Chapter 5: CPU Scheduling

http://www2.cs.uregina.ca/~hamilton/courses/330/notes/scheduling/scheduling.html

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

CPU Scheduler

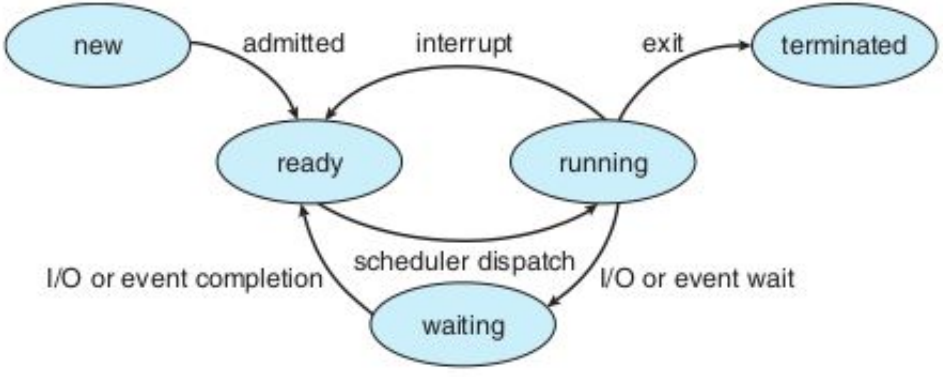


Figure 3.2 Diagram of process state.

CPU scheduling 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

nonpreemptive schedulers use 1 & 4 only

preemptive schedulers run at all four points

- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - How to order **Queue?**

Preemptive and Nonpreemptive Scheduling

- **nonpreemptive:** once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
  - Otherwise, it is **preemptive**.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use **preemptive** scheduling algorithms.

Exercise

Which scheduling may cause Race Conditions?

A. Preemptive
B. Non-preemptive

Preemptive scheduling

- **Example:** two processes **P1** and **P2** that share data.
  - P1 is updating data,
    - before it is finished, it is preempted.
  - P2 starts running and tries to read data
    - the data is in an inconsistent state

- more detail next week.

Why do we care?

- What goals should we have for a scheduling algorithm?

Scheduling criteria

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming

CPU–I/O Burst Cycle

- Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

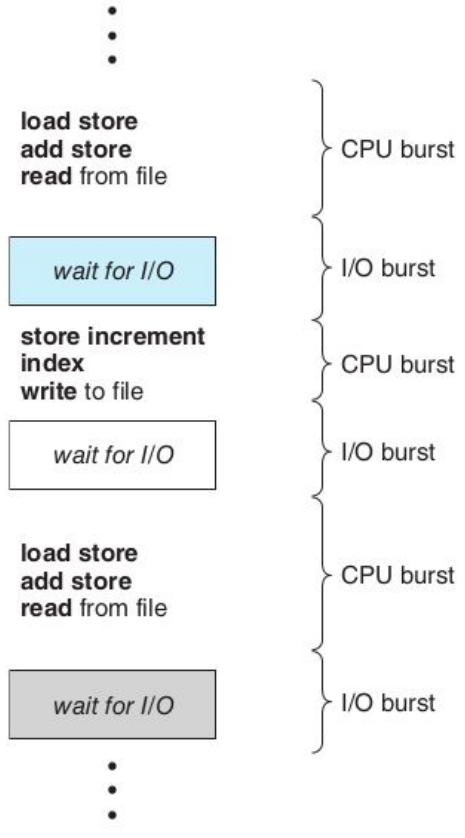


Figure 5.1 Alternating sequence of CPU and I/O bursts.

Histogram of CPU-burst Times

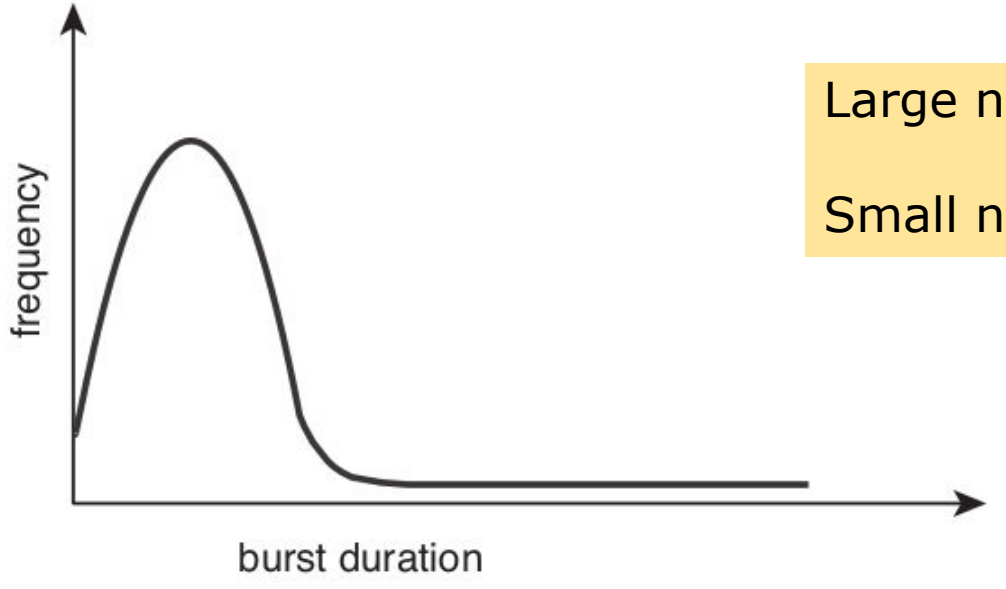


Figure 5.2 Histogram of CPU-burst durations.

Large number of short bursts

Small number of longer bursts

Dispatcher

Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

in linux to see #context switches:

- **vmstat 1 3**
- for process 2166
  - **cat /proc/2166/status**

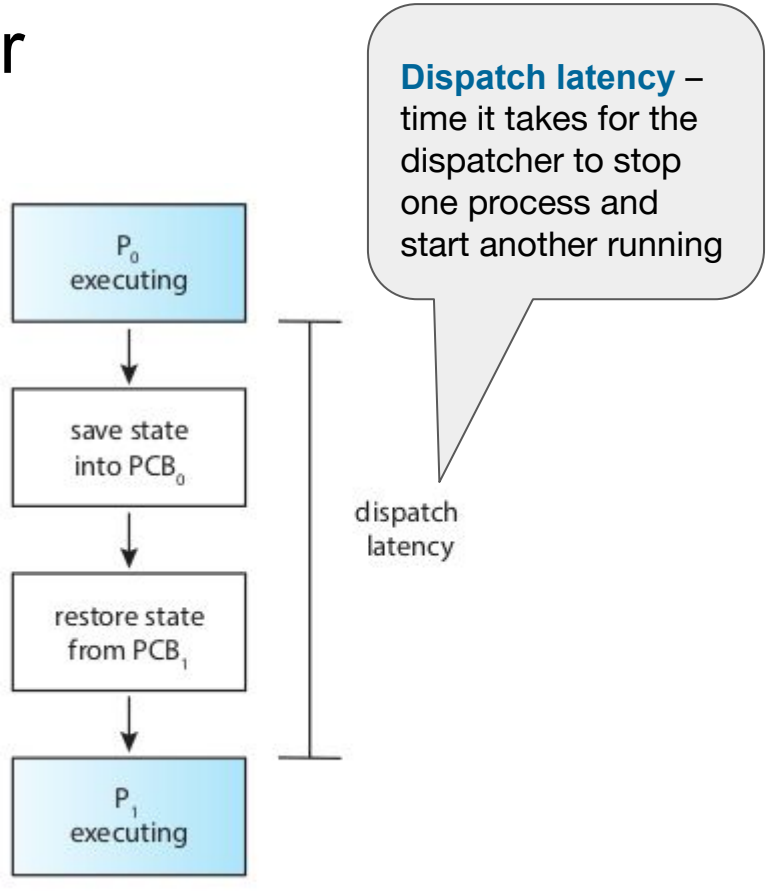


Figure 5.3 The role of the dispatcher.

Scheduling Criteria

What goals should we have for a scheduling algorithm? How to compare one algorithm with another?

- **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.

Scheduling Algorithm Optimization Criteria

What goals should we have for a scheduling algorithm?

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

Scheduling Algorithms

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 times
    - $WTP_1 = 0$
    - $WTP_2 = 24$
    - $WTP_3 = 27$
  - Average waiting time: AWT = (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

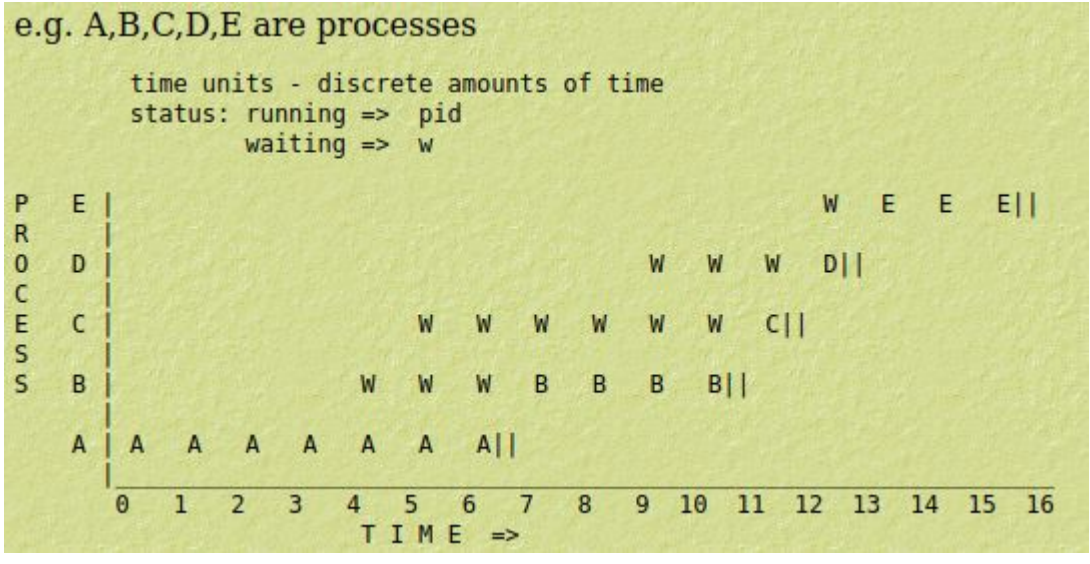
P_2 , P_3 , P_1

- Waiting times? Awt?

