Term Project

- In this project, you will design and implement a program that simulates the job scheduling and CPU scheduling of an operating system. In addition to the scheduling algorithms, you must implement a deadlock avoidance method by implementing the Banker's Algorithm.

- You may work in teams of three; you must sign-up on Canvas

- You may use C/C++

- You will have time in class to work on the project

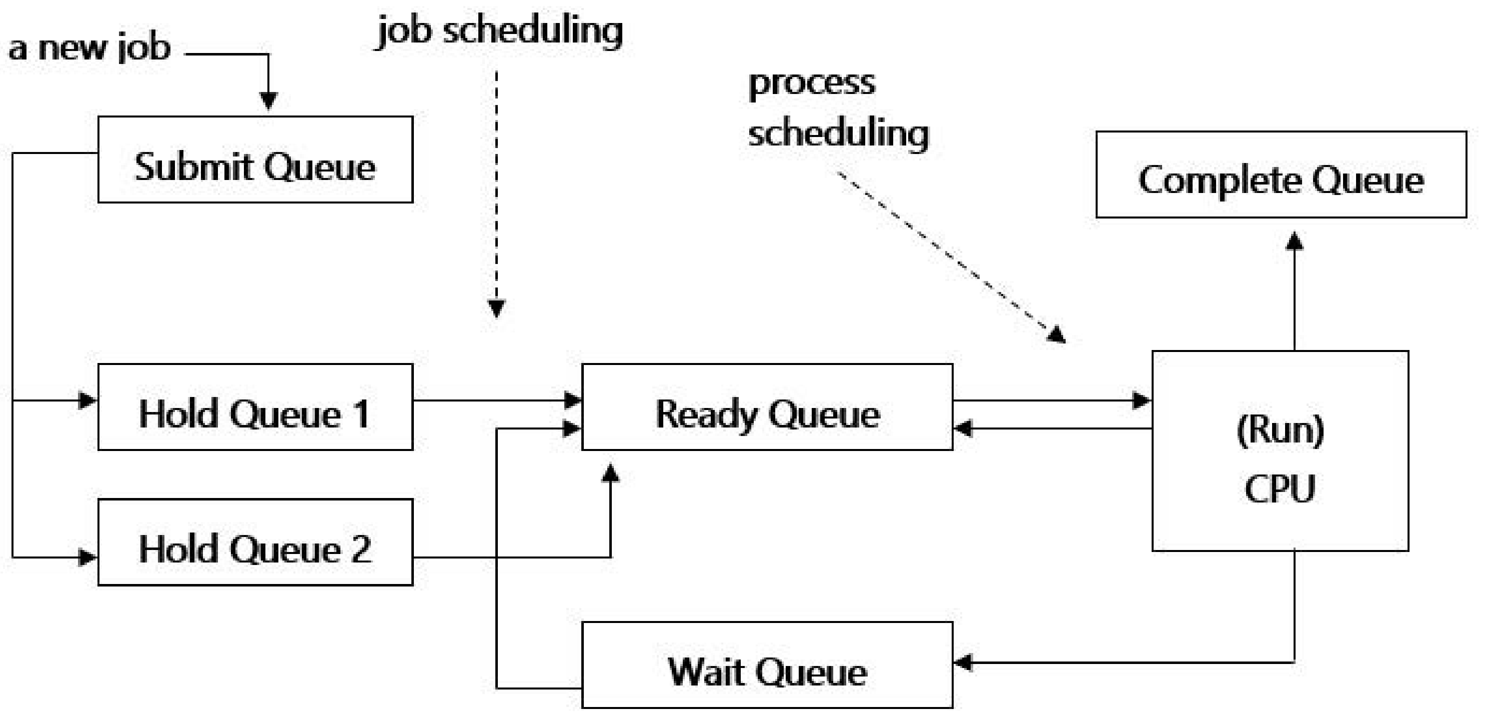
Term Project: Grading

- Source Code – 70%
- Report & Code Quality – 30%

- Describe your design and the output of your program using the sample input provided to you.

