Ch5 Practice Exercises

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

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

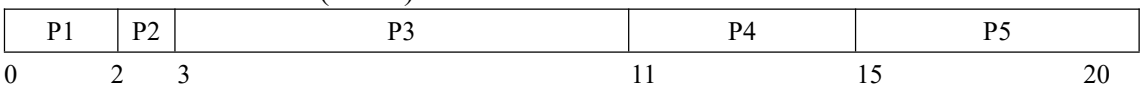
c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

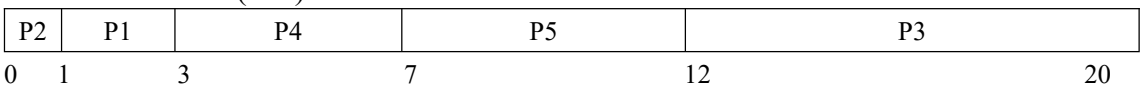
Answer:

a. Gantt Charts:

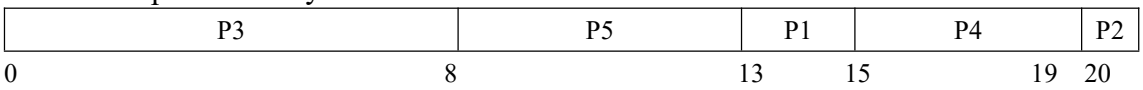
First Come First Served (FCFS):



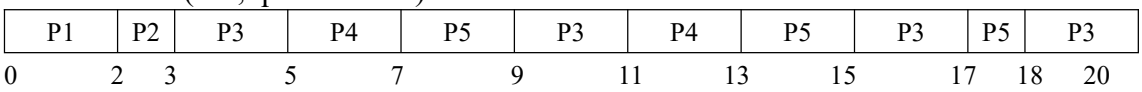
Shortest Job First (SJF):



Non-Preemptive Priority:



Round Robin (RR, quantum = 2):



b. Turnaround Time:

	FCFS	SJF	N-P Priority	RR
P1	2	3	15	2
P2	3	1	20	3
P3	11	20	8	20
P4	15	7	19	13
P5	20	12	13	18

c. Waiting Time:

	FCFS	SJF	N-P Priority	RR
P1	0	1	13	0
P2	2	0	19	2
P3	3	12	0	12
P4	11	3	15	9
P5	15	7	8	13

d. From the result of item c, SJF has the shortest waiting time.

5.5 The following processes are being scheduled using a preemptive, round-robin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an **idle task** (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.

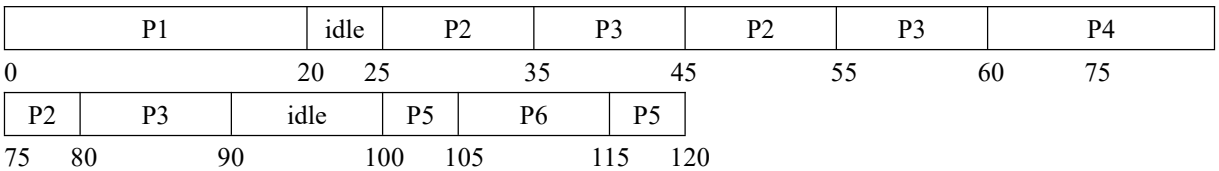
b. What is the turnaround time for each process?

c. What is the waiting time for each process?

d. What is the CPU utilization rate?

Answer:

a. Gantt Charts:



b. Turnaround Time:

Turnaround Time = Completion Time - Arrival Time:

P1: 20 - 0 = 20, P2: 80 - 25 = 55, P3: 90 - 30 = 60,

P4: 75 - 60 = 15, P5: 120 - 100 = 20, P6: 115 - 105 = 10.

c. Waiting Time:

Waiting Time = Completion Time - Burst Time - Arrival Time
= Turnaround Time - Burst Time:
P1: 20 - 20 = 0, P2 :55 - 25 = 30, P3: 60 - 25 = 35,
P4: 15 - 15 = 0, P5: 20 - 10 = 10, P6: 10 - 10 = 0.