- Waiting times
  - $P_1 = 6$
  - $P_2 = 0$
  - $P_3 = 3$
- Average waiting time:
  - AWT = (6 + 0 + 3)/3 = 3

- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Arrival time	Process	required time (t)
0	A	7
4	B	4
5	C	1
9	D	1
12	E	3



FCFS summary

- wonderful for long processes when they finally get on
- terrible for short processes if they are behind a 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

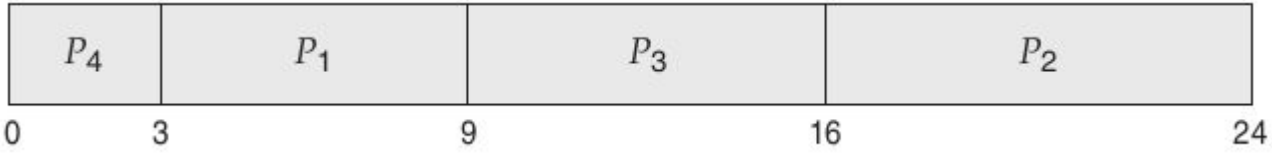
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Shortest-Job-First (SJF) Scheduling

- Preemptive version called **shortest-remaining-time-first**
  - when a process arrives at the ready queue with an expected CPU-burst-time that is less than the expected remaining time of the running process, the new one preempts the running process

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

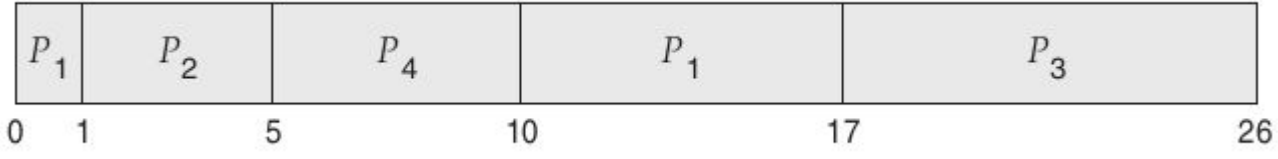


Waiting times?
P1 = 3
P2 = 16
P3 = 9
P4 =0
Average waiting time = 7

- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

Example of Shortest-remaining-time-first (preemptive SJF)

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



- Average waiting time = 6.5
- non-preemptive SJF awt = 7.75

Determining Length of Next CPU Burst

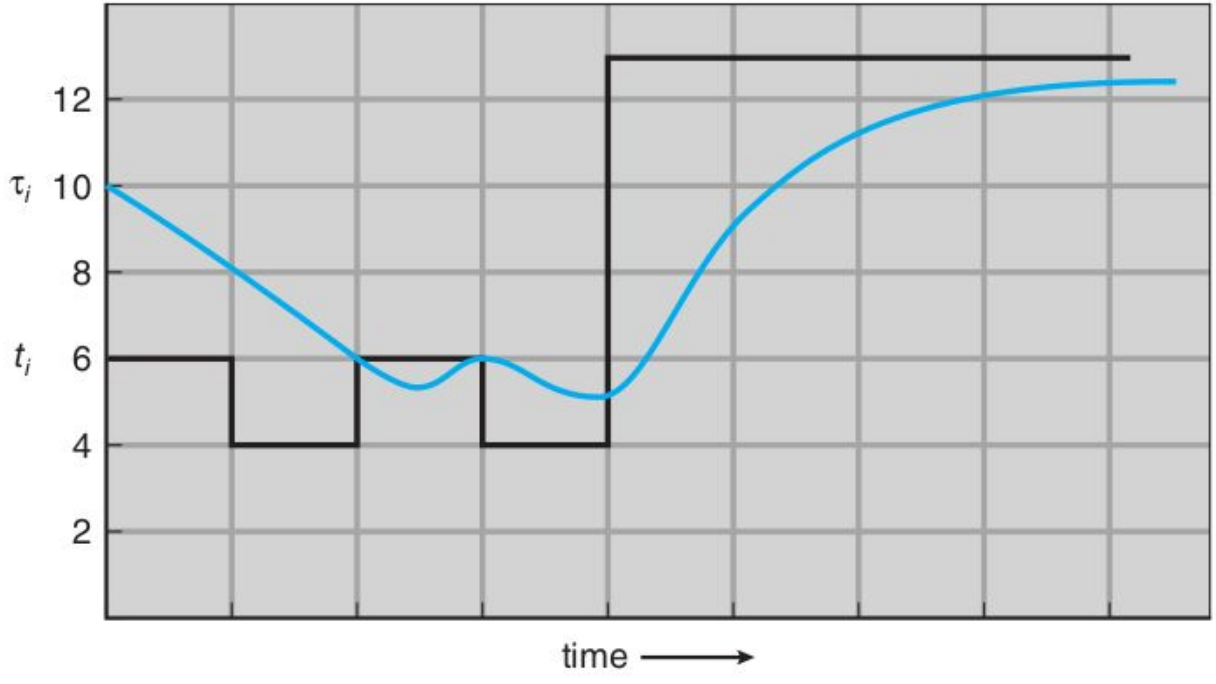
The next CPU-burst is generally predicted as a**n exponential average** of the previous CPU-bursts

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

- **tn**
  - the most recent CPU-burst,
- **τn**
  - the past predicted value (history)
- **α**
  - is a parameter 0 ≤ α ≤ 1

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

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	...

Figure 5.4 Prediction of the length of the next CPU burst.

the expected burst length is calculated as follows:

- **a(t) =** actual amount of time required during cpu burst t
- **e(t) =** amount of time that was expected for cpu burst t
- **e(t+1) =** expected time during the next cpu burst

α is generally 0.1

- if α = 0.5,

e(t+1) = 0.5 * e(t) + 0.5 * a(t)

http://www2.cs.uregina.ca/~hamilton/courses/330/notes/scheduling/scheduling.html

Suppose a process p is given a **default expected burst length** of **5** time units.

- When it is run, the actually burst lengths are
  - 10,
  - 10,
  - 10,
  - 1,
  - 1,
  - 1
  - (this information is not known in advance to any algorithm).
- The prediction of burst times for this process?

Let e(1) = 5, as a default value.

When process p runs,

- its 1st burst actually runs 10 time units, so,

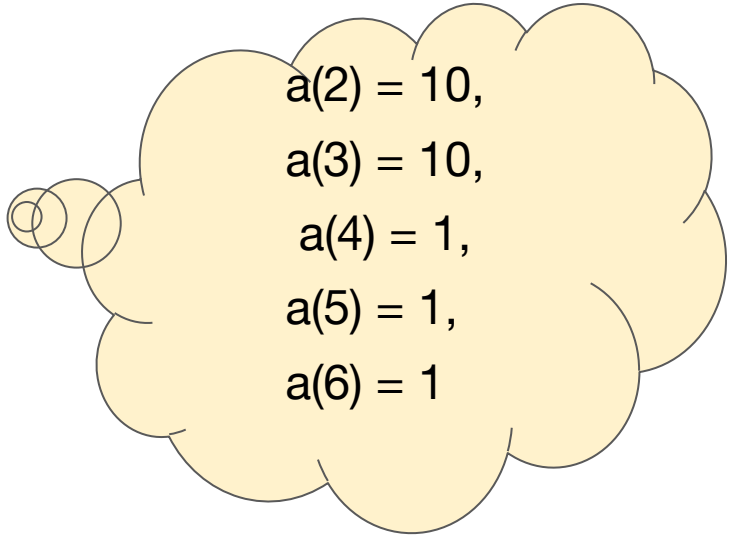
  `a(1) = 10.`

- the prediction for the 2nd cpu burst

  `e(2) = 0.5*e(1) + 0.5*a(1) = 0.5*5 + 0.5*10 = 7.5`

  `e(3) = 0.5*e(2) + 0.5*a(2) = 0.5*7.5 + 0.5*10 = 8.75`

  `e(4) = 0.5*e(3) + 0.5*a(3) = 0.5*8.75 + 0.5*10 = 9.38`



- So, we predict that the next burst will be close to 10 (9.38) because we recent bursts have been of length 10.

At this point, it happens that the process starts having shorter bursts, with a(4) = 1

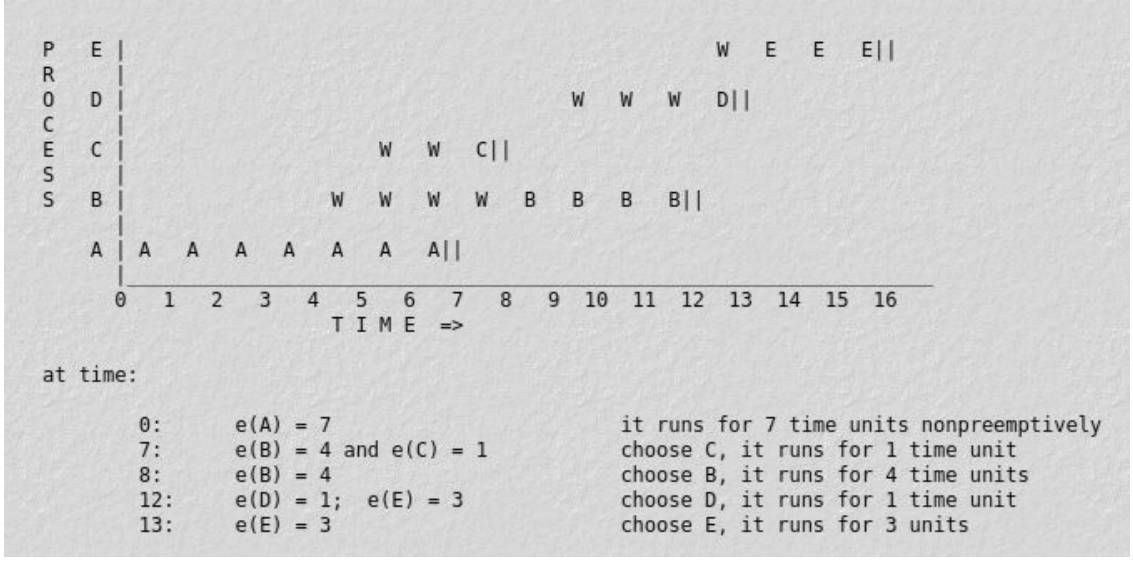
```
e(5) = 0.5*e(4) + 0.5*a(4) = 0.5*9.38 + 0.5*1 = 5.19

e(6) = 0.5*e(5) + 0.5*a(5) = 0.5*5.19 + 0.5*1 = 3.10

e(7) = 0.5*e(6) + 0.5*a(6) = 0.5*3.10 + 0.5*1 = 2.05
```

- Once again, the algorithm has gradually adjusted to the process's recent burst lengths.
- If the bursts lengths continued to be 1, the estimates would continue to adjust until, by rounding, they reached 1.00.

suppose that based on previous information about the processes, our estimates are exactly correct,

- i.e., we expect process A to take 7 units, B to take 4 units, etc.