Term Project: Overview Diagram

A graphic view of the simulator



Term Project: Input

- Your program will need to handle five types of commands
  - System Configuration (C)
  - Job Arrival (A)
  - Request for Devices (Q)
  - Release for Devices (L)
  - Display State (D)

- Each command will have parameters: <cmd> <time> <param1> <param2> …
  - E.g., System Configuration
    - C 9 M=45 S=12 Q=1
    - The system should start at time ‘9’, have 45 units of memory, 12 serial devices, and the CPU should use a time quantum of 1

You do not need to worry about invalid input.

Term Project: Sample Input

- C 1 M=200 S=12 Q=4
- A 3 J=1 M=20 S=5 R=10 P=1
- A 4 J=2 M=30 S=2 R=12 P=2
- A 9 J=3 M=10 S=8 R=4 P=1
- Q 10 J=1 D=5
- A 13 J=4 M=20 S=4 R=11 P=2
- Q 14 J=3 D=2
- A 24 J=5 M=20 S=10 R=9 P=1
- A 25 J=6 M=20 S=4 R=12 P=2
- Q 30 J=4 D=4
- Q 31 J=5 D=7
- L 32 J=3 D=2
- D 9999

This will appear at the end of every file; in addition to print the state, provide turnaround time information.

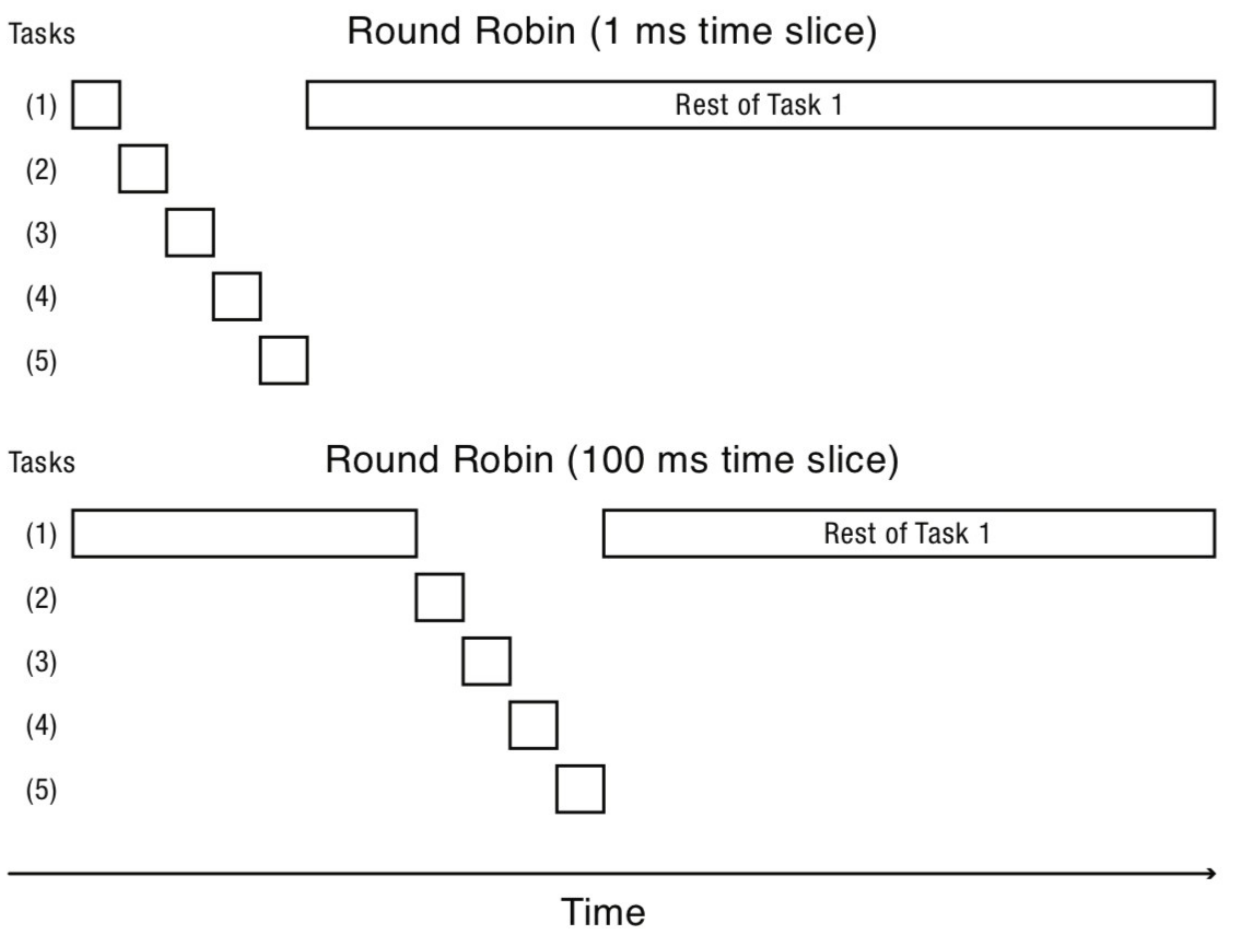
Term Project: Hold Queues

Assume that the two Hold Queues are based on priority. There are two external priorities: 1 and 2

- with 1 being the highest priority. Priority is only used for the Hold Queue.
- Job scheduling for Hold Queue 1 is Shortest Job First (SJF).
- Job scheduling for Hold Queues 2 is First In First Out (FIFO).
- Process scheduling will be Round Robin (FIFO).

Hint: Implement the Hold Queues as sorted linked lists.

Round Robin



Tasks
Round Robin (1 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Tasks
Round Robin (100 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. Number of available resources of each type
  - If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: n x m matrix. Defines the maximum **demand** of each process
  - If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. Defines the number of resources of each type currently allocated to each process
  - If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. Indicates the remaining resource need of each process
  - If $Need[i,j] = k$, then P_i may need k more instances of R_j to

$$Need\ [i,j] = Max[i,j] – Allocation\ [i,j]$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   ***Finish [i] = false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish [i] = false***

   (b) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish[i] = true
   go to step 2

4. If ***Finish [i] == true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise **error** condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must **wait**, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe ⇒ the resources are allocated to P_i
