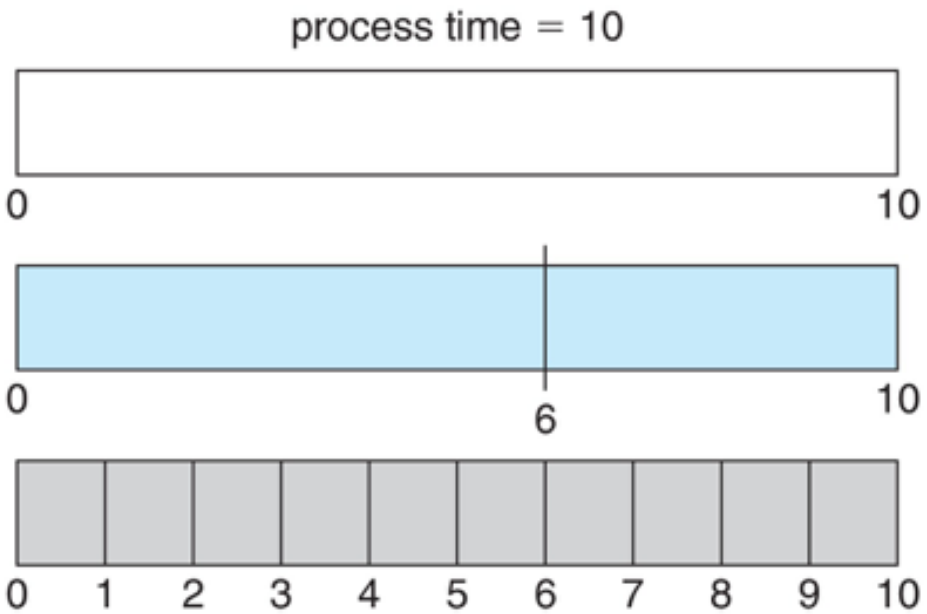


Typically, higher average turnaround than SJF, but better response

q should be large compared to context switch time

q usually 10ms to 100ms, context switch < 10 μsec

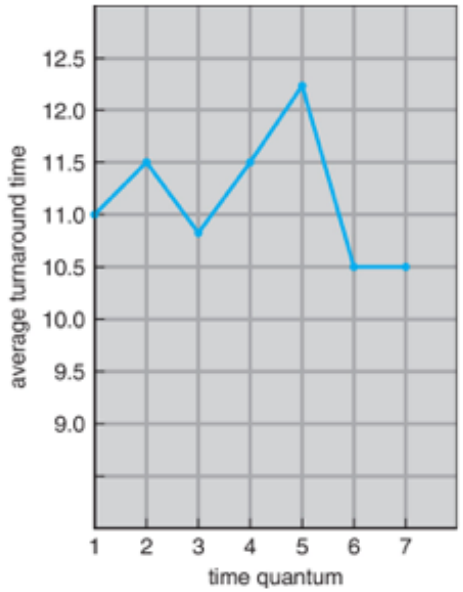
Time Quantum and Context Switch Time



quantum	cntx switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Given three processes of 10 time units each.

- q = 1 time unit - the average turnaround time is 29.
- q = 10 time units - the average turnaround time is 20.

