CPU Scheduling

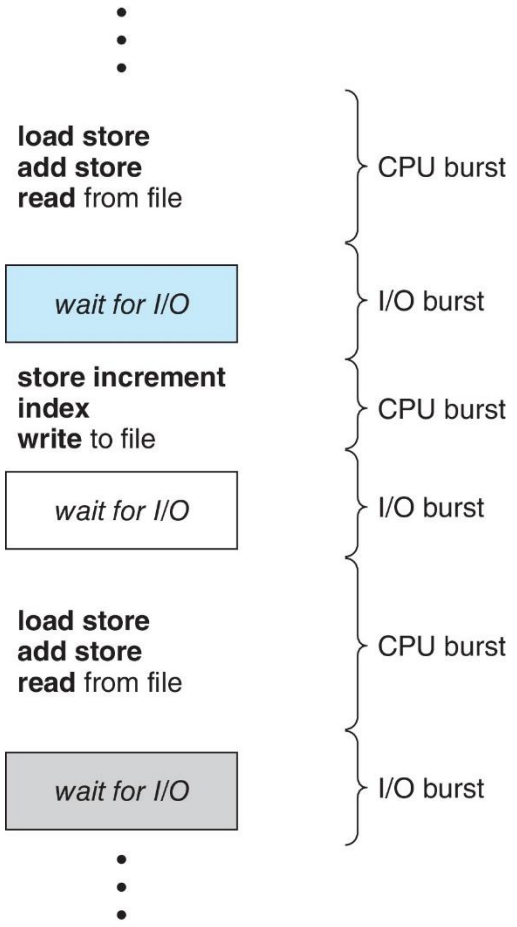
Chapter 5

Recall Schedulers

- Scheduling is a fundamental O.S. function.
- Schedulers affect the performance of the system.
- **Long term scheduler:** Determine which processes are admitted into the system.
- **Short term scheduler:** Select a process in the ready queue for CPU allocation.
- **Medium term scheduler** (swapper): determine which process to swap in/out of disk to/from memory.
- We will discuss short-term schedulers (CPU scheduling).

CPU-I/O Burst Cycles

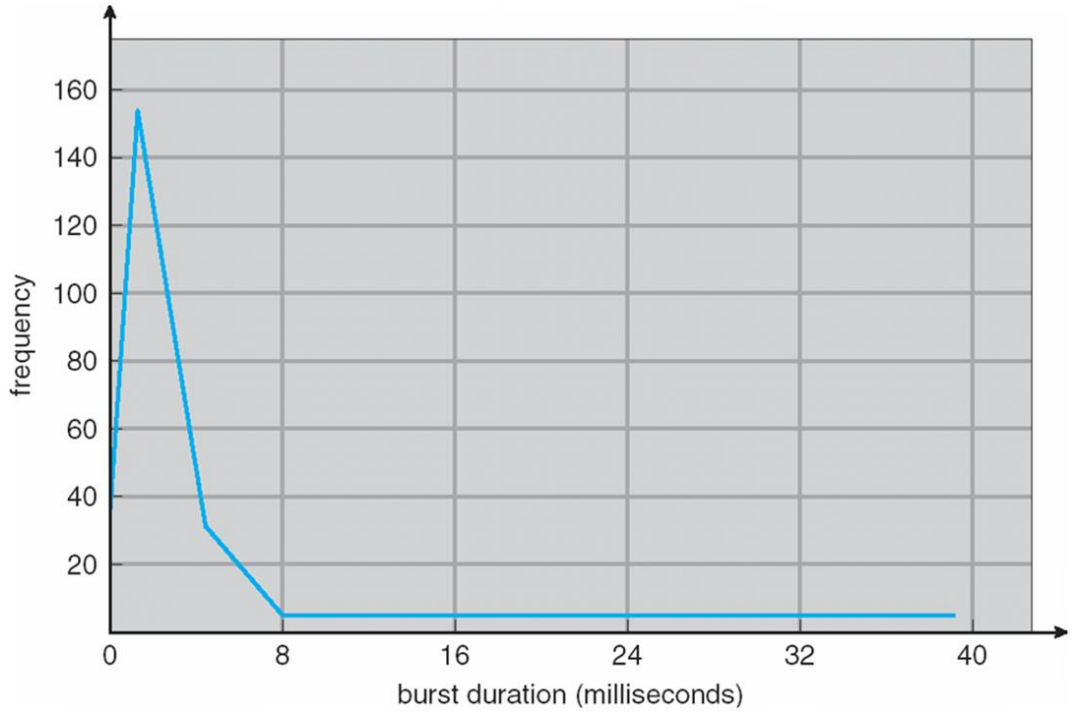
- Maximum CPU utilization obtained with multiprogramming
- Process execution consists of a number of CPU execution and I/O waiting bursts.
  - A process begins with a CPU burst.
  - When a process waits for I/O, it enters an I/O burst.
  - A process alternates between CPU bursts and I/O bursts.
  - Eventually a CPU burst will end with process termination.
- The length of CPU bursts vary by process and computer.
- CPU burst distribution is of main concern.



Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts



Preemptive vs. Non-preemptive scheduling

- Non-preemptive scheduling:
  - Each process completes its full CPU burst cycle before the next process is scheduled.
  - No time slicing or CPU stealing occurs.
  - Once a process has control of the CPU, it keeps control until it gives it up (e.g. to wait for I/O or to terminate).
  - Works OK for batch processing systems, but not suitable for time sharing systems.
- Preemptive scheduling:
  - A process may be interrupted during a CPU burst and another process scheduled. (E.g. if the time slice of the first process expires).
  - More expensive implementation due to process switching.
  - Used in all time sharing and real time systems.

