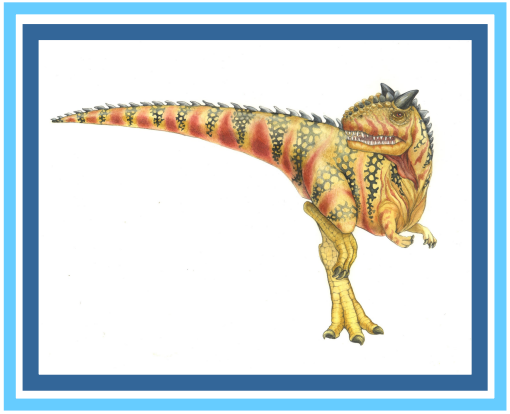
Chapter 6: CPU Scheduling



Chapter 6: CPU Scheduling

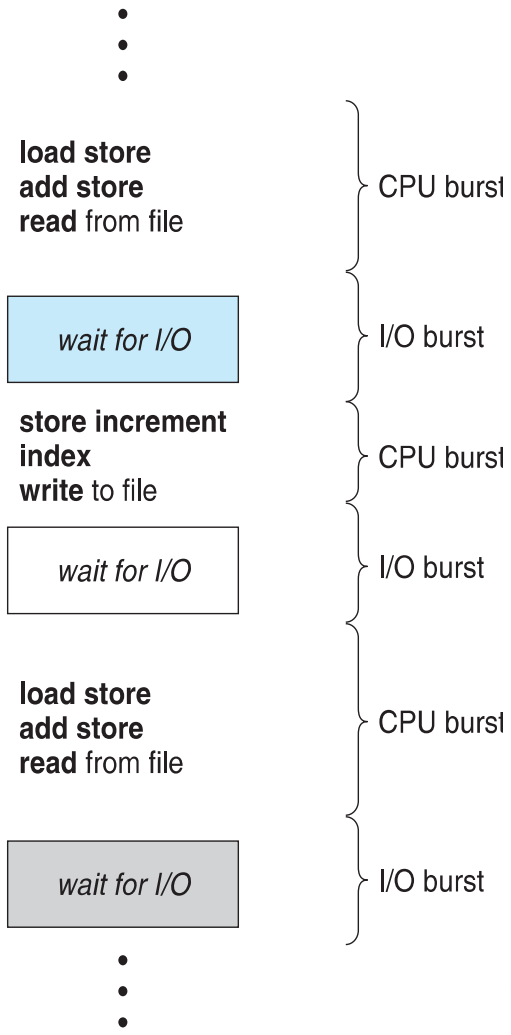
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

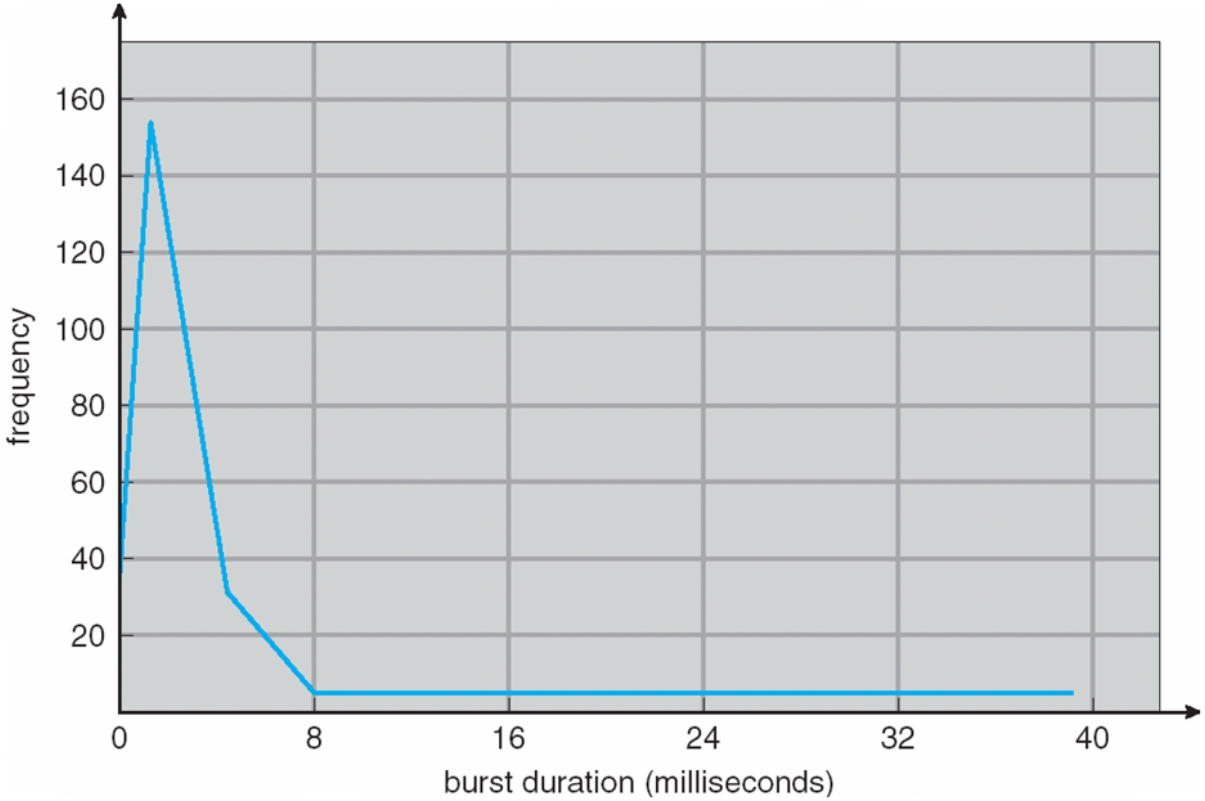
- To introduce CPU scheduling, which is the basis of multi-programmed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

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



Histogram of CPU-burst Times



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways: FIFO, priority, tree, linked list
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (e.g. I/O request or wait for child)
  2. Switches from running to ready state (e. g. timed interrupt)
  3. Switches from waiting to ready (e.g. I/O completion)
  4. Terminates
- If scheduling is done only under 1 and 4, it is **non-preemptive (cooperative)**

  Once a process gets the CPU it keeps it until it releases it
- Otherwise, scheduling is **preemptive** (requires hardware support)
  - Requires hardware support (e.g. timer)
  - Race condition when multiple processes access shared data
  - Consider preemption while in kernel mode modifying kernel data

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler. This involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another.

