Chapter 5: CPU Scheduling

How the OS Chooses Which Process to Run Next

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

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

The Fundamental Problem

One CPU, Many Processes

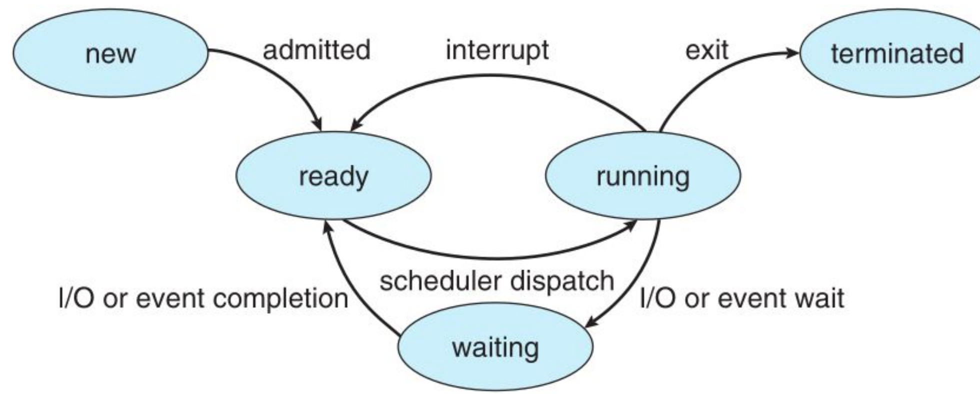
Question: In what order should we run them?

Goals?

- ★ Keep CPU busy ✅
    - ○ **Utilization**
- ★ Be fair to all processes ⚖️
    - ○ **fairness**
- ★ Respond quickly to users ⚡
    - ○ **responsiveness**

CPU Scheduler

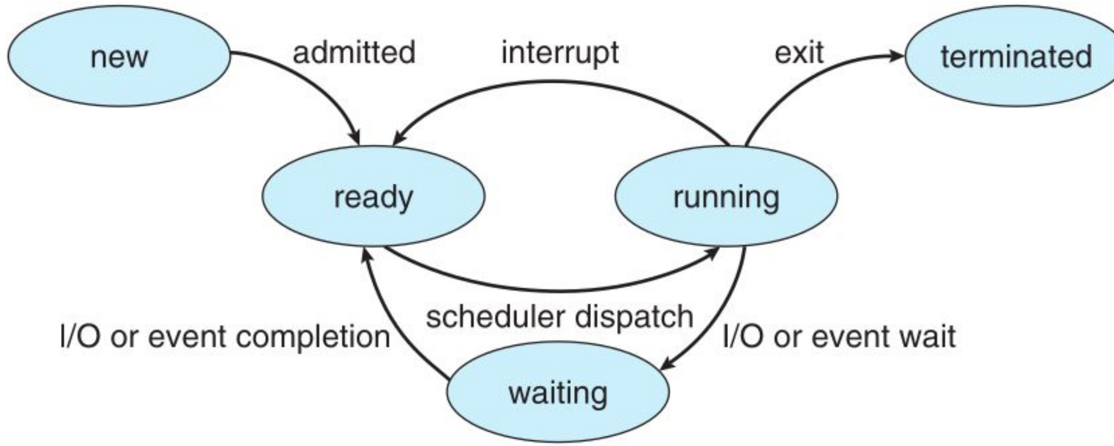
The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them



Preemptive vs. Non-preemptive Scheduling:

- **Non-preemptive:** Once started, process keeps CPU until done
- **Preemptive:** OS can interrupt running process
- **Modern OSs:** All use preemptive scheduling

When Does Scheduling Happen?



CPU scheduling may take place,

1. `Running → Waiting      (Non-preemptive)`
2. `Running → Ready        (Preemptive)`
3. `Waiting → Ready        (Preemptive)`
4. `Process Terminates    (Non-preemptive)`

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

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 details next week.

Why do we care?

- What goals should we have for a scheduling algorithm?

Scheduling criteria

Useful Concepts to Consider: CPU Burst Cycles

- 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

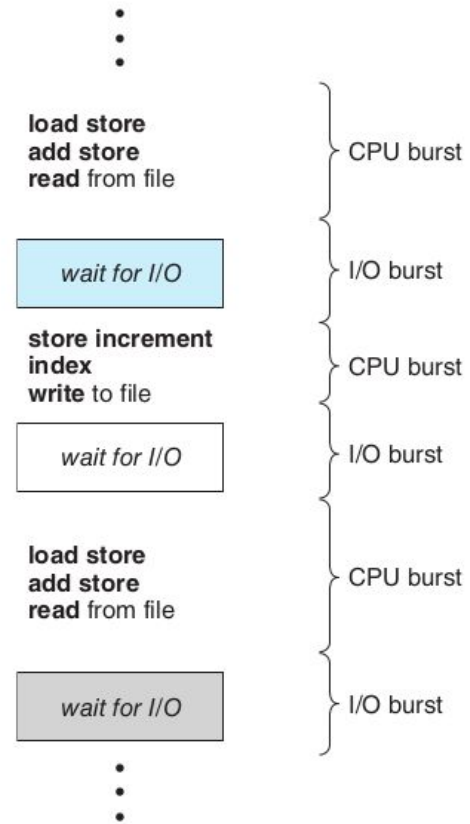


Figure 5.1 Alternating sequence of CPU and I/O bursts.

Histogram of CPU-burst Times

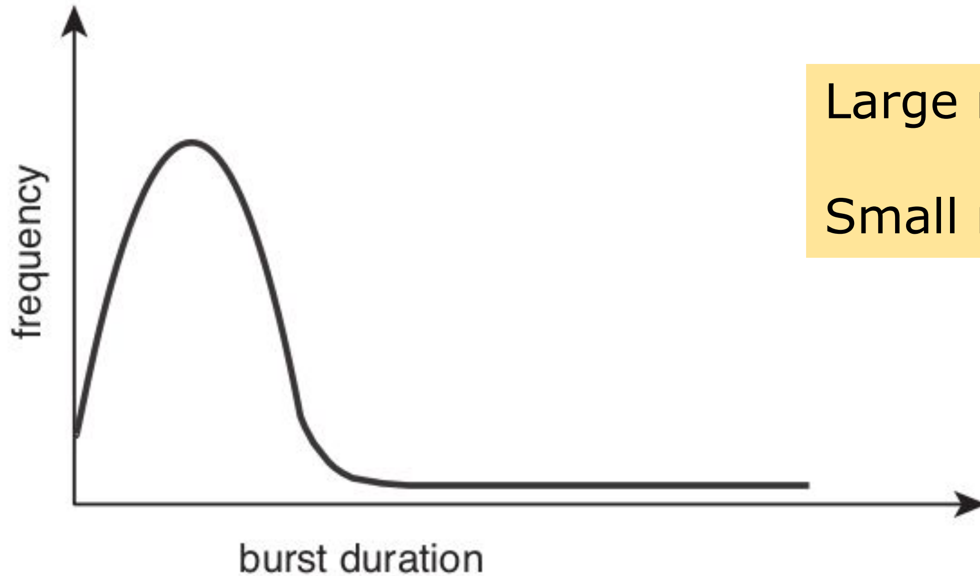


Figure 5.2 Histogram of CPU-burst durations.

Large number of short bursts

Small number of longer bursts

Useful Concepts to Consider: 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**

Dispatch latency – time it takes for the dispatcher to stop one process and start another running

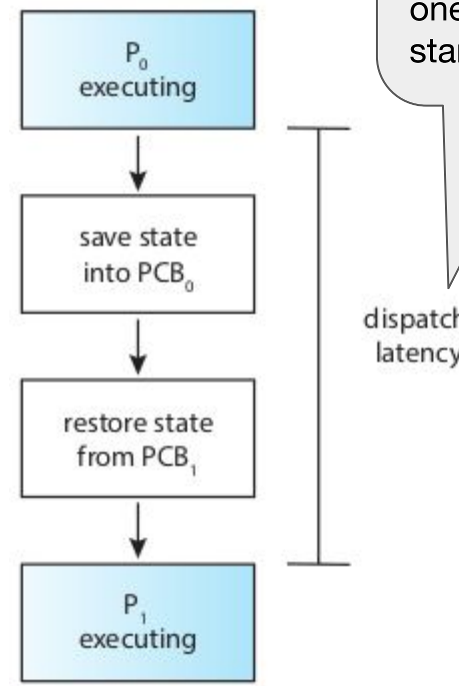


Figure 5.3 The role of the dispatcher.

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 Criteria - What Matters?

Metric	**What it Measures**	**Ideal**
CPU Utilization	How busy is the CPU?	**Maximize**
Throughput	Processes completed/time	**Maximize**
Turnaround Time	Submit → Finish	**Minimize**
Waiting Time	Time in ready queue	**Minimize**
Response Time	Request → First response	**Minimize**

Trade-off: Throughput 🆚 Latency (response time, turnaround time)

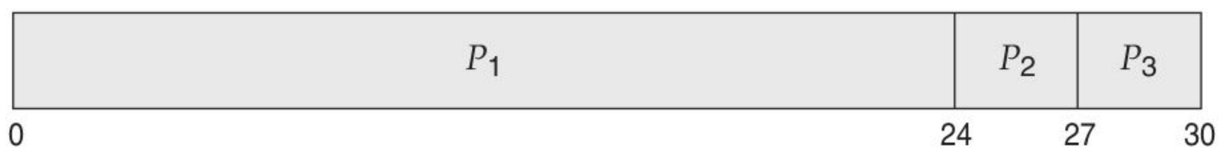
Scheduling Algorithms

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

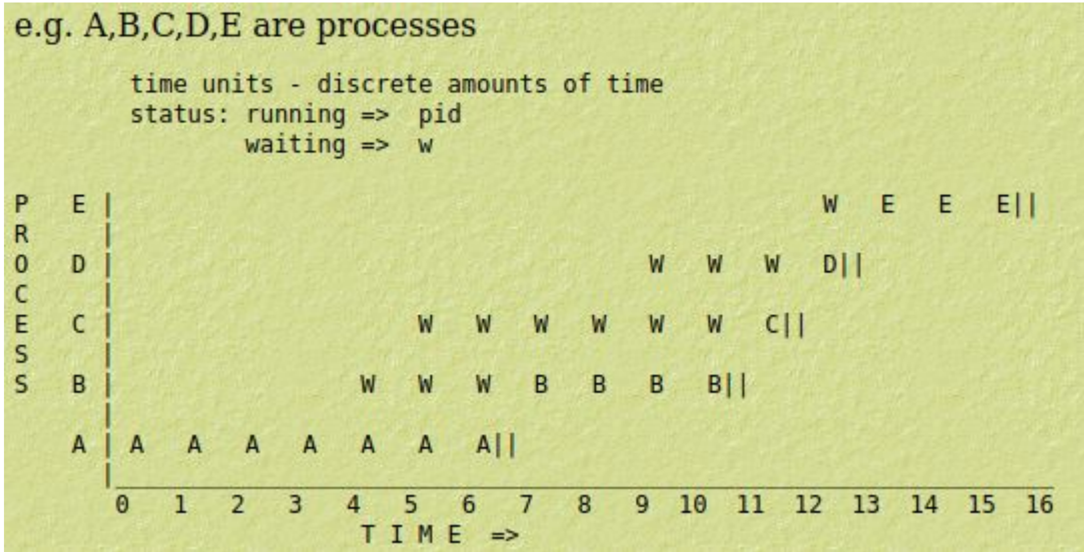
Same Processes, Different Order: P_2 (3), P_3 (3), P_1 (24)