CPU Scheduling

- Scheduler selects from among the processes 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
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Which of those cases would happen for a non-preemptive scheduler?

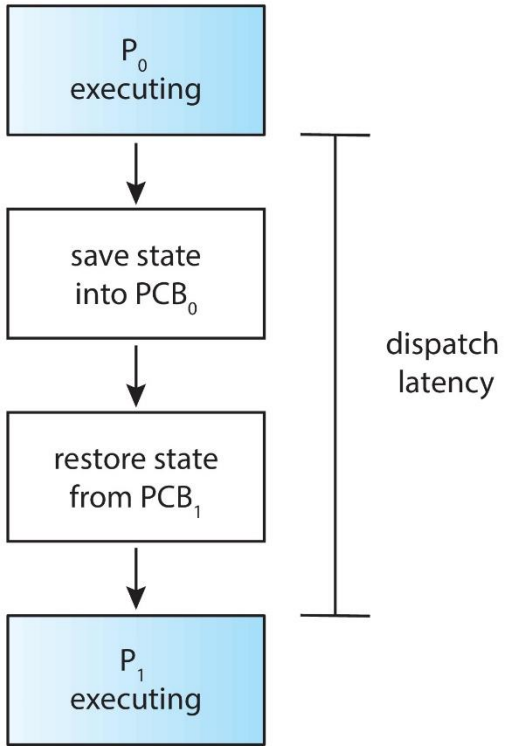
A	1 only
B	2 only
C	2 and 3
D	1 and 4
E	All of them

Costs of Preemptive Scheduling

- Preemptive scheduling leads to some problems that the OS must deal with:
- Problem 1: inconsistent data:
  - Suppose process 1 is updating data when preempted by process 2.
  - Process 2 may then try to read the data, which is in an inconsistent state.
  - The OS needs mechanisms to coordinate shared data.
- Problem 2: Kernel preemption:
  - Suppose the kernel is preempted while updating data (e.g. I/O queues) used by other kernel functions. This could lead to chaos.
  - UNIX solution – non-preemptive kernel: Wait for the system call to complete or an I/O block take place if in kernel mode.
  - Problem with UNIX solution: Not good for real time computing.

Dispatcher

- Process scheduling determines the order in which processes execute.
- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running



Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes 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, **not** output (for time-sharing environment)

Optimization Criteria

- The scheduling criteria are optimization problems. We would like to maximize or minimize each.
- Question: Maximize or Minimize?
  - CPU utilization:
  - throughput:
  - turnaround time:
  - waiting time:
  - response time:
- Can all criteria be optimized simultaneously?
- Usually try to optimize average times (although sometimes optimize minimum or maximum)

Example: Computing Criteria

- Suppose two processes arrive at time zero.
- P1: CPU burst t1 = 7P2: CPU burst t2 = 11
- Assume 1 unit of time for dispatch of process.
- Assume non-preemptive scheduling.
- Compute the following:
  - Throughput:
  - CPU utilization:
  - Avg. Turnaround:
  - Avg. Wait time:
- Suppose P2 arrives at time 4. What statistics change? What are their new values?

Recall Scheduling Criteria

- CPU utilization –fraction of time the CPU is busy.
- **CPU efficiency** – fraction of time the CPU is executing user code.
- Throughput – # of processes completed per unit time
- Average Turnaround time – average delay between job submission and job completion.
- 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.

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

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

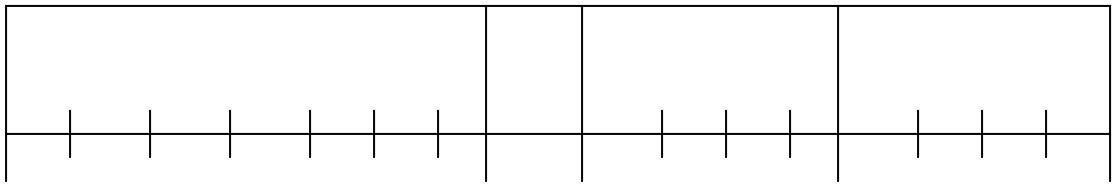
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- 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 Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

