CMPS 405: OPERATING SYSTEMS

Linux CPU Scheduling

Related System Calls and C Library Functions

This material is based on the operating system books by Silberschatz and Stallings
And other Linux, system programming, and simulation books.

Objectives

- Motivations
- Invoking CPU Scheduler
- Preemptive and Non-Preemptive Scheduling
- Performance measures of interest
- Basic Scheduling Algorithms
- Other Scheduling Algorithms
- Linux scheduling related policies and system calls
- The actual Linux CPU scheduler

Linux Scheduler

- The process scheduler is the component of a kernel that selects which process to run next. In other words, the process .
- The scheduler is the subsystem of the kernel that divides the finite resource of processor time among a system's processes.
- The behavior of the Linux scheduler with respect to a process depends on the process's scheduling policy, also called the scheduling class.
- A preprocessor macro from the header <sched.h> represents each policy: the macros are SCHED_FIFO, SCHED_RR, and SCHED_OTHER (the default policy).
  - Every process possesses a static priority, unrelated to the nice value. For normal applications, this priority is always 0. For the real-time processes, it ranges from 1 to 99, inclusive.
  - The Linux scheduler always selects the highest-priority process to run (i.e., the one with the largest numerical static priority value).
  - Because normal processes have a priority of 0, any real-time process that is runnable will always preempt a normal process and run.

Preemptive and non-preemptive Scheduling

- In Preemptive scheduling: a process under execution in the CPU can be interrupted and brought out of the CPU (Examples include most of modern OSs).
- In Nonpreemptive (cooperative) scheduling: once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by
  - terminating or
  - switching to waiting state.

Comments on Preemptive Scheduling

- These situations must be taken care of when using preemptive scheduling:
  - In the case of two processes sharing data an inconstant state of the shared data could be caused by one of them interrupting the other.
    - Example before the first process finish updating the data the second interrupted to read it.
  - Interrupting an activity of a process that may involve changing important kernel data.
    - What if the interrupting process is accessing that kernel data?
      - Currently, most modern OS disable interrupts during the execution of such activity.
        - » A drawback will be the performance of this kernel is poor with real time systems.
  - Guarding sections of code from being accessed concurrently by other process is done by disabling interrupts at the entry of these sections and enabling it at exit. These sections should not contain more than few instructions.

Dispatching

- The dispatch latency is the time needed by the dispatcher module to stop one process and start another running in the CPU.

- The dispatcher involves:
  - context switching,
  - switching to user mode,
  - jumping to proper location in the new user program to restart it.

Scheduling Algorithms

- The set of basic scheduling algorithms includes:

Basic Scheduling Algorithms
First-Come, First-Served (FCFS)
Nonpreemptive Shortest-Job First (SJF)
Preemptive SJF Shortest-Remaining Time First (SRTF)
Nonpreemptive Priority
Preemptive Priority
Round-Robin (RR)
Multilevel queueing
Multilevel feedback-queueing

Evaluation of CPU scheduling algorithms

- The evaluation is concern with determining the level of the goodness of the performance of a system under a given CPU scheduling algorithm.
  - Such goodness is measured by the averages of a set of performance measurements including:
    - average waiting time,
    - average turnaround time
    - Etc.

Performance Measurements of interest

- CPU utilization: keep the CPU as busy as possible.
- Throughput: # of processes that complete their execution per time unit
- Turnaround time: amount of time to execute a particular process
- Waiting time: amount of time a process has been waiting in the ready queue
- Response time: amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling Algorithms

- First-Come, First-Served (FCFS)
  - Processes are processed based on their arrival times FIFO.
- Shortest-Job First (SJF)
  - Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time first.
    - Nonpreemptive: once CPU given to the process it cannot be preempted until completes its CPU burst.
    - Preemptive SJF or Shortest-Remaining Time First (SRTF): if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
  - SJF is optimal – gives minimum average waiting time for a given set of processes.

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

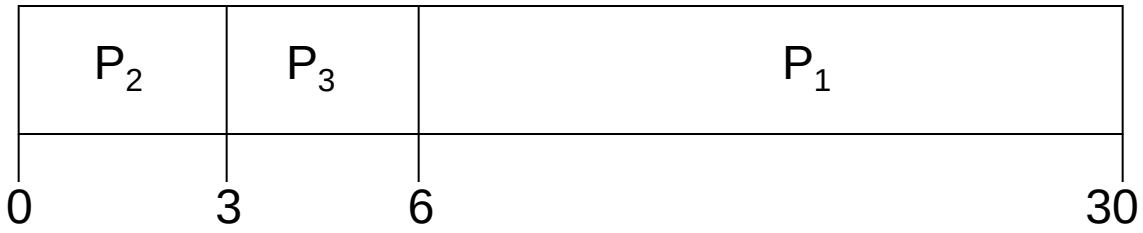
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:

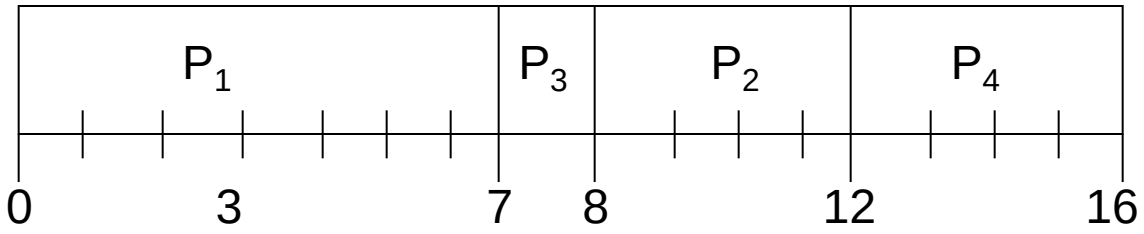


- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (non-preemptive)

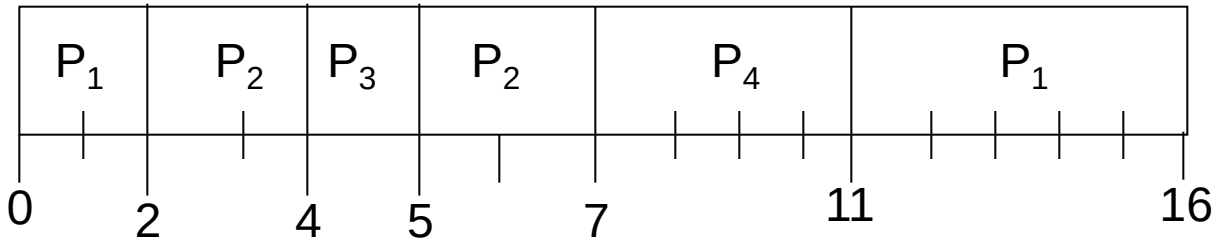


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

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

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

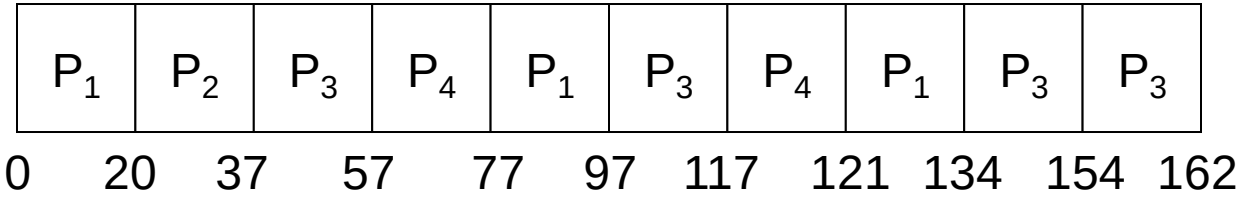
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Context switching time is about 0.1-1 ms leading to 1% overhead due to context switching.
- 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.
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

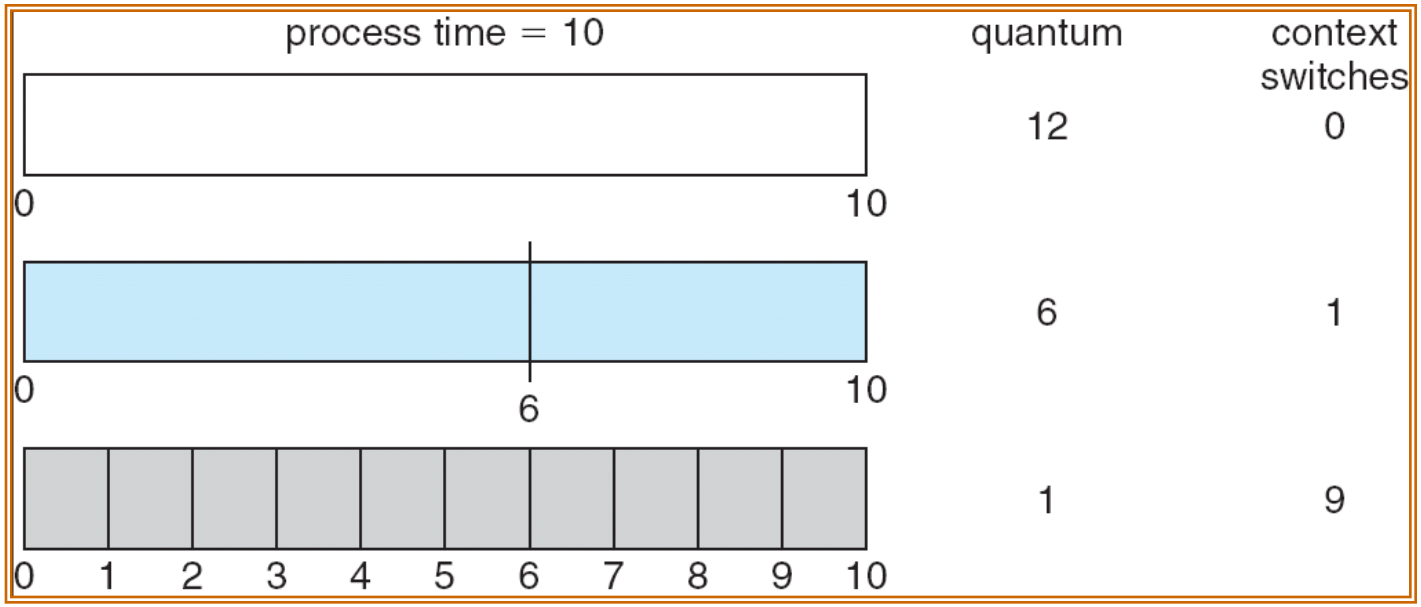
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Q. *What constraints would you set on the value of the quantum time of round robin scheduling algorithm. Explain.*