- 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

- Good for: Long processes
- Bad for: Short processes, response time

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)

Example: Processes: P1(6), P2(8), P3(7), P4(3)

```
SJF Order: P4(3), P1(6), P3(7), P2(8)

Gantt Chart:
P4:     (0-3)
P1:     (3-9)
P3:     (9-16)
P2:     (16-24)

Waiting Times: P4=0, P1=3, P3=9, P2=16
AWT = (0+3+9+16)/4 = 7ms
```

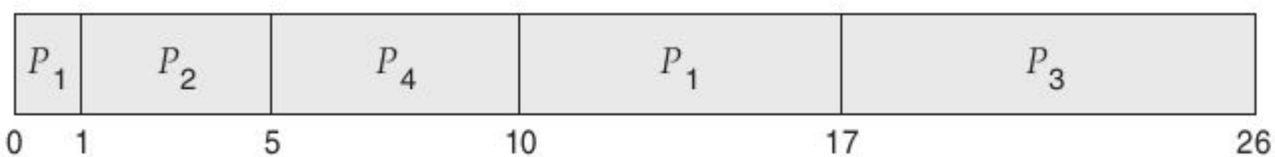
Theory: Optimal for Average Waiting Time
Problem: Need to know future burst lengths!

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

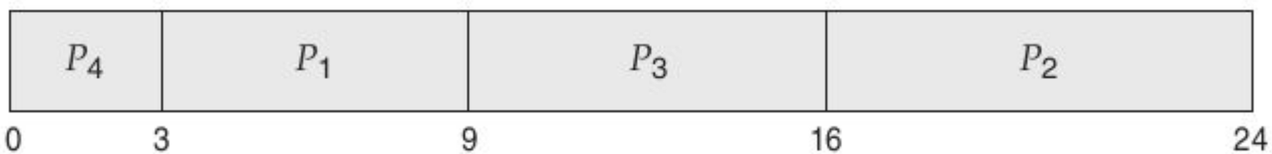
Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



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

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

SJF: Predicting CPU Bursts

Exponential Averaging: Learn from History

$$\tau_{\square+1} = \alpha \times \text{actual}_{\square} + (1-\alpha) \times \tau_{\square}$$

$\text{actual}_{\square}$ (or $t_{\square}$) the most recent CPU-burst,
$\tau_{\square}$: the past predicted value (history)

α : a parameter $0 \le \alpha \le 1$

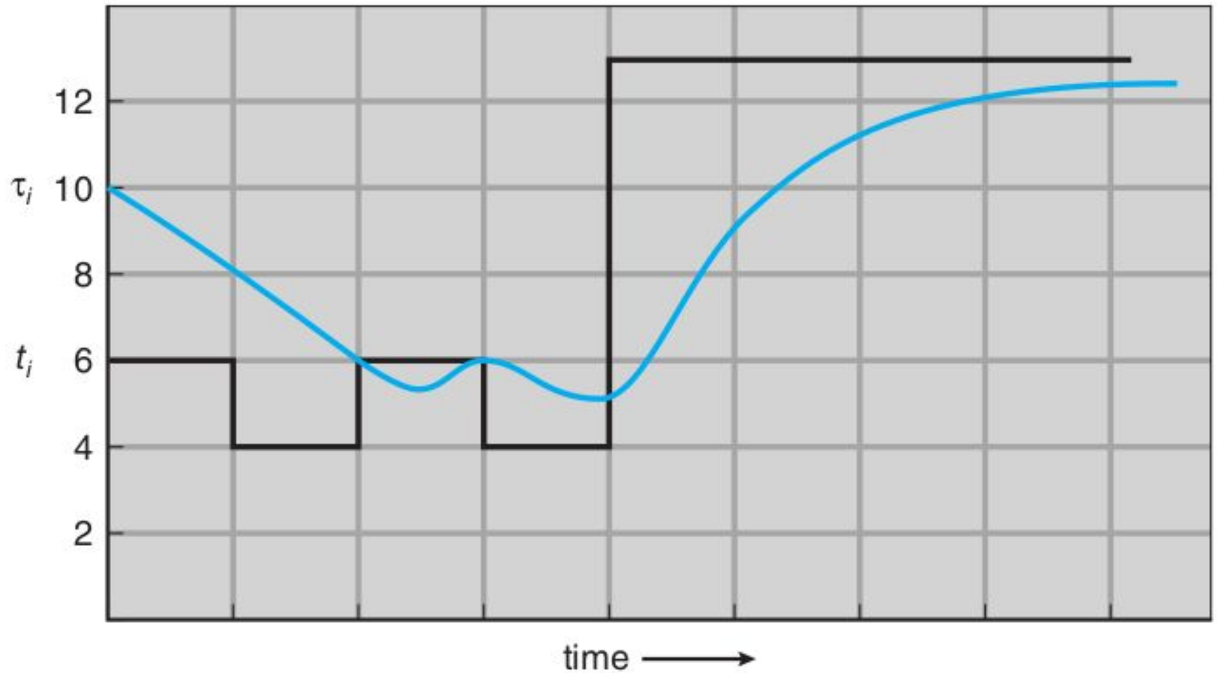
Example: α=0.5, initial prediction=5
Bursts: 10, 10, 10, 1, 1, 1

Predictions:
$\tau_1 = 0.5\times10 + 0.5\times5 = 7.5$
$\tau_2 = 0.5\times10 + 0.5\times7.5 = 8.75$
$\tau_3 = 0.5\times10 + 0.5\times8.75 = 9.38$
$\tau_4 = 0.5\times1 + 0.5\times9.38 = 5.19$
$\tau_5 = 0.5\times1 + 0.5\times5.19 = 3.10$

Gradually adapts to changing behavior

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

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.

SJF Example-2

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

```
P  E |                                                       W   E   E   E||
R    |
O  D |                                             W   W   W   D||
C    |
E  C |                                 W   W   C||
S    |
S  B |                             W   W   W   W   B   B   B   B||
     |
   A | A   A   A   A   A   A   A||
     |_________________________________________________________________________
     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
                             T I M E  =>

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
- the penalty ratios are small; this algorithm works extremely well in most cases

- 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)
- Poor performance for processes with short burst times arriving after a process with a long burst time has started
- SJF cannot handle **infinite loops**
- 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

```
P   E |                                         W   W   W   E   E   E||
R     |
O   D |                                 W   D||
C     |
E   C |                     C||
S     |
S   B |             W   W   W   W   W   B   W   B   B   B||
      |
    A | A   A   A   A   A   W   A   A||
      |____________________________________________________________________
      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
                    T I M E  =>

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
- 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.

- 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)

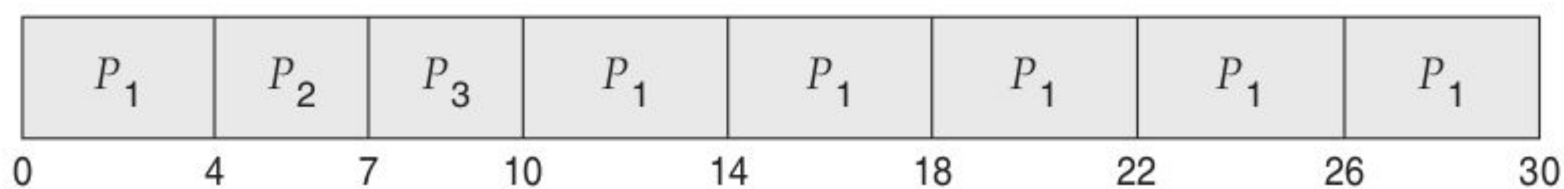
Round Robin (RR)

- Fairness Through Time Slicing
  - Each process gets a small unit of CPU time (time quantum q),
    - usually 10-100 milliseconds.

Example:

Processes: P1(24), P2(3), P3(3), Time Quantum=4ms

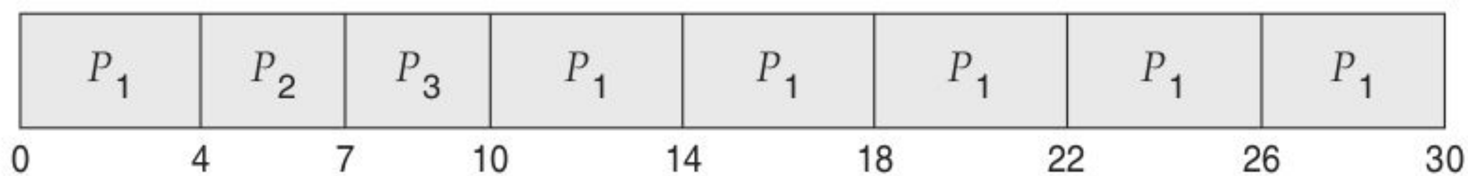
Gantt Chart:




```
Waiting Times:
P1: (4-0)+(10-4)=10?
P1: runs at 0-4, 10-14, 14-18, 18-22, 22-26, 26-30
   Waiting: (10-4)+(30-26)=6+4=10?
```

Round Robin (RR)

- Fairness Through Time Slicing
  - Each process gets a small unit of CPU time (time quantum q),
    - usually 10-100 milliseconds.