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

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

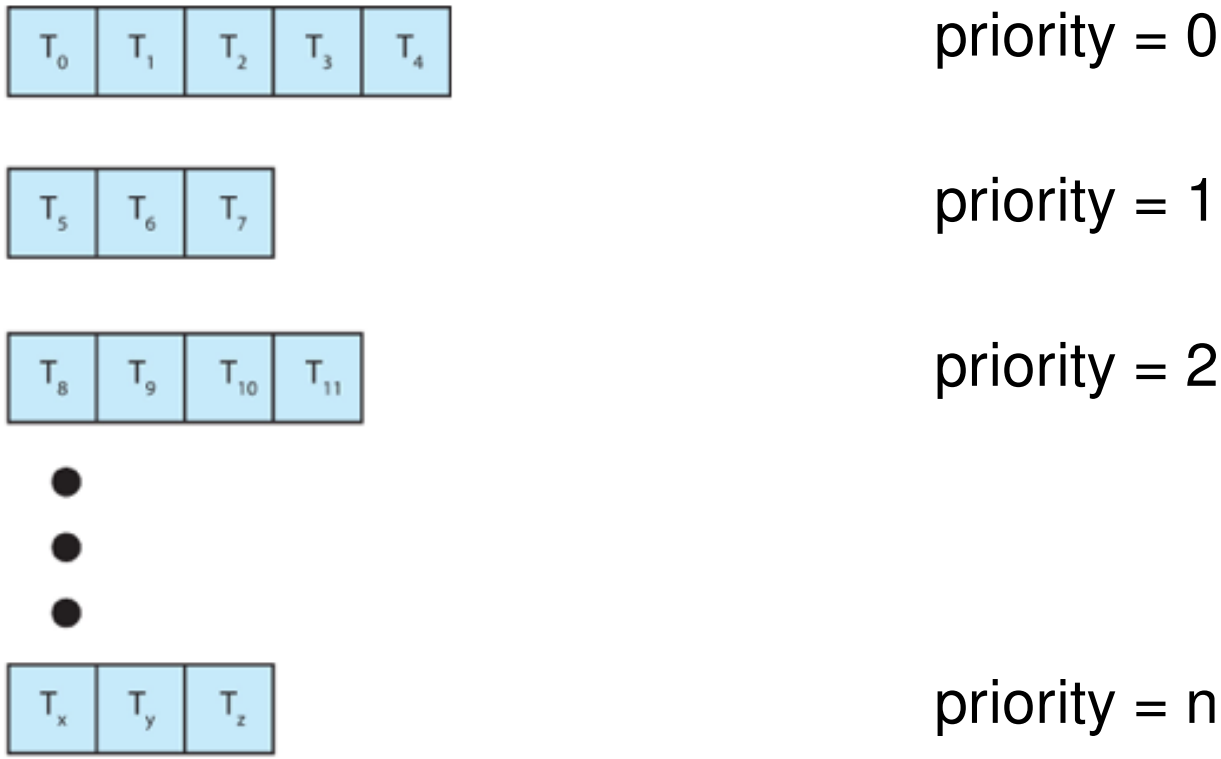
Run the process with the highest priority. Processes with the same priority run round-robin

Gantt Chart with 2 ms time quantum

Multilevel Queue

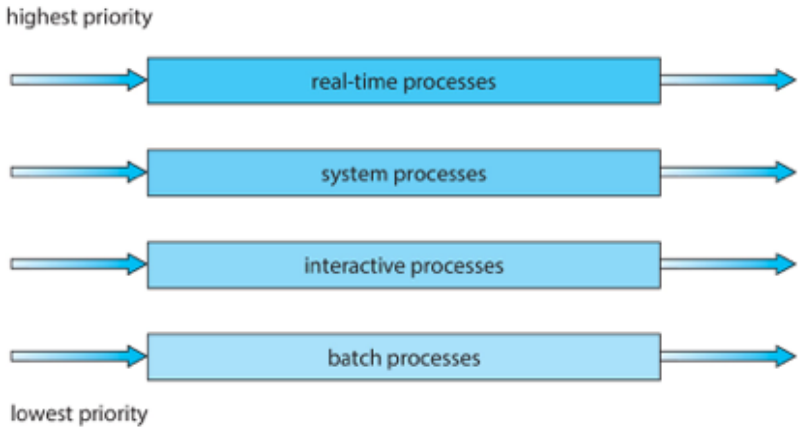
With priority scheduling, have separate queues for each priority.

Works well when priority scheduling is combined with round-robin.



Multilevel Queue

Prioritization based upon process type



Foreground (interactive) processes and background (batch) processes.

Different response-time requirements => different scheduling needs.

Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm.

Multilevel Feedback Queue

The idea is to separate processes according to the characteristics of their CPU bursts.

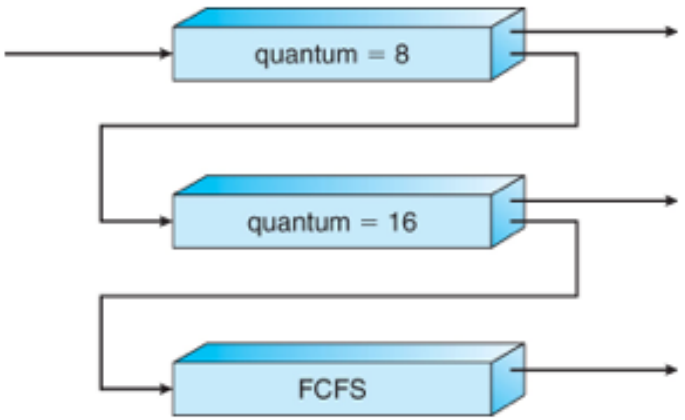
A process can move between the various queues - aging can be implemented this way.

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

1. Q_0 - RR with time quantum 8 milliseconds
2. Q_1 - RR time quantum 16 milliseconds
3. 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

Thread Scheduling

Distinction between user-level and kernel-level threads.