Arrival time	Process	time (t) required
0	A	7
4	B	4
5	C	1
9	D	1
12	E	3



at time:

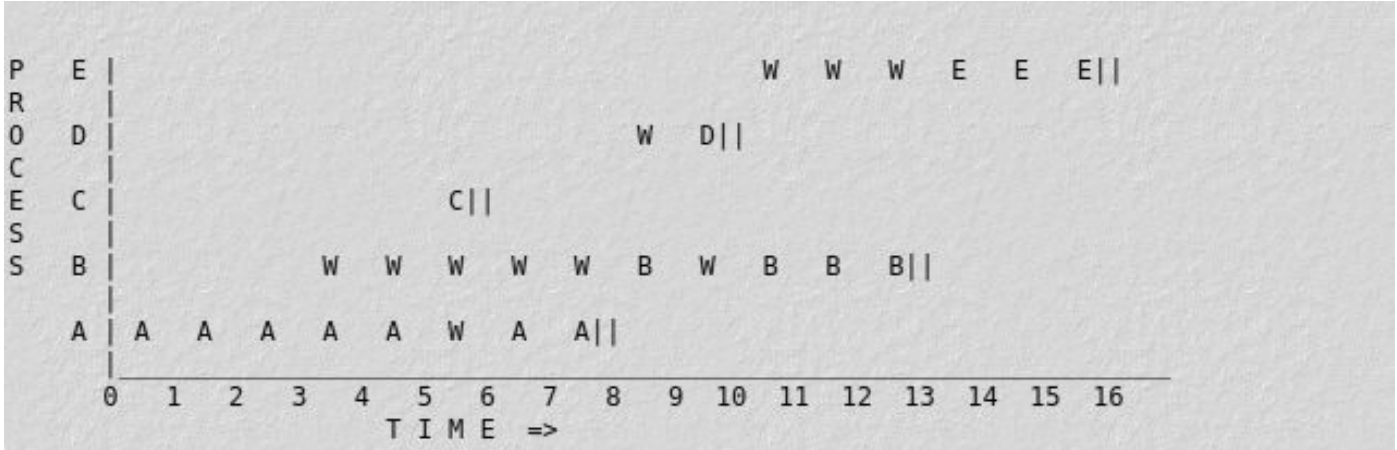
0:	e(A) = 7	it runs for 7 time units nonpreemptively
7:	e(B) = 4 and e(C) = 1	choose C, it runs for 1 time unit
8:	e(B) = 4	choose B, it runs for 4 time units
12:	e(D) = 1; e(E) = 3	choose D, it runs for 1 time unit
13:	e(E) = 3	choose E, it runs for 3 units

Summary of SJF

- very short processes get very good service
- a process may mislead the scheduler if it previously had a short bursts, but now may be cpu intensive (this algorithm fails very badly for such a case)
- the penalty ratios are small; this algorithm works extremely well in most cases
- SJF cannot handle infinite loops
- poor performance for processes with short burst times arriving after a process with a long burst time has started
- processes with long burst times may starve
    - **starvation** - when a process is indefinitely postponed from getting on the processor

Shortest remaining time (SRT) algorithm

- **the new one preempts the running process**
  - when a process arrives at the ready queue with an expected CPU-burst-time that is less than the expected remaining time of the running process,
- long processes can starve



at time:

0:	e(A) = 7			choose A
1:				(no decision required)
2:				
3:				
4:	e(A) = 3;	e(B) = 4		choose A
5:	e(A) = 2;	e(B) = 4;	e(C) = 1	choose C -> done
6:	e(A) = 2;	e(B) = 4		choose A
7:				no decision; A -> done
8:		e(B) = 4		choose B
9:		e(B) = 3;	e(D) = 1	choose D -> done
10:		e(B) = 3		choose B
11:				
12:		e(B) = 1;	e(E) = 3	choose B -> done
13:			e(E) = 3	choose E
14:				
15:				E -> done

Summary of SRT

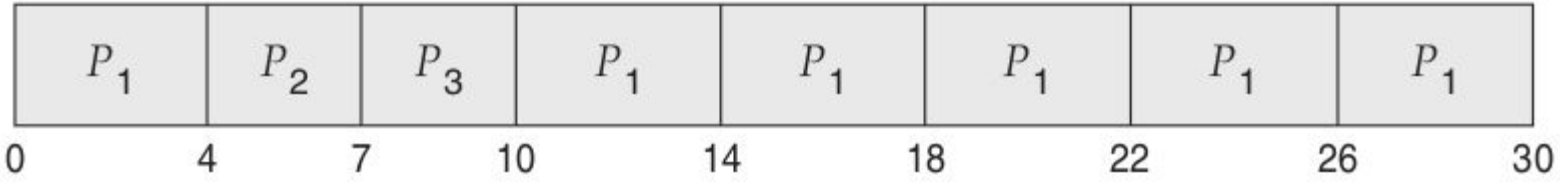
- very short processes get very good service
- a process may mislead the scheduler if it previously ran quickly but now may be cpu intensive (this algorithm fails very badly for such a case)
- the penalty ratios are small;
- this algorithm works extremely well in most cases
- this algorithm provably gives the highest throughput (number of processes completed) of all scheduling algorithms if the estimates are exactly correct.

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 a process finishes early, before its quantum expires, the next process starts immediately and gets a full quantum
    - in some implementations, the next process may get only the rest of the quantum
      - assume a new process arrives and goes into the queue before the process is removed from the processor

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3



- AWT = 5.66,
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

- 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.
- process **waiting time < `(n-1)q`**
- Timer interrupts every quantum to schedule next process
- Performance
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high

Time Quantum and Context Switch Time

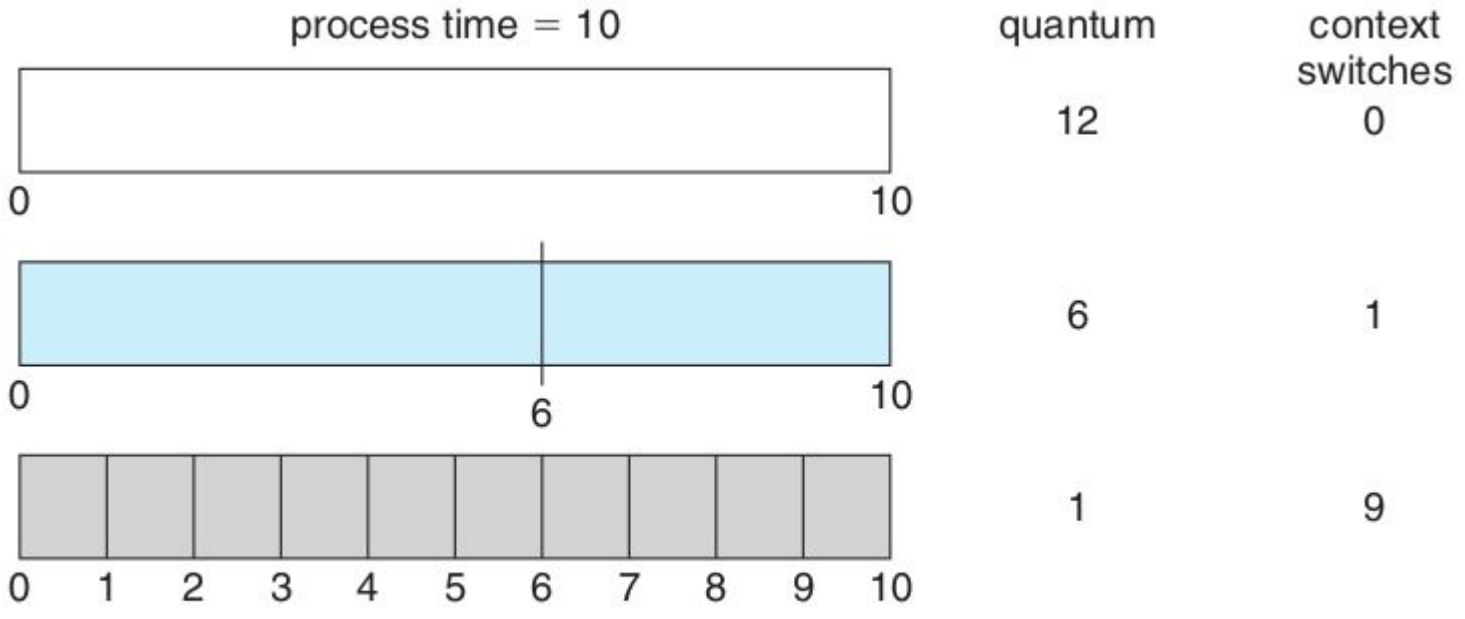
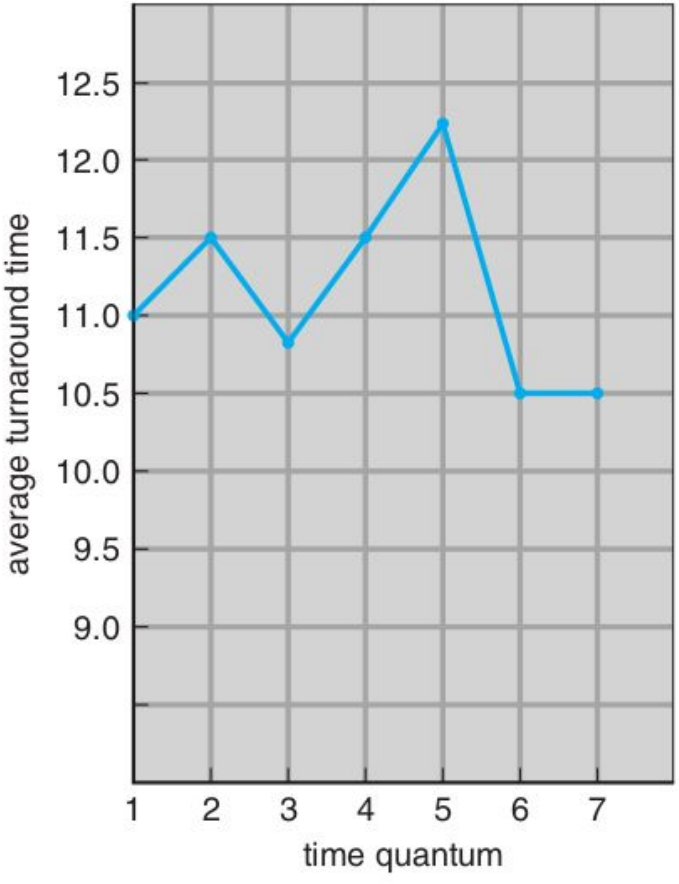


Figure 5.5 How a smaller time quantum increases context switches.

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- q should be > the context switch time,
- But, it should not be too large.
  - if the time quantum is too large,
    - RR scheduling degenerates to an FCFS policy.
- A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

Figure 5.6 How turnaround time varies with the time quantum.

e.g. let quantum (q) = 1.