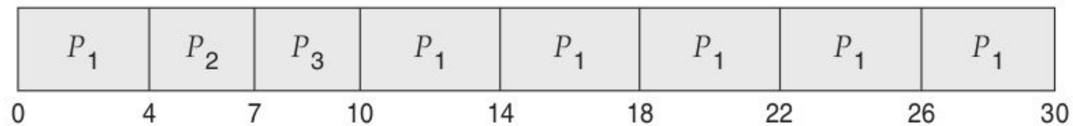
Actually simpler:

P1: **Turnaround=30, Burst=24 → Waiting=6**

P2: Turnaround=7, Burst=3 → Waiting=4

P3: Turnaround=10, Burst=3 → Waiting=7

Process	Burst Time
P_1	24
P_2	3
P_3	3



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

RR - Choosing Time Quantum

Quantum Too Large:

Becomes like FCFS

Quantum Too Small:

High overhead from context switches

```
Rule of thumb: 80% of CPU bursts < time quantum
Typical: 10-100ms
Context switch: ~0.01ms
```

Time Quantum and Context Switch Time

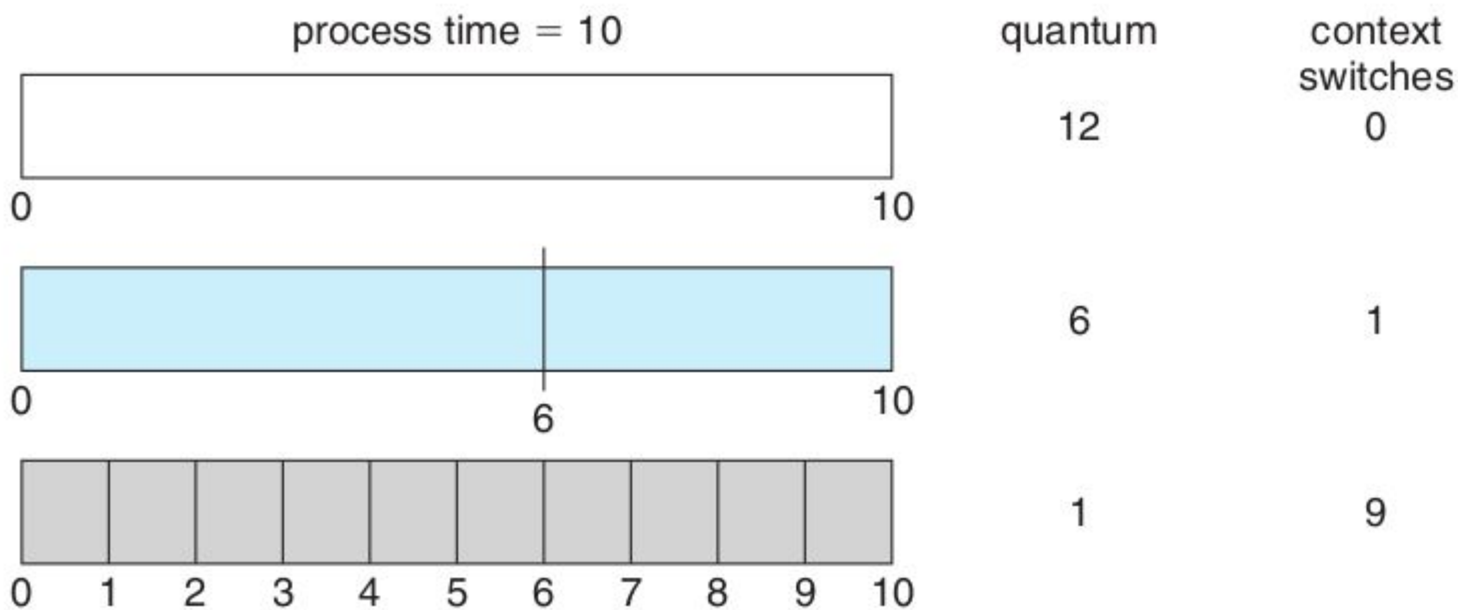
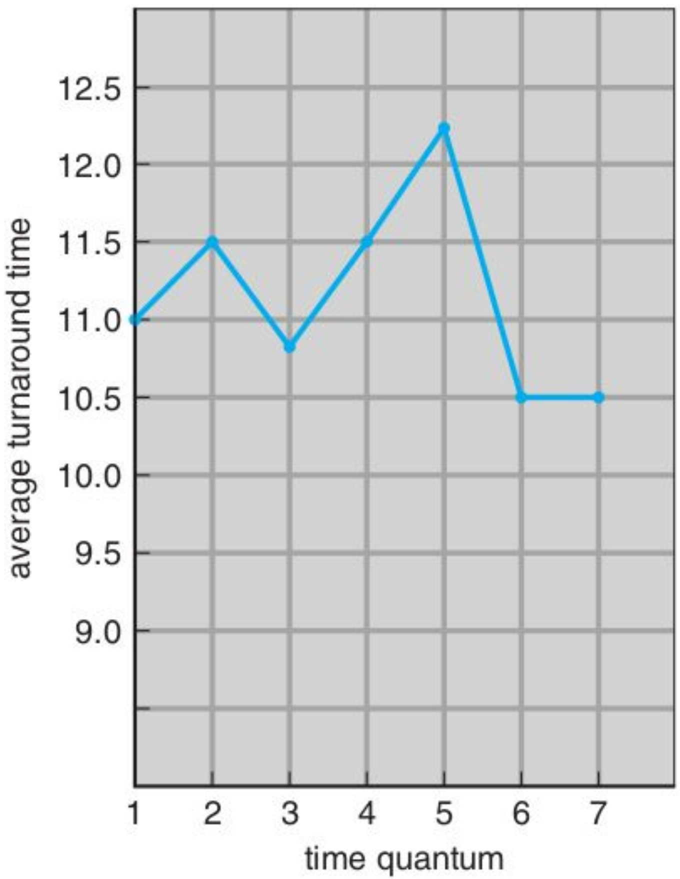


Figure 5.5 How a smaller time quantum increases context switches.

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 5.6 How turnaround time varies with the time quantum.

- 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.

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

```
P   E |                                                   W   E   E   E   ||
R     |
O   D |                                       W   D||
C     |
E   C |                       W   C||
S     |
S   B |                   B   W   W   B   W   B   W   W   B||
      |
    A |   A   A   A   A   W   A   W   W   A   W   W   A||
      |--------------------------------------------------------------------
        0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
                        T I M E  =>
```

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

RR Summary

- n processes,
  - each process gets `1/n` of the CPU time.
- 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

✅ Excellent for response time

✅ Fair to all processes

❌ Poor turnaround time vs SJF

❌ Overhead from frequent switches

Priority Scheduling

Not All Processes Are Equal!

```
Priority 1 (Highest):   Mouse driver
Priority 2:             Video player
Priority 3:             Web browser
Priority 4 (Lowest):    Batch job
```

Priority Scheduling

Not All Processes Are Equal

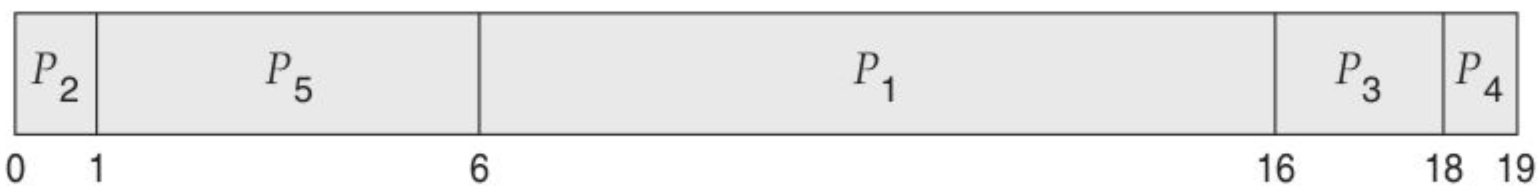
- **Priority number**
  - an integer is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

- **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

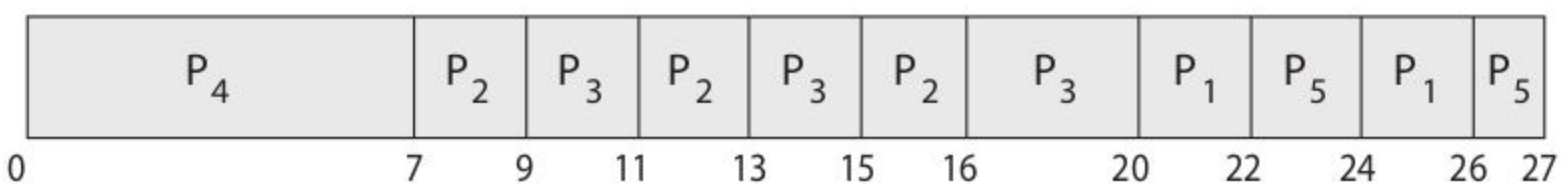


The average waiting time is 8.2 milliseconds.

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.