On most OS its kernel-level threads are being scheduled.

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.

Systems using the one-to-one model, such as Windows and Linux schedule threads using only SCS.

Multiple-Processor Scheduling

CPU scheduling more complex when multiple CPUs are available

Multiprocess may be any one of the following architectures:

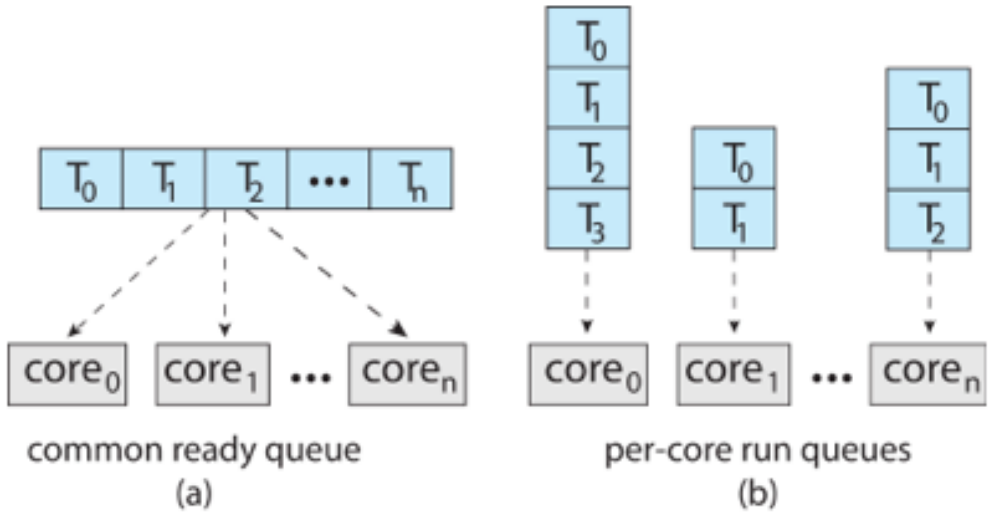
- Multicore CPUs
- Multithreaded cores
- NUMA (non-uniform memory access) systems. The CPUs are connected by a shared system interconnect, so that all CPUs share one physical address space.
- Heterogeneous multiprocessing (i.e. combination of a multicore processor with a GPU)

Multiple-Processor Scheduling

Asymmetric multiprocessing, only one core (master) accesses the system data structures, but master server becomes a potential bottleneck.

Symmetric multiprocessing (SMP) is where each processor is self scheduling - all modern OS support SMP.

a) All threads may be in a common ready queue
b) Each processor may have its own private queue of threads



