# Computer Operating Systems

## Midterm Exam Spring 2025

Department of Computer Science,

Hofstra University

# Q1. Multiple-choice. (20 pts)

1. What is a process in an operating system?
A) A static program stored on disk
B) An active entity with a program counter and stack
C) A thread of execution
D) A section of memory
Answer: B

2. What is dual-mode operation in an operating system?
A) Running two operating systems simultaneously
B) Providing two modes: kernel mode and user mode
C) Allowing two users to access the same process
D) Switching between two CPUs dynamically
Answer: B

3. What does "protection" in an operating system ensure?
A) That processes cannot interfere with each other or the OS itself
B) That all applications run in kernel mode for efficiency
C) That users have unrestricted access to hardware resources
D) That only one application can run at a time on the machine
Answer: A

4. Which of the following is NOT typically included in a process control block (PCB)?
A) Process ID
B) Program counter
C) Source code
D) Open file descriptors
Answer: C

5. What is the purpose of the fork() system call?
A) To create a new thread
B) To make a duplicate copy of the calling process
C) To execute a new program
D) To terminate a process
Answer: B

6. What is the purpose of the wait() system call?
A) Puts the calling process to sleep
B) Waits for any child process to terminate
C) Waits for any parent process to terminate
D) Waits for available CPU time
Answer: B

7. What is the main difference between a process and a thread?
A) Processes are in the kernel space, and threads are in the user space
B) Threads cannot have multiple instances, but processes can
C) Processes have their own address spaces, while threads can share an address space
D) Threads are managed by the kernel, while processes are managed by the user
Answer: C

8. What prevents starvation in the ticket lock implementation?
A) Random backoff
B) FIFO queue based on ticket numbers
C) Priority inheritance
D) Timeout mechanisms
Answer: B

9. What ensures fairness in ticket locks?
A) Test-and-Set instruction
B) Fetch-and-Add atomic operation
C) Compare-and-Swap
D) Disabling interrupts
Answer: B

10. What happens when sem_wait() is called on a semaphore with value 1?
A) Decrements the value to 0 and does not block
B) Increments the value to 2 and does not block
C) Blocks until sem_post() is called
D) It does not modify the value and does not block
Answer: A

# Q1. Multiple-choice. (20 pts)

11. Which of the following is NOT a necessary condition for deadlock?
A) Mutual exclusion
B) Hold and wait
C) No preemption
D) Fair Scheduling
Answer: D

12. In a Resource-Allocation Graph (RAG), a deadlock is certain if:
A) There is a cycle and each resource has multiple instances
B) There is no cycle
C) There is a cycle and all resources have single instances
D) A thread requests two resources simultaneously
Answer: C

13. In the context of the Dining Philosophers problem, which solution can prevent deadlock?
A) Allowing each philosopher to pick up forks in any order
B) Requiring each philosopher to pick up both forks simultaneously in one atomic operation
C) Requiring each philosopher to pick up his left fork before his right fork
D) Removing all forks from the table
Answer: B

14. In the context of the two-armed lawyers problem, where there is a pile of chopsticks at the center of the table, wher which solution can prevent deadlock?
A) Allowing each lawyer to pick up chopsticks in any order
B) Requiring each lawyer to pick up both chopsticks simultaneously in one atomic operation
C) Requiring each lawyer to pick up his left chopstick before his right chopstick
D) Removing all chopsticks from the table
Answer: B

15. In the producer-consumer problem, why must the mutex be acquired after sem_wait(emptySlots)?
A) To prevent buffer overflow
B) To avoid deadlock
C) To ensure proper scheduling
D) To maintain thread priority
Answer: B

16. What is the main difference between spinlocks and semaphores?
A) Spinlocks use busy waiting, while semaphores allow threads to sleep
B) Spinlocks are faster in all scenarios
C) Semaphores can only be used for mutual exclusion
D) Spinlocks can only be used on single-core systems
Answer: A