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes that complete their execution per time unit
- **Turnaround time** – amount of time from submission to completion of a particular process
- **Waiting time** – sum of periods a process spends waiting in the ready queue
- **Response time** – amount of time from the submission of a request to the first response (not output) is produced
  - Important in interactive, time-sharing environments

Scheduling Algorithm Optimization Criteria

- Maximize
  - CPU utilization
  - throughput
- Minimize
  - turnaround time
  - **waiting time** (main criterion)
  - response time
- Usually, we optimize the average measure, but, under some circumstances, we optimize the minimum or maximum value (e.g. minimize the maximum response time in an interactive system)
- Measure of algorithm comparison will be the **average waiting time**
- For simplicity, assume one CPU burst per process

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

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

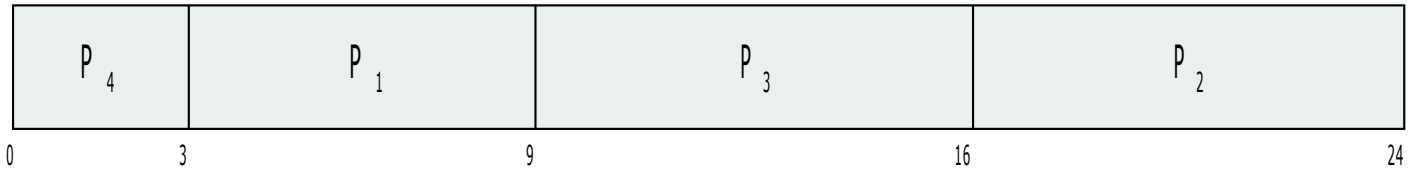
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Assign the CPU to the process with the smallest next CPU burst
  - Ties broken using FCFS
  - More appropriate name would be **shortest-next-CPU-burst**
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- Compare with FCFS time = (6 + 6+8 + 6+8+7) / 4 = 10.25

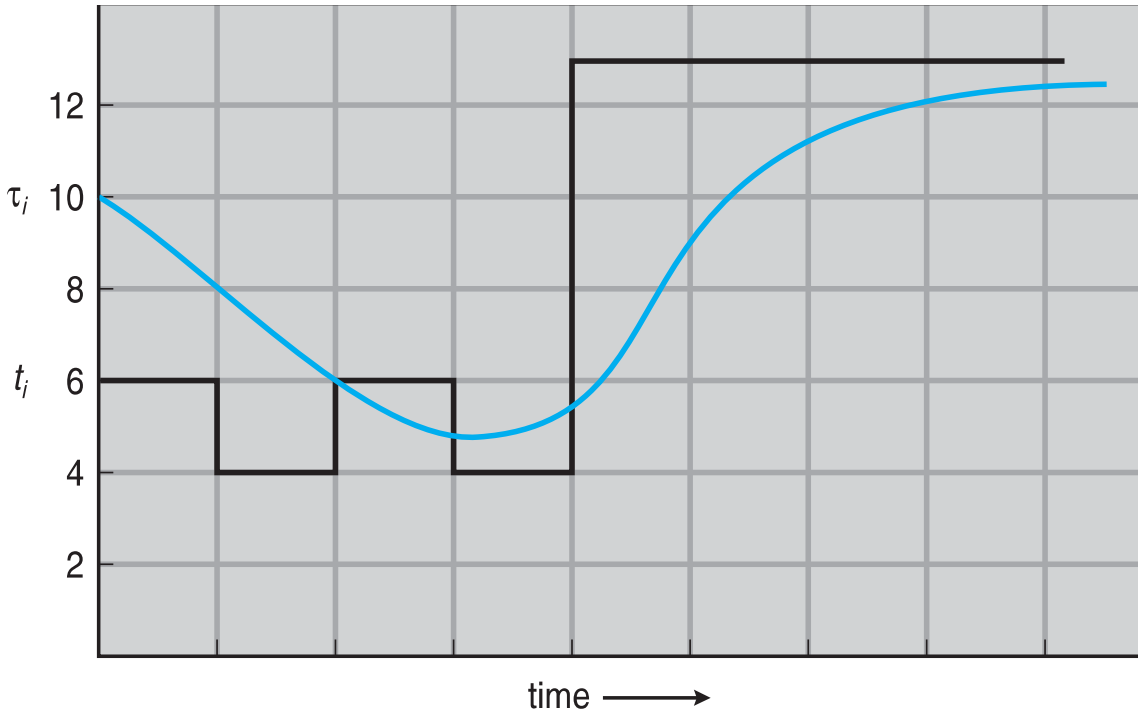
Determining Length of Next CPU Burst

- Can only estimate the length – guess it is similar to previous length
  - Then pick process with shortest predicted next CPU burst
- Can be predicted as an **exponential average** of previous CPU bursts

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.