Scheduling Algorithms

- 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).
  - Processes of higher priority are served before those with lower priority.
    - Nonpreemptive Priority: once CPU given to the process it cannot be preempted until completes its CPU burst.
    - Preemptive Priority: current executing process is preempted if a new process arrives with a higher priority.

Comparison of CPU Scheduling

- SJF is the optimal CPU scheduling among nonpreemptive algorithms and SRTF is the optimal CPU Scheduling among preemptive algorithms.
- Since we can never know how long a job will run, SJF and SRTF cannot be accurately implemented. Therefore, they can only be used to benchmark the goodness of an algorithm since they are the optimal.
- When the jobs are of similar (almost equal) burst times, SJF and SRTF become the same as FIFO.
- RR behaves the same as FIFO when the quantum time is very long with respect to burst times.
- RR has less average response time than FIFO when the jobs are of varying burst times.
- SRTF can lead to starvation if many small jobs. Large jobs will never run.
- Priority scheduling can lead to starvation is many high priority jobs. Low priority jobs will never run.
-

CPU Scheduling Ex.1:

- Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

  The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

Process	Burst Time	Priority
***P*1**	**10**	**3**
***P*2**	**1**	**1**
***P*3**	**2**	**3**
***P*4**	**1**	**4**
***P*5**	**5**	**2**

1. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.
2. What is the turnaround time of each process for each of the scheduling algorithms in part 1?
3. What is the waiting time of each process for each of the scheduling algorithms in part 1?
4. Which of the schedules in part 1 results in the minimal average waiting time (over all processes)? ***ANS: Shortest Job First!***

1. Gantt charts

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Time
P1										P2	P3		P4	P5					FCFS
P1	P2	P3	P4	P5	P1	P3	P5	P1	P5	P1	P5	P1	P5	P1					RR
P2	P4	P3		P5					P1										SJF
P2	P5					P1										P3	P4		Prior-ity

Process	Burst Time	Priority
*P*1	10	3
*P*2	1	1
*P*3	2	3
*P*4	1	4
*P*5	5	2

2. Turnaround Times

P.No.	FCFS	RR	SJF	Priority
*P*1	10	19	19	16
*P*2	11	2	1	1
*P*3	13	7	4	18
*P*4	14	4	2	19
*P*5	19	14	9	6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Time
P1										P2	P3		P4	P5					FCFS
P1	P2	P3	P4	P5	P1	P3	P5	P1	P5	P1	P5	P1	P5	P1					RR
P2	P4	P3		P5					P1										SJF
P2	P5					P1										P3		P 4	Prior-ity