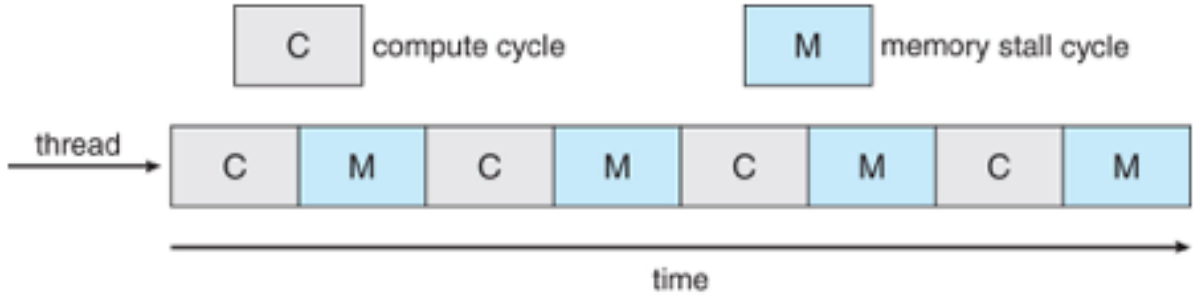
Multicore Processors

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

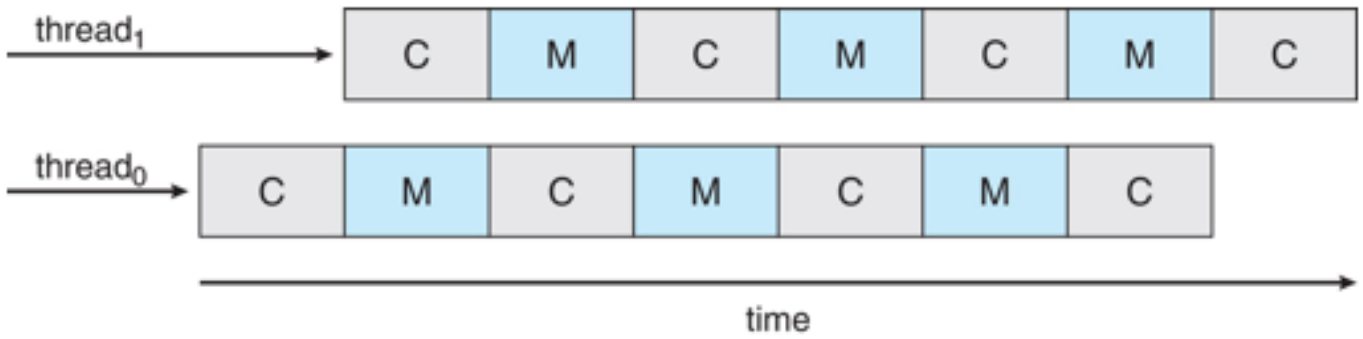


A memory stall - the processor can spend up to 50 percent of its time waiting for data to become available from memory.

Multithreaded Multicore System

Each core has > 1 hardware threads.

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



From an OS perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread.

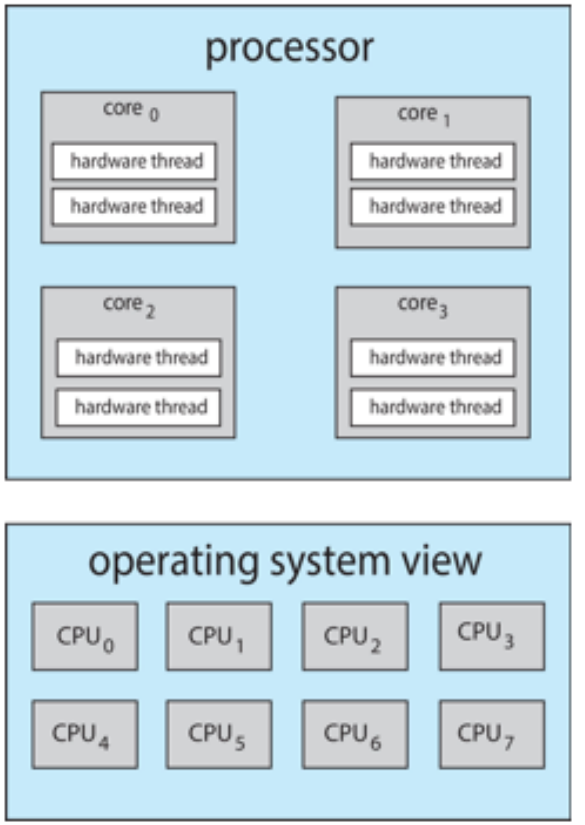
This technique is known as chip multithreading (CMT)

Example of Multilevel Feedback Queue

Chip-multithreading (CMT) assigns each core multiple hardware threads.

Intel refers to this as hyperthreading or simultaneous multithreading or (SMT).

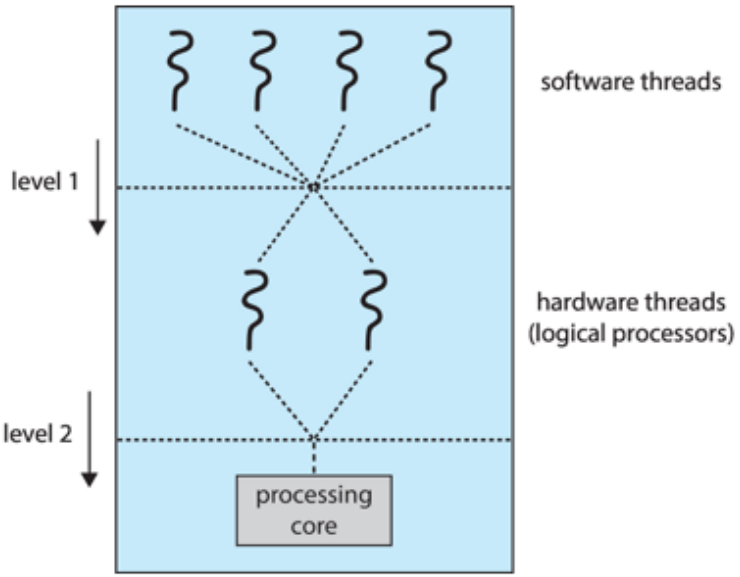
On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.



Multithreaded Multicore System

The resources of the physical core (such as caches and pipelines) must be shared among its hardware threads therefore a processing core can only execute one hardware thread at a time.