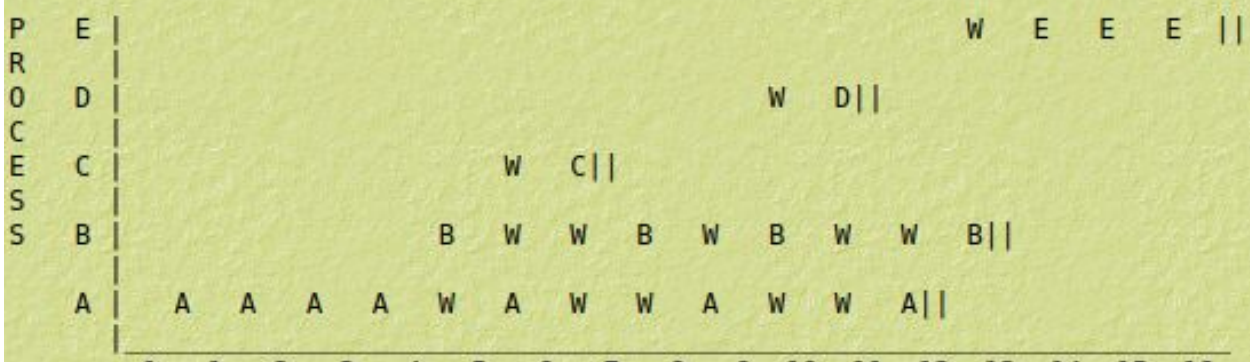


TABLE:

T = elapsed time (includes waiting)
t = processing time required
M = T-t = missed (idle) time
R = t/T = ratio (response) time
P = T/t = penalty rate = 1/R

Arrival time	Process	time (t) required	elapsed time (T)	missed time	ratio	penalty
0	A	7	12	5	7/12	12/7
4	B	4	9	5	4/9	9/4
5	C	1	2	1	1/2	2
9	D	1	2	1	1/2	2
12	E	3	4	1	3/4	4/3

Comparison to FCFS

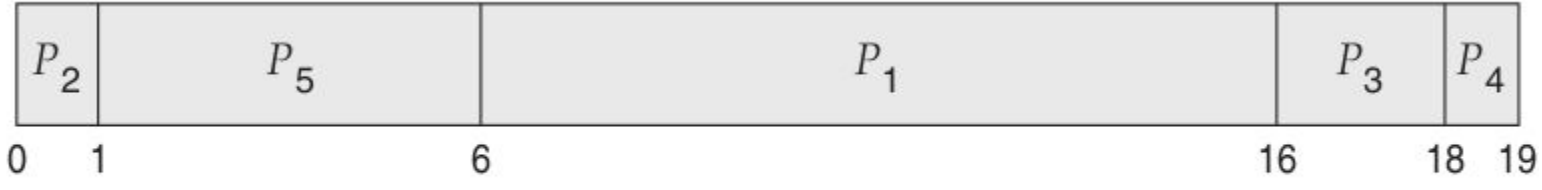
- RR has a much lower penalty ratio than FCFS for processes with short cpu bursts
- RR gives the processes with short bursts (interactive work) much better service
- RR gives processes with long bursts somewhat worse service, but greatly benefits processes with short bursts and the long processes do not need to wait that much longer
- RR is preemptive, that is sometimes the processor is taken away from a process that can still use it
- FCFS is not preemptive.

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

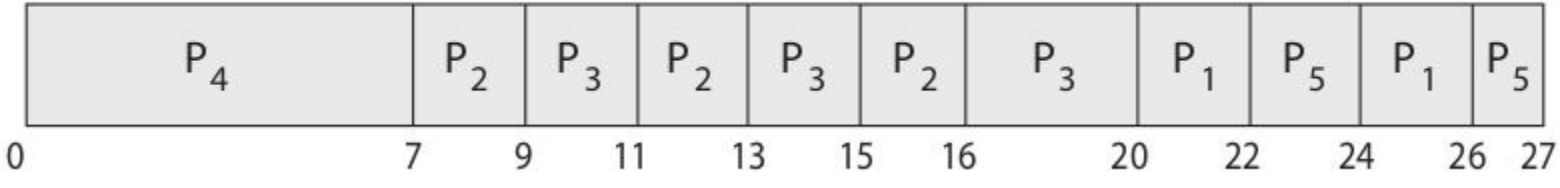


The average waiting time is 8.2 milliseconds.

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

using a time quantum of 2 milliseconds:



Multilevel Queue

- In practice, it is often easier to have separate queues for each distinct priority,
- and priority scheduling simply schedules the process in the highest-priority queue.

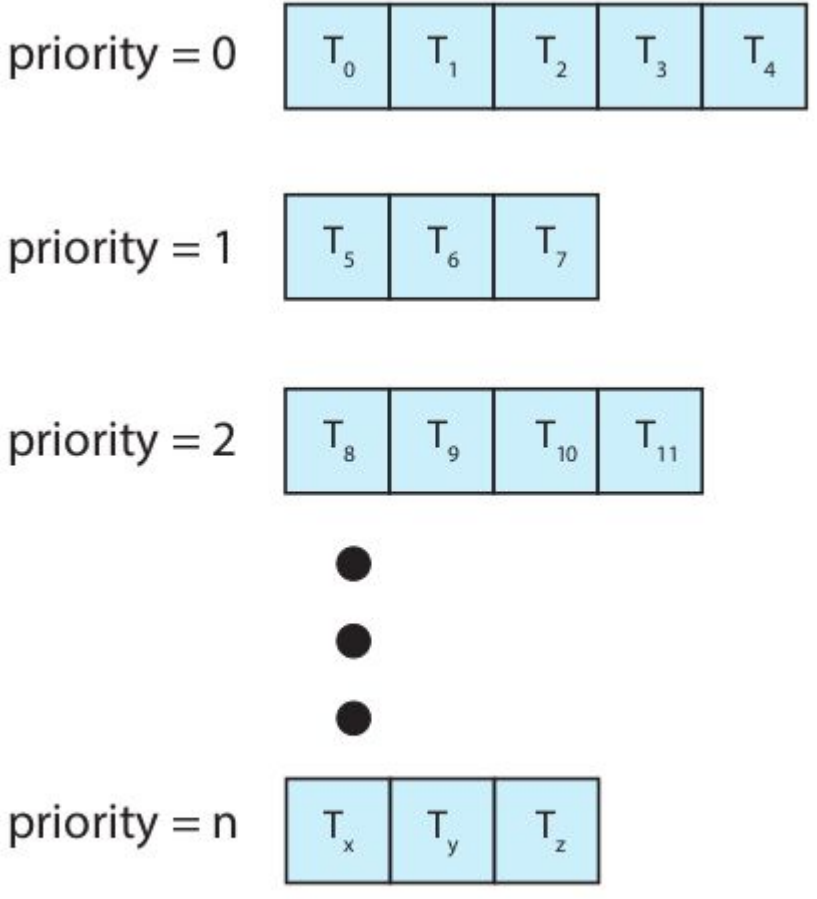
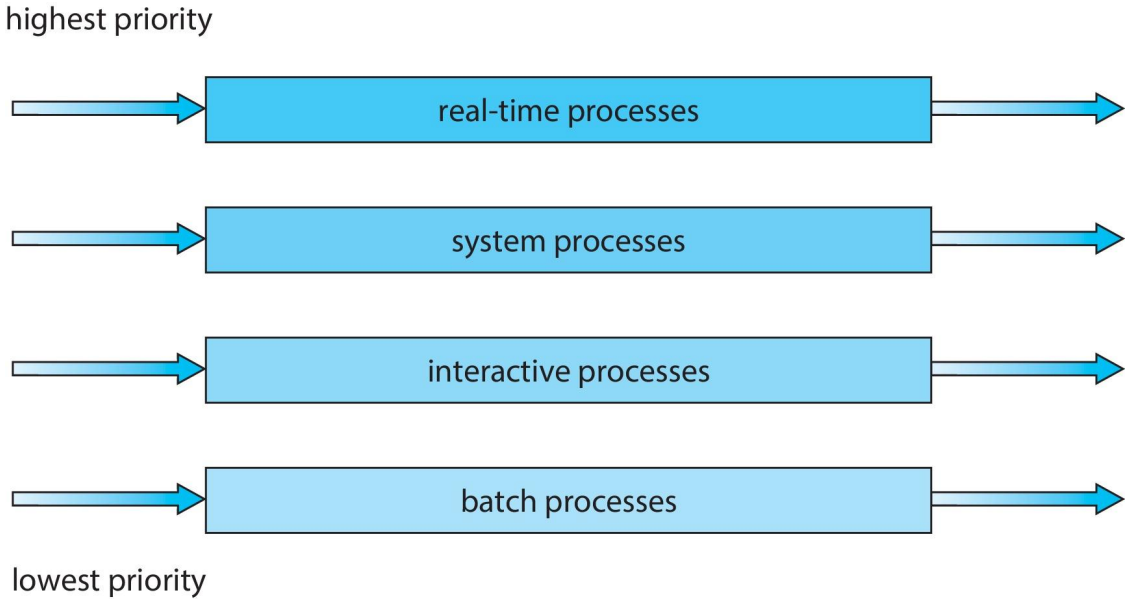
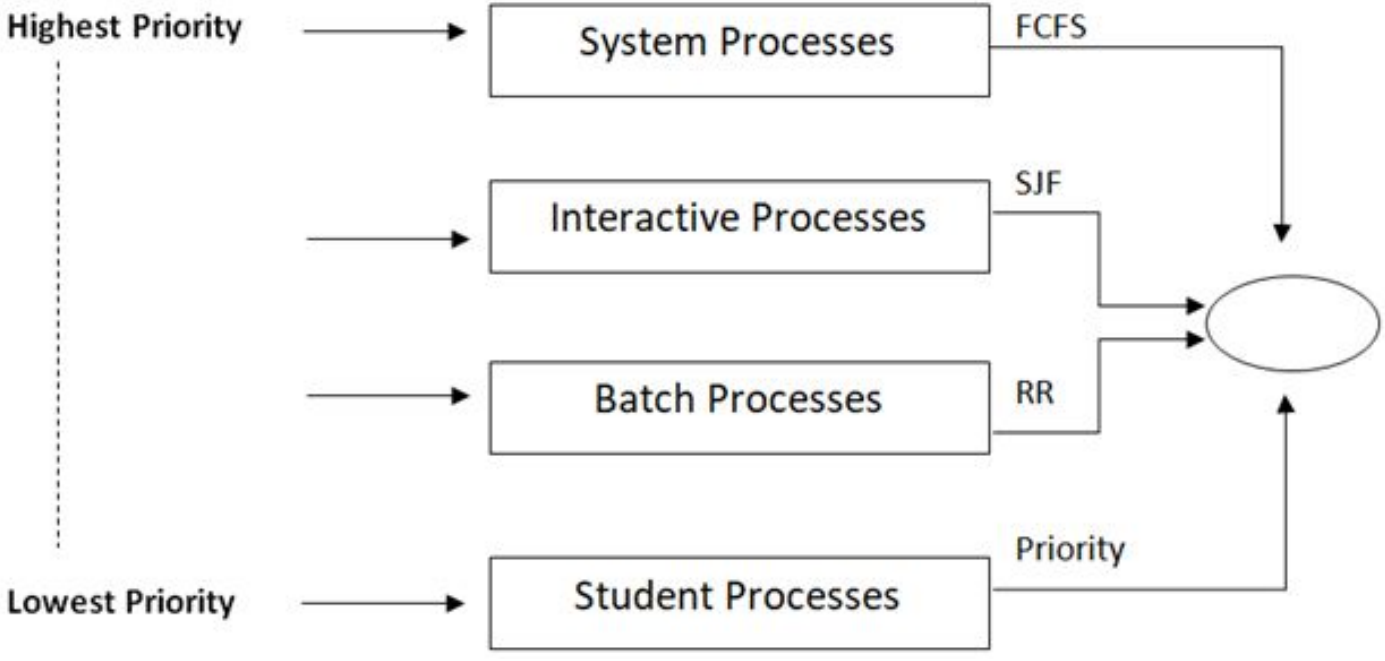


Figure 5.7 Separate queues for each priority.