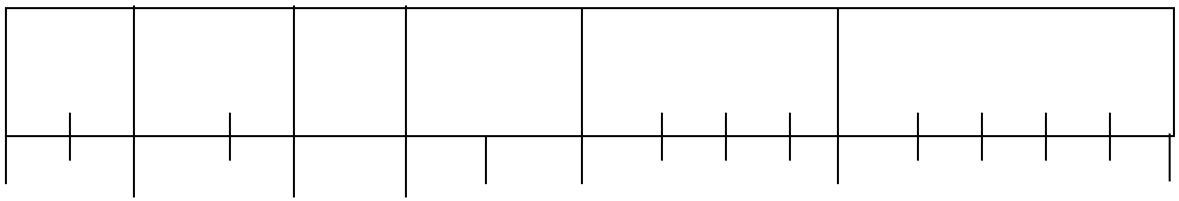


- Average waiting time =

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time =

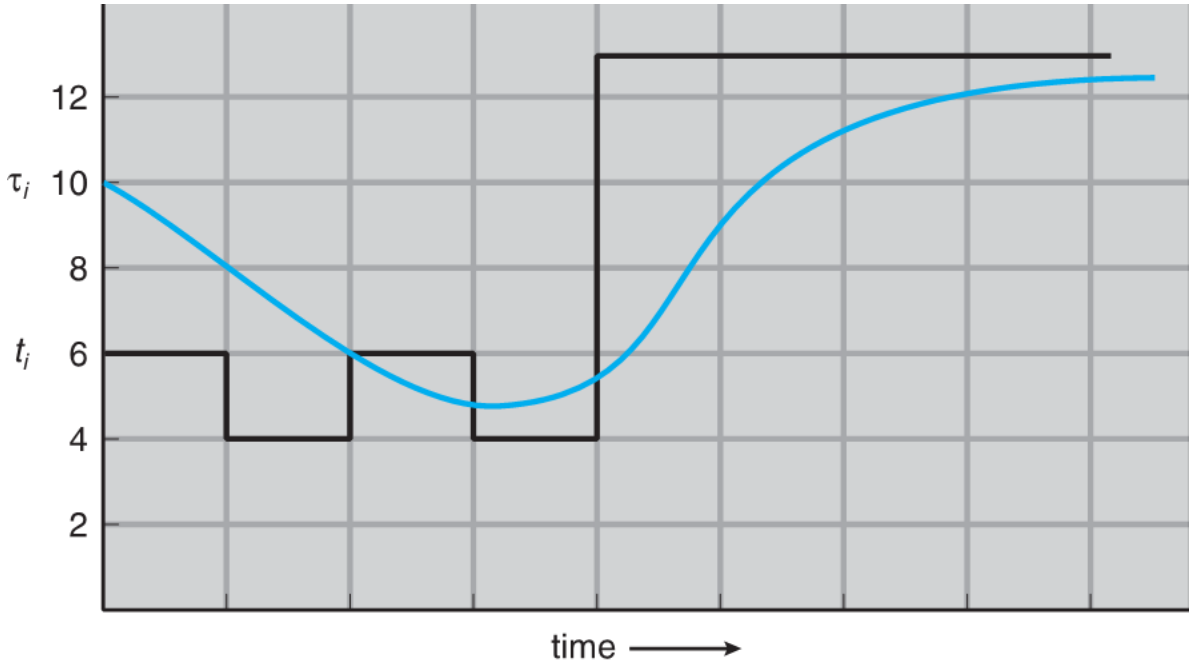
Determining Length of Next CPU Burst

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.

1. $t_n = \text{actual lenght of } n^{th} \text{CPU burst}$
2. $\tau_{n+1} = \text{predicted value for the next CPU burst}$
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

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

Prediction of the Length of the Next CPU Burst



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

Examples of Exponential Averaging

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

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

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

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (e.g. smallest integer ≡ highest priority)
  - Preemptive
  - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process
- Question: Is starvation possible with SJF? How about FCFS?

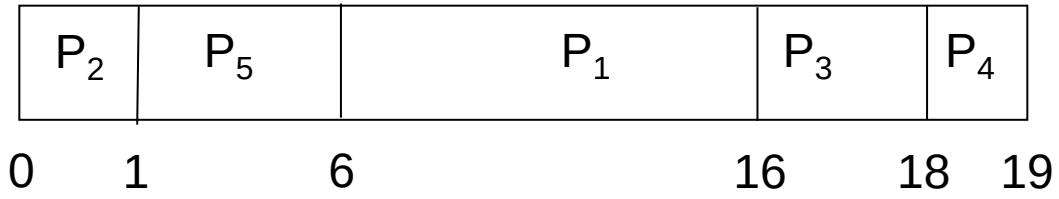
Assigning Priorities

- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Other bases for assigning priority:
  - Memory requirements
  - Number of open files
  - Avg I/O burst / Avg CPU burst
  - External requirements (amount of money paid, political factors, etc).

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 (lower value means higher priority)

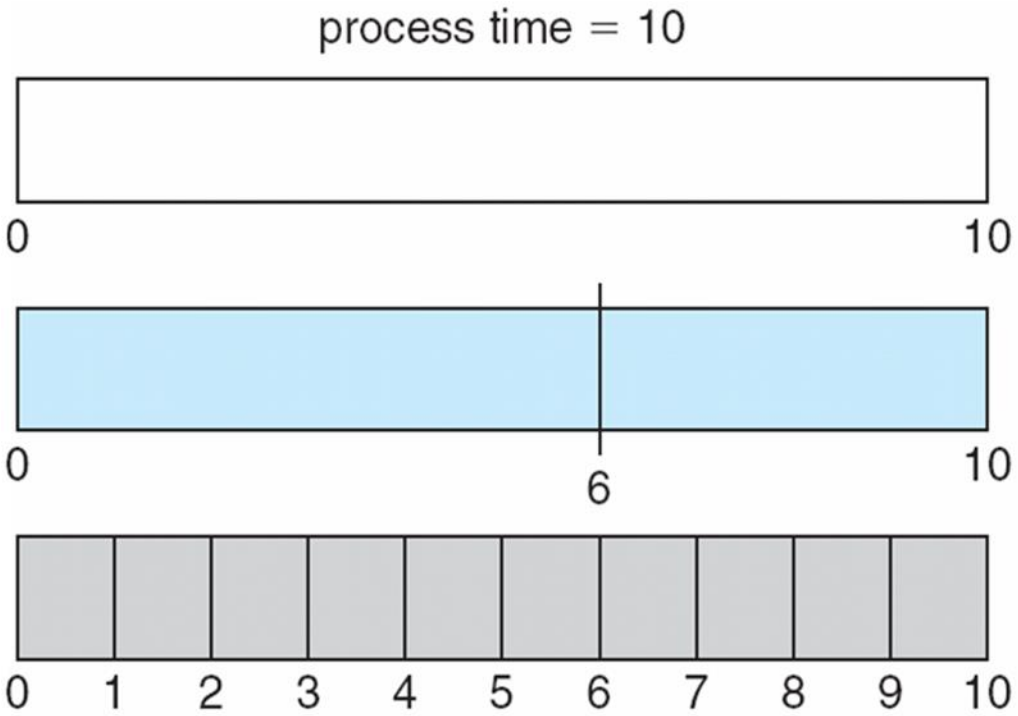


Average waiting time = 8.2

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum/slice*), 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.
- RR makes the implicit assumption that all processes are equally important.
  - Cannot use RR directly if you want different processes to have different priorities.

