

GENERAL INSTRUCTIONS

This homework has two parts: one part with a set of theory questions and another part, which is programming. Programming can be done in either C or Java. Submissions should include the **annotated** source code. Programs that do not compile will get a low grade. Make sure your programs do not crash when given bad input, but rather produce a useful warning or error message and take the appropriate action (recover or quit).

SUBMISSION INSTRUCTIONS

The compressed files (either tar or zip) will be submitted using email to the TA. Please make sure that you send me a **single compressed file with all the documents and code. The compressed file should not exceed 1MB.**

THEORY/WRITTEN QUESTIONS (40 points)

1. Consider a system having two resource types RT1 and RT2. Let there be two resources of type RT1 and let there be a single resource of type RT2. Let P1, P2, P3 and P4 be four processes competing for these resources.

In the following sequences of request events, let (P_i, RT_j) denote a request event indicating that process P_i requests a resource of type RT_j . Consider each sequence of request events and determine if it causes a deadlock in the system. Can you identify a sequence that does NOT cause a deadlock?

- a) $(P_4, RT_2), (P_3, RT_1), (P_2, RT_1), (P_4, RT_1), (P_2, RT_2), (P_3, RT_2)$
- b) $(P_4, RT_1), (P_1, RT_2), (P_3, RT_1), (P_2, RT_2), (P_3, RT_2), (P_1, RT_1)$
- c) $(P_3, RT_2), (P_2, RT_1), (P_4, RT_2), (P_3, RT_1), (P_2, RT_2), (P_1, RT_1)$
- d) $(P_2, RT_1), (P_4, RT_2), (P_3, RT_1), (P_2, RT_2), (P_3, RT_2), (P_4, RT_1)$

2. In order to prevent deadlock, let a system adopt the policy of forcing processes to request resources in ascending order of request type rank. In particular, when a process is requesting a resource of a certain type, the process cannot be holding other higher ranked resources.

Consider four resource types with the following profile in this system:

Resource Type	Rank	Number of Instances
RT1	4	2
RT2	7	4
RT3	8	3
RT4	10	2

Let the following sequence of resource requests be processed by the system (note that REQ(A, B, k) represents the request of k resources of type B by process A and REL(A, B, k) represents process A releasing k resources of type B):

- * REQ(P4, RT2, 3)
- * REQ(P3, RT3, 2)
- * REQ(P2, RT4, 1)
- * REL(P4, RT2, 3)
- * REQ(P4, RT1, 2)
- * REL(P2, RT4, 1)
- * REL(P3, RT3, 2)
- * REQ(P3, RT2, 3)
- * REQ(P2, RT3, 2)
- * REQ(P1, RT4, 2)

Note that all of the above request and release operations can be processed without any violation of the deadlock-prevention policy of the system.

For each of the following sequences of request/release operations, examine if it can or cannot be processed without violating the deadlock-prevention scheme above. Can you identify a sequence that is allowed by the scheme and consequently can be processed without leading to system deadlock?

- a) REL(P2, RT3, 2), REQ(P4, RT4, 2), REQ(P2, RT4, 2), REQ(P1, RT1, 2)
- b) REL(P2, RT3, 2), REQ(P4, RT2, 2), REQ(P3, RT3, 2), REQ(P2, RT2, 3)
- c) REQ(P2, RT4, 2), REL(P3, RT2, 3), REQ(P4, RT2, 2), REQ(P3, RT1, 2)
- d) REQ(P4, RT2, 3), REL(P2, RT3, 2), REQ(P2, RT4, 2), REQ(P3, RT1, 2)
- e) REQ(P3, RT3, 2), REL(P4, RT1, 2), REQ(P4, RT4, 2), REQ(P2, RT2, 3)

3. Consider a system with four resource types as follows:

Resource Type	Number of Instances
RT1	6
RT2	9
RT3	12
RT4	5

The following table presents the maximum resource needs for each process for each of the four resource types:

Process	Max RT1 Need	Max RT2 Need	Max RT3 Need	Max RT4 Need
P1	5	6	8	4
P2	3	7	4	3
P3	4	5	7	2

In the following sequence of resource requests, let REQ(A, N1, N2, N3, N4) represent a request from process A for N1 resources of type RT1, N2 resources of type RT2, N3 resources of type RT3 and N4 resources of type RT4:

- REQ(P1, 0, 1, 3, 0)
- REQ(P3, 3, 0, 3, 0)
- REQ(P1, 0, 1, 0, 1)
- REQ(P2, 0, 1, 0, 0)
- REQ(P3, 0, 3, 1, 0)
- REQ(P2, 1, 0, 0, 1)

The above sequence of resource requests can be granted safely without any danger of leading the system into a deadlock situation. Determine the set of resources available after processing the above sequence of requests and the remaining resource needs of the processes. Verify that a safe sequence of resource allocation to the processes does exist in this situation.