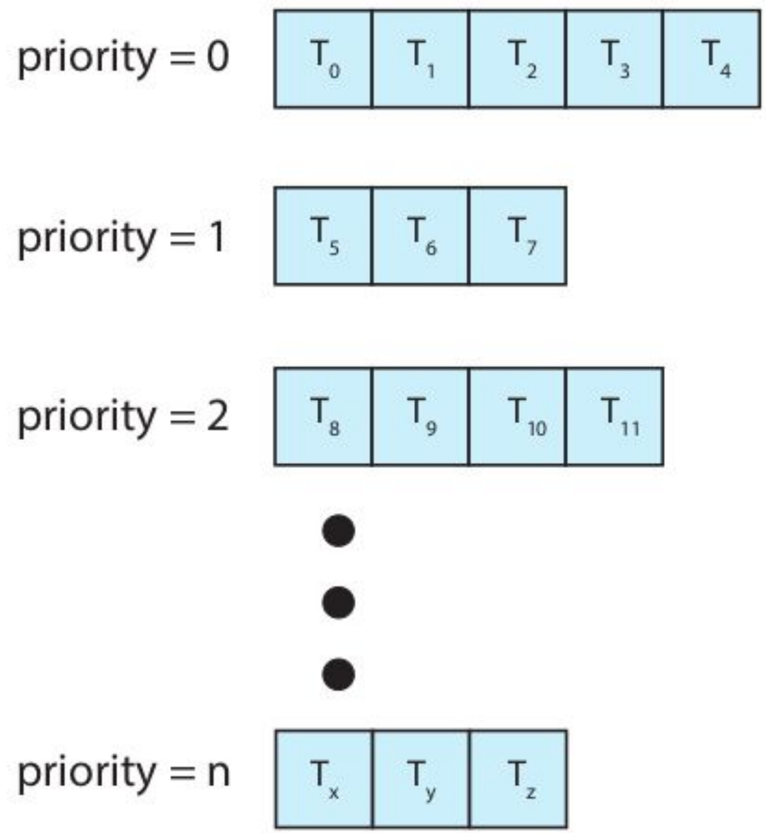
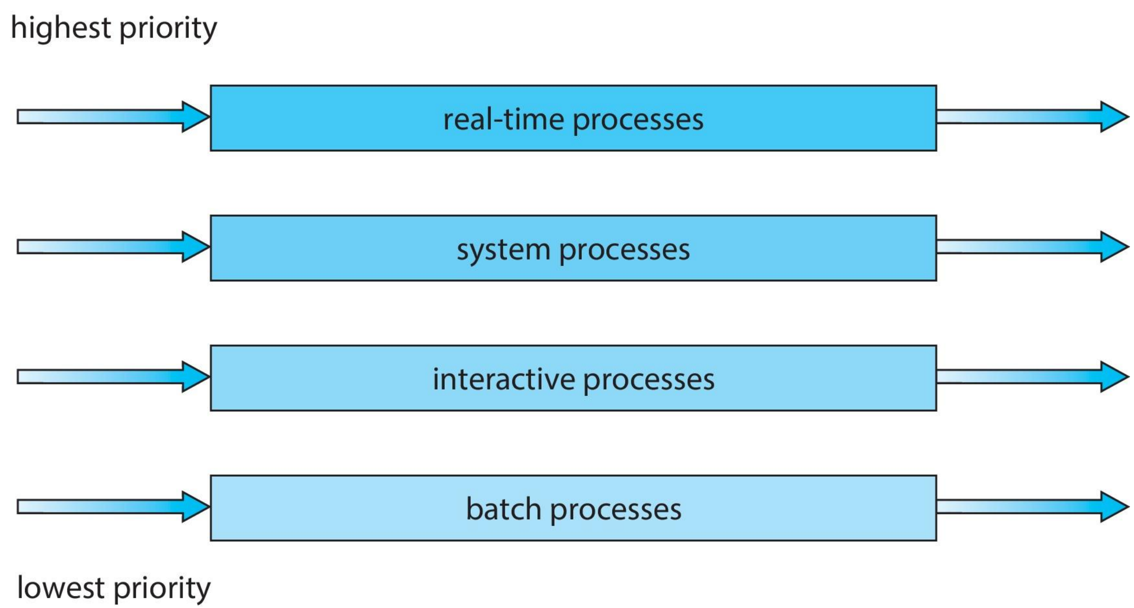
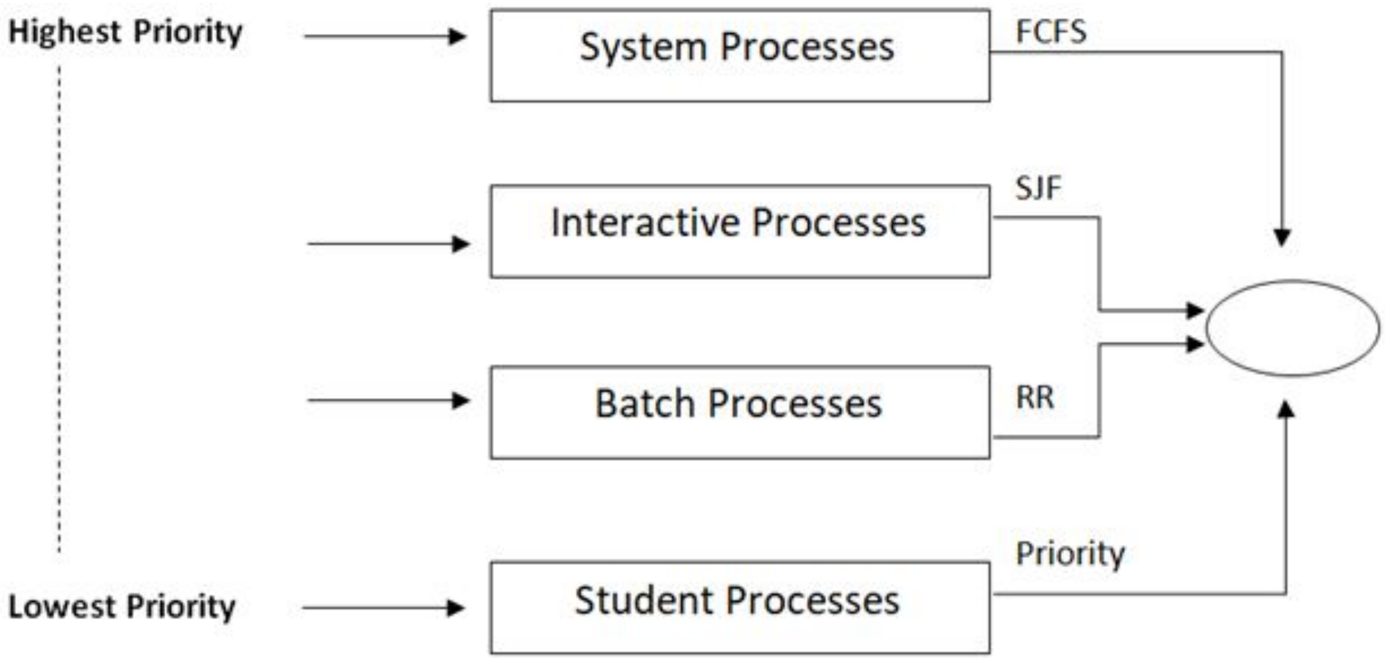


Figure 5.7 Separate queues for each priority.

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.

starvation: Lowest priority process can starve

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

The Best of All Worlds: Separate queues for each priority

Q0: RR (q=8ms)
High Priority ← New processes enter here

↓

Q1: RR (q=16ms)
Medium Priority

↓

Q2: FCFS
Low Priority

- **Aging:** Long-waiting processes move up
- **Demotion:** CPU-hog processes move down

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

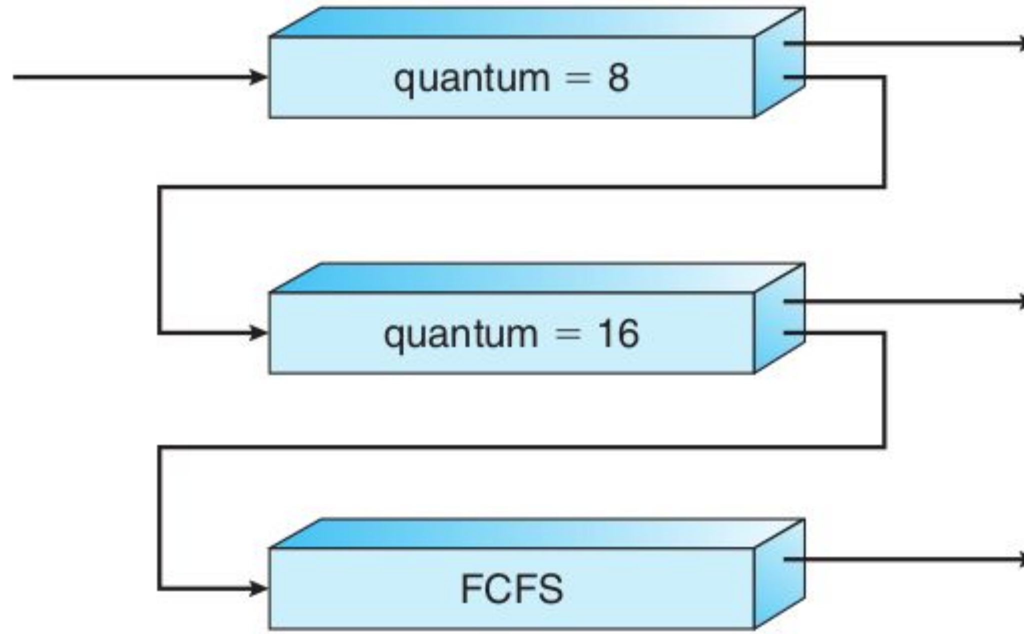


Figure 5.9 Multilevel feedback queues.

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");
   }
```

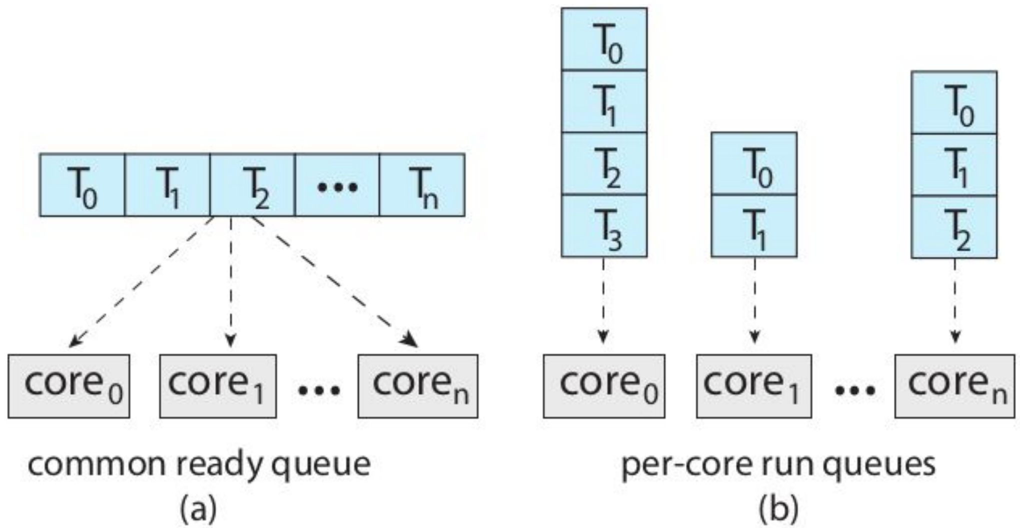
Pthread Scheduling API

```
    /* 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.
- (a) All threads may be in a common ready queue
- (b) Each processor may have its own private queue of threads

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

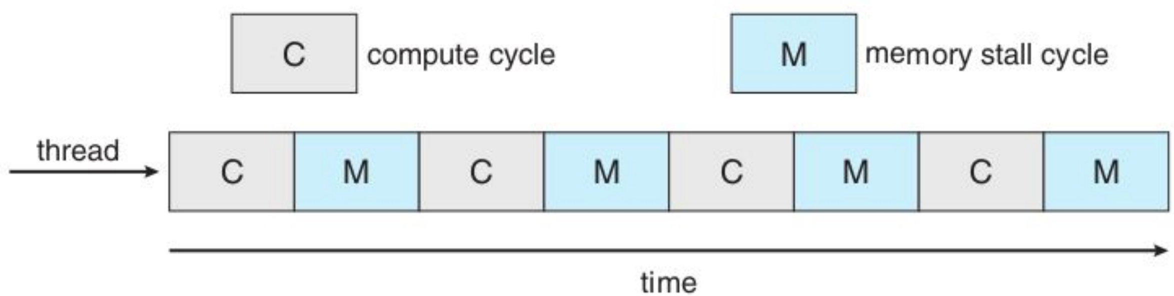


Figure 5.12 Memory stall.

Multithreaded Multicore System

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

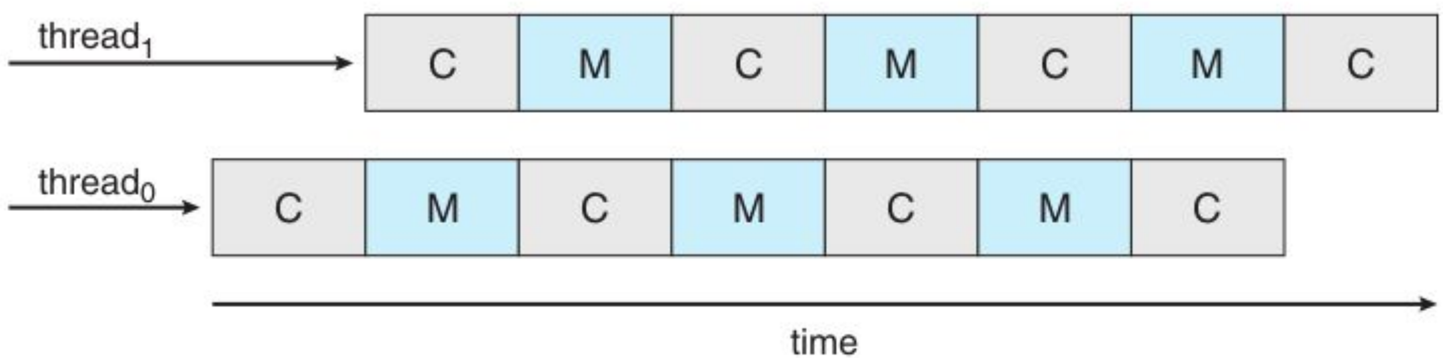


Figure 5.13 Multithreaded multicore system.