- Commonly, α set to ½

Prediction of the Length of the Next CPU Burst
α =0.5



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

Examples of Exponential Averaging

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

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

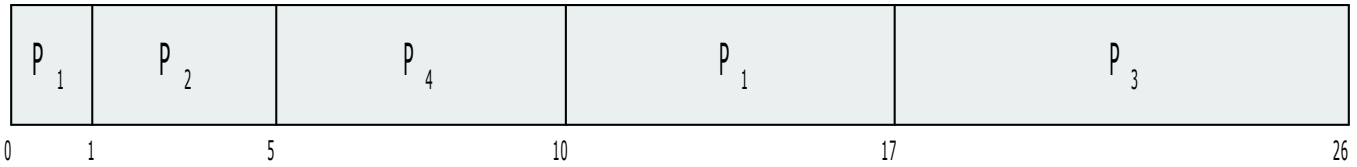
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

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

- *Preemptive* Shortest Remaining Time First:



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

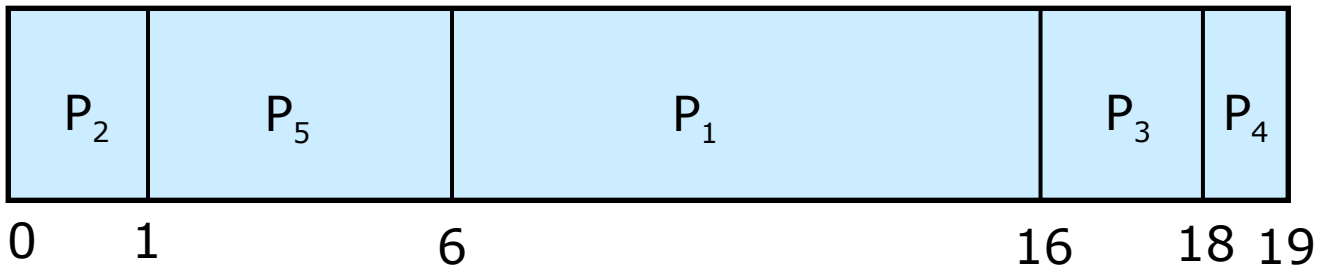
Priority Scheduling

- 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: If a higher priority process arrives, preempt the current process
  - Non-preemptive: If a higher priority process arrives, put it at the head of the ready queue
- SJF is priority scheduling where priority is the predicted next CPU burst time
- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



Average waiting time = (6+0+16+18+1)/5 = 8.2

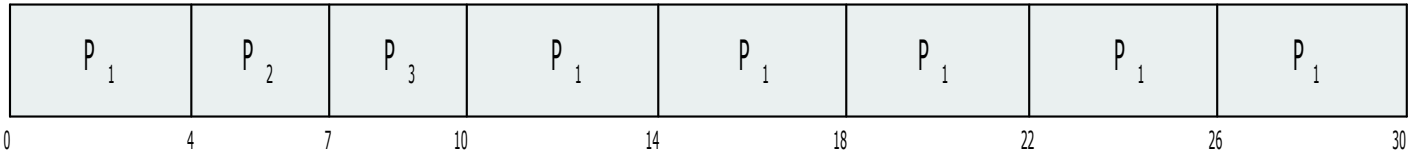
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