1. The operating system deciding which software thread to run on a logical CPU - OS can use any scheduling algorithm (FCFS, SJF, RR,..)
2. How each core decides which hardware thread to run on the physical core? (RR, dynamic urgency value ranging from 0 to 7,...)



A multithreaded, multicore processor requires two levels of scheduling.

Multiple-Processor Scheduling - Load Balancing

If SMP, need to keep all CPUs loaded for efficiency

Load balancing attempts to keep workload evenly  distributed

There are two general approaches to load balancing

- Push migration - periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- 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.

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.

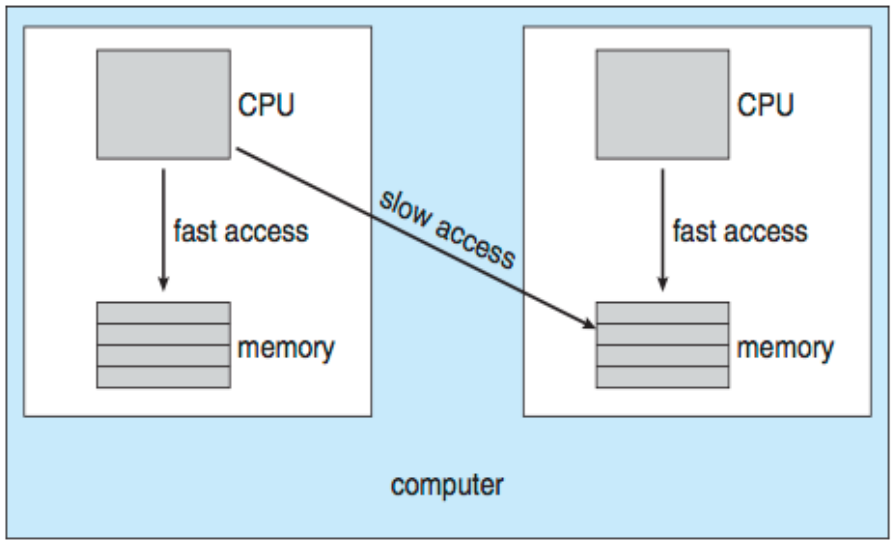
Soft affinity - the operating system attempts to keep a thread running on the same processor, but no guarantees.

Hard affinity - allows a process to specify a set of processors it may run on.

NUMA and CPU Scheduling

A system Interconnect allows all CPUs in a NUMA system to share one physical address space.

Two physical processor chips each with their own CPU and local memory.



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

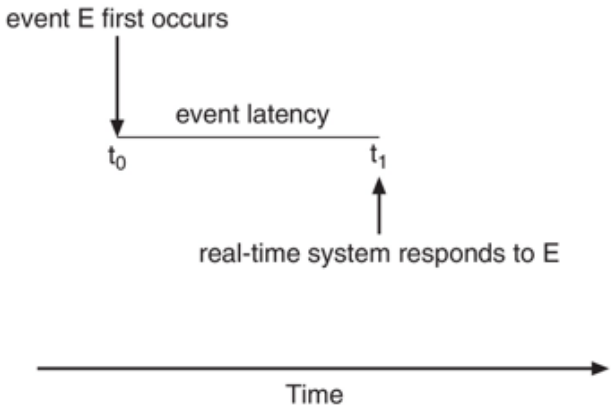
Real-Time CPU Scheduling

Soft real-time systems task may be serviced by its deadline, but no guarantee.

Hard real-time systems task **must** be serviced by its deadline.