Multilevel Queue Scheduling

- A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type





Different type of process has different scheduling algorithm, as per requirement.

Lowest priority process gets starvation for the higher priority process because here priority is static.

https://mycareerwise.com/content/multi-level-queue-scheduling/content/exam/gate/computer-science

Multilevel Feedback Queue

- To solve **the starvation problem** in Multilevel Queue
  - multilevel feedback queue scheduling, allows a process to move between queues

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process to higher level queue
    - usually this is some type of aging, whereby a process that has waited a long time in a queue (say 15 minutes) gets upgraded by one level)
  - Method used to determine when to demote a process to a lower level queue
    - in the simple case, move down one level at the end of each quantum
  - Method used to determine which queue a process will enter when that process needs service
    - in the simple case, all ready processes enter at the end of queue 0
- Aging can be implemented using multilevel feedback queue

Example of Multilevel Feedback Queue

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

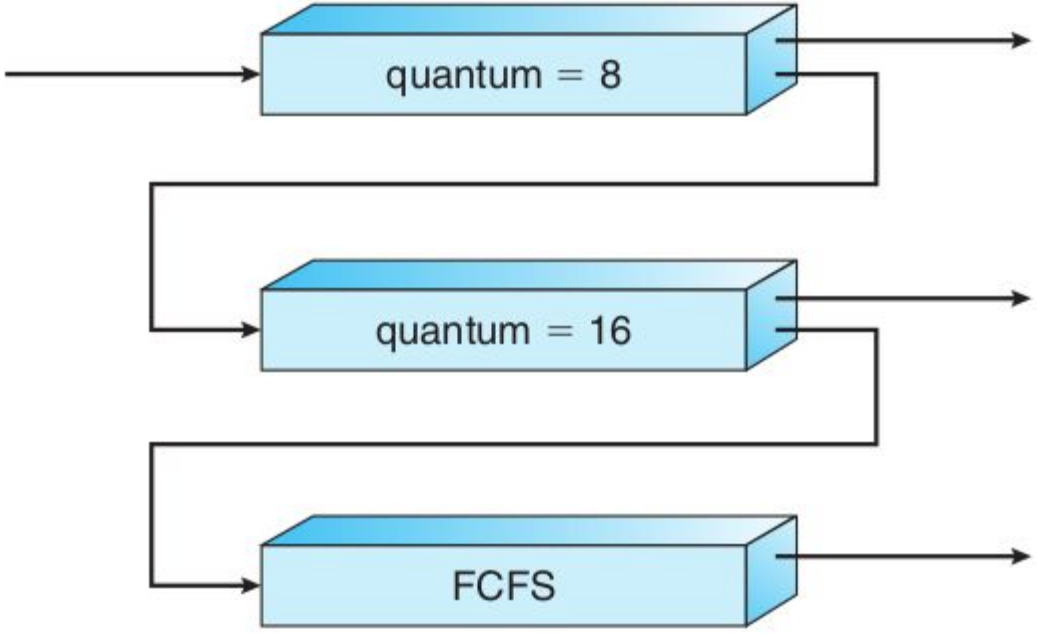


Figure 5.9 Multilevel feedback queues.

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - **PTHREAD_SCOPE_PROCESS**
    - schedules threads using PCS scheduling
  - **PTHREAD_SCOPE_SYSTEM**
    - schedules threads using SCS scheduling
- Can be limited by OS
  - – Linux and macOS **only support** **PTHREAD_SCOPE_SYSTEM**
  - does not support **PTHREAD_SCOPE_PROCESS**

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

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 systems
  - Heterogeneous multiprocessing

Multiple-Processor Scheduling

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

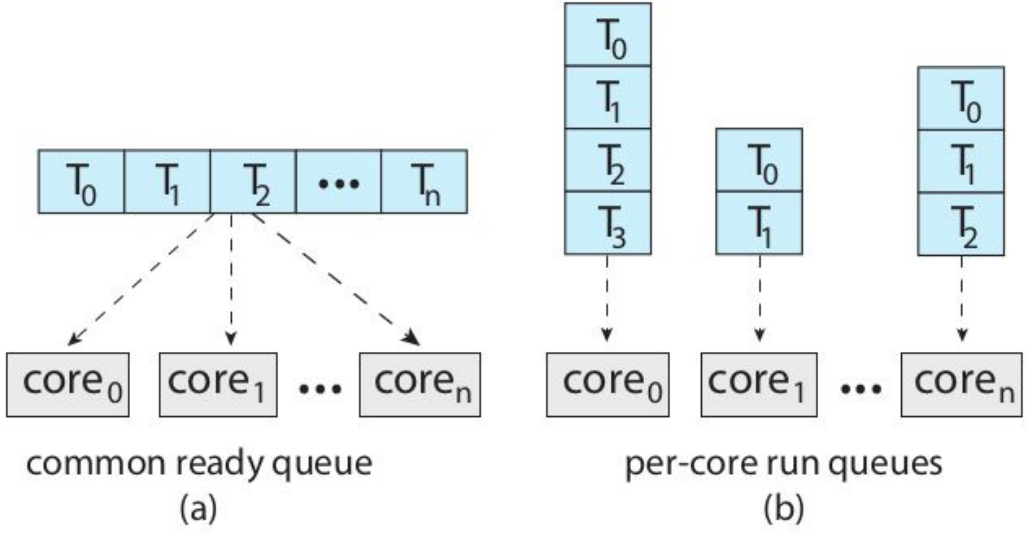


Figure 5.11 Organization of ready queues.

Multicore Processors

- Multiple threads per core also growing
  - Takes advantage of memory stall(cpu works faster than memory) to make progress on another thread while memory retrieve happens

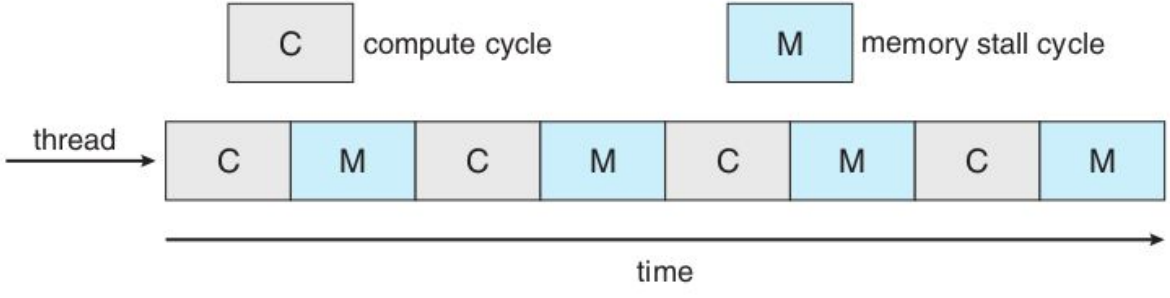


Figure 5.12 Memory stall.

Multithreaded Multicore System

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

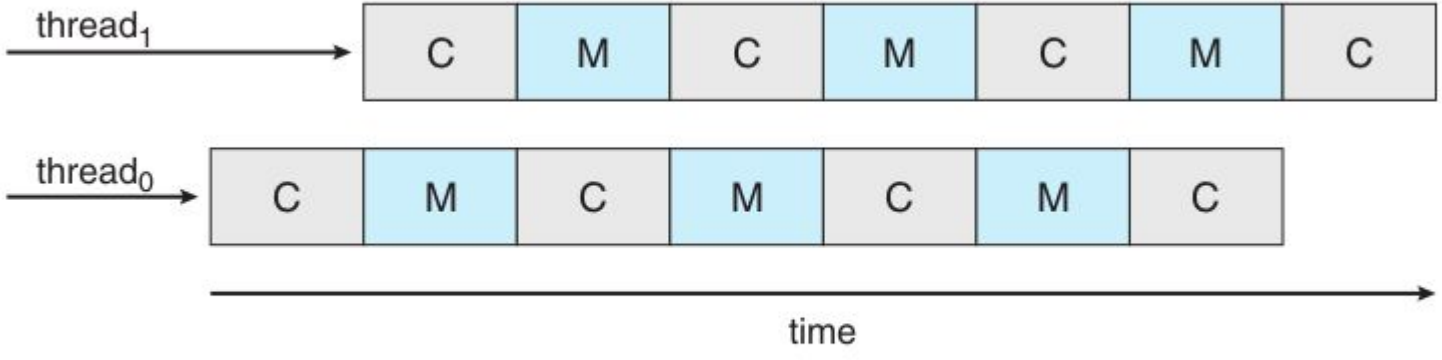


Figure 5.13 Multithreaded multicore system.

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.

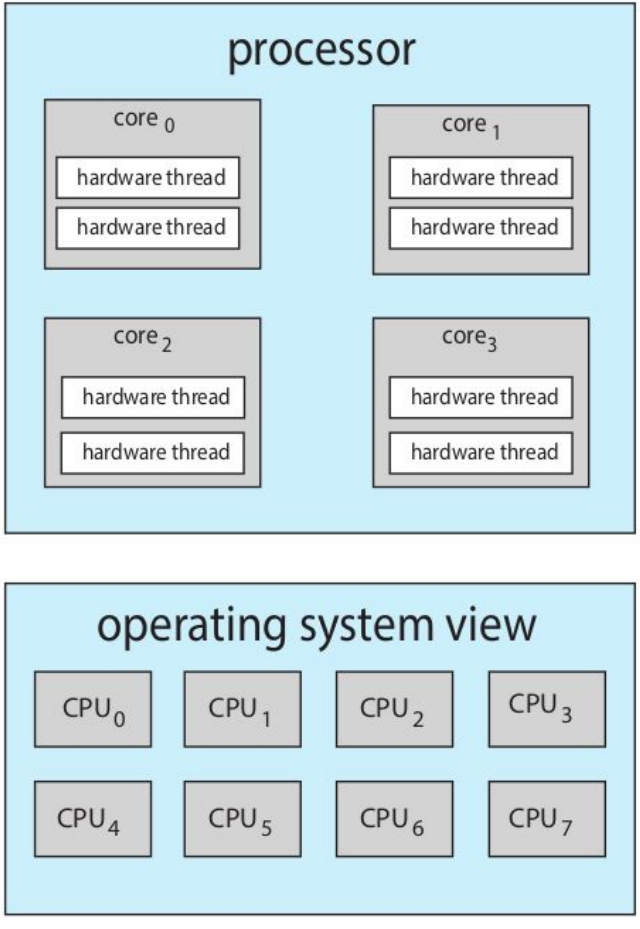


Figure 5.14 Chip multithreading.