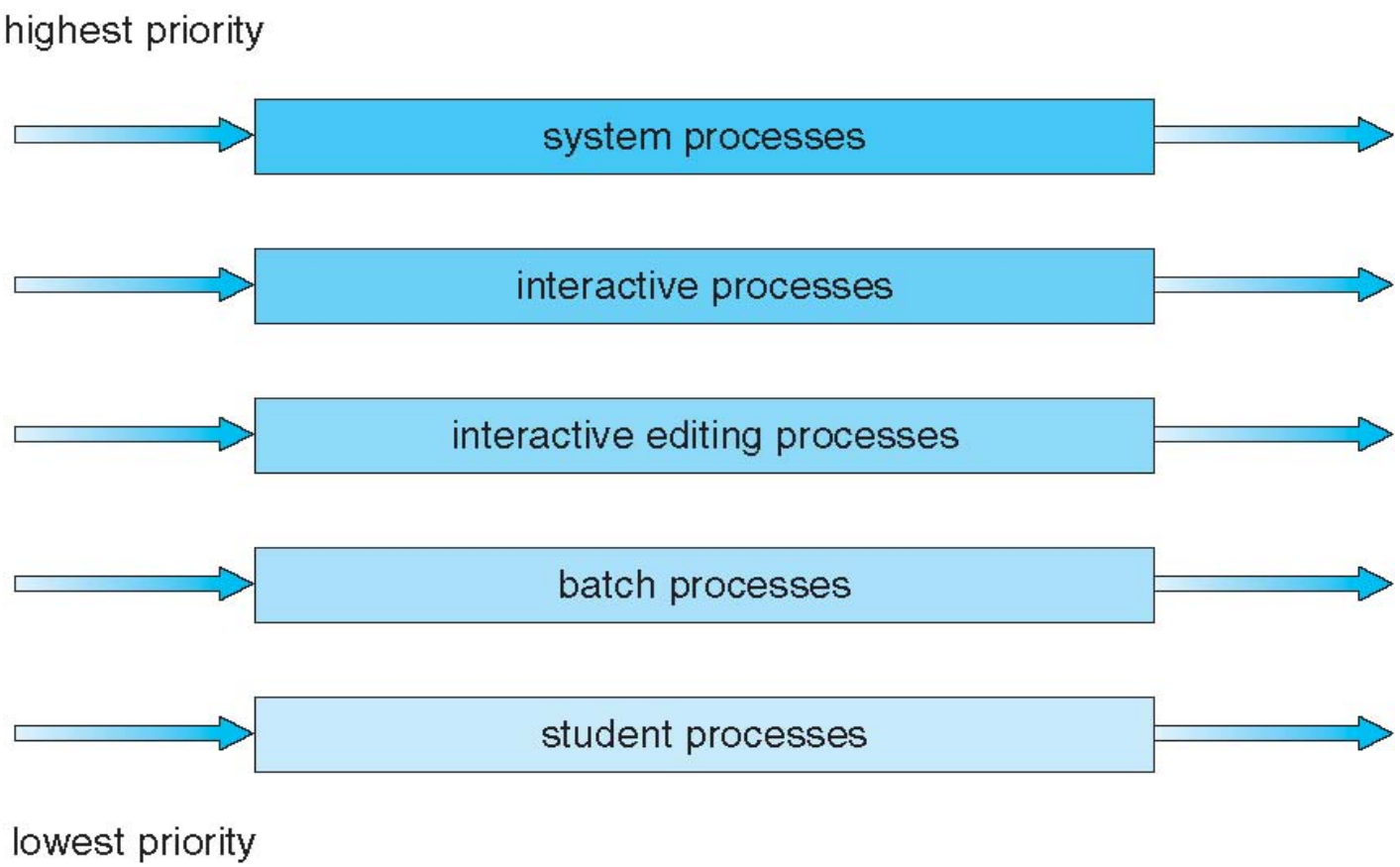


- Average waiting time = (6+4+7)/3 = 5.66 ms
- Typically, higher average turnaround time than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch is typically microseconds

Multilevel Queue

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

Multilevel Queue Scheduling

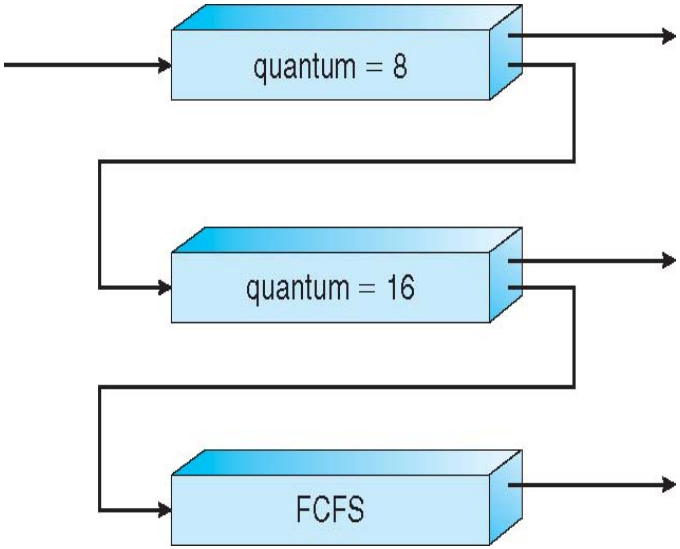


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to promote a process, e.g. more interactive, waited too long in a low priority queue (aging)
  - method used to determine when to demote a process, e.g. used too much CPU time
  - method used to determine which queue a process will enter when that process needs service

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
- Jobs in Q_1 won't be run unless Q_0 is empty
- Jobs in Q_2 won't be run unless Q_0 and Q_1 are empty
- Jobs in Q_0 preempt jobs in Q_1 or Q_2
- Jobs in Q_1 preempt jobs in Q_2
- Scheduling
  - A new job enters queue Q_0
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- **Process-contention scope (PCS):** scheduling competition is within the process
  - Typically done via priority set by programmer
- **System-contention scope (SCS)** – competition among all threads in the 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. Some systems only allow PTHREAD_SCOPE_SYSTEM

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
- Focus on **homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for kernel data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common (Windows, Linux and Mac OS)
- **Processor affinity** – process has affinity for processor on which it is currently running (an important reason is caching)
  - **soft affinity:** no guarantee
  - **hard affinity:** process provides a subset of processors on which it may run. e.g. sched_setaffinity() system call on Linux