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.

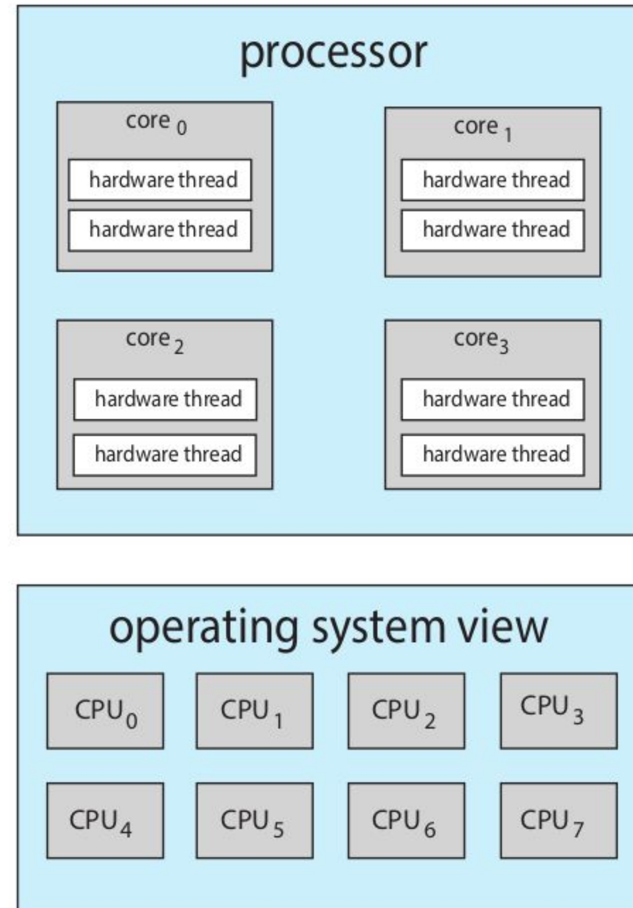


Figure 5.14 Chip multithreading.

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.

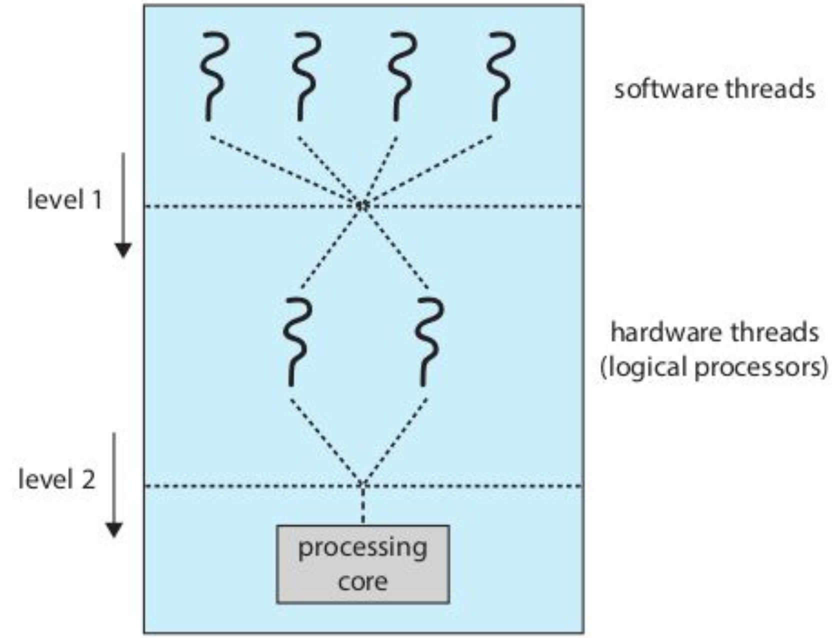


Figure 5.15 Two levels of scheduling.

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
-
- **Load balancing** attempts to keep workload evenly distributed
  - when necessary, pack tasks to allow cores to sleep
- **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.
  - sched_setaffinity(2) - Linux manual page
  - pthread_setaffinity_np(3) - Linux manual page

NUMA and CPU Scheduling

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

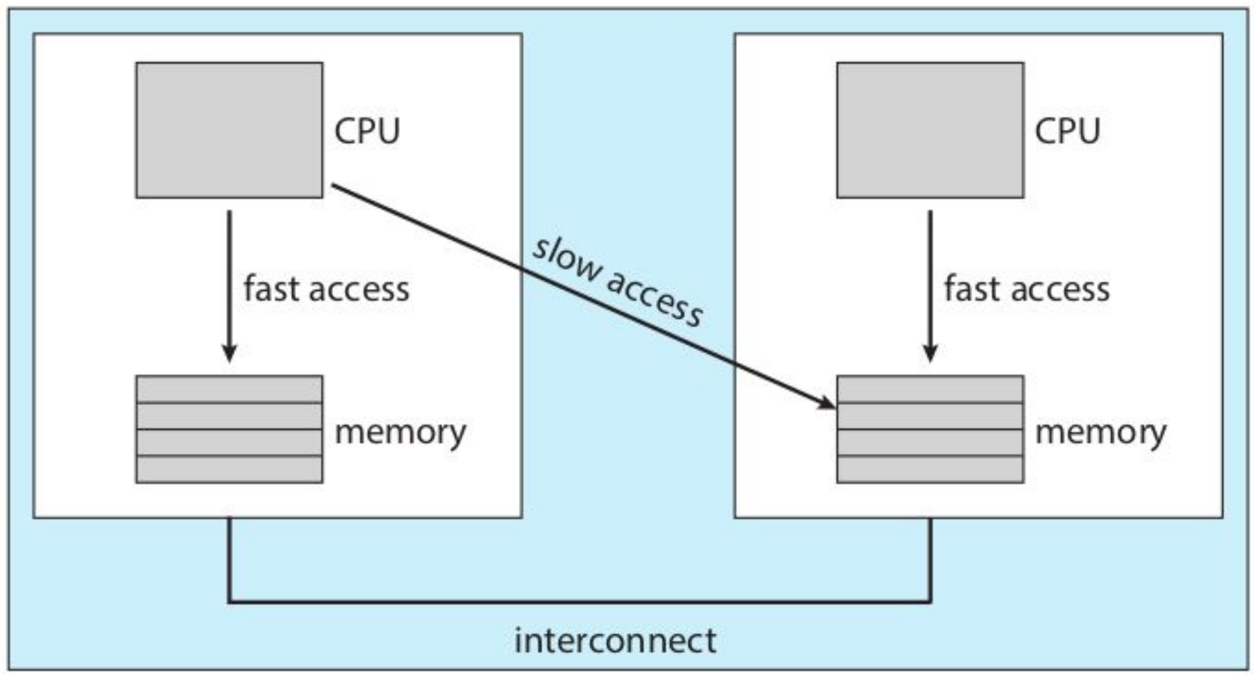


Figure 5.16 NUMA and CPU scheduling.

Real-Time CPU Scheduling

When Timing Matters

Example: Anti-lock Braking System

- Must check wheel slip every 10ms
- Calculation takes 2ms
- Deadline: 10ms period

- **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**

Algorithms:

- **Rate Monotonic: Shorter period = higher priority**
- **Earliest Deadline First: Closer deadline = higher priority**

Real-Time CPU Scheduling: Event Latency

Event latency

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

➔ Two types of latencies affect performance

- **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
- **Dispatch latency** – time for schedule to take current process off CPU and switch to another

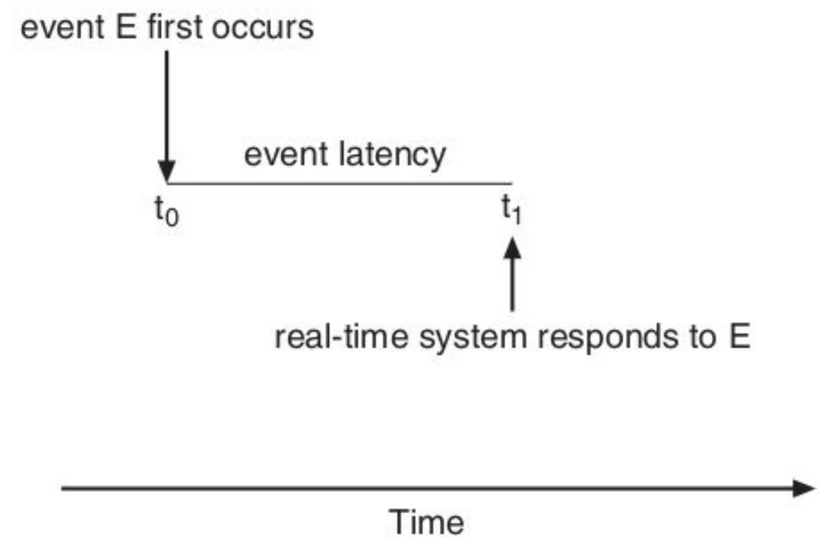


Figure 5.17 Event latency.