Time Quantum and Context Switch

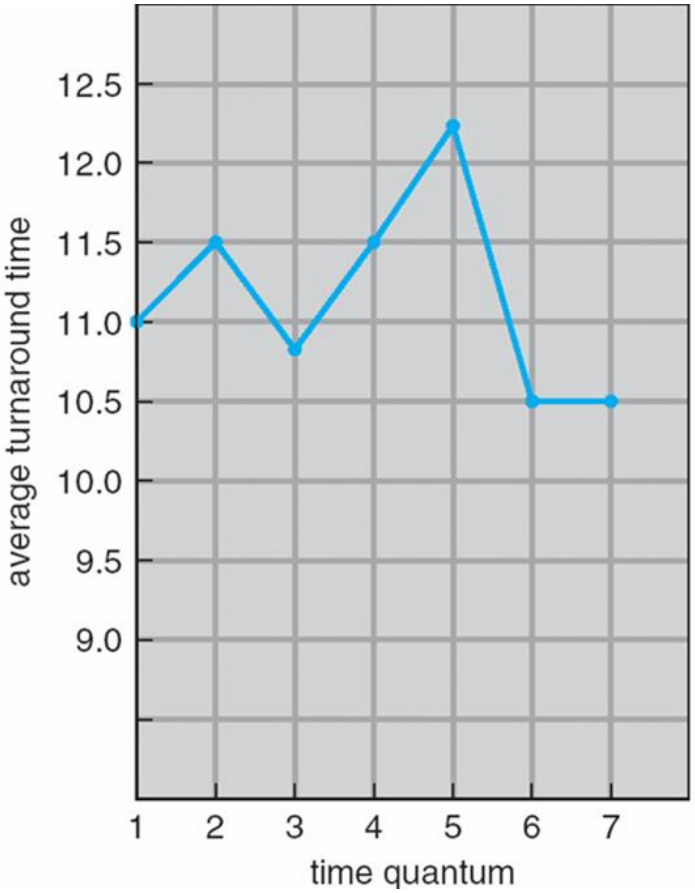


quantum	context switches
12	0
6	1
1	9

RR Performance

- Performance varies with the size of the time slice, but not in a simple way.
- Short time slice leads to faster interactive response.
  - Problem: Adds lots of context switches. High Overhead.
  - If time slice = 20 and process switch time = 5, then 5/25 = 20% of CPU time spent on overhead.
  - The time slice should be large enough compared to the process switch time.
  - q usually 10ms to 100ms, context switch < 10 µsec
- Longer time slice leads to better system throughput (lower overhead), but response time is worse.
  - If time slice is too long, RR becomes just like FCFS.

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	6
P_2	3
P_3	1
P_4	7

- Draw The Gantt chart of RR with time quantum = 4
  - Compute: Avg turnaround, avg waiting time
- Suppose P_1 arrives at 0, P_2 at 3, P_3 at 8 and P_4 at 9. What are the new values of avg turnaround time for RR?
- Typically, RR has higher average turnaround than SJF, but better *response*

RR Example

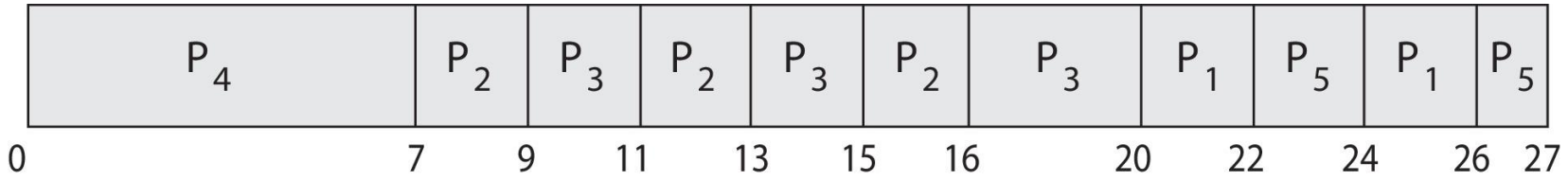
Process	Burst Time	Arrival time
P_1	6	0
P_2	3	3
P_3	1	8
P_4	7	9