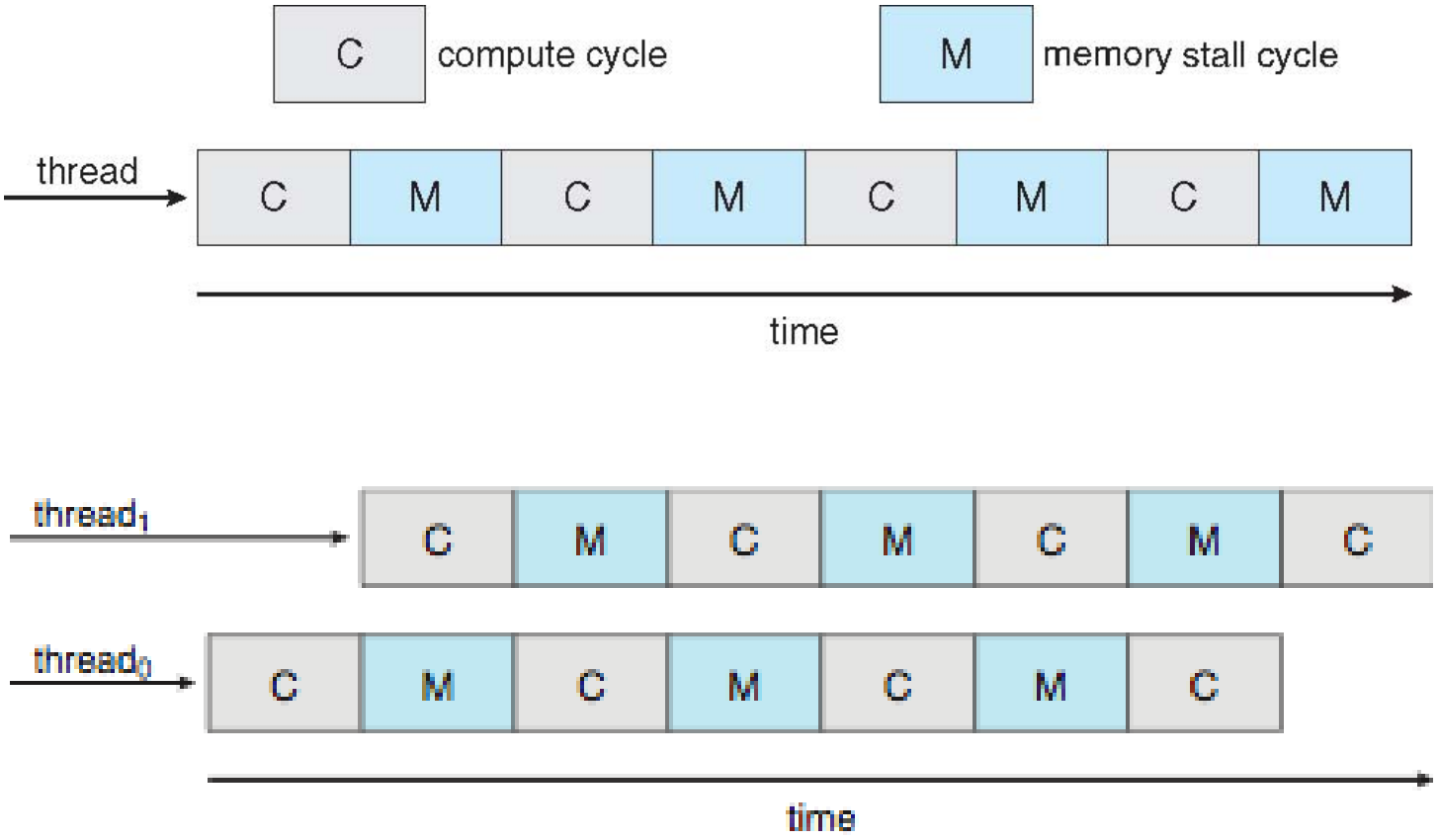
Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- Often unnecessary if there is a common queue, because once a processor becomes idle, it picks a process from the queue
- **Push migration** – periodic task checks load on each processor, and pushes tasks from overloaded CPU to idle or less loaded CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor
- Conflicts with processor affinity:
  - On some systems, idle processor always pulls processes from non-idle processor
  - On other systems, move is made only if imbalance exceeds a certain threshold

Multicore Processors

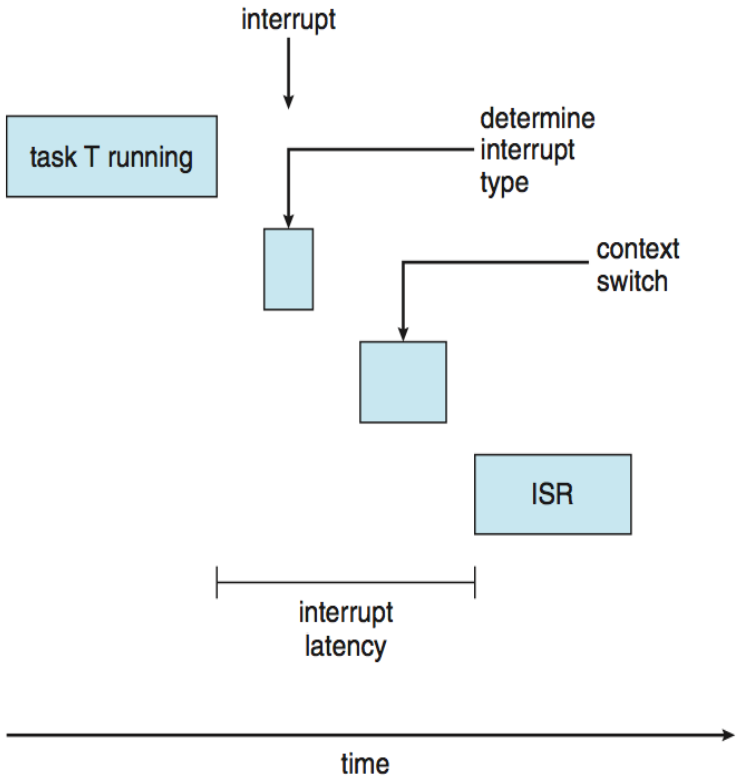
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while waiting on memory