Interrupt Latency

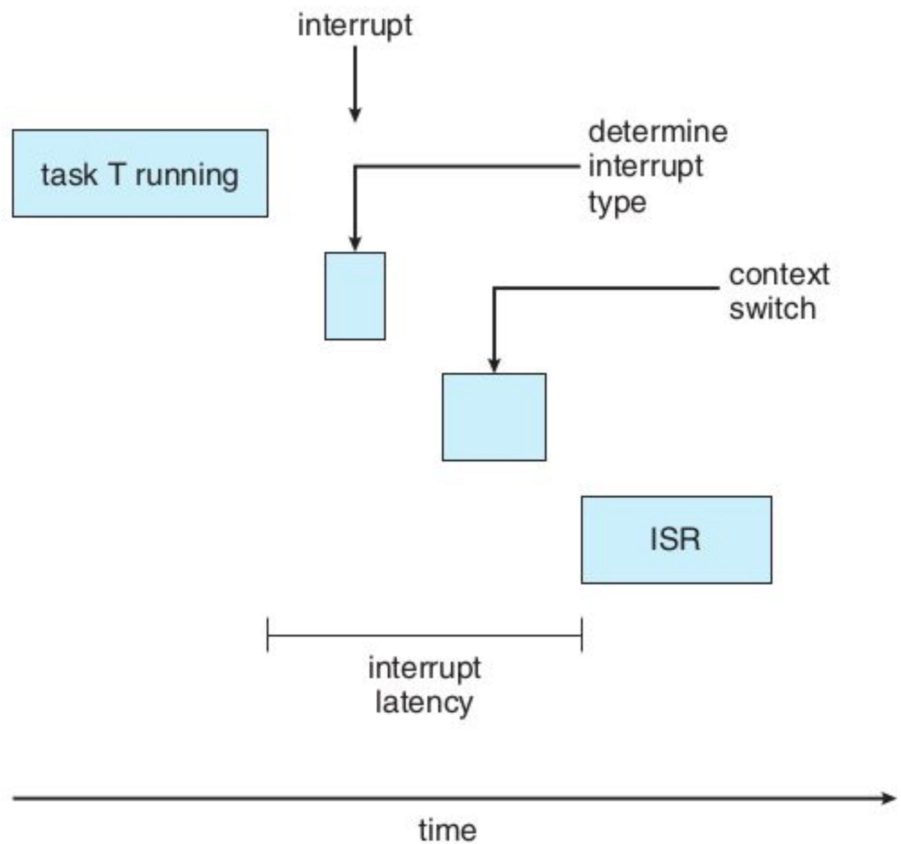


Figure 5.18 Interrupt latency.

Dispatch Latency

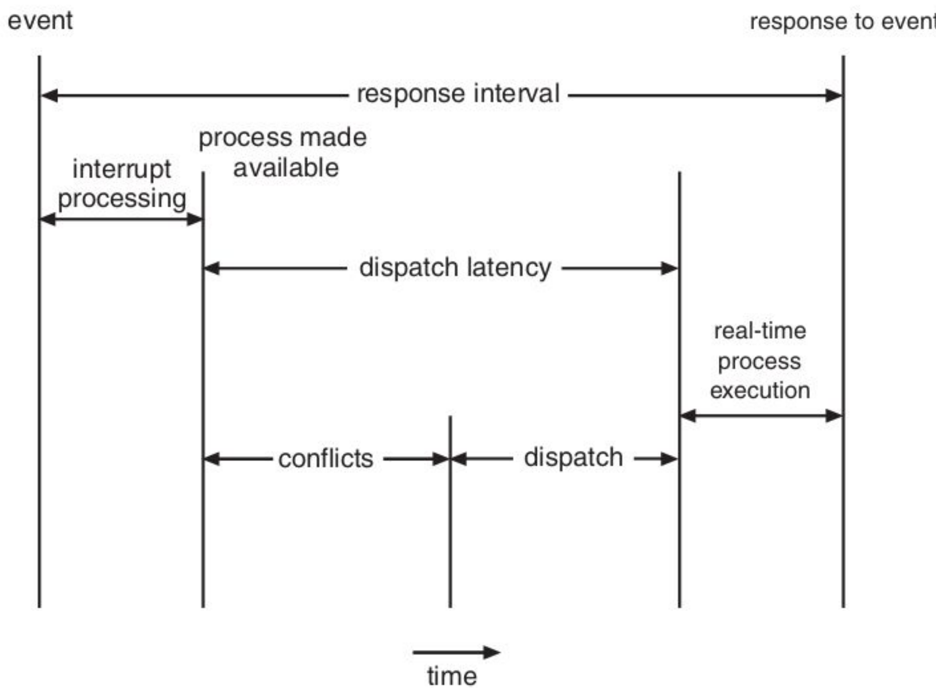


Figure 5.19 Dispatch latency.

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

Priority-based Scheduling

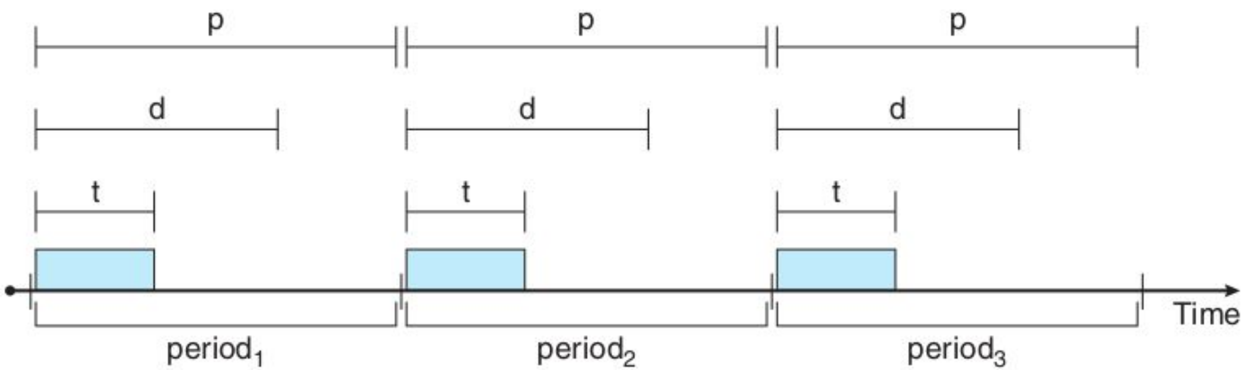


Figure 5.20 Periodic task.

- 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 $1/p$
- preemptive-priority-based scheduling
  - **guarantees soft real-time!**
  - **For hard real-time;** must also provide ability to meet deadlines

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

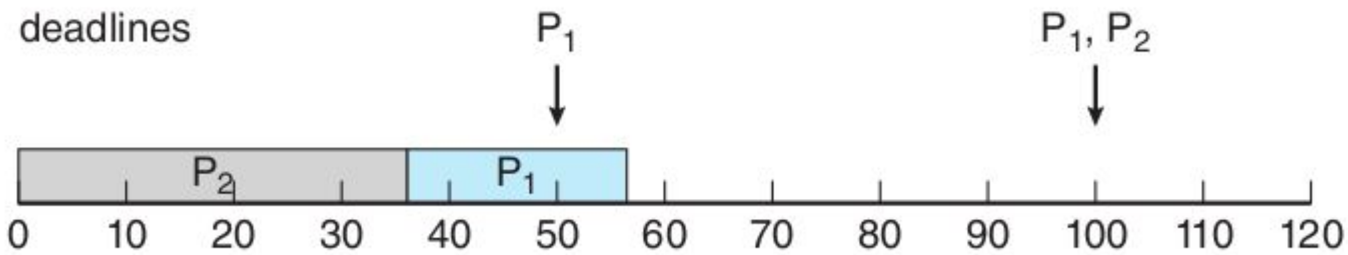


Figure 5.21 Scheduling of tasks when P_2 has a higher priority than P_1.

- A priority is assigned based on the inverse of its period
  - **Shorter periods = higher priority;**
  - **Longer periods = lower priority**
  - **priorities fixed: not changing during the execution**

Example-1:
Periods p and processing time t

P1: p1 = 50, t1 = 20

P2: p2 = 100, t2 = 35

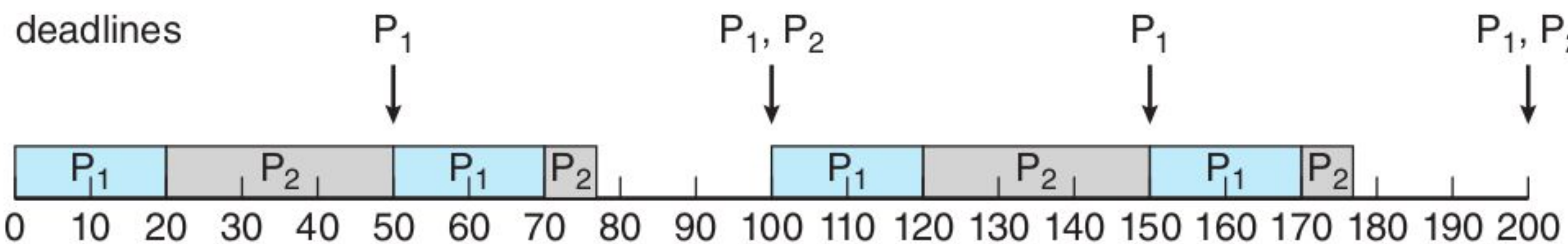


Figure 5.22 Rate-monotonic scheduling.

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

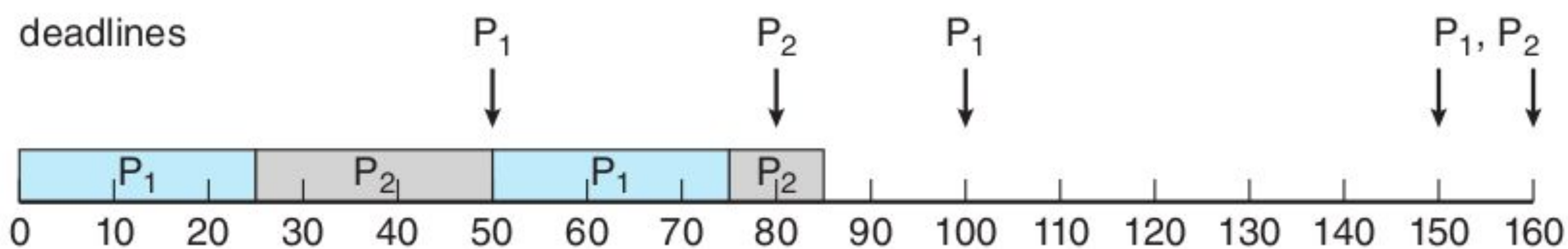


Figure 5.23 Missing deadlines with rate-monotonic scheduling.

Deadline-monotonic priority assignment

Similar to rate monotonic

- fixed-priority pre-emptive scheduling.
  - priorities according to their deadlines.

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
  - **Dynamic priority scheduling**

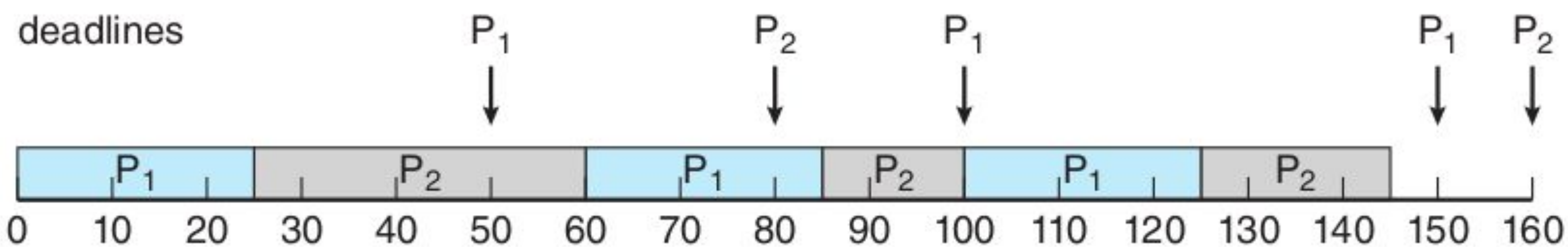


Figure 5.24 Earliest-deadline-first scheduling.