Real-Time CPU Scheduling

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

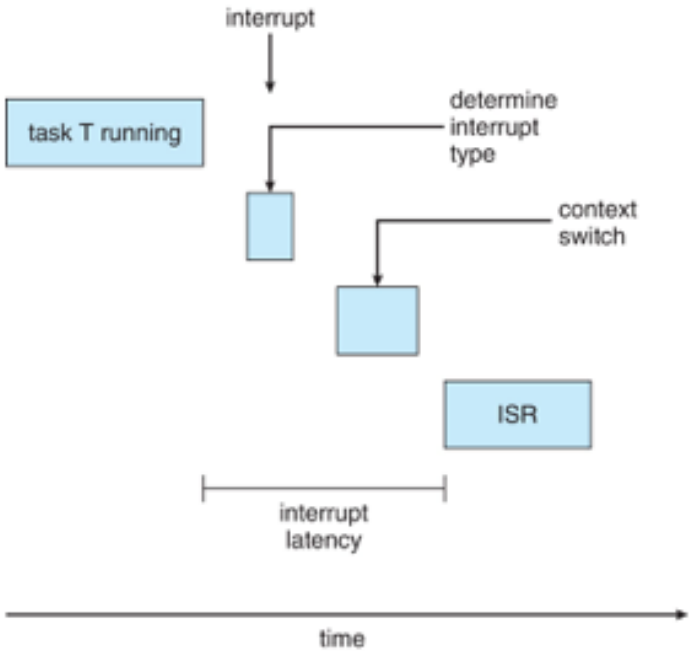


Two types of latencies affect performance

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

Interrupt Latency

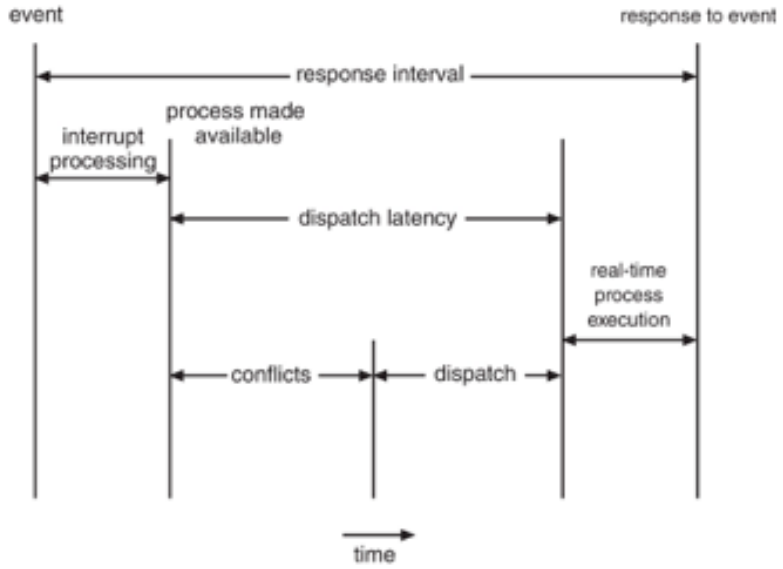
The period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.



- Complete the instruction it is executing and determine the type of interrupt that occurred.
- Save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR).

Dispatch Latency

The amount of time required for the scheduling dispatcher to stop one process and start another.



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

The dispatch phase schedules the high-priority process onto an available CPU.

Priority-based Scheduling

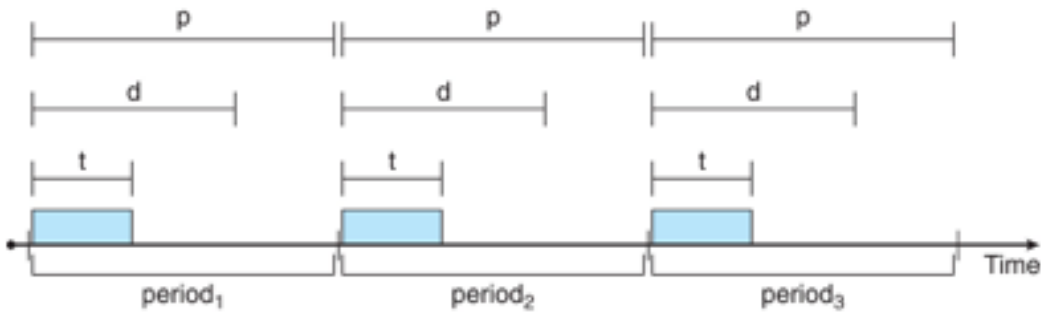
For real-time scheduling, scheduler must support preemptive, priority-based scheduling

- 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

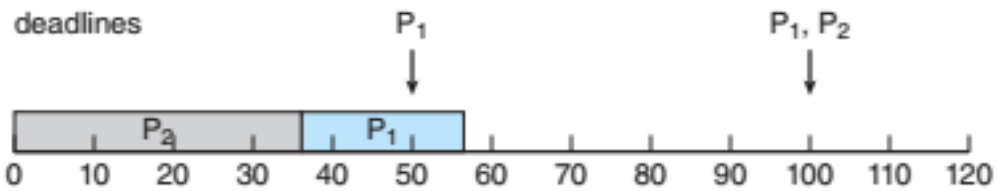


No Rate Monotonic Scheduling

Two processes: P_1, P_2; the periods: $p_1 = 50$, $p_2 = 100$; processing times: $t_1 = 20$, $t_2 = 35$

Is it possible to schedule these tasks so that each meets its deadlines?