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

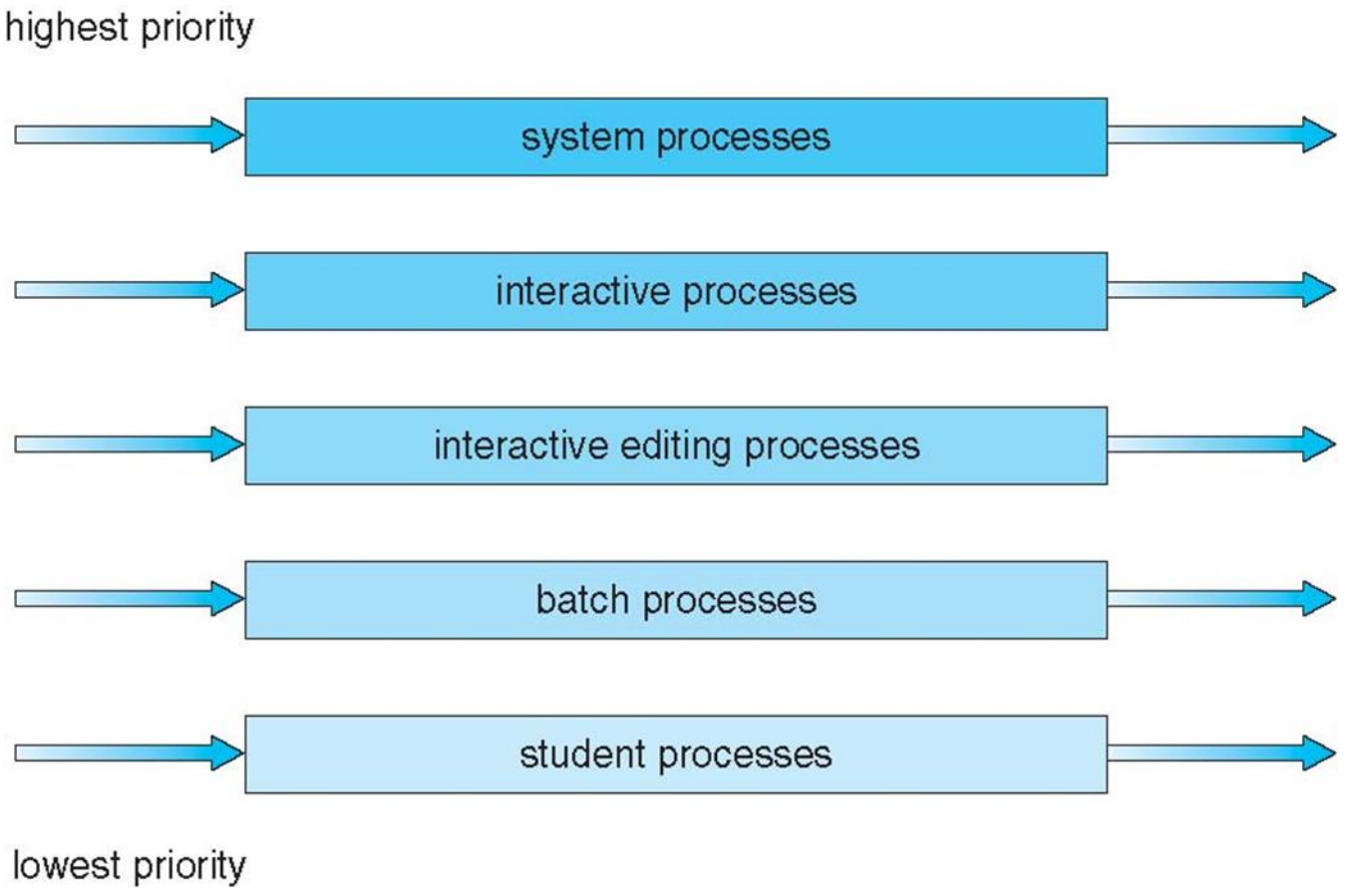
- Gantt Chart wit 2 ms time quantum



Multilevel Queue

- Ready queue is partitioned into separate queues: e.g. foreground (interactive) and background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling



Multilevel Feedback Queue

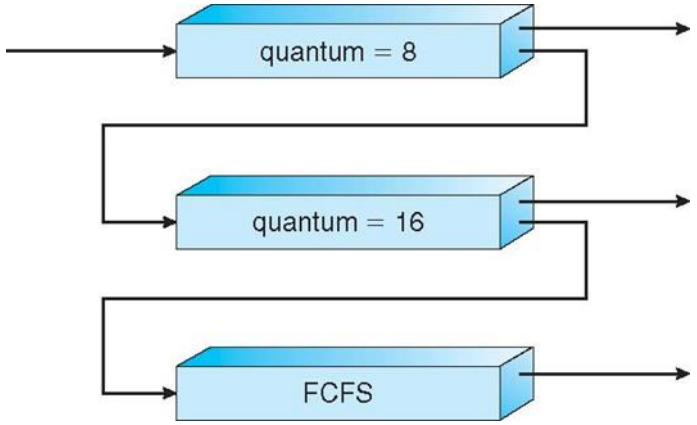
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine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

Typical behavior for multilevel feedback queues

- New processes go to the highest priority queue for that job type.
  - Top priorities reserved for system processes.
- Higher priority queues have smaller time slices than lower priority queues.
- If process uses full time slice, it moves down a priority.
- If process blocks before using full time slice, it remains at the same priority.

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served RR. 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 RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