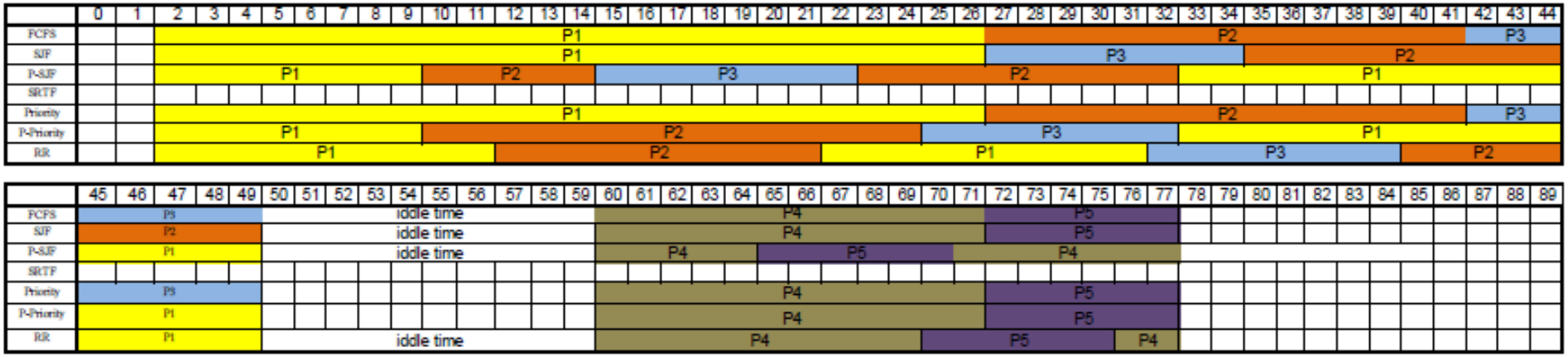
3. Waiting Times

	FCFS	RR	SJF	Priority
***P*1**	**0**	**9**	**9**	**6**
***P*2**	**10**	**1**	**0**	**0**
***P*3**	**11**	**5**	**2**	**16**
***P*4**	**13**	**3**	**1**	**18**
***P*5**	**14**	**9**	**4**	**1**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Time
P1										P2	P3		P4	P5					FCFS
P1	P2	P3	P4	P5	P1	P3	P5	P1	P5	P1	P5	P1	P5	P1					RR
P2	P4	P3		P5					P1										SJF
P2	P5					P1										P3	P4		Prior-ity

Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Arrival Time	Burst Time	Priority
P_1	2	25	5
P_2	10	15	3
P_3	15	8	3
P_4	60	12	2
P_5	65	6	2



Delay/Queueing/waiting time

	FCFS	SJF	P-SJF	Priority	P-Priority	RR
P_1	0	0	23	0	23	23
P_2	17	25	8	17	0	20
P_3	27	12	0	27	10	17
P_4	0	0	6	0	0	6
P_5	7	7	0	7	5	5
Avr	10.2	8.8	7.4	10.2	7.6	14.2

Response Time

	FCFS	SJF	P-SJF	Priority	P-Priority	RR
P_1	0	0	0	0	0	0
P_2	17	25	0	17	0	2
P_3	27	12	0	27	10	17
P_4	0	0	0	0	0	0
P_5	7	7	0	7	7	5
Avr	10.2	8.8	0	10.2	3.4	4.8

Turnaround Time

	FCFS	SJF	P-SJF	Priority	P-Priority	RR
P_1	25	25	48	25	48	48
P_2	32	40	23	32	15	35
P_3	35	20	8	35	18	25
P_4	12	12	18	12	12	18
P_5	13	13	6	13	11	11
Avr	23.4	22	20.6	23.4	21	27.4