CPU utilization for $P_1 = t_1/p_1 = 20/50 = 0.4\%$
CPU utilization for $P_2 = t_2/p_2 = 35/100 = 0.35\%$
Total CPU utilization = 75%



Suppose P_2 has a higher priority than $P_1 \Rightarrow P_1$ will miss the deadline

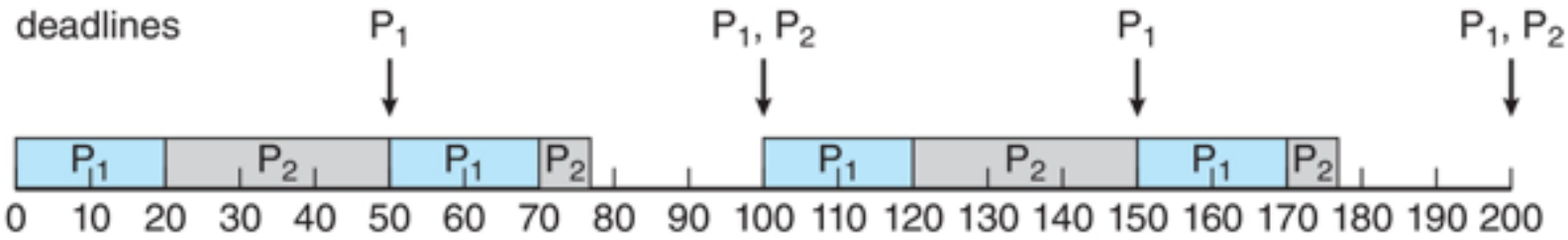
Rate Monotonic Scheduling

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.

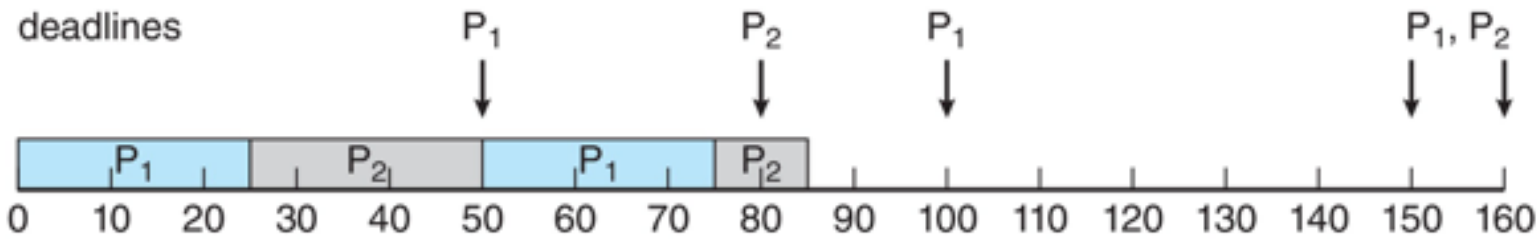


Both P_1 and P_2 meets the deadline.

Missed Deadlines with Rate Monotonic Scheduling

$p_1 = 50$, $p_2 = 80$
$t_1 = 25$, $t_2 = 35$

The total CPU utilization of the two processes is (25/50) + (35/80) = 0.94%



Process P2 misses finishing its deadline at time 80

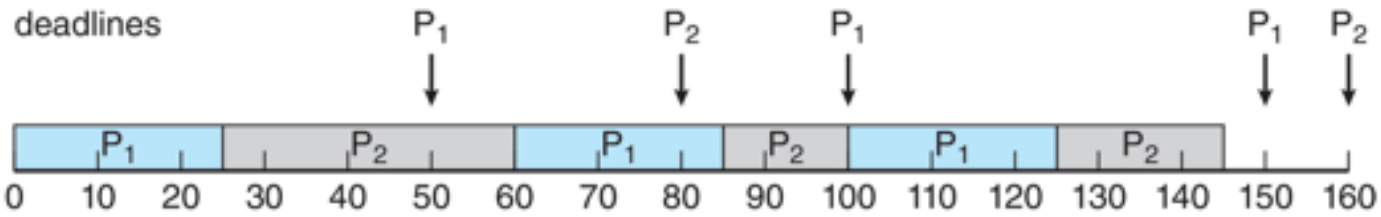
Despite being optimal, rate-monotonic scheduling has a limitation.

For N processes, the worst-case CPU utilization: $N(2^{1/N} - 1)$

Earliest Deadline First Scheduling (EDF)

$p_1 = 50$, $p_2 = 80$
$t_1 = 25$, $t_2 = 35$
Priorities are assigned according to deadlines:

- the earlier the deadline $\Rightarrow$ the higher the priority
- the later the deadline $\Rightarrow$ the lower the priority



At the beginning of its next period at time 50, EDF scheduling allows process P_2 to continue running.