Multithreaded Multicore System

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

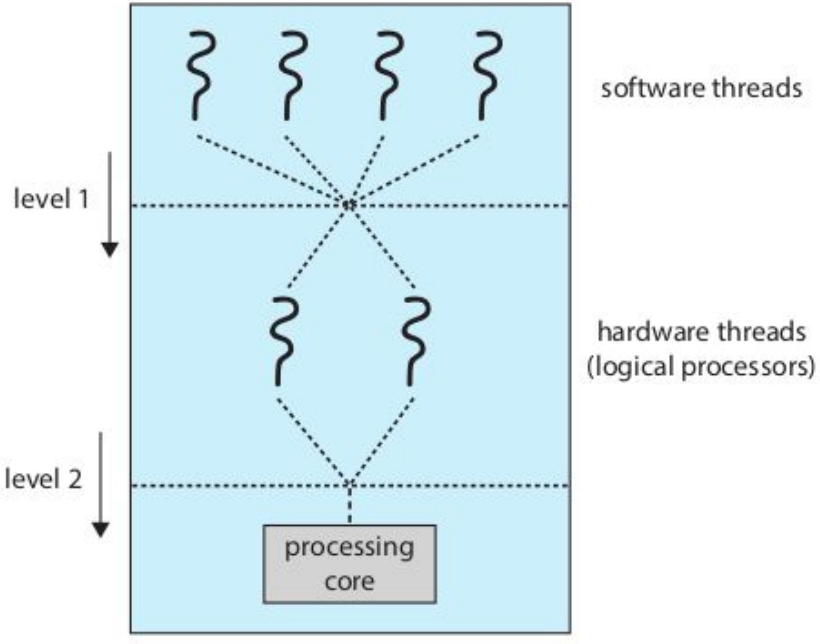


Figure 5.15 Two levels of scheduling.

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **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.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.

- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

NUMA and CPU Scheduling

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

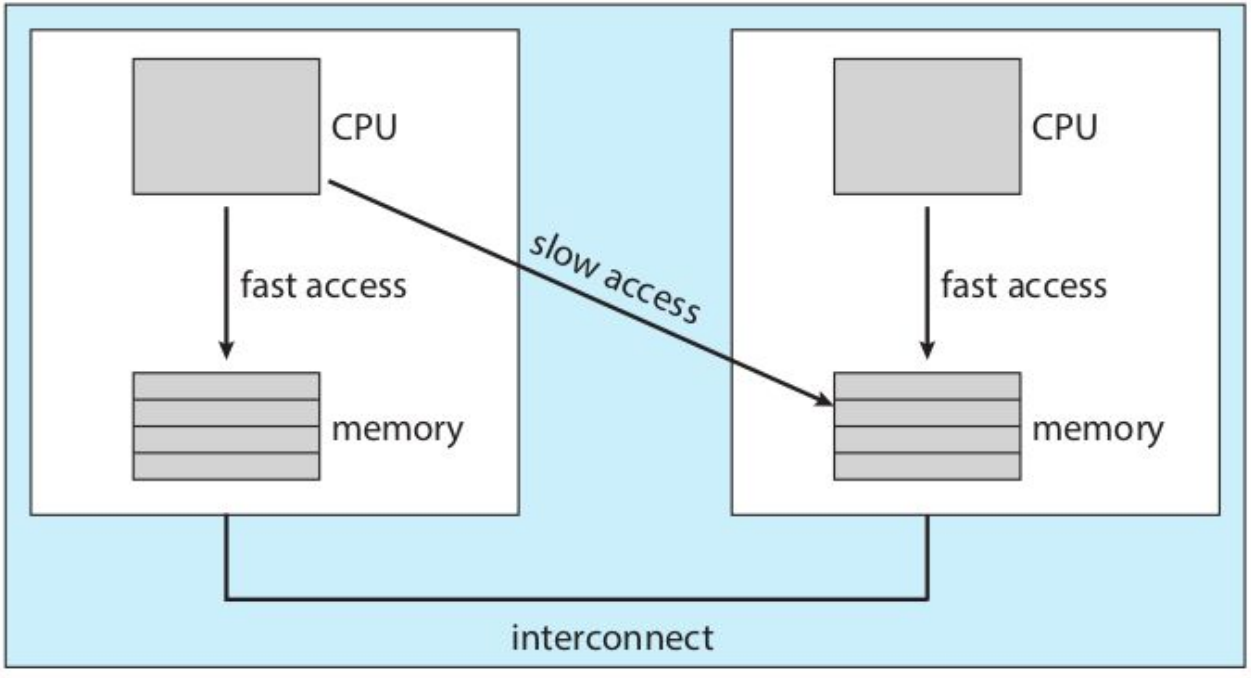


Figure 5.16 NUMA and CPU scheduling.

Real-Time CPU Scheduling

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

Real-Time CPU Scheduling

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

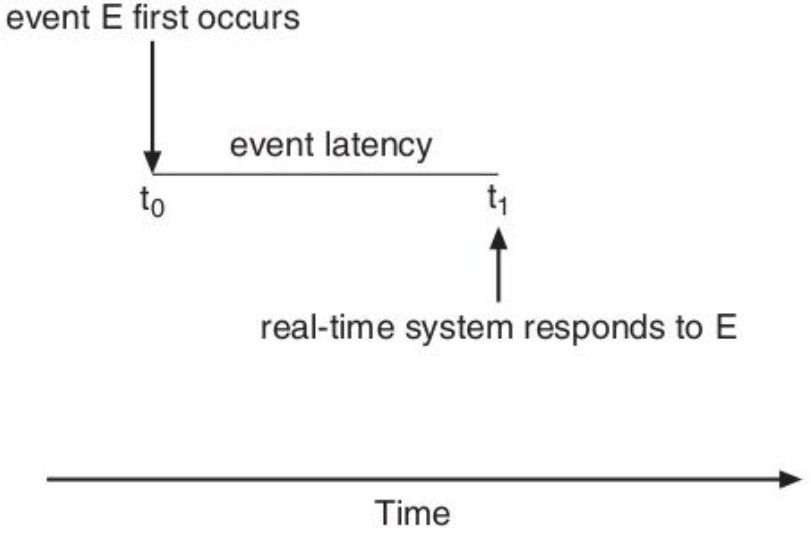


Figure 5.17 Event latency.

Interrupt Latency

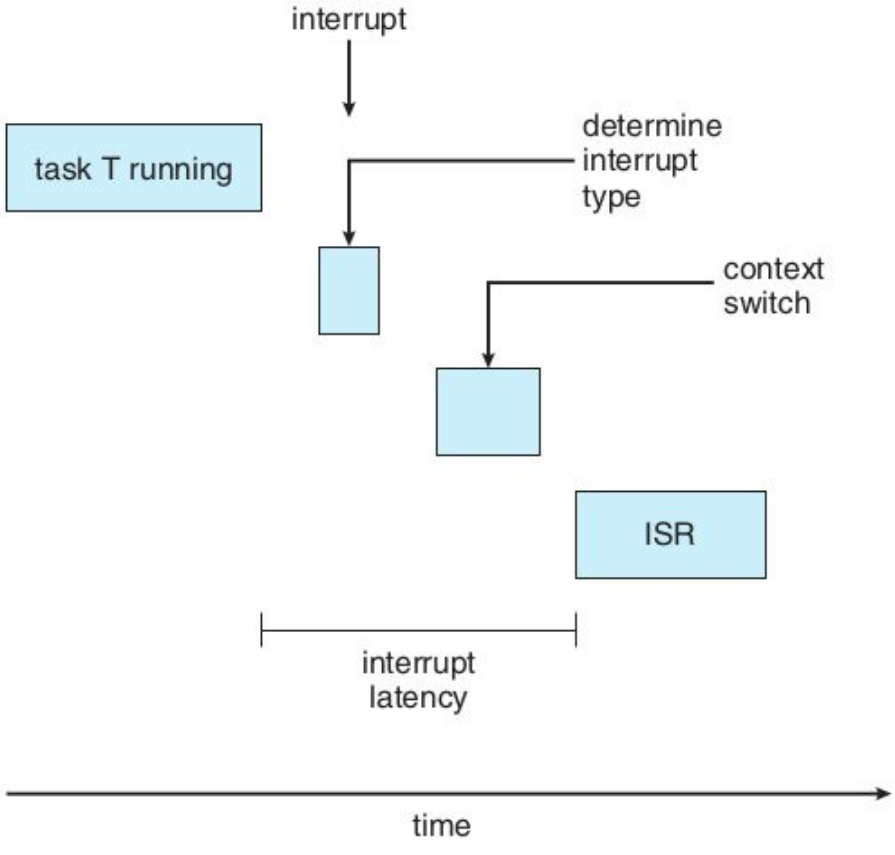


Figure 5.18 Interrupt latency.

Dispatch Latency

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

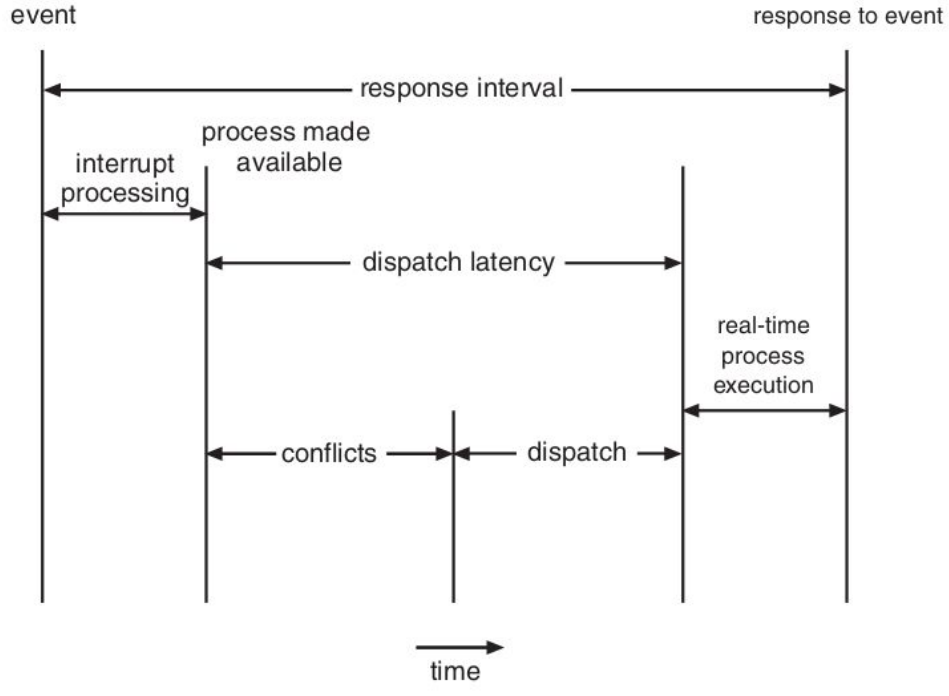


Figure 5.19 Dispatch latency.