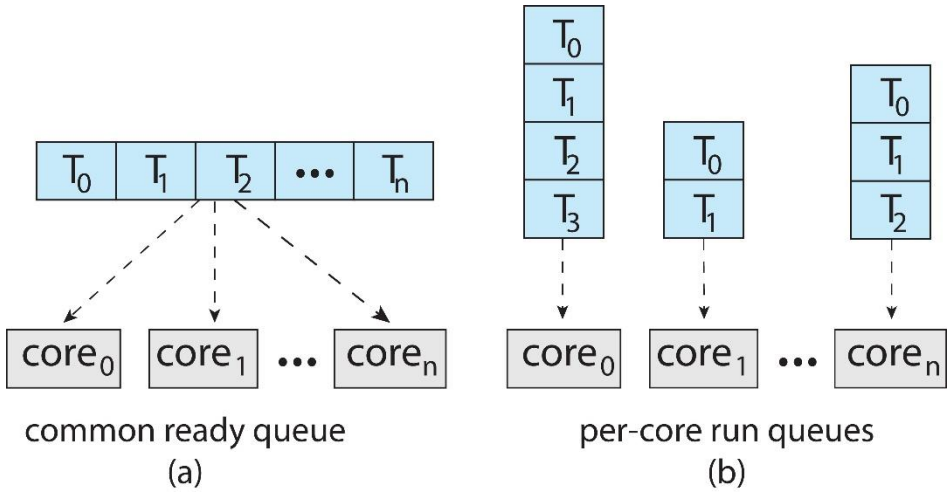


Notes on Multilevel Feedback Queues

- Short processes are favored.
- Good for interactive processes with short CPU bursts.
- Is starvation Possible?
  - Yes--Long processes may wait forever.
  - To avoid--increase the priority of a process if it has been waiting for some period of time in some queue.

Scheduling Multiple Processors

- CPU scheduling more complex when multiple CPUs are available. Want **load sharing** between processors
  - **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
  - **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes

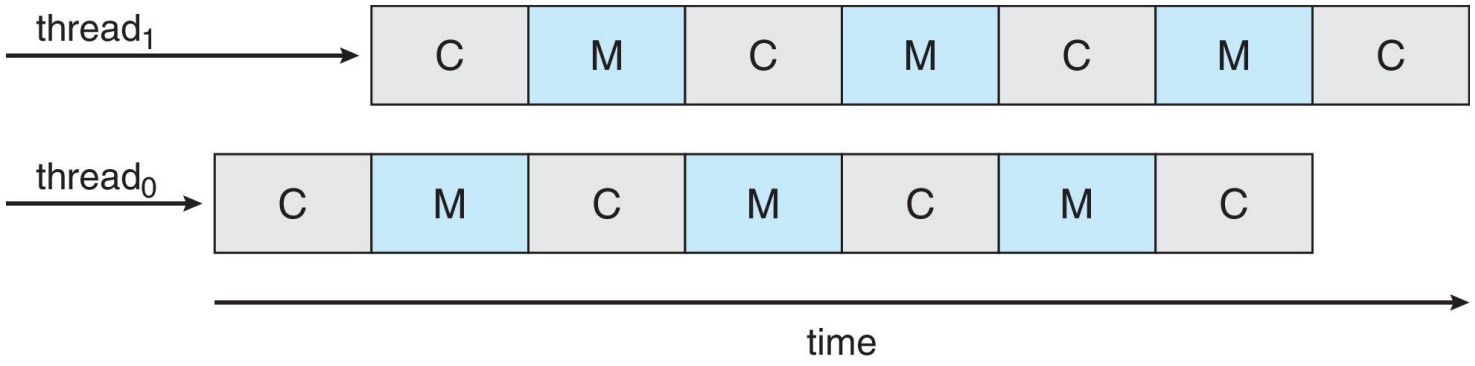


Issues in Multiple processor scheduling

- **Processor affinity** – process has affinity for processor on which it is currently running. Why?
  - A thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
  - **soft affinity**
  - **hard affinity**
- **Load balancing** attempts to keep workload evenly distributed
  - 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.
  - **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

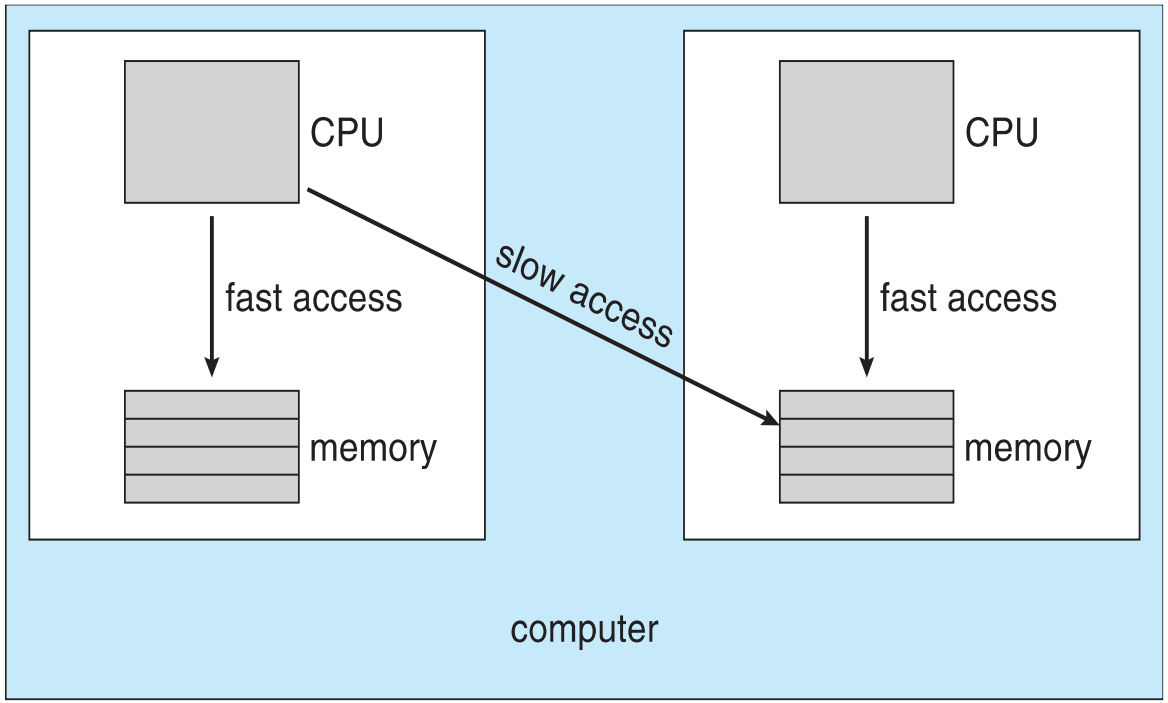
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - assigns each core multiple hardware threads. (Intel refers to this as hyperthreading.)
  - If one thread has a memory stall, switch to another thread!