Proportional Share Scheduling

- *fair-share scheduler,*
- *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
  - or gets time based on their weights
- Can be used for soft-real time systems

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 +

- **Scheduling classes (include/asm-generic/vmlinux.lds.h)**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class(pick_next_task() in kernel/sched/core.c)
    - #define SCHED_DATA \
    - STRUCT_ALIGN(); \
    - __sched_class_highest = .; \
    - *(__stop_sched_class) \
    - *(__dl_sched_class) \
    - *(__rt_sched_class) \
    - *(__fair_sched_class) \
    - *(__idle_sched_class) \
    - __sched_class_lowest = .;
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time

Linux Scheduling in Version 2.6.23 +

- Completely Fair Scheduler (CFS) [CFS Scheduler — The Linux Kernel documentation]
- CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.
  - it always tries to run the task with the smallest **p->se.vruntime**
    - the task which executed least so far
  - CFS always tries to split up CPU time between runnable tasks as close to "ideal multitasking hardware" as possible.
  - later versions use **latency-nice** patches for latency

Fairness

Kernel 6.6 new The **"Earliest Eligible Virtual Deadline First"**

- [EEVDF Scheduler — The Linux Kernel documentation]
- **lag value** to determine fair shares
  - assigns vruntime
  - -lag means it ran more
    - not eligible to be the next
  - +lag means it ran less
    - computes VD based on +lags and picks the earliest VD as the next

latency

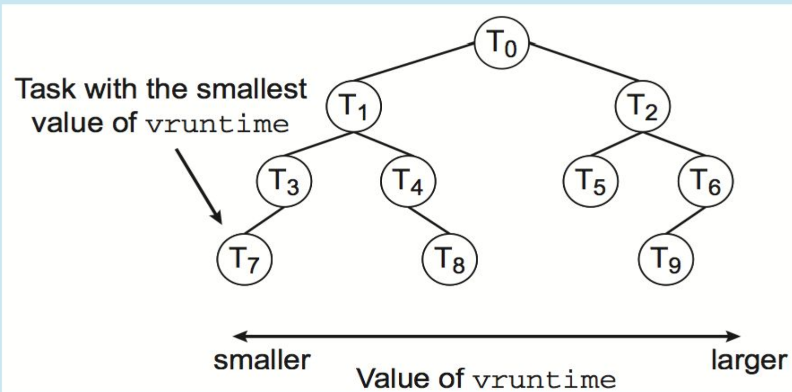
- some process need CPU asap
- interactive processes

Linux Scheduling in Version 2.6.23 + (Cont.)

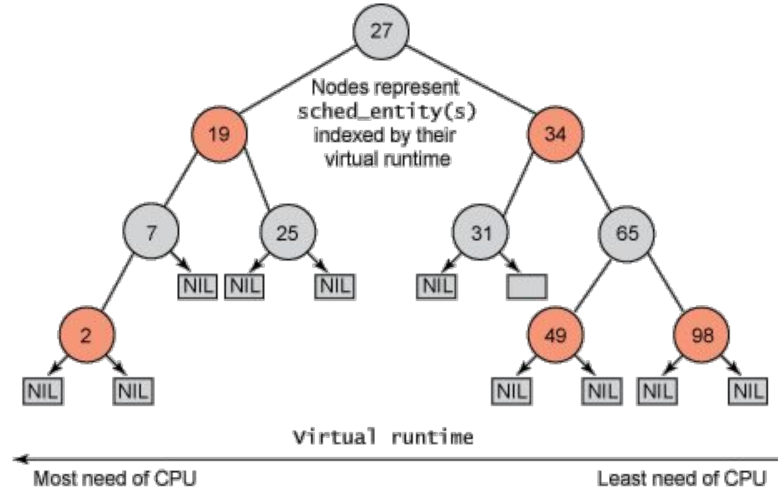
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime (in nanoseconds, not in jiffies or HZ)`**
  - 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.

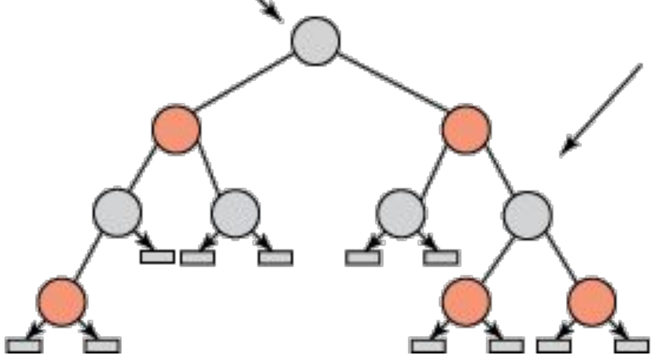


https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

```
struct task_struct {
  volatile long state;
  void *stack;
  unsigned int flags;
  int prio, static_prio normal_prio;
  const struct sched_class *sched_class;
  struct sched_entity se;
  ...
};
```

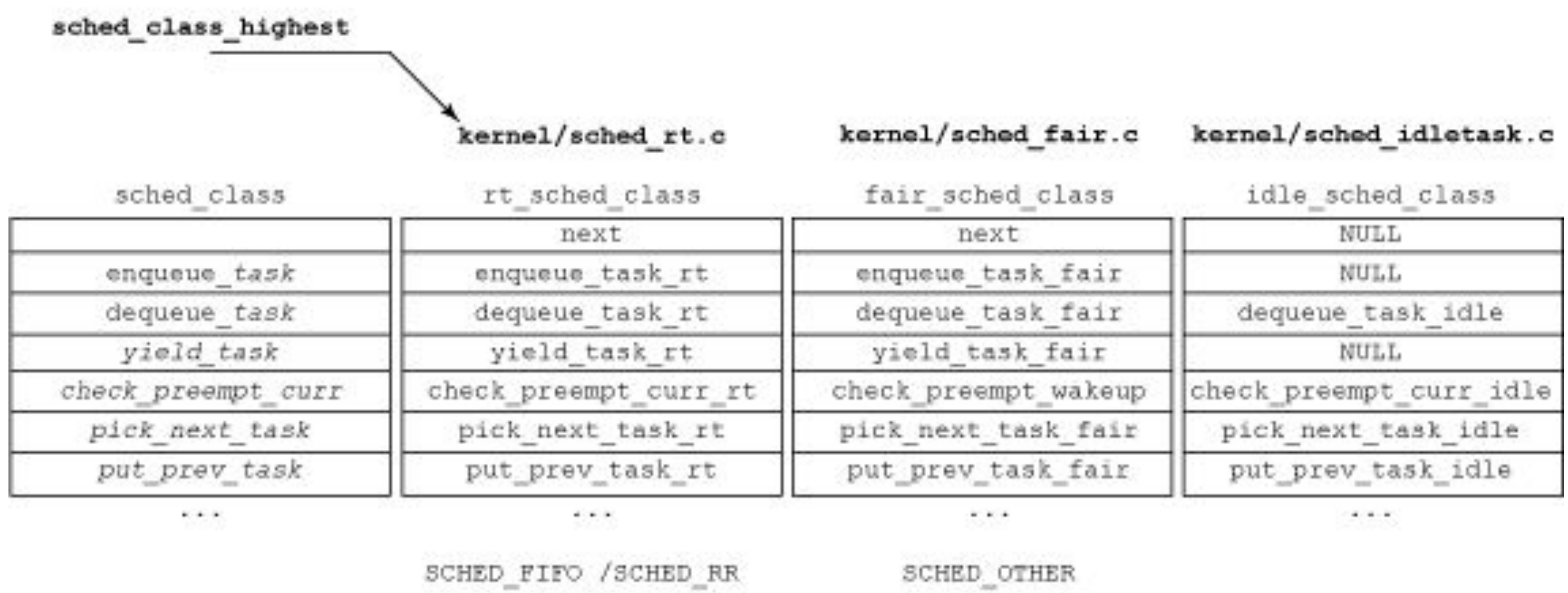
```
struct sched_entity {
  struct load_weight load;
  struct rb_node run_node;
  struct list_head group_node;
  ...
};
```

```
struct ofs_rq {
   ...
   struct rb_root tasks_timeline;
   ...
};
```



```
struct rb_node {
  unsigned long rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};
```

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



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.

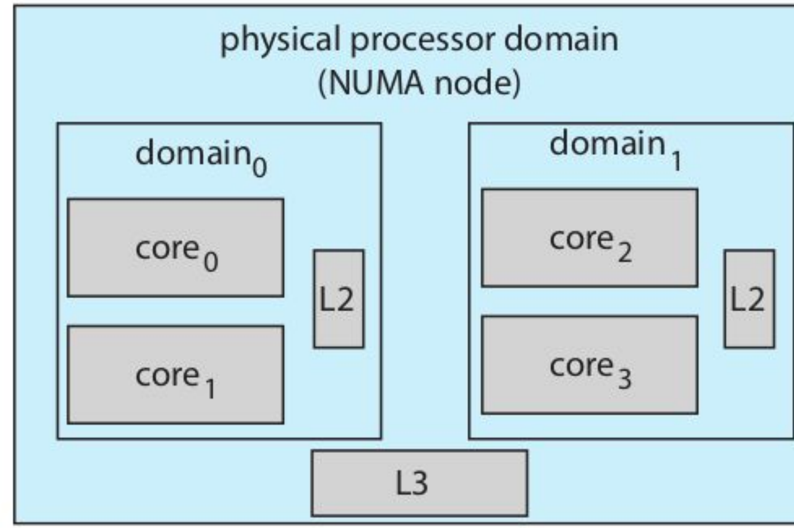


Figure 5.27 NUMA-aware load balancing with Linux CFS scheduler.

[Scheduler Domains — The Linux Kernel documentation]()

Linux Scheduler Implementation Details

How scheduler works

- 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

system calls sched(7) - Linux manual page

FreeBSD(ULE) scheduler

Priority	Class	Thread type
0 – 47	ITHD	Bottom-half kernel (interrupt)
48 – 79	REALTIME	Real-time user
80 – 119	KERN	Top-half kernel
120 – 223	TIMESHARE	Time-sharing user
224 – 255	IDLE	Idle user

- ITHD and KERN classes are managed by the kernel
- REALTIME and IDLE classes are managed by user processes
- TIMESHARE class management shared by kernel and user processes

[An Overview of Scheduling in the FreeBSD Kernel Brought to you by Dr. Marshall Kirk McKusick BSD Canada Conference 2020 June 5,]()

FreeBSD Scheduling Choices

Real time

- processes set specific priorities
- kernel does not change priorities

Interactive scheduler (ULE)

- ULE uses 3 run queues
  - priorities 0-171(real time queue)
  - 172-223(batch queue)
  - 224-255 (idle queue)
- processor affinity
- kernel sets priority based on interactivity score

Share scheduler (4BSD)

- multi-level feedback queues
  - from highest to lowest priority (0-255)
- kernel changes priority based on run behavior

Idle scheduler

- administrator set specific priorities
- kernel does not change priorities

CFS vs ULE

From [The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS]

CFS tries to be fair to all threads, while ULE gives priority to interactive threads.