17. What is the purpose of using a "while" loop instead of an "if" statement when checking a condition variable in a monitor?
A) To improve performance
B) To handle spurious wakeups
C) To reduce code complexity
D) To allow more threads to enter the critical section
Answer: B

18. Only in preemptive scheduling (not in non-preemptive scheduling), a process can transition directly from:
A) Running → Waiting
B) Running → Ready
C) Ready → Terminated
D) Waiting → Ready
Answer: B

19. Which of the following scheduling algorithm suffers from the Convoy effect, where short jobs are stuck behind long jobs?
A) SJF
B) SRTF
C) RR
D) FP
Answer: A

20. In Round Robin scheduling, if there are 10 jobs in the ready queue and time quantum=10ms, what's the maximum wait time for any job?
A) 40ms
B) 80ms
C) 90ms
D) 100ms
Answer: C

## Q1 Multiple-choice questions: enter your answer keys here

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| B | B | A | C | B | B | C | B | B | A |
| **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** |
| D | C | B | B | B | A | B | B | A | C |

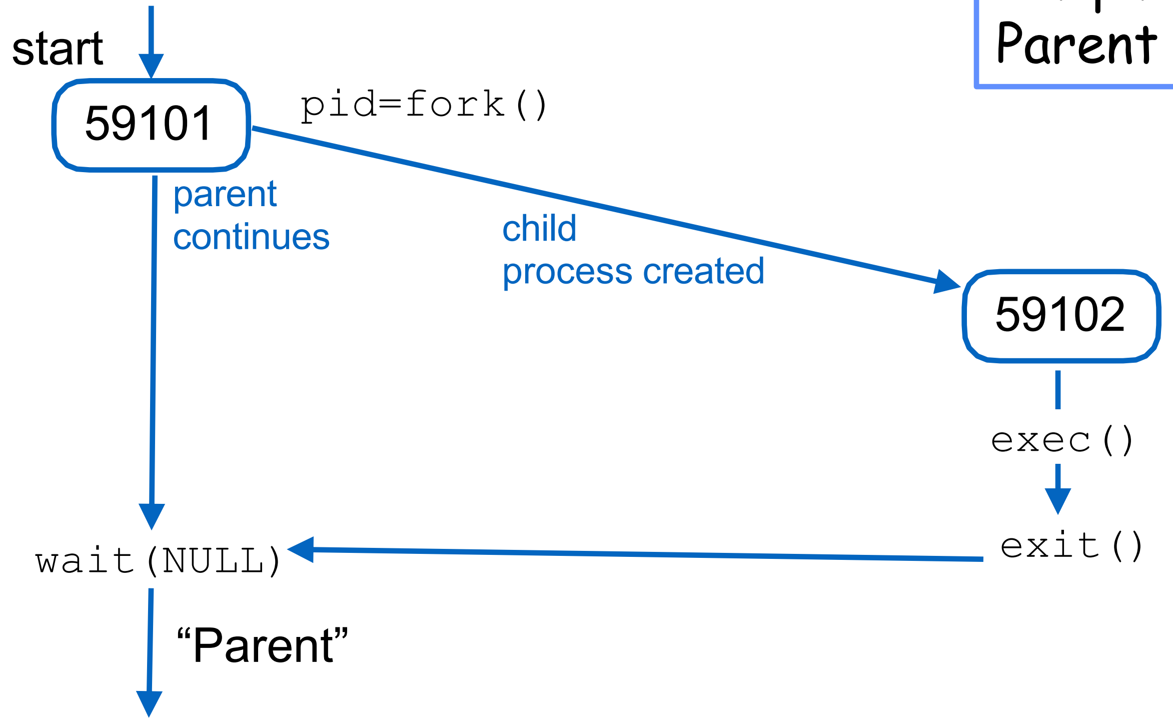
## Q2 Processes and Threads (20 pts)

- For these questions, assume there is no error, i.e., the return value of fork() is never negative. Assume round-robin scheduling algorithm like in Windows or Linux.
- What is the output of the program below? If there may be multiple possible outputs, list ALL possible outputs, and explain why.
- (You just need to provide the possible outputs and explain why. You do not need to draw the figures in the next slides to show the parent child relationships.)