NUMA and CPU Scheduling

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



Operating System Examples

- Windows scheduling
- Solaris scheduling
- Linux scheduling

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- 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
- Queue for each priority
- If no run-able thread, runs **idle thread**

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
- 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
- Default base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x longer time quantum

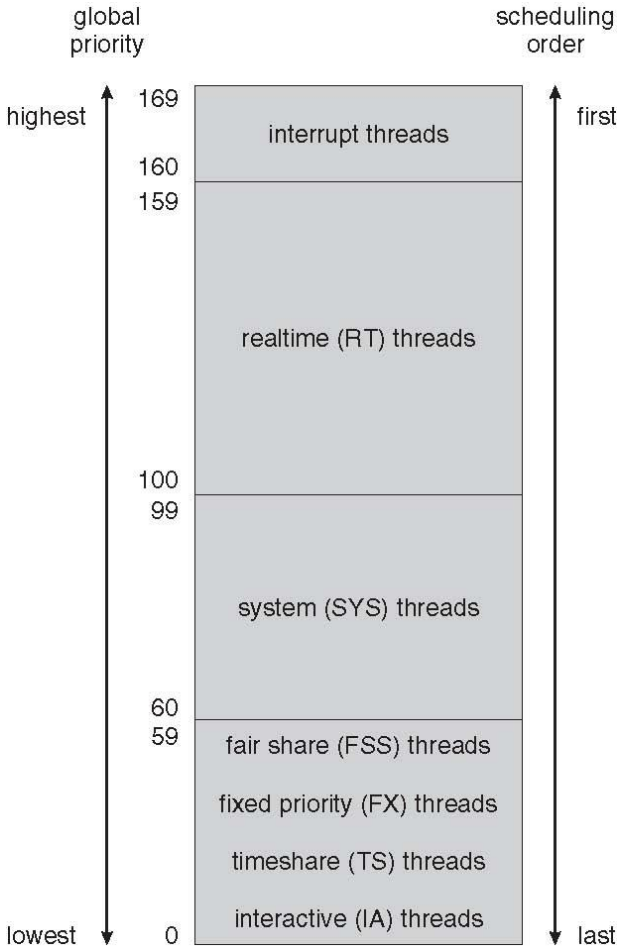
Windows Priorities

	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

Solaris 2 Scheduling

- Priority-based thread scheduling
- Thread in one of six classes
  - Each class has its own scheduling algorithm
- Interrupt and real-time processes have the highest priority
- Multi-level feedback queue is used for interactive and time-sharing processes.
  - Configurable 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

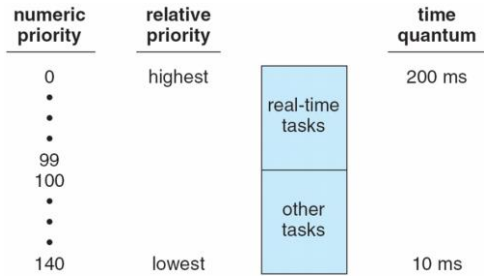


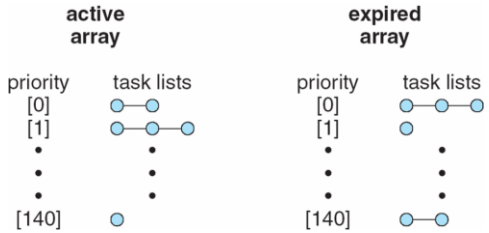
Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Linux Scheduling Through Version 2.5

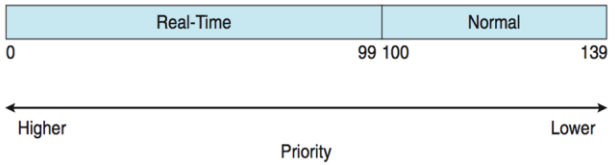
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 - 99 and **other tasks** value from 100 -140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger time slice q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes





Linux Scheduling in Version 2.6.23 +

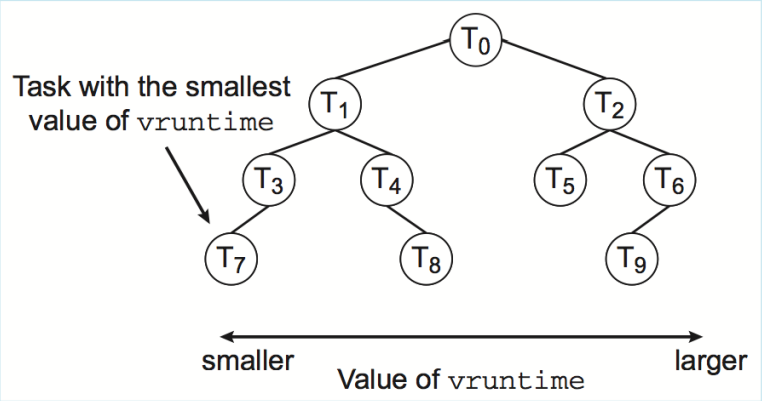
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - 2 scheduling classes included, others can be added
    1. Default – using CFS scheduling
    2. real-time



- ***Completely Fair Scheduler*** (CFS)
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority, which gets larger proportion of CPU time.
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- 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.