- If unsafe ⇒ P_i must **wait**, and the old resource-allocation state is **restored**

Banker's Algorithm for Multiple Resources

1. Look for a row, ***Need$_i$***, whose unmet resource needs are all smaller than or equal to ***Available***. If no such row exists, system will eventually deadlock.
2. Assume the process of row chosen requests all resources needed and finishes. Mark that process as terminated, add its resources to the ***Available*** vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated (**safe state**) or no process is left whose resource needs can be met (**deadlock**)

Example (Banker's Algorithm)

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5 instances), and C (7 instances)

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

How many resources do the processes need?

What is the state of the system (safe or unsafe)?

Can request for (1,0,2) by $\boldsymbol{P_1}$ be granted?

Based on the updated table, can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?

Based on the updated table, can request for (0,2,0) by $\boldsymbol{P_0}$ be

Example (Banker's Algorithm)

How many resources do the processes need?

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*ABC*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Example (Banker's Algorithm)

What is the state of the system (safe or unsafe)?

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*ABC*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

$< P_{1,} P_{3,} P_{4,} P_{2,} P_o >$ = Safe

Example (Banker's Algorithm)

- Can request for (1,0,2) by ***P*$_1$** be granted?

Available

	Allocation		*Max*	*Need*
	A B C	*A B C*	*ABC*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

$< P_1, P_3, P_4, P_o, P_2 >$ = Safe, Request Granted

Example (Banker's Algorithm)

- Based on the updated table, can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?

Available

	Allocation		*Max*	*Need*
	A B C	*A B C*	*ABC*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Resources are unavailable…

Example (Banker's Algorithm)

- Based on the updated (prior) table, can request for (0,2,0) by $\boldsymbol{P_0}$ be granted? If we pretend to grant the request…

Available

	Allocation		*Max*	*Need*
	A B C	*A B C*	*ABC*	*A B C*
P_0	0 3 0	7 5 3	7 2 3	2 1 0
P_1	3 0 2	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Resources are available… but resulting state is unsafe: requesting process will need to wait.