Multithreaded Multicore System



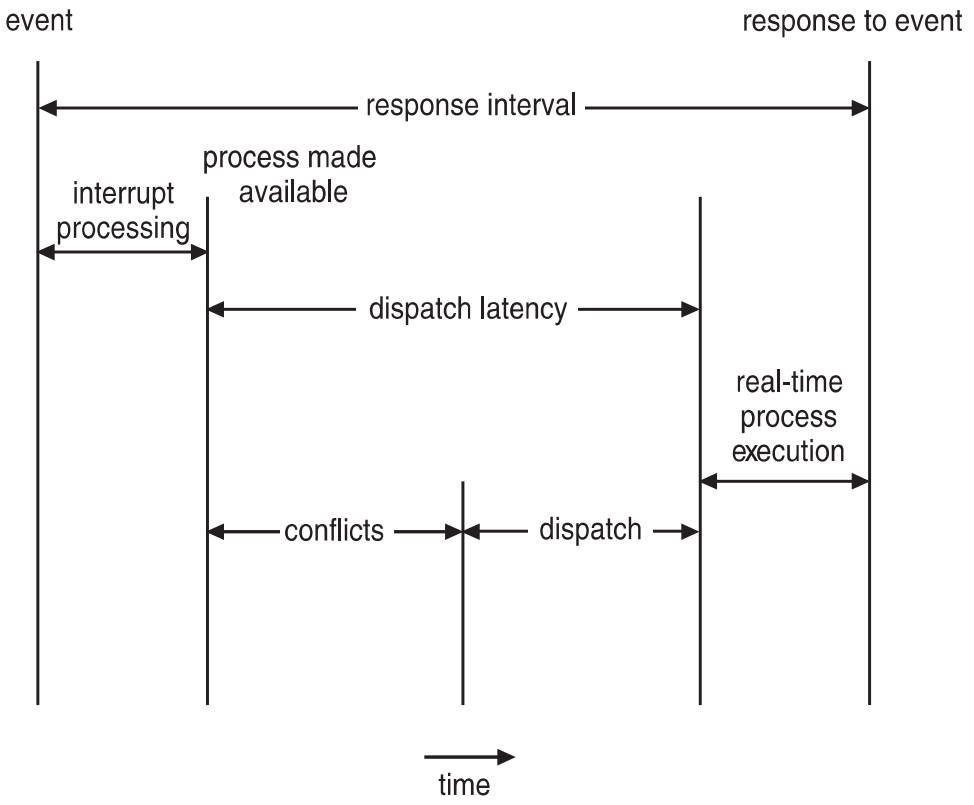
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- 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



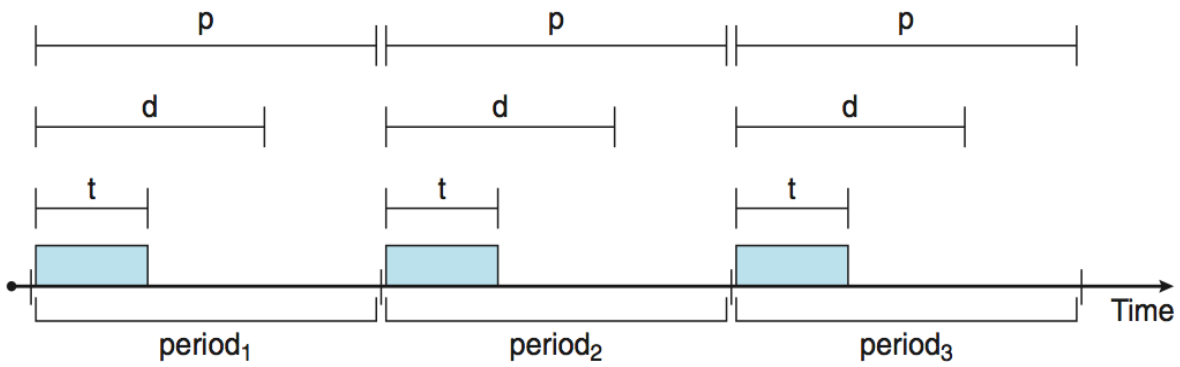
Real-Time CPU Scheduling (Cont.)

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



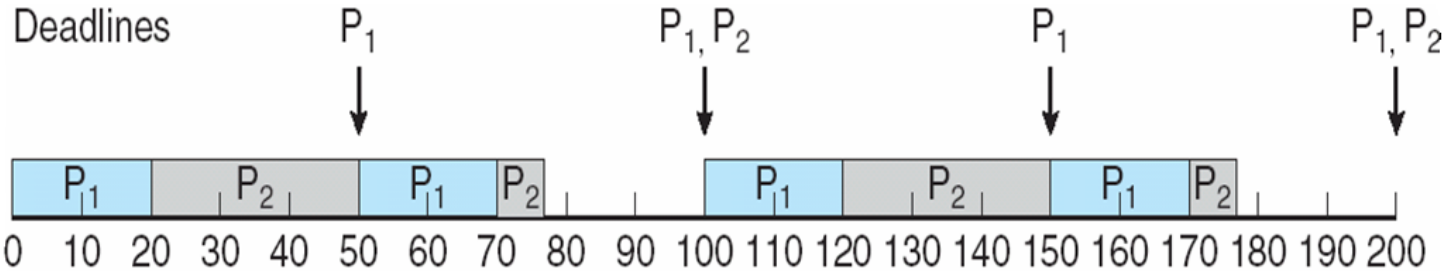
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - Only gives soft real-time scheduling
- For hard real-time must also provide ability to meet deadlines
  - Process must announce its deadline to the system
- Processes have special 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 $1/p$



Rate Montonic Scheduling

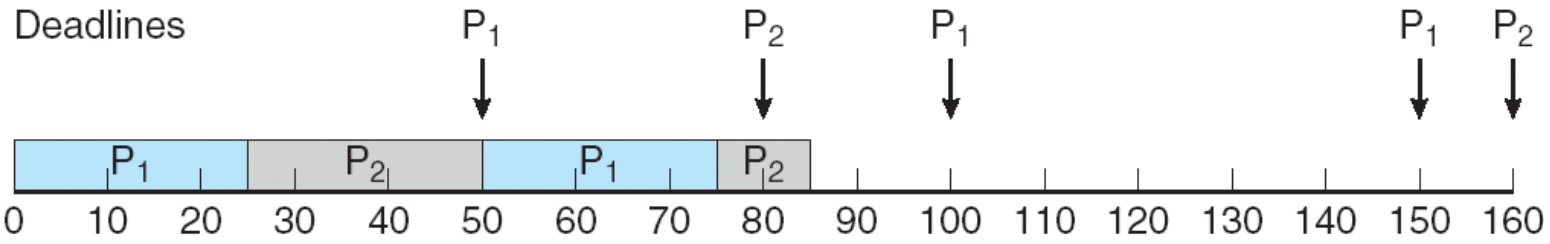
- Static priority with preemption
- Priority is assigned based on period:
  - Shorter period gets higher priority
  - Longer period gets lower priority
- Example:
  - p_1=50, t_1=20, d_1=50
  - p_2 = 100, t_2=35, d_2=100
- P_1 is assigned a higher priority than P_2.