Now, consider the following additional requests for resources. For each request, determine if it can be granted safely in a manner that avoids deadlocks. Identify the request that can be safely granted.

- | | |
|------------------------|------------------------|
| a) REQ(P1, 1, 0, 2, 0) | d) REQ(P1, 0, 0, 2, 1) |
| b) REQ(P2, 0, 1, 0, 2) | e) REQ(P2, 0, 2, 1, 0) |
| c) REQ(P2, 0, 1, 3, 0) | f) REQ(P2, 0, 2, 2, 0) |

4. Consider a system with eight resources currently allocated as follows:

Resource	Allocated to Process
R1	P4
R2	P1
R3	P5
R4	P7
R5	P2
R6	P8
R7	P3
R8	P6

The following sequence of additional resource requests is then processed.
(Let REQ(A,B) denote process A's request for resource B.)

- * REQ(P4, R2)
- * REQ(P3, R6)
- * REQ(P2, R1)
- * REQ(P7, R7)
- * REQ(P6, R1)
- * REQ(P5, R7)

The above sequence of requests do not cause a deadlock. Verify this fact by constructing a resource-allocation graph involving R1 through R8 and P1 through P8.

For each of the following resource requests, add an arc to the resource-allocation graph and examine if the resource request causes a deadlock in the system. Identify the requests, among the following, that does NOT cause a deadlock.

- REQ(P1, R1)
- REQ(P1, R8)
- REQ(P1, R3)

PROGRAMMING (60 points)

This lab project addresses the implementation of CPU-scheduling algorithms in an operating system.

Each process in an operating system is managed by using a data structure called the Process Control Block (PCB). A PCB contains the process ID, arrival timestamp, total burst time, execution start time, execution end time, remaining burst time and the priority of the process in the system. The PCB class is defined as follows and is accessible from the rest of the code in this lab project:

```
public class PCB {
    public int processID;
    public long arrivalTimeStamp;
    public long totalBurstTime;
    public long executionStartTime;
    public long executionEndTime;
    public long remainingBurstTime;
    public int processPriority;
}
```

or for C:

```
struct PCB {
    int processID;
    long arrivalTimeStamp;
    long totalBurstTime;
    long executionStartTime;
    long executionEndTime;
    long remainingBurstTime;
    int processPriority;
    struct PCB *next;
}
```

The set of processes in the operating system that are ready to execute are maintained in a data structure called the Ready Queue. This data structure is an List or Array (you can select one of the two in your implementation) of PCBs of the processes that are waiting for the CPU to become available so that they can execute.

To determine the schedule of execution for the processes in an operating system, we consider three policies:

1. Priority-based Preemptive Scheduling (PP)
2. Shortest-Remaining-Time-Next Preemptive Scheduling (SRTP)
3. Round-Robin Scheduling (RR)

In order to implement the above policies, we need to develop methods that handle arrival of processes for execution and the completion of process execution. Whenever a process arrives for execution, it can either join the ready queue, and wait for its chance to execute or execute immediately (if there is no other process currently executing or if the currently-running process can be preempted). Whenever a process completes execution, another process from the ready queue is chosen to execute next, based on the scheduling policy. The details of these methods are described below in the specification and you need to develop code for these methods that follows the specification.

Hints:

Note that the following Java classes are relevant for this lab project: `java.util.ArrayList`. For C, linked lists or dynamic arrays are going to be useful.

handleProcessArrival_PP:

This function/method implements the logic to handle the arrival of a new process in a Priority-based Preemptive Scheduler. Specifically, it takes four inputs: 1) the ready queue (an `ArrayList` of PCB objects), 2) the PCB of the currently running process, 3) the PCB of the newly arriving process, and 4) the current timestamp. The method determines the process to execute next and returns its PCB.