P_2 now has a higher priority than P_1 because its next deadline at time 80 is earlier than that of P_1 at time 100.

Proportional Share Scheduling

T shares are allocated among all processes in the system

An application receives N shares where $N < T$

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

Example: T = 100 shares is to be divided among three processes, A, B, and C.
A = 50, B =15 shares, C = 20

This scheme ensures that A will have 50% of total processor time, B will have 15%, and C will have 20%.

What happens if new process D requested 30 shares?

Algorithm Evaluation

How to select CPU-scheduling algorithm for an OS?

Determine criteria, then evaluate algorithms

Deterministic modeling

- Type of analytic evaluation
- 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

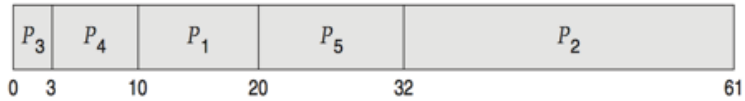
Deterministic Evaluation

For each algorithm, calculate minimum average waiting time

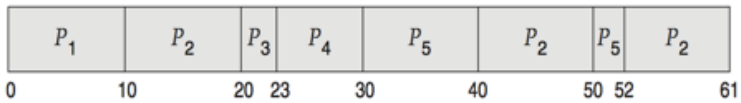
Simple and fast, but requires exact numbers for input, applies only to those inputs

- FCFS is 28ms:

- Non-preemptive SJF is 13ms:



- RR is 23ms:



Queueing Models

Describes the arrival of processes, and CPU and I/O bursts probabilistically

- Commonly exponential, and described by mean
- Computes average throughput, utilization, waiting time, etc

Computer system described as network of servers, each with queue of waiting processes

- Knowing arrival rates and service rates
- Computes utilization, average queue length, average wait time, etc

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 $\lambda = 7$ processes arrive per second, and normally $n = 14$ processes in queue.
Then average wait time per process W = $\frac{n}{\lambda}$ = 2s

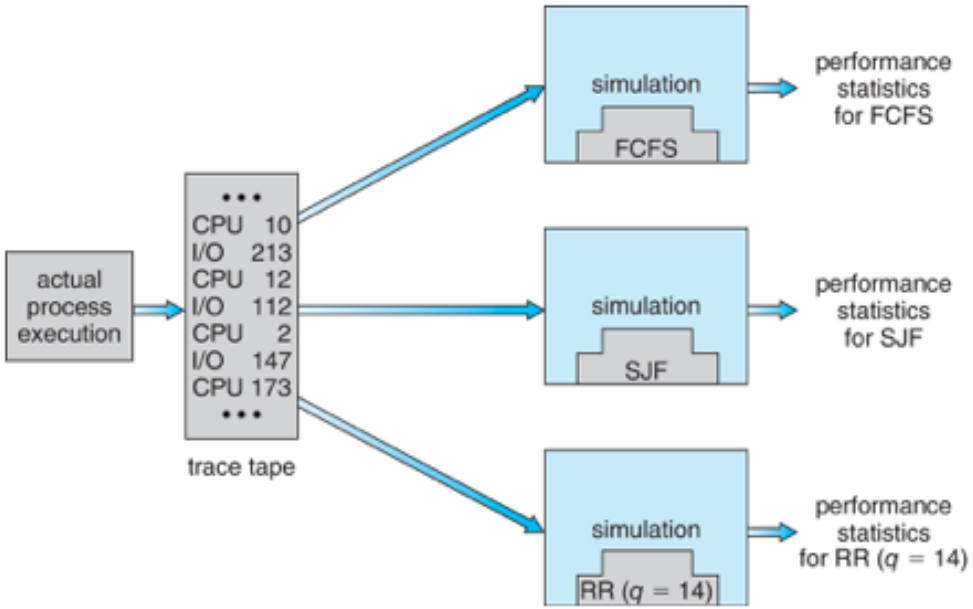
Simulations

Queueing models are limited 

Simulations are more accurate

- Programmed model of computer system
- Clock is a variable
- Gather statistics indicating algorithm performance
- Data to drive simulation gathered via
  - Random number generator according to probabilities
  - Distributions defined mathematically or empirically
  - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



Distribution does not indicate order of event instances.

To correct this problem, we can use trace files.

Implementation

Even simulations have limited accuracy 

Implement new scheduler and test in real systems

- High cost, high risk
- Environments vary

Most flexible schedulers can be modified per-site or per-system. 

Thank you !

Operating Systems are among the most complex pieces of software ever developed !