Missed Deadlines with Rate Monotonic Scheduling

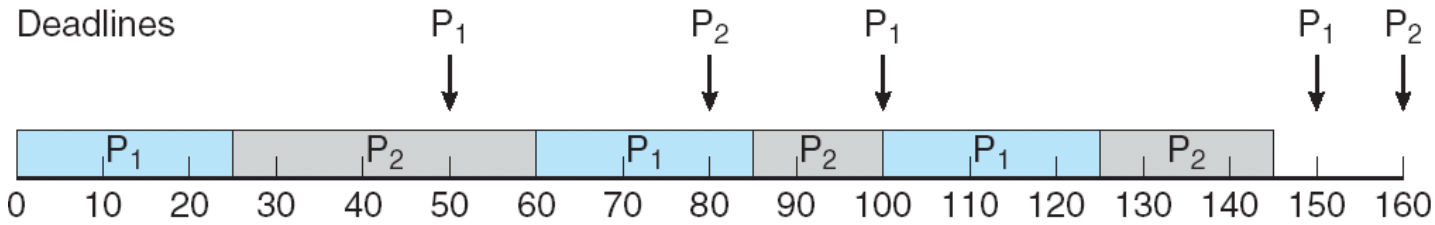
- Example:
  p_1=50, t_1=25, d_1=50
  p_2 = 80, t_2=35, d_2=80

  With Rate Monotonic Scheduling,
  P_2 will miss its deadline



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - Earlier deadline, higher priority
  - Later deadline, lower priority
- Does not require processes to be periodic or have constant CPU bursts
- Same example:
  p_1=50, t_1=25, d_1=50
  p_2 = 80, t_2=35, d_2=80



POSIX Real-Time Scheduling

- The POSIX.1b API standard 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:

n `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`

n `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

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

Linux Scheduling Through Version 2.5

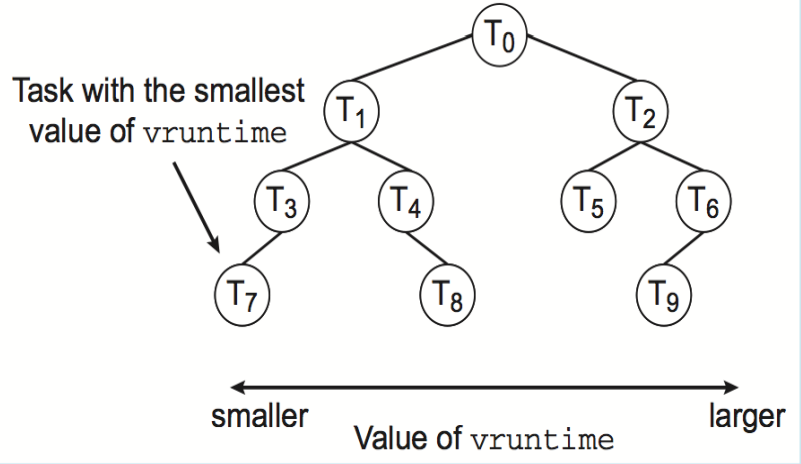
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
  - Did not perform well on SMP
- Version 2.5 moved to a constant-time (*O*(1)) scheduling algorithm:
  - Support for SMP, including processor affinity and load balancing
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - 2 scheduling classes included, others can be added
    1. Default using the ***Completely Fair Scheduler*** (CFS)
    2. real-time
- Time Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Normal default priority (nice value = 0) : virtual run time = actual run time
  - Low priority (nice value > 0) : virtual run time > actual run time
  - High priority (nice value < 0) : virtual run time < actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time
- A higher priority task that becomes ready can preempt a lower priority task
- Example: CPU-bound task and I/O-bound task with same initial nice value
  - I/O-bound task will have a smaller virtual run time
  - I/O-bound task will get higher priority and may preempt CPU-bound task

CFS Performance

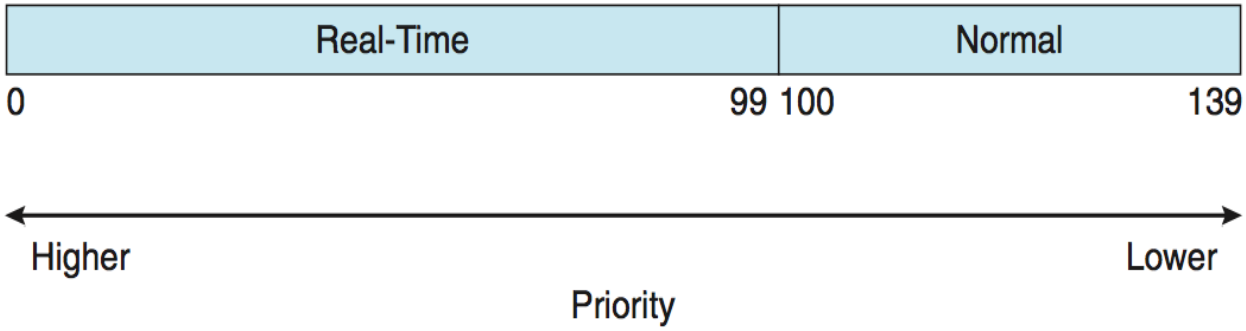
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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.