If there is no currently-running process (i.e., the second argument is `NULL`), then the method returns the PCB of the newly-arriving process, indicating that it is the process to execute next. In this case, the PCB of the new process is modified so that the execution start time is set to the current timestamp, the execution end time is set to the sum of the current timestamp and the total burst time and the remaining burst time is set to the total burst time.

If there is a currently-running process, the method compares the priority of the newly-arriving process with the currently-running process. If the new process has lower priority (smaller integers for the priority field in the PCB indicate higher priority), then its PCB is simply added to the ready queue and the return value is the PCB of the currently-running process. As the newly-arriving process is added to the ready queue, its execution start time and execution end time are set to 0, and the remaining burst time is the same as its total burst time.

If the new process has higher priority, then the PCB of the currently-running process is added to the ready queue and the return value is the PCB of the new process. In this case, the PCB of the new process is modified so that the execution start time is set to the current timestamp, the execution end time is set to the sum of the current timestamp and the total burst time and the remaining burst time is set to the total burst time. Also, the PCB of the currently-running process is added to the ready queue after marking its execution start time and execution end time as 0, and adjusting its remaining burst time.

handleProcessCompletion_PP:

This function/method implements the logic to handle the completion of execution of a process in a Priority-based Preemptive Scheduler. Specifically, it takes two inputs: 1) the ready queue (an ArrayList of PCB objects) and 2) the current timestamp. The method determines the process to execute next and returns its PCB.

If the ready queue is empty, the method returns null, indicating that there is no process to execute. Otherwise, the method finds the PCB of the process in the ready queue with the highest priority (smaller integers for the priority field in the PCB mean higher priorities), removes this PCB from the ready queue and returns it. Before returning the PCB of the next process to execute, it is modified to set the execution start time as the current timestamp and the execution end time as the sum of the current timestamp and the remaining burst time.

handleProcessArrival_SRTP:

This function/method implements the logic to handle the arrival of a new process in a Shortest-Remaining-Time-Next Preemptive Scheduler. Specifically, it takes four inputs: 1) the ready queue (an ArrayList of PCB objects), 2) the PCB of the currently-running process, 3) the PCB of the newly-arriving process, and 4) the current timestamp. The method determines the process to execute next and returns its PCB.

If there is no currently-running process (i.e., the second argument is NULL), then the method returns the PCB of the newly-arriving process, indicating that it is the process to execute next. In this case, the PCB of the new process is modified so that the execution start time is set to the current timestamp, the execution end time is set to the sum of the current timestamp and the total burst time and the remaining burst time is set to the total burst time.

If there is a currently-running process, the method compares the remaining burst time of the currently-running process and the total burst time of the newly-arriving process. If the new process does not have a shorter burst time, then its PCB is simply added to the ready queue and the return value is the PCB of the currently running process. As the newly-arriving process is added to the ready queue, its execution start time and execution end time are set to 0, and the remaining burst time is set to the total burst time.

If the new process has a shorter burst time, then the PCB of the currently-running process is added to the ready queue and the return value is the PCB of the new process. In this case, the PCB of the new process is modified so that the execution start time is set to the current timestamp, the execution end time is set to the sum of the current timestamp and the total burst time and the remaining burst time is set to the total burst time. Also, the

PCB of the currently-running process is added to the ready queue, after marking its execution start time and execution end time as 0, and adjusting its remaining burst time.

handleProcessCompletion_SRTP:

This method implements the logic to handle the completion of execution of a process in a Shortest-Remaining-Time Preemptive Scheduler. Specifically, it takes two inputs: 1) the ready queue (an ArrayList of PCB objects) and 2) the current timestamp. The method determines the process to execute next and returns its PCB.

If the ready queue is empty, the method returns null, indicating that there is no process to execute next. Otherwise, the method finds the PCB of the process in the ready queue with the smallest remaining burst time, removes this PCB from the ready queue and returns it. Before returning the PCB of the next process to execute, it is modified to set the execution start time as the current timestamp and the execution end time as the sum of the current timestamp and the remaining burst time.

handleProcessArrival_RR:

This function/method implements the logic to handle the arrival of a new process in a Round-Robin Scheduler. Specifically, it takes five inputs: 1) the ready queue (an ArrayList of PCB objects), 2) the PCB of the currently-running process, 3) the PCB of the newly-arriving process, 4) the current timestamp, and 5) the time quantum. The method determines the process to execute next and returns its PCB.