# Q2 a) (5 pts)

```
int ret = fork();
if(ret == 0) {
  exec(SOME_COMMAND); //SOME_COMMAND is a Linux
command that does not print anything
  printf("Child\n");
}
else {
  wait(NULL);
  printf("Parent\n")
}
```

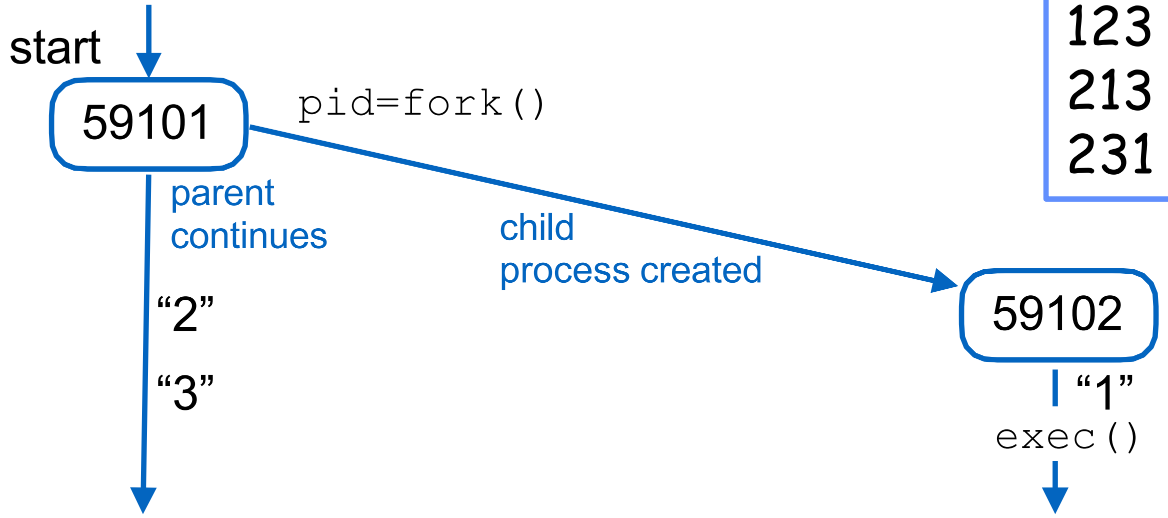
Output:
Parent



- ANS: Parent\n
- Code inside if(ret == 0){} block is executed by Child process; code inside if(ret > 0){} block is executed by Parent process; code outside of if-then-else blocks is executed by both Child and Parent
- In Child process: exec() replaces the current process image with a new program called SOME_COMMAND. The child process will execute the command and terminate. The code following it (e.g., printf("Child\n")) will not be executed because it is now running SOME_COMMAND, not the code shown in the text box.
- In Parent process: wait for the child to finish using wait(NULL) and then prints "Parent\n“
  - wait(NULL) waits for any child process to terminate, not a specific one. it returns PID of the terminated child process
  - It works as a “synchronization barrier” between two processes
- If non-preemptive scheduling, then output is still “Parent”

## Q2 b) (5 pts)

```
int ret = fork();
if(ret==0) {
  printf("1");
  exec(SOME_COMMAND); //SOME_COMMAND is a Linux
command that does not print anything
} else {
  printf("2");
}
printf("3");
```