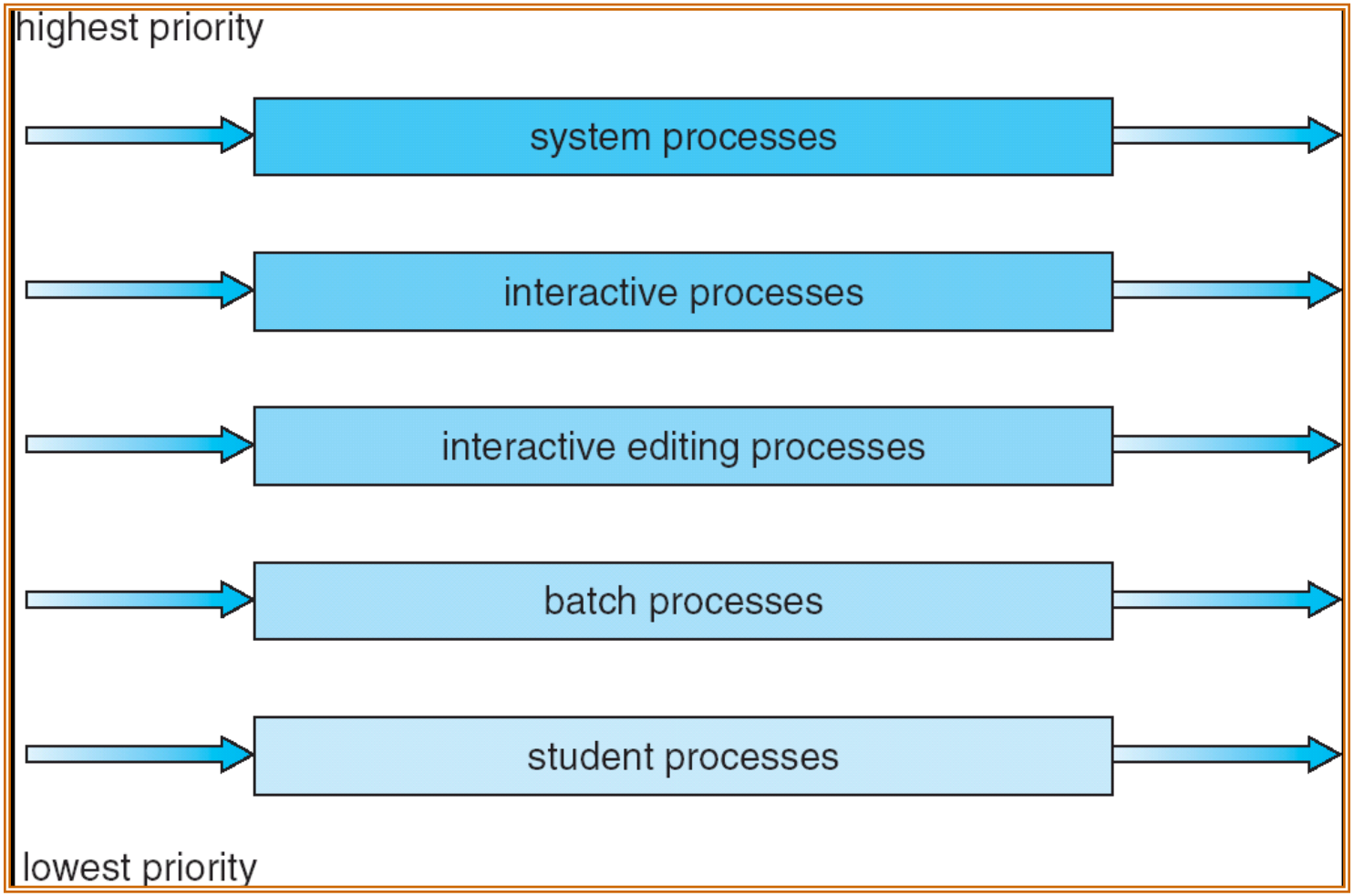
Practice

- Examples of tracing CPU scheduling in tabular format ordered by time.

Objectives

- Motivations
- Invoking CPU Scheduler
- Preemptive and Non-Preemptive Scheduling
- Performance measures of interest
- Basic Scheduling Algorithms
- Other Scheduling Algorithms
- Linux scheduling related policies and system calls
- The actual Linux CPU scheduler
- Evaluation of CPU Scheduling Algorithms
- Simulation of CPU scheduling algorithms