- **No overall winner!**
- **ULE may cause starvation,**
- **ULE is better in load balancing**

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 runnable thread, runs **idle thread**

[Scheduling - Win32 apps | Microsoft Learn]

Windows Priority Classes

Each process belongs to one of the following priority classes:

- IDLE_PRIORITY_CLASS
  - **priority levels**
- BELOW_NORMAL_PRIORITY_CLASS
  - **priority levels**
- NORMAL_PRIORITY_CLASS
  - **priority levels**
- ABOVE_NORMAL_PRIORITY_CLASS
  - **priority levels**
- HIGH_PRIORITY_CLASS
  - **priority levels**
- REALTIME_PRIORITY_CLASS
  - **priority levels**

priority levels within each priority class:

- **THREAD_PRIORITY_IDLE**
- **THREAD_PRIORITY_LOWEST**
- **THREAD_PRIORITY_BELOW_NORMAL**
- **THREAD_PRIORITY_NORMAL**
- **THREAD_PRIORITY_ABOVE_NORMAL**
- **THREAD_PRIORITY_HIGHEST**
- **THREAD_PRIORITY_TIME_CRITICAL**

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.

- If wait occurs, priority boosted depending on what was waited for
- **Foreground window** given 3x **priority boost**

see [Scheduling Priorities - Win32 apps | Microsoft Learn]

- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
    - UMS thread has its own thread context
    - Unlike fibers sharing the thread context of a single thread.
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework
  - recommended for applications with high performance requirements that need to efficiently run many threads concurrently on multiprocessor or multicore systems.
  - As of Windows 11, UMS is not supported. User-Mode Scheduling - Win32 apps | Microsoft Learn
- Fibers are still supported (as of 2025): Fibers - Win32 apps | Microsoft Learn

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

Overview of the Scheduler - Oracle® Solaris 11.3 Programming Interfaces Guide

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

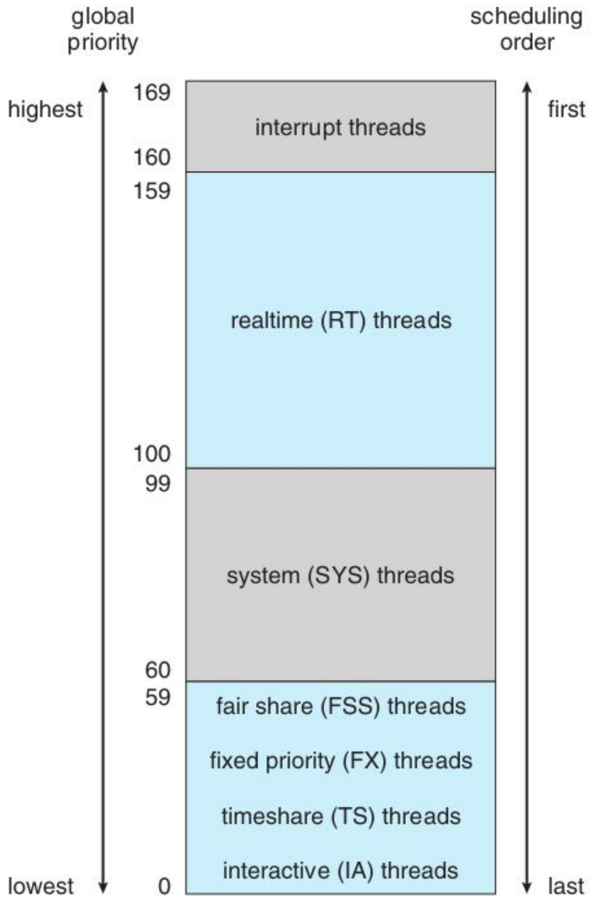
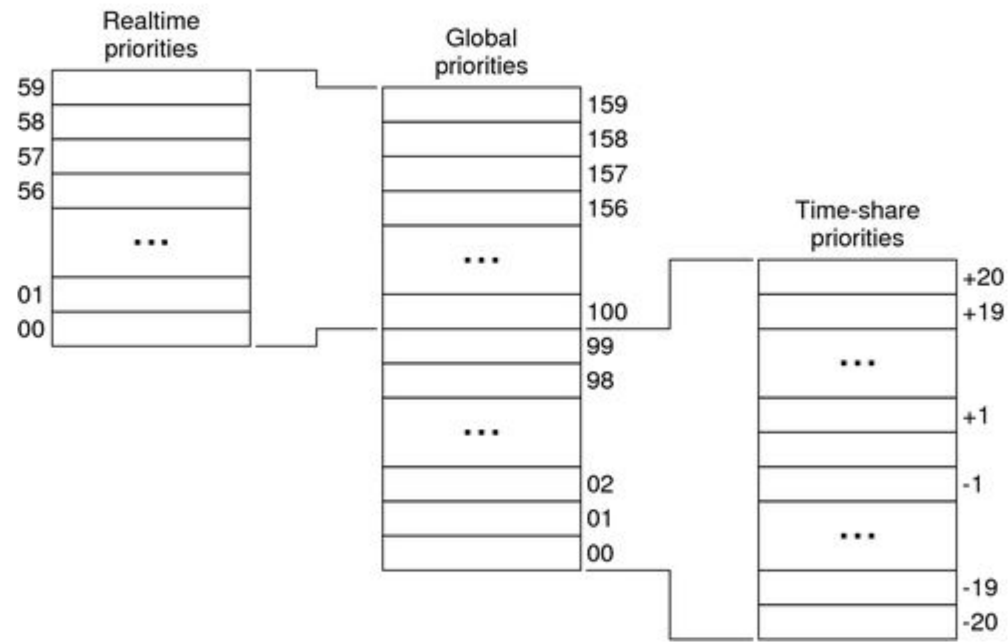


Figure 5.30 Solaris scheduling.



[Real-Time Scheduler - Oracle® Solaris 11.3 Programming Interfaces Guide]

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

Evaluating algorithms

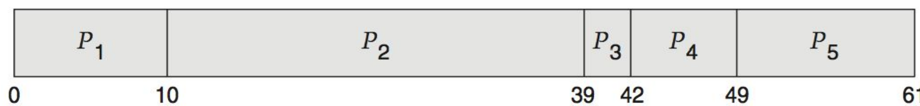
- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
  - deterministic evaluation
  - queueing models
  - simulations
  - implementation

Deterministic Evaluation

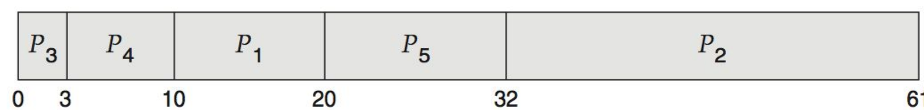
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload

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

- 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

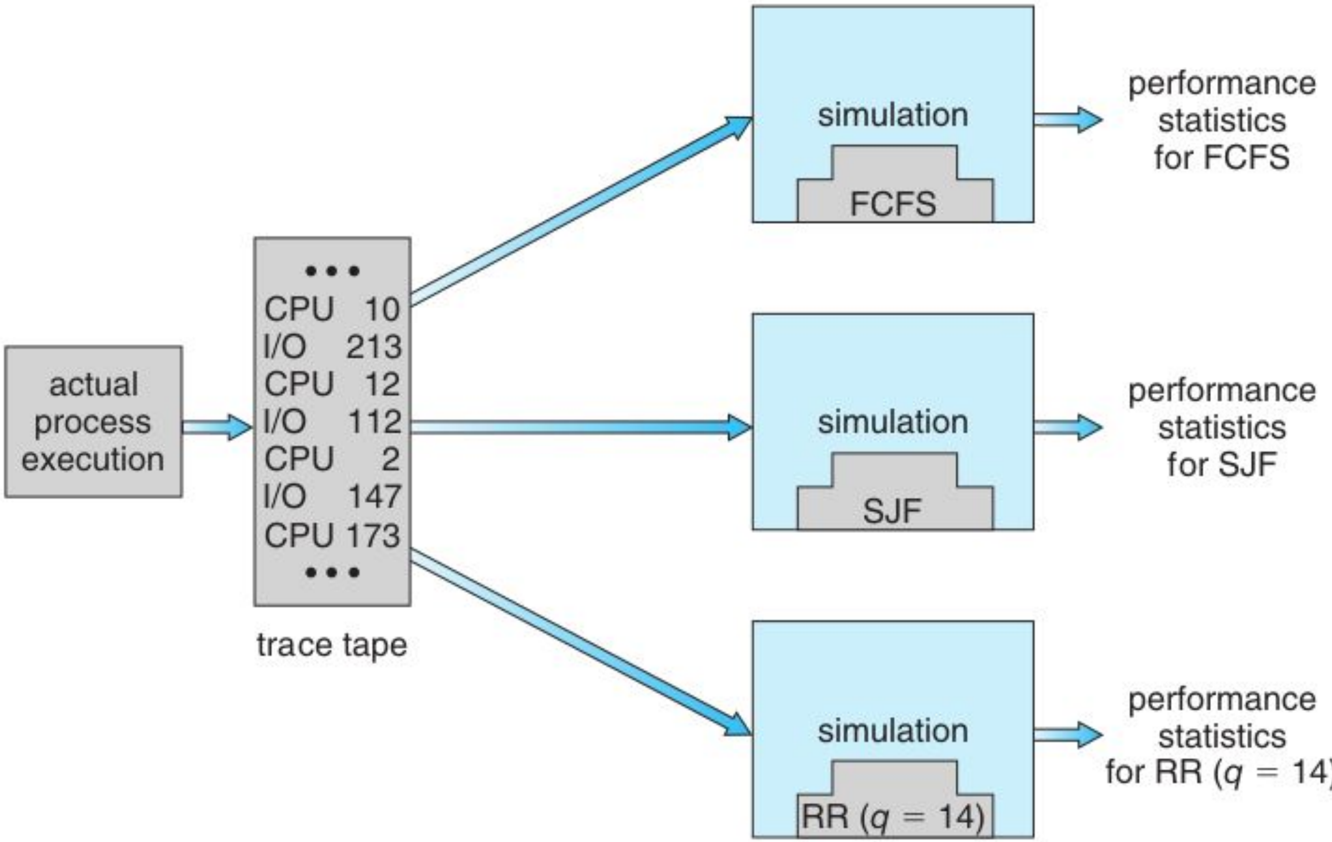


Figure 5.31 Evaluation of CPU schedulers by simulation.

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