5.10 The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

Priority = (recent CPU usage / 2) + base

where base = 60 and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process P_1 is 40, for process P_2 is 18, and for process P_3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

Answer:

The priorities assigned to the processes are:
Priority(P1) = 80, Priority(P2) = 69, Priority(P3) = 65.
The relative priority of a CPU-bound process is decreased as a result of this recalculation..

Ch5 Exercises

5.15 Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

a. $\alpha = 0$ and $\tau_0 = 100$ milliseconds

b. $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds

Answer:

Exponential Average Formula: $\tau(n + 1) = \alpha * \tau n + (1 - \alpha) * \tau n$

a. When $\alpha = 0$ and $\tau 0 = 100$ milliseconds, the formula always makes a prediction of 100 milliseconds for the next CPU burst.

b. When $\alpha = 0.99$ and $\tau 0 = 10$ milliseconds, the most recent behavior of the process is given much higher weight than the past history associated with the process. Consequently, the scheduling algorithm is almost memoryless, and simply predicts the length of the previous burst for the next quantum of CPU execution, which is τn.

Ch6 Exercises

6.17 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

Answer:

It Depends on how interrupts are implemented, but regardless of how, it is a poor choice of techniques.

1) Case 1 -- interrupts are disabled for ONE processor only -- result is that threads running on other processors could ignore the synchronization primitive and access the shared data.
2) Case 2 -- interrupts are disabled for ALL processors -- this means task dispatching, handling I/O completion, etc. is also disabled on ALL processors, so threads running on all CPUs can grind to a halt.

In conclusion:

Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

Ch6 Programming

6.33 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and will return them once finished. As an example, many commercial software packages provide a given number of ***licenses***, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such a request will be granted only when an existing license holder terminates the application and a license is returned.

The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the `decrease_count()` function:

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
  if (available_resources < count)
    return -1;
  else {
    available_resources -= count;

    return 0;
  }
}
```

When a process wants to return a number of resources, it calls the `increase_count()` function:

```
/* increase available_resources by count */
int increase_count(int count) {
  available_resources += count;

  return 0;
}
```

The preceding program segment produces a race condition. Do the following:

a. Identify the data involved in the race condition.

b. Identify the location (or locations) in the code where the race condition occurs.

c. Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the `decrease_count()` function so that the calling process is blocked until sufficient resources are available.

Answer:

a. Identify the data involved in the race condition:

The data involved in the race condition is the variable “int available_resources”.

b. Identify the locations in the code where the race condition occurs:

When commands “available_resources -= count” in the function “decrease_count()” and “available_resources += count” in the function “increase_count()” are executed at the same time, the race condition will occur.

c. Using a semaphore or mutex lock, fix the race condition.

Method 1:

```
int decrease_count (int count) {
    mutex.lock()
    if (available_resources < count) {
        mutex.unlock();
        return -1;
    }
    else{
        availabl_resources -= count;
        mutex.unlock();
        return 0;
    }
}
int increase_count (int count) {
    mutex.lock()
    if(available_resources + count <= MAX_RESOURCES) {
        available_resources += count;
        mutex_unlock();
        return 0;
    }
    else {
        mutex.unlock()
        return -1;
    }
}
```

Explanation:

1) Whenever one of these two functions starts to execute, the mutex will be locked by “mutex.lock()”, and the other function cannot execute until “mutex.unlock()”. Thus the shared memory “available_resources” will not be access at the same time, the shared memory is protected, and the race condition disappear.
2) I also edited the function “increase_count()” to let it check whether the summation of “available_resources” and “count” is greater than the “MAX_RESOURCES” before we increase the count.

Method 2:

```
int decrease_count (int count) {
    if (available_resources < count) {
       wait(synch);
   }
   available_resources -= count;
   return 0;
}
int increase_count (int count) {
   available resources += count;
   signal(synch);
   return 0;
}
```

Explanation:

1) The changes I made here could only avoid the specific situation of error that "available_resources - count < 0", and at this situation the function "decrease_count()" will only be executed after the function "increase_count()" is finished and send a signal of synch.
2) In this way, the race condition is fixed, however, the shared memory may not be protected.