Output:
123
213
231

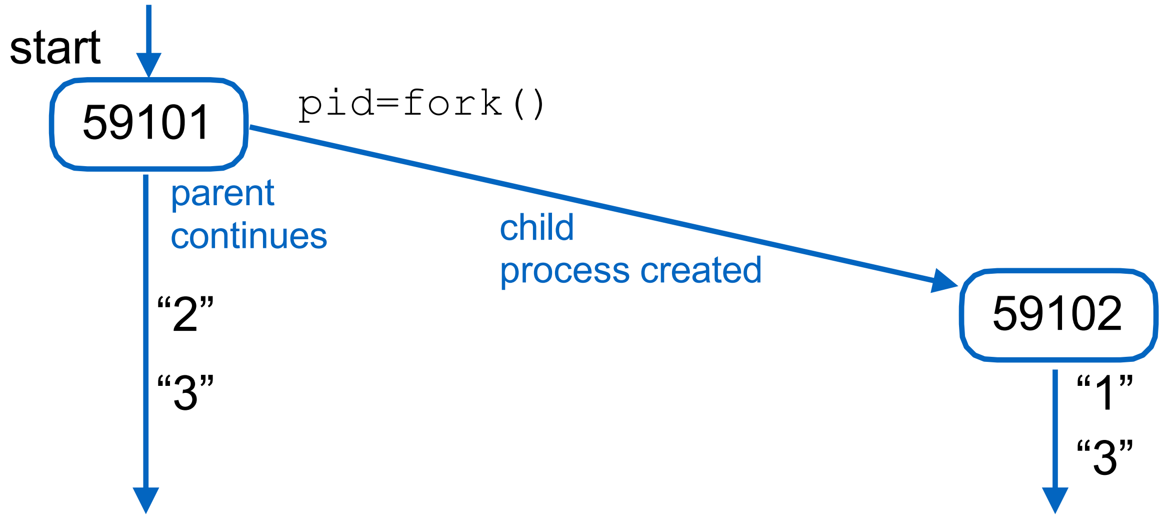
- ANS: 123 or 213 or 231
- In Child process: printf("1") → exec(SOME_COMMAND). printf("3") in child will not be executed
- In Parent process: printf("2") → printf("3"), output 23
- The end result is any interleaving of 23 and 1, which has 3 alternatives
- If non-preemptive scheduling, then output is 231, since Parent runs to completion before Child can start to run

## Q2 c) (5 pts)

```
int i;
int ret = fork();
if(ret==0) {
  printf("1");
} else {
  printf("2");
}
printf("3");
```

```
Output:
1323
2313
1233
2133
```

- ANS: 1323 or 2313 or 1233 or 2133
- In Child process: printf("1") → printf("3"), output 13
- In Parent process: printf("2") → printf("3"), output 23
- The end result is any interleaving of 23 and 13, which has 4 alternatives
- If non-preemptive scheduling, then output is 2313, since Parent runs to completion before Child can start to run