Priority-based Scheduling

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

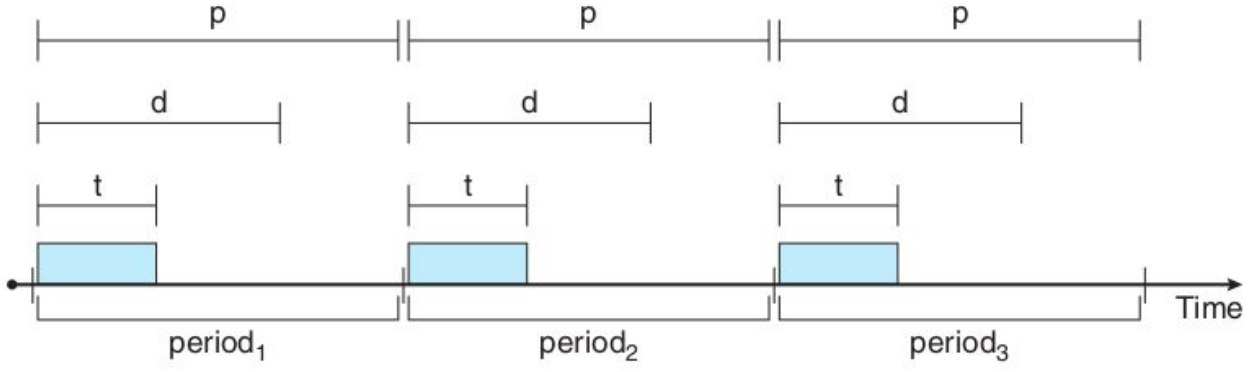


Figure 5.20 Periodic task.

For example,Windows has 32 different priority levels. The highest levels—priority

values 16 to 31—are reserved for real-time processes.

Solaris and Linux have similar prioritization schemes.

Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority

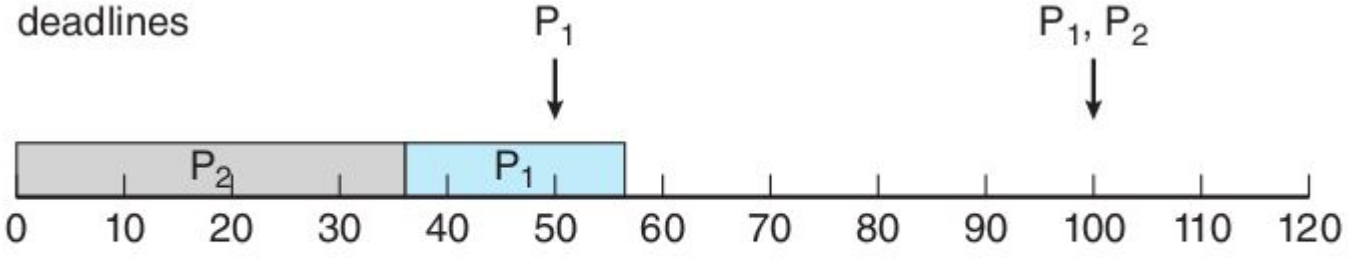


Figure 5.21 Scheduling of tasks when P_2 has a higher priority than P_1.

Example-1:
P1: p1 = 50, t1 = 20
P2: p2 = 100, t2 = 35

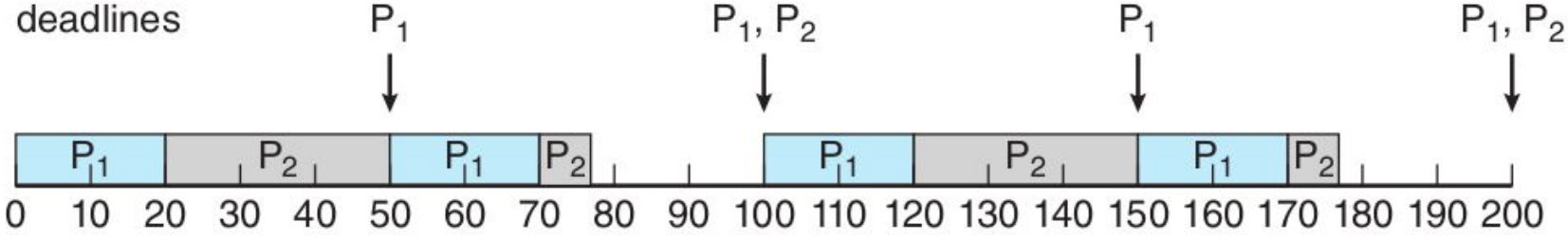


Figure 5.22 Rate-monotonic scheduling.

Example-2:
P1: p1 = 50, t1 = 25
P2: p2 = 80, t2 = 35

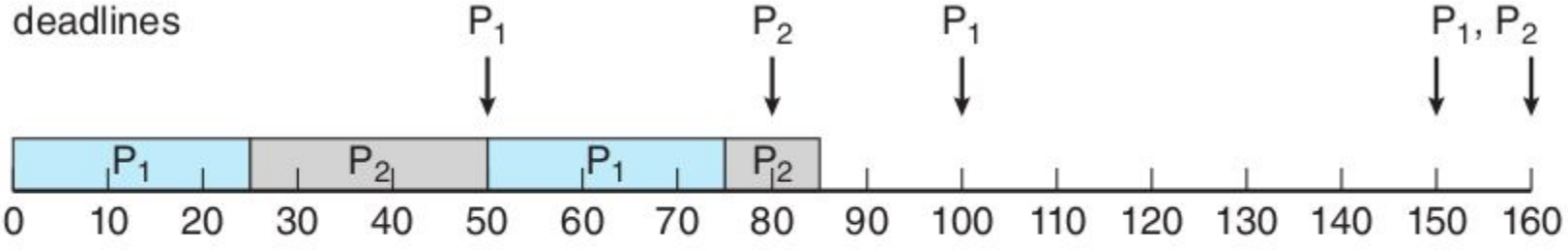


Figure 5.23 Missing deadlines with rate-monotonic scheduling.

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority

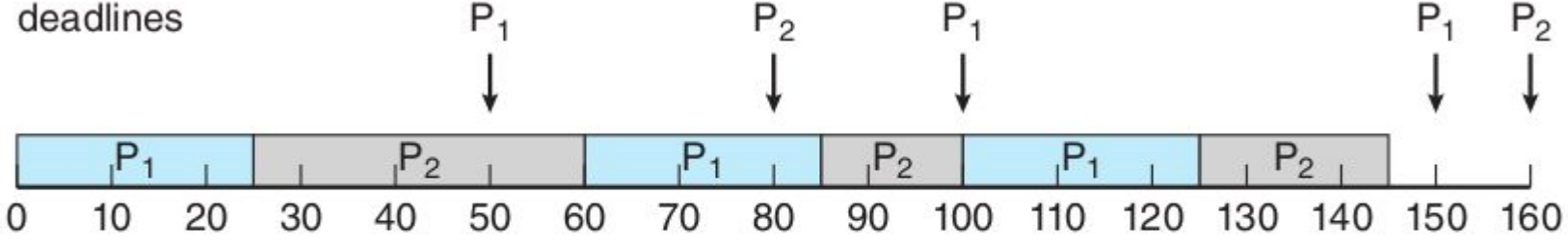


Figure 5.24 Earliest-deadline-first scheduling.