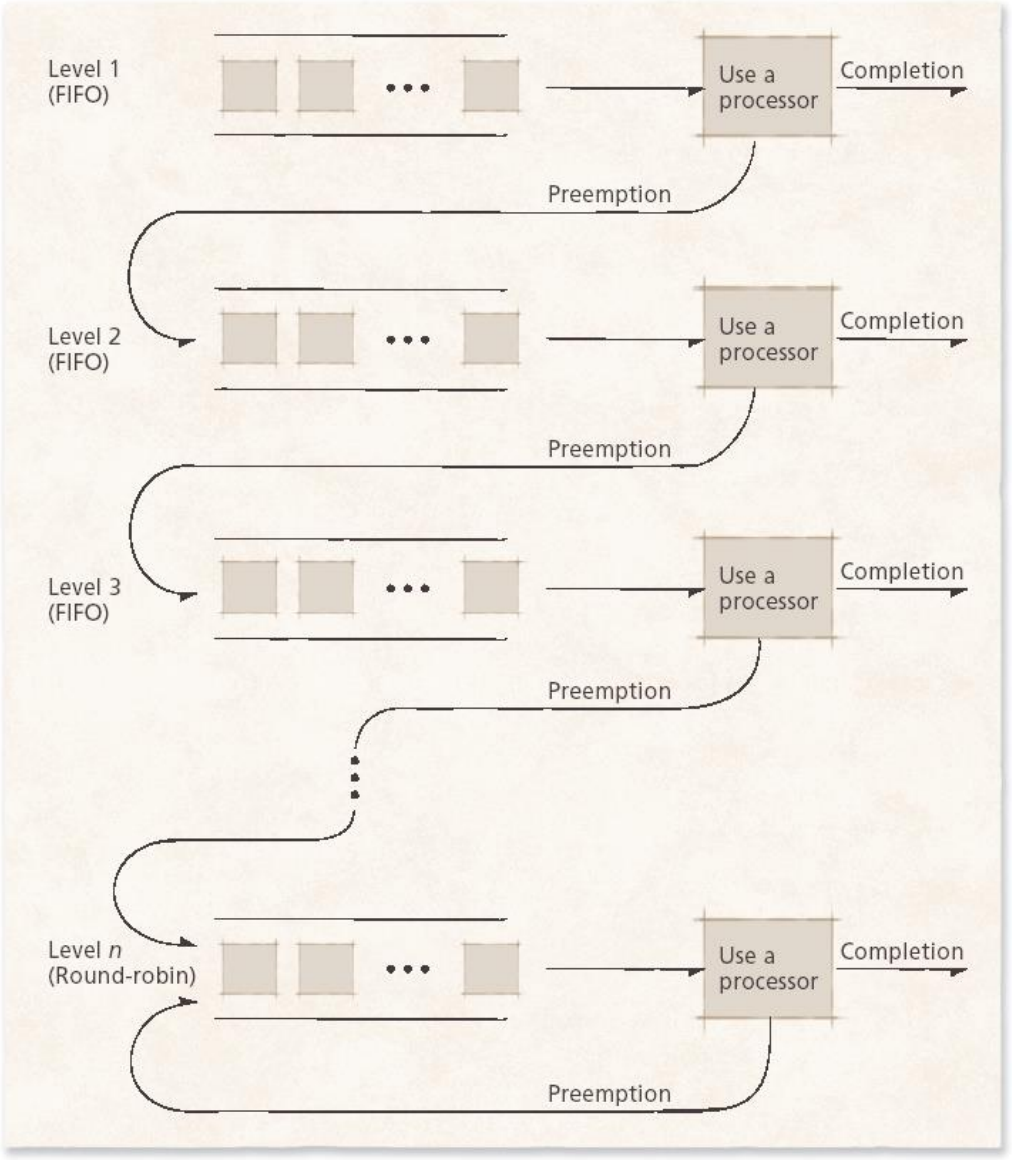
Multilevel Queue Scheduling



Multilevel Queue

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

Multilevel Feedback Queue



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- Multiple queues with different priorities. Each queue has its own scheduling policy.
- Job starts at high priority queue then if it's not finished moved to one level to lower priority and so on.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

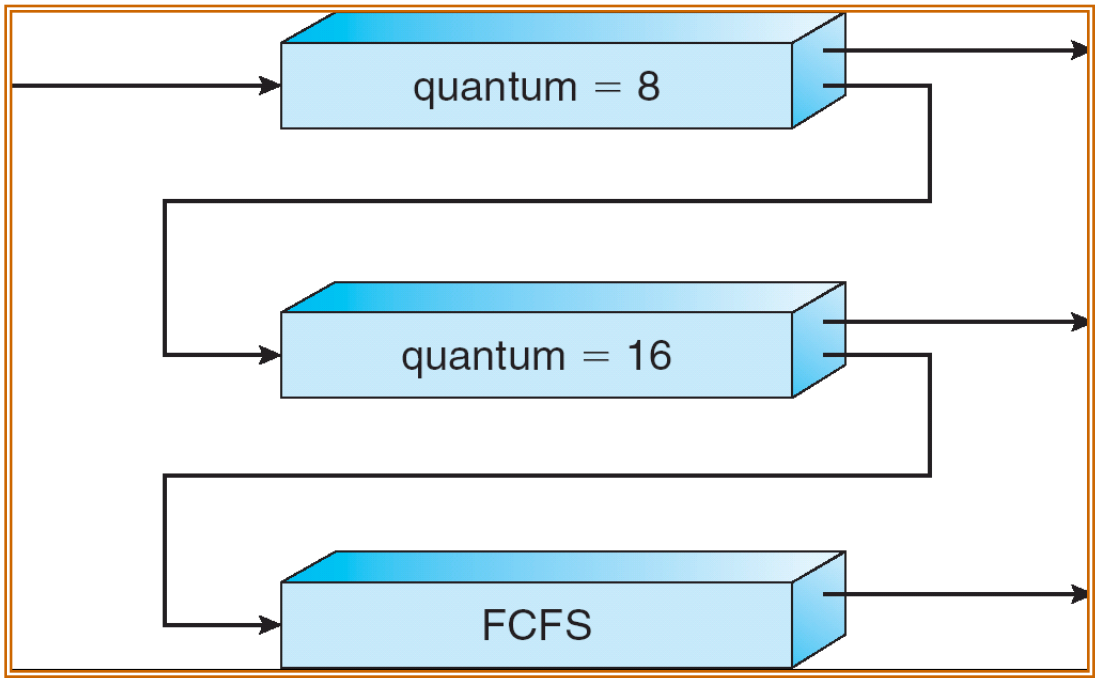
Multilevel Feedback Queue: Example

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

- Scheduling:
  - A new job enters queue Q0 which is served RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q1.

  - At Q1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q2and is served as FCFS.

Multilevel Feedback Queues: Example



Ex. *What questions are to be asked/answered by you if you are to design a CPU scheduling algorithm for a multilevel queueing System/multilevel feedback-queueing system?*

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
  - *Homogeneous processors* within a multiprocessor.
  - *Load sharing.*
  - *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.