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b (FIFO and RR)
- Both real-time and normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- Thread runs until it
  - Blocks (e.g. I/O) or
  - Terminates
  - Uses time quantum or
  - Gets preempted by a higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
  - Larger number indicates higher priority
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no ready thread found, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS
  - BELOW_NORMAL_PRIORITY_CLASS
  - IDLE_PRIORITY_CLASS
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class

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

Windows Priority Classes (Cont.)

- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for (keyboard, disk, etc)
- Foreground window given 3x priority boost

Algorithm Evaluation

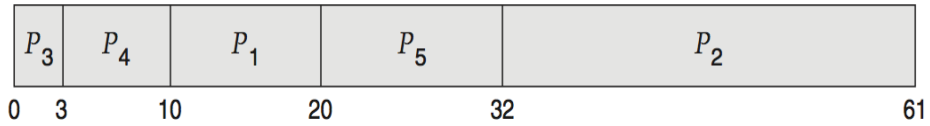
- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving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

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

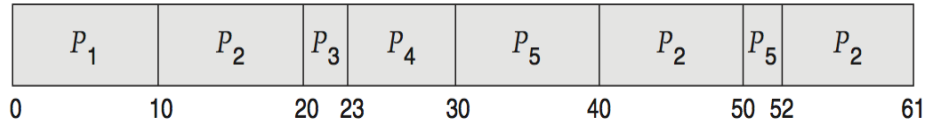
Deterministic Evaluation

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

  - Non-preemptive SJF is 13ms:



  - RR (time quantum = 10) is 23ms:



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

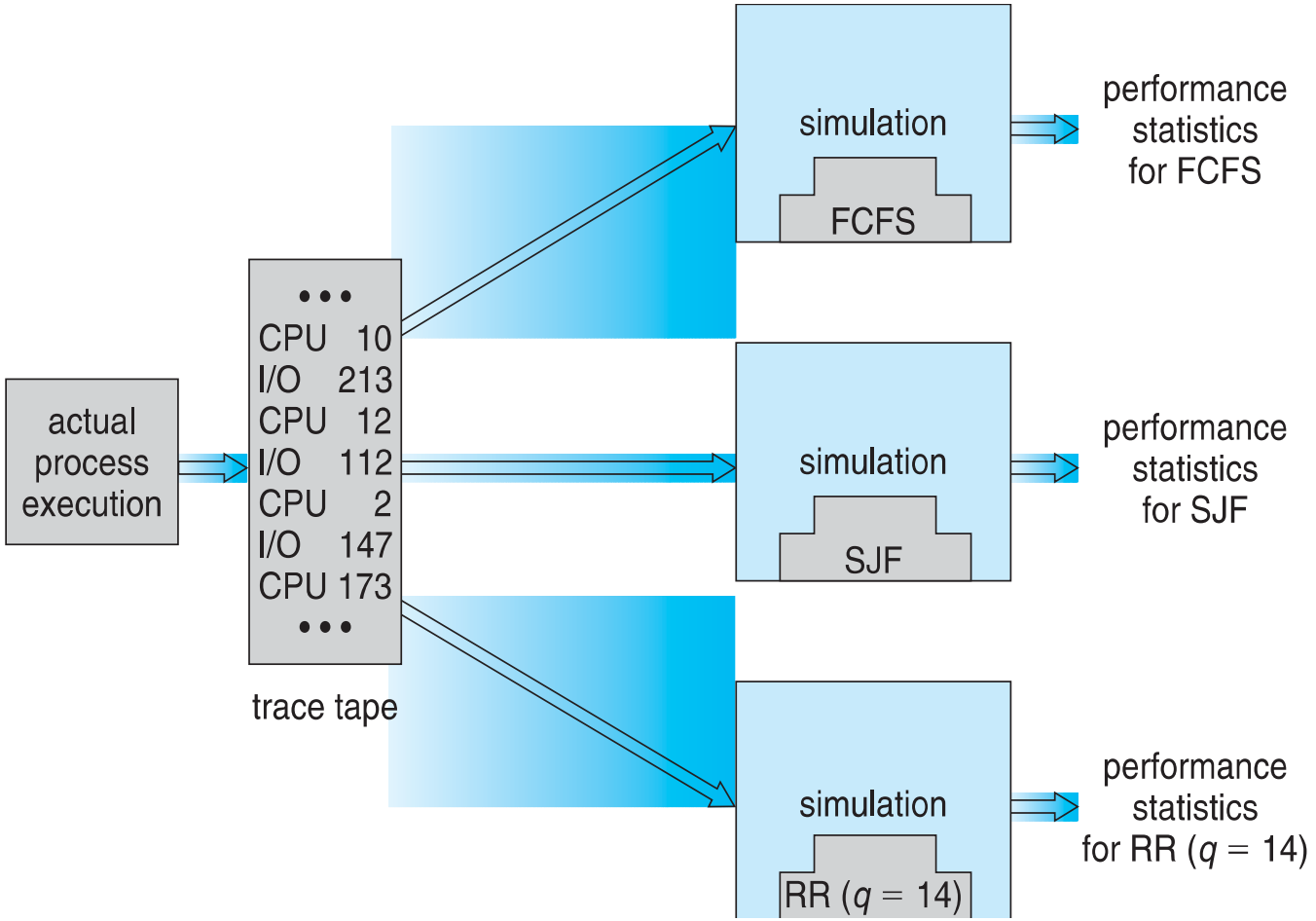
  $n = \lambda \times W$

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

Simulations

- Queueing models limited, because they often make unrealistic assumptions to avoid mathematical complexity, thus giving inaccurate results
- **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



Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
  - Users may adjust their applications to get more CPU time
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

End of Chapter 6