Algorithm Evaluation

- How do we choose a scheduling algorithm?
- First, choose the **criteria** you want to use to judge the algorithm. E.g.
  - Maximize the CPU utilization under the constraint that the maximum response time is 1 second.
  - Maximize throughput such that turnaround time is on average linearly proportional to total execution time.
- Next, evaluate the algorithms under consideration.
- Methods of evaluation:
  - Analytic evaluation (Deterministic modeling)
  - Queueing models
  - Simulations
  - Implementation

Deterministic Modeling

- **Analytic Evaluation** is a method in which a given algorithm and system workload are used to produce a formula or number that evaluates the performance of the algorithm **for that particular workload.**

- **Deterministic modeling** is one type of analytic evaluation.

Deterministic modeling

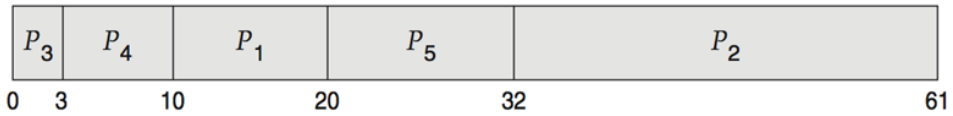
- **Analytic Evaluation** is a method in which a given algorithm and system workload are used to produce a formula or number that evaluates the performance of the algorithm **for that particular workload.**
- **Deterministic modeling** is one type of analytic evaluation.
- Consider 5 processes arriving at time 0:

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

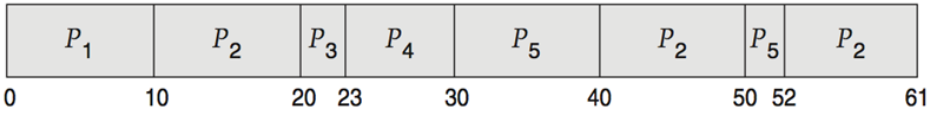
Deterministic Evaluation

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

  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Pros and Cons of Deterministic Modeling

- Advantages of deterministic modeling
  - Simple and fast
  - Gives an exact number
  - Makes it easy to compare algorithms

- Disadvantages:
  - Requires exact numbers for input.
  - Answers only apply to the specific case tested.
  - Does not give a general answer (too specific).

Queueing Models

- **Network queueing analysis** models the system as a set of servers with associated queues of waiting processes:
  - CPU and ready queue
  - I/O system and device queues
  - etc.
- This method uses information about the **distribution** of CPU and I/O bursts (determined by measurement or estimation).
- The method also requires a distribution of process arrival times (either measured or estimated).
- Using the above information, queueing analysis can be used to compute CPU 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:

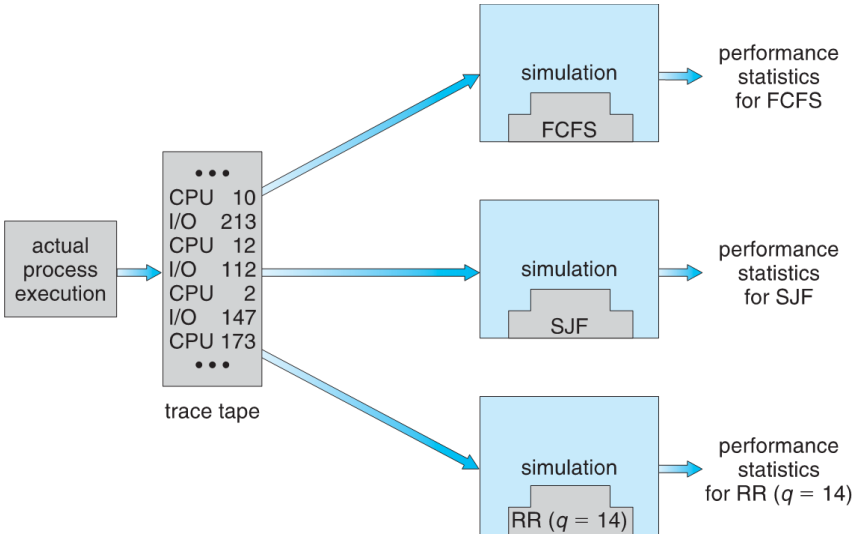
$$n = \lambda \times W$$

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

Limitations of Queueing analysis

- There are limited classes of algorithms and distributions that can be handled by queueing analysis.
- Mathematics can be difficult to work with.
  - Distributions are often defined in mathematically tractable ways that are unrealistic.
- To work with the mathematics, inaccurate assumptions may be made.
- The theory therefore only yields approximate answers.

Simulations



- Simulations are programs that models the components of the computer system. (e.g. a variable is used to represent the system clock).
- The simulator modifies the system state to reflect the activities of the devices, scheduler and processes.
- The simulator keeps track of statistics about system performance.
- Input data (arrival times and burst times) can be generated by either:
  - A random number generator
  - A trace tape (a record of arrival and burst times on an actual system).
- Drawbacks of simulation:
  - Simulation is expensive. It can take hours of computing time.
  - More detail yields more accurate results, but takes more time to compute.

Implementation

- The most accurate way to measure performance is to implement the algorithm and run it.
- Drawbacks:
  - Expensive--Coding the algorithm and modifying the O.S. to support it can take many hours of programming.
  - The environment in which the algorithm is used may change, so that the initial results no longer apply.
- Most flexible schedulers can be altered per-site or per-system.
  - Separate mechanism and policy