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

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

https://www.cs.unc.edu/~anderson/teach/comp790/papers/posix-rt.pdf

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```

POSIX Real-Time Scheduling API (Cont.)

```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling

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 to 99 and nice value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger 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 +

- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Two scheduling classes included, others can be added
    1. default
    2. real-time

CFS Scheduler — The Linux Kernel documentation

CFS basically models an “ideal, precise multi-tasking CPU” on real hardware.

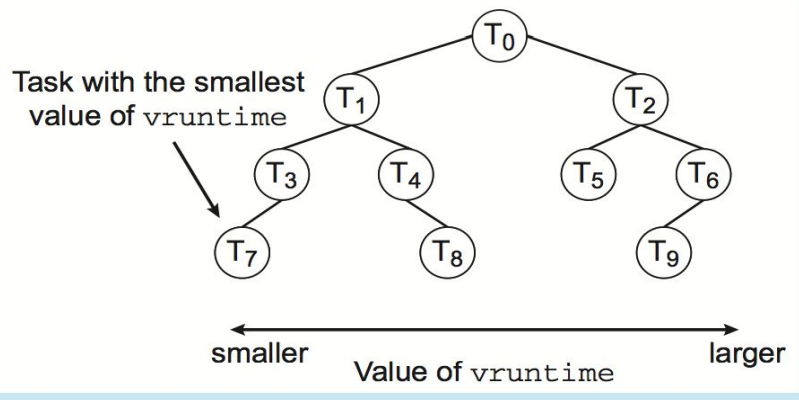
Kernel 6.6 new EEVDF Scheduler — The Linux Kernel documentation

Linux Scheduling in Version 2.6.23 + (Cont.)

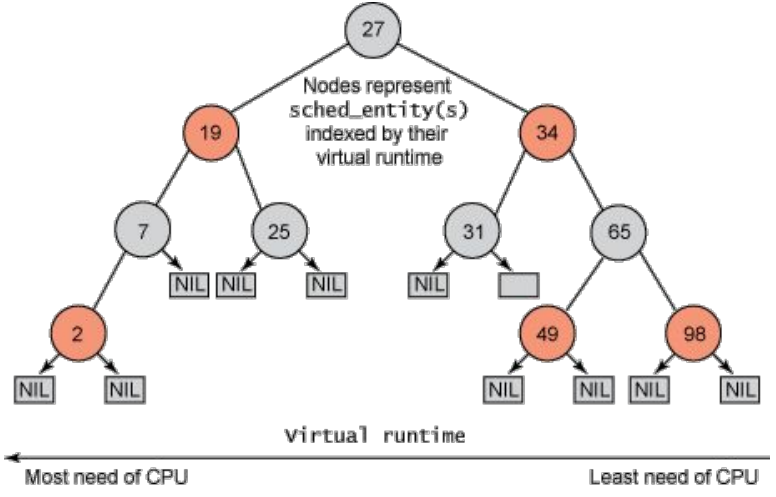
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual 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.



https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

Linux Scheduling (Cont.)

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

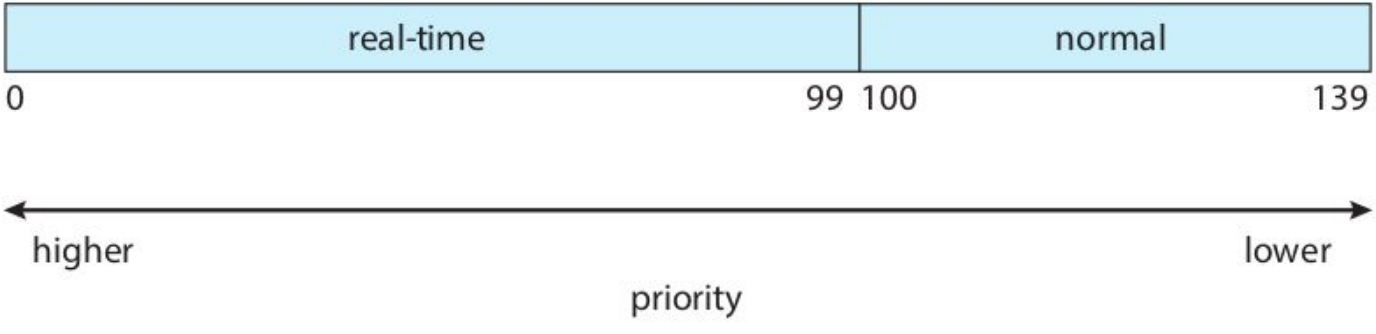


Figure 5.26 Scheduling priorities on a Linux system.

Linux Scheduling (Cont.)

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

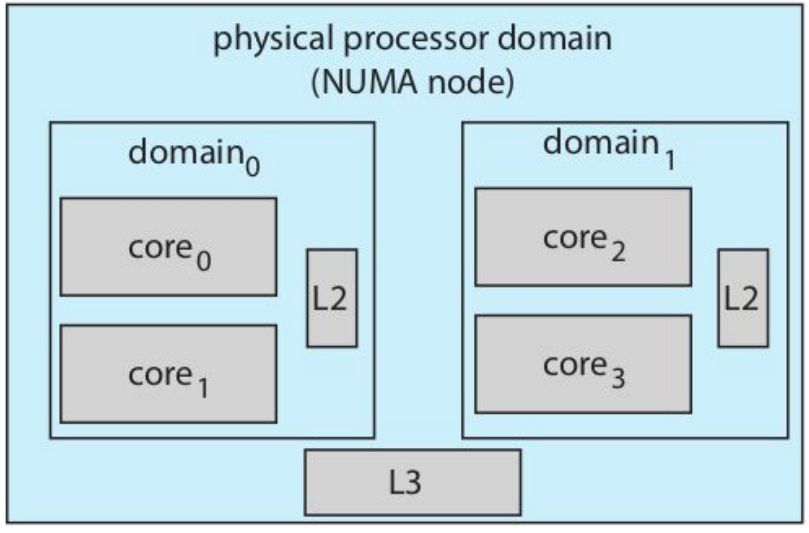


Figure 5.27 NUMA-aware load balancing with Linux CFS scheduler.

Linux Scheduler Implementation Details

https://cs4118.github.io/www/2023-1/lect/16-linux-sched-class.pdf

https://www3.cs.stonybrook.edu/~youngkwon/cse306/Lecture16_Linux_Process_Scheduling.pdf

https://www.cs.columbia.edu/~jae/4118-LAST/L17-linux-sched-class.pdf

Windows Scheduling

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

Windows Priority Classes

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

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

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

Figure 5.28 Windows thread priorities.

Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

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

Figure 5.29 Solaris dispatch table for time-sharing and interactive threads.

Solaris Scheduling

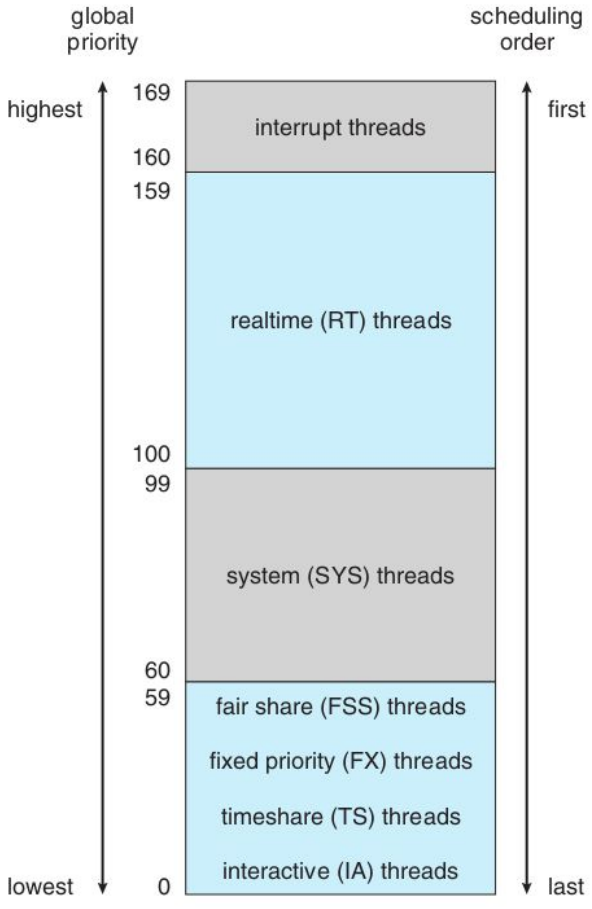


Figure 5.30 Solaris scheduling.

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

Algorithm Evaluation

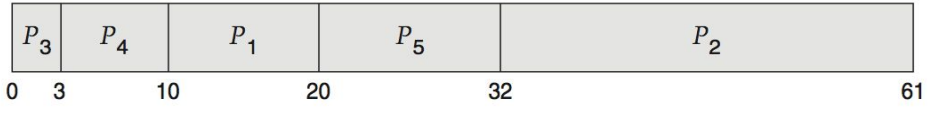
- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving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

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

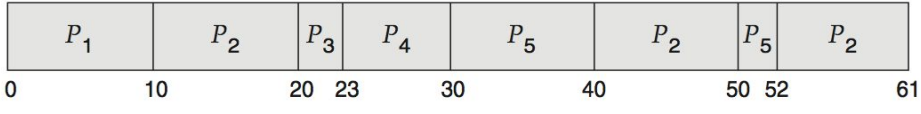
Deterministic Evaluation

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

  - Non-preemptive SFJ 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 law** – in steady state, processes leaving queue must equal processes arriving, thus:

  $$L = \lambda \times W$$

  - *L*: average queue length, the average number of customers in the queue
  - *W:* average waiting time in queue
  - *λ:* average arrival rate into queue
- Valid for any scheduling algorithm and arrival distribution

- For example, if on average 10 processes arrive per second, and there are normally 2 processes in queue,

$$W = \frac{L}{\lambda} = \frac{2}{10} = 0.2$$

Simulations

- Queueing models limited
- **Simulations** 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

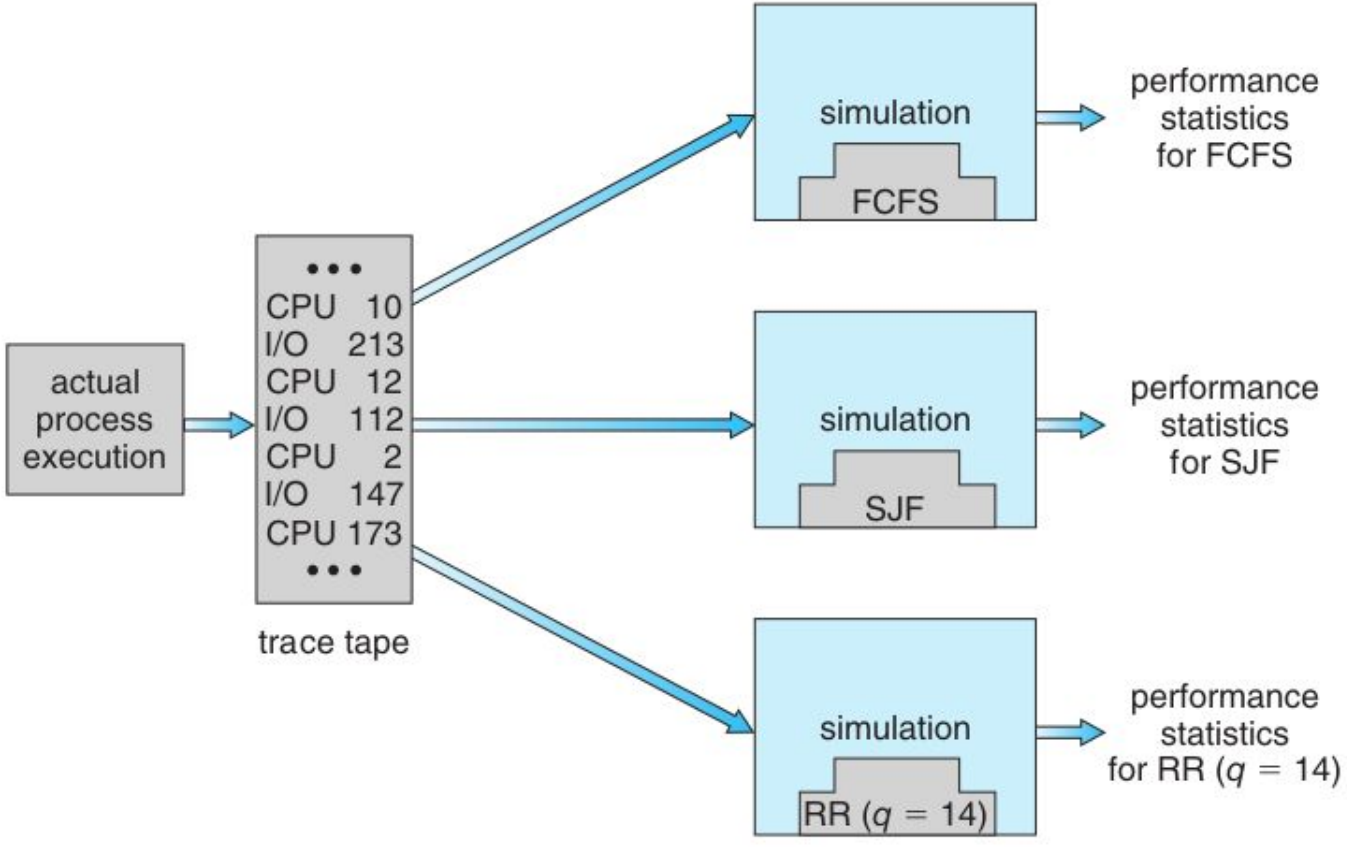


Figure 5.31 Evaluation of CPU schedulers by simulation.

Implementation

- Even simulations have limited accuracy
- Just 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
- Or APIs to modify priorities
- But again environments vary