Real-Time Scheduling

- *Hard real-time* systems: required to complete a critical task within a guaranteed amount of time.

- *Soft real-time* computing: requires that critical processes receive priority over less fortunate ones.

Thread Scheduling

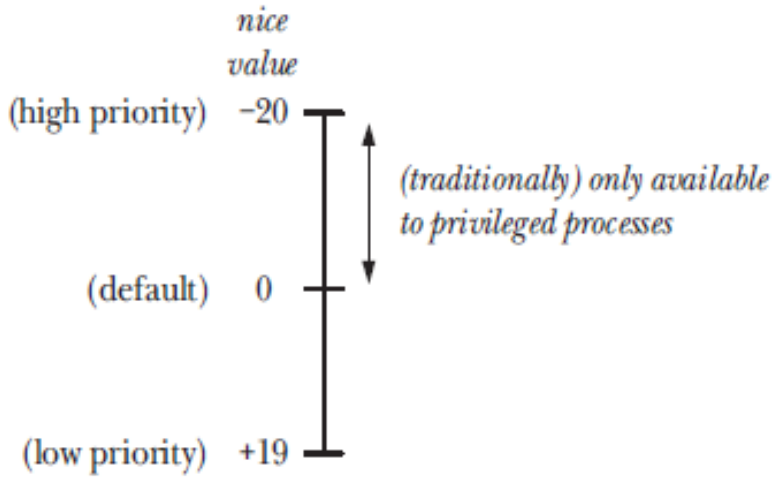
- Local Scheduling: How the threads library decides which thread to put onto an available LWP.
- Global Scheduling: How the kernel decides which kernel thread to run next.

Linux Scheduling

- The process scheduler is the component of a kernel that selects which process to run next. In other words, the process .
- The scheduler is the subsystem of the kernel that divides the finite resource of processor time among a system's processes.
- The behavior of the Linux scheduler with respect to a process depends on the process's scheduling policy, also called the scheduling class.
- A preprocessor macro from the header <sched.h> represents each policy: the macros are SCHED_FIFO, SCHED_RR, and SCHED_OTHER (the default policy).
  - Every process possesses a static priority, unrelated to the nice value. For normal applications, this priority is always 0. For the real-time processes, it ranges from 1 to 99, inclusive.
  - The Linux scheduler always selects the highest-priority process to run (i.e., the one with the largest numerical static priority value).
  - Because normal processes have a priority of 0, any real-time process that is runnable will always preempt a normal process and run.

Process Priorities (Nice Values)

- The nice value is a process attribute that allows a process to indirectly influence the kernel's scheduling algorithm.



- Each process has a nice value in the range –20 (high priority) to +19 (low priority); the default is 0.
- Rather than returning the actual nice value, the getpriority() system call service routine returns a number in the range 1 (low priority) to 40 (high priority), calculated according to the formula unice = 20 – knice.
- This is done to avoid having a negative return value from a system call service routine, which is used to indicate an error.

Process Priorities (Nice Values)

- Linux provides several system calls for retrieving and setting a process' nice value. The simplest is nice( ):

```
#include <unistd.h>
int nice (int inc);
```

  - A successful call to nice( ) increments a process' nice value by inc, and returns the newly updated value.
  - Only a process with the CAP_SYS_NICE capability (effectively, processes owned by root) may provide a negative value for inc, decreasing its nice value, and thereby increasing its priority.
  - Consequently, nonroot processes may only lower their priorities (by increasing their nice values).
  - The nice value is inherited by a child created via fork() and preserved across an exec().
  - Passing 0 for inc is an easy way to obtain the current nice value.
  - To set an absolute nice value , val, rather than a relative increment:

```
nice (val-nice(0));
```

Linux Realtime Scheduling

- A system is "real-time" if it is subject to operational deadlines: minimum and mandatory times between stimuli and responses.
  - hard real-time system requires absolute adherence to operational deadlines. Exceeding the deadlines constitutes failure, and is a major bug.
  - A soft real-time system, on the other hand, does not consider overrunning a deadline to be a critical failure.
- Latency refers to the period from the occurrence of the stimulus until the execution of the response.
  - If latency is less than or equal to the operational deadline, the system is operating correctly.
  - In many hard real-time systems, the operational deadline and the latency are equal—the system handles stimuli in fixed intervals, at exact times.
  - In soft real-time systems, the required response is less exact, and latency exhibits some amount of variance—the aim is simply for the response to occur within the deadline.
- Jitter refers to the variation in timing between successive responses.
  - Hard real-time systems often exhibit very low jitter because they respond to stimuli after—not within—an exact amount of time. Such systems aim for a jitter of zero, and a latency equal to the operational delay.