If there is no currently-running process (i.e., the second argument is null), then the method returns the PCB of the newly-arriving process, indicating that it is the process to execute next. In this case, the PCB of the new process is modified so that the execution start time is set to the current timestamp, the execution end time is set to the sum of the current timestamp and the smaller of the time quantum and the total burst time. The remaining burst time is set to the total burst time.

If there is a currently-running process, the method simply adds the PCB of the newly-arriving process to the ready queue and the return value is the PCB of the currently running process. As the newly-arriving process is added to the ready queue, its execution start time and execution end time are set to 0, and the remaining burst time is set to the total burst time.

handleProcessCompletion_RR:

This function/method implements the logic to handle the completion of execution of a process in a Round-Robin Scheduler. Specifically, it takes three inputs: 1) the ready queue (an ArrayList of PCB objects), 2) the current timestamp, and 3) the time quantum. The method determines the process to execute next and returns its PCB.

If the ready queue is empty, the method returns null, indicating that there is no process to execute next. Otherwise, the method finds the PCB of the process in the ready queue with the earliest arrival time, removes this PCB from the ready queue and returns it. Before returning this PCB, it is modified to set the execution start time as the current timestamp and the execution end time as the sum of the current timestamp and the smaller of the time quantum and the remaining burst time.

handleEndOfTimeQuantum_RR:

This function/method implements the logic to handle the completion of a time quantum in a Round-Robin Scheduler. Specifically, it takes four inputs: 1) the ready queue (an ArrayList of PCB objects), 2) the PCB of the currently running process, 3) the current timestamp, and 4) the time quantum. The method determines the process to execute next and returns its PCB.

If there is a currently-running process (i.e., the second argument is not null), its PCB is added to the ready queue. Its execution start time and execution end time are set to 0 and its remaining burst time is reduced by the time quantum. The arrival time of its PCB is set to the current timestamp, as if the process is just entering the ready queue now for the first time. The method then finds the PCB of the process in the ready queue with the earliest arrival time, removes this PCB from the ready queue and returns it. Before returning this PCB, it is modified to set the execution start time as the current timestamp and the execution end time as the sum of the current timestamp and the smaller of the time quantum and the remaining burst time.

EXTRA CREDIT (10 points)

Suppose a system handles deadlocks by detecting them periodically and resolving them. In such a system, when a process requests a resource, it is granted to the process if the resource is free. Otherwise, an entry for the requesting process is added to the end of the wait queue for the resource.

A process whose request could not be granted is allowed to request other resources and proceed. However it cannot finish all its work until it obtains all the resources it needs/requests. When a process terminates (either after finishing all its work or aborting prematurely), it releases all its resources. When a process releases a resource, the resource is granted to the first process in the wait queue. The wait-queue is maintained on a First-Come First-Served (FCFS) basis.

On a periodic check for deadlocks, the system identifies all the processes that are currently deadlocked and selects a subset of them to abort (i.e., terminate prematurely) in order to resolve all the existing deadlocks.

Now, consider the following sequence of resource requests in the system (note that REQ(A, B) indicates a request by process A for resource B):

- * REQ(P3, R7)
- * REQ(P5, R5)
- * REQ(P4, R6)
- * REQ(P7, R3)
- * REQ(P2, R8)
- * REQ(P8, R2)
- * REQ(P1, R9)
- * REQ(P6, R4)
- * REQ(P9, R1)
- * REQ(P5, R6)
- * REQ(P4, R8)
- * REQ(P2, R2)
- * REQ(P6, R9)
- * REQ(P1, R8)
- * REQ(P3, R3)
- * REQ(P4, R1)
- * REQ(P7, R6)
- * REQ(P9, R3)
- * REQ(P3, R5)
- * REQ(P2, R4)
- * REQ(P8, R9)

At the end of processing the above sequence of requests, some of the processes in the system are deadlocked. Construct a wait-for graph for the sequence of requests above and identify the processes that are deadlocked. Note that each process in the system is a vertex in the wait-for graph. An arc $P_j \rightarrow P_i$ represents the fact that P_j is waiting for a request that P_i is holding. Cycles in the wait-for graph correspond to deadlocks in the system. Based on the resource and process dependency graph, determine which of pairs of processes can be aborted in order to resolve all the existing deadlocks in the system.