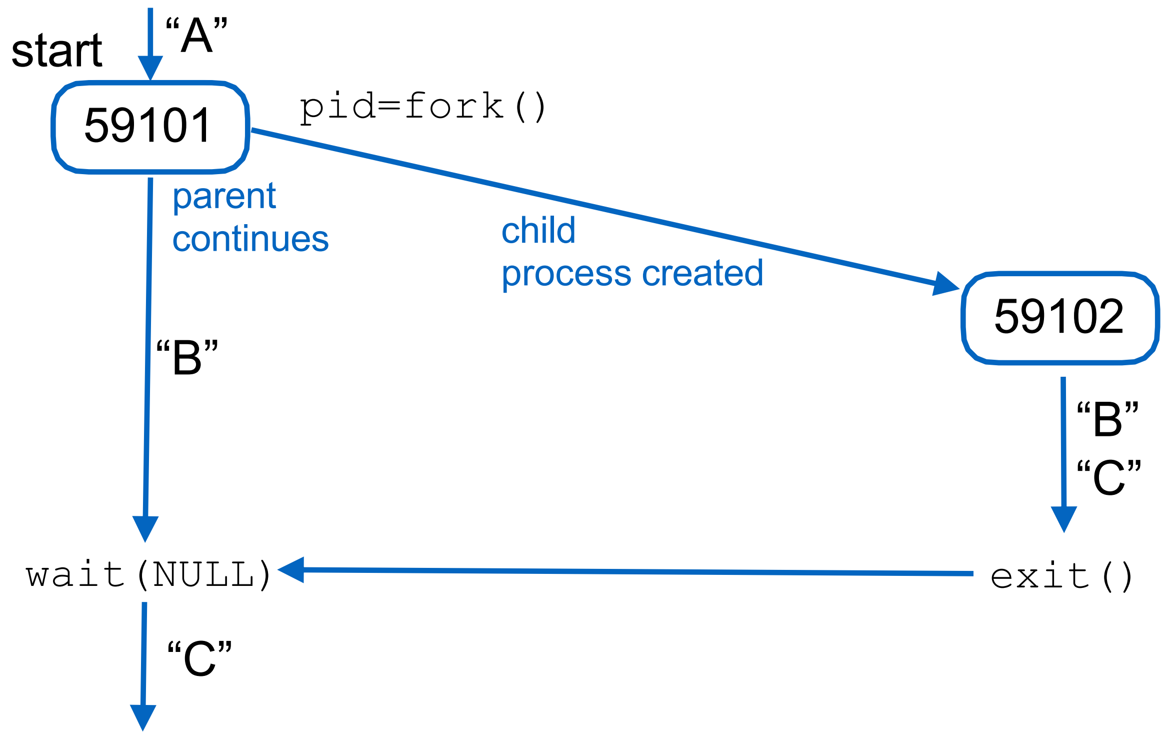


# Q2 d) (5 pts)

```
int main() {
  printf("A");
  int pid = fork();
  printf("B");
  if (pid>0) wait(NULL);
  printf("C");
}
```

Output:
ABBCC
ABCBC

- ANS: ABBCC or ABCBC
- Parent prints A
- fork() is called. Concurrently:
  - Parent prints B, then waits for child to finish
  - Child prints B, then prints C
- Parent continues after child finishes execution, and prints C.
- Output sequence can be either ABBCC and ABCBC
  - Initially parent prints A first
  - Parent's printing of B and child's printing of B, C can interleave in arbitrary order, which has 3 alternatives, but only with two possible outputs BBC or BCB, since the output is the same if Parent prints B before Child prints B then C, or if Child prints B before Parent prints B, then Child prints C
  - The parent's wait(NULL) ensures that parent prints C last.
- If non-preemptive scheduling, then output is ABBCC, since Parent runs, prints AB until wait(NULL); then Child starts to run and prints BC; then Parent continues to run and prints C after Child exits.
  - When Parent calls fork(), it does not yield to Child due to non-preemptive scheduling.
  - When Parent calls wait(NULL), it blocks and yields to Child. This is not pre-emption since Parent gives up the CPU voluntarily.



## Q3 Synchronization (10 pts)

- b) (5 pts) Consider the following concurrent program, where three threads access a shared variable x within critical sections protected by mutex locks. What are the possible final values of x after all threads finish execution? Explain why.

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
int x=0; //x is a global shared variable

//Thread T1:
pthread_mutex_lock(&mutex);
x = x + 1;
pthread_mutex_unlock(&mutex);

//Thread T2:
pthread_mutex_lock(&mutex);
x = x - 1;
pthread_mutex_unlock(&mutex);

//Thread T3:
pthread_mutex_lock(&mutex);
x = x * 2;
pthread_mutex_unlock(&mutex);
```

## Q3 Synchronization b) ANS

- ANS: 0, 1, and -1
- With mutex protection, no update to x will be erased. We consider all possible interleavings of the three threads.
- Case 1: if T1 and T2 both run before T3, then x=0
- Case 2: if T1 and T2 both run after T3, then x=0
- Case 3: if T1 before T3 before T2, then x=1
- Case 4: if T2 before T3 before T1, then x=-1

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
int x=0; //x is a global shared variable

//Thread T1:
pthread_mutex_lock(&mutex);
x = x + 1;
pthread_mutex_unlock(&mutex);

//Thread T2:
pthread_mutex_lock(&mutex);
x = x - 1;
pthread_mutex_unlock(&mutex);

//Thread T3:
pthread_mutex_lock(&mutex);
x = x * 2;
pthread_mutex_unlock(&mutex);
```