Linux Realtime Scheduling Policies

- The first in, first out (FIFO) class is a very simple real-time policy without timeslices. A FIFOclassed process will continue running so long as no higher-priority process becomes runnable. The FIFO class is represented by the macro SCHED_FIFO.
- The round-robin (RR) class is identical to the FIFO class, except that it imposes additional rules in the case of processes with the same priority. The macro SCHED_RR represents this class.
  - The scheduler assigns each RR-classed process a timeslice. When an RR-classed process exhausts its timeslice, the scheduler moves it to the end of the list of processes at its priority.
- SCHED_OTHER represents the standard round-robin time-sharing scheduling and is the default scheduling policy for nonreal-time class. All normal-classed processes have a static priority of 0.
  - Consequently, any runnable FIFO- or RR classed process will preempt a running normal-classed process.
  - The scheduler uses the nice value, discussed earlier, to prioritize processes within the normal class. The nice value has no bearing on the static priority, which remains 0.

Linux Realtime Scheduling

- A process employing the SCHED_RR policy maintains control of the CPU until either:
  - it reaches the end of its time slice;
  - it voluntarily relinquishes the CPU, either by performing a blocking system call or by calling the sched_yield() system call
  - it terminates; or
  - it is preempted by a higher-priority process. In both the SCHED_RR and the SCHED_FIFO policies, the currently running process may be preempted for one of the following reasons:
    - a higher-priority process that was blocked became unblocked (e.g., an I/O operation on which it was waiting completed);
    - the priority of another process was raised to a higher level than the currently running process; or
    - the priority of the currently running process was decreased to a lower value than that of some other runnable process.

Retrieving and modifying priorities

- The getpriority() and setpriority() system calls allow a process to retrieve and change its own nice value or that of another process.

  #include <sys/resource.h>

  int getpriority(int which, id_t who);

  - Returns (possibly negative) nice value of specified process on success, or –1 on error

  int setpriority(int which, id_t who, int prio);

  - Returns 0 on success, or –1 on error
  - The which argument determines how who is interpreted. This argument takes one of the following values:

PRIO_PROCESS	**Operate on the process whose process ID equals who. If who is 0, use the caller's process ID.**
PRIO_PGRP	Operate on all of the members of the process group whose process group ID equals who. If who is 0, use the caller's process group.
PRIO_USER	Operate on all processes whose real user ID equals who. If who is 0, use the caller's real user ID.

Realtime Process Scheduling API

- Realtime Priority Ranges

```
#include <sched.h>
int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

  - Both return nonnegative integer priority on success, or –1 on error
  - The sched_get_priority_min() and sched_get_priority_max() system calls return the available priority range for a scheduling policy.
  - On Linux, these system calls return the numbers 1 and 99, respectively, for both the SCHED_RR and SCHED_FIFO policies.

Realtime Process Scheduling API

- Modifying and Retrieving Policies and Priorities

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
```

  - Returns 0 on success, or –1 on error
  - If pid is 0 means the calling process.
  - The param argument is a pointer to a structure of the following form:

```
struct sched_param {
int sched_priority; /* Scheduling priority */
};
```

  - The policy argument determines the scheduling policy for the process.

Policy	Description
SCHED_FIFO	Realtime first-in first-out
SCHED_RR	Realtime round-robin
SCHED_OTHER	Standard round-robin time-sharing

  - A successful sched_setscheduler() call moves the process specified by pid to the back of the queue for its priority level.

Realtime Process Scheduling API

- Modifying and Retrieving Policies and Priorities

```
#include <sched.h>
int sched_setparam(pid_t pid, const struct sched_param *param);
```

  - Returns 0 on success, or –1 on error
  - The arguments are the same as for sched_setscheduler().
  - A successful sched_setparam() call moves the process specified by pid to the back of the queue for its priority level.

```
#include <sched.h>
int sched_getscheduler(pid_t pid);
```

  - Returns scheduling policy, or –1 on error

```
int sched_getparam(pid_t pid, struct sched_param *param);
```

  - Returns 0 on success, or –1 on error
  - The sched_getscheduler() and sched_getparam() system calls retrieve the scheduling policy and priority of a process.

Realtime Process Scheduling API

- Modifying and Retrieving Policies and Priorities

```
#include <sched.h>
int sched_rr_get_interval(pid_t pid, struct timespec *tp);
```

  - Returns 0 on success, or –1 on error
  - The sched_rr_get_interval() system call enables us to find out the length of the time slice allocated to a SCHED_RR process each time it is granted use of the CPU.
  - The time slice is returned in the timespec structure pointed to by tp:

```
struct timespec {
        time_t tv_sec; /* Seconds */
        long tv_nsec; /* Nanoseconds */
};
```

The Linux Scheduler

- Two priority ranges: time-sharing and real-time.
  - Real-time range from 0 to 99 and nice value from 100 to 140.
- Every processor has its own active and expired arrays.