## Q3 Synchronization (10 pts)

- a) (5 pts) Consider the following concurrent program, where three threads access a shared variable x without mutex locks. What are the possible final values of x after all threads finish execution? Explain why.

```
int x=0; //x is a global shared variable

//Thread T1:
x = x + 1;

//Thread T2:
x = x - 1;

//Thread T3:
x = x * 2;
```

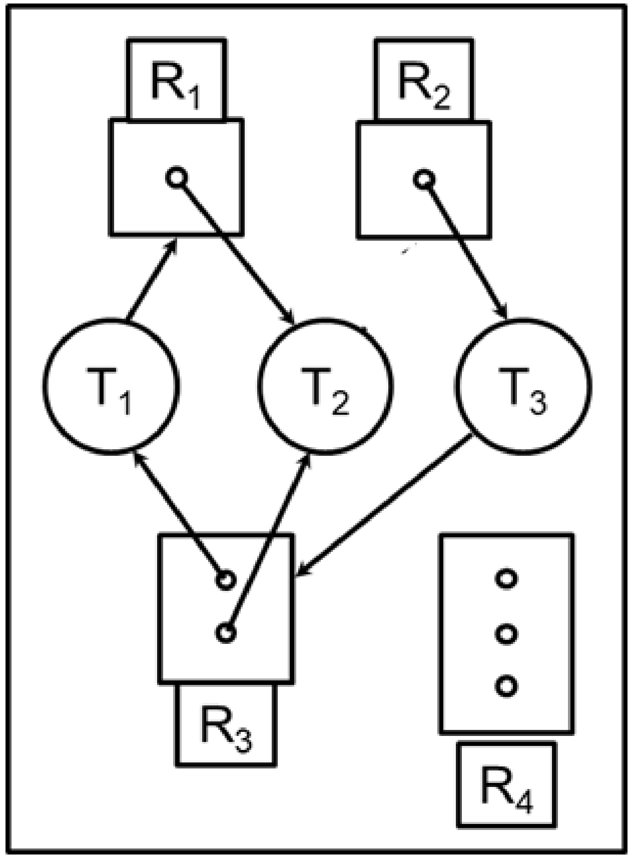
## Q3 Synchronization a) ANS

- ANS: -2,-1,0,1, or 2.
- Each update x statement to x can be "erased" by "sneaking in between" the load and store of another update x statement. The x=x+1 statement can either do nothing (if erased) or increase x by 1. The x=x-1 statement can either do nothing (if erased) or decrease x by 1. The x=x*2 statement can either do nothing (if erased) or multiply x by 2.
- Case 1: none of the update statements are erased, so we have 3 possible outputs 0, 1, and -1 as in part b)
- Case 2: x=x+1 in T1 is erased, and T2 runs before T3, then x=-2. This is the minimum possible value of x.
- Case 3: x=x-1 in T2 is erased, and T1 runs before T3, then x=2. This is the maximum possible value of x.
- You may think of other cases, e.g., x=x*2 in T3 is erased, and T1, T2 run, then x=0; or x=x+1 in T1 and x=x-1 in T2 are both erased, then x=0, and so on. But it is not necessary to enumerate all these cases, as we already have the possible outputs 0, 1, and -1 from part b), so these cases do not result in any new values.

```
int x=0; //x is a global shared variable

//Thread T1:
x = x + 1;

//Thread T2:
x = x - 1;

//Thread T3:
x = x * 2;
```

## Q4 Deadlocks (20 pts) Morning Section

- Consider the following Resource Allocation Graph with 3 processes and 4 resource types. Number of small circles in the box of resource Rj indicates the number of instances of resource Rj. An arrow from process Ti to resource Rj indicates that Ti requests 1 instance of Rj; an arrow from resource Rj to process Ti indicates that Ti is holding 1 instance of Rj. Run Banker's algorithm to check if the current state is safe, by writing out the matrices Max, Allocation and Need, and vectors Total and Available. If yes, give a safe sequence of process completions and fill in the table with the sequence of process completions without deadlock, and available resources after the completion of each process.



# Q4 Deadlocks (20 pts) Morning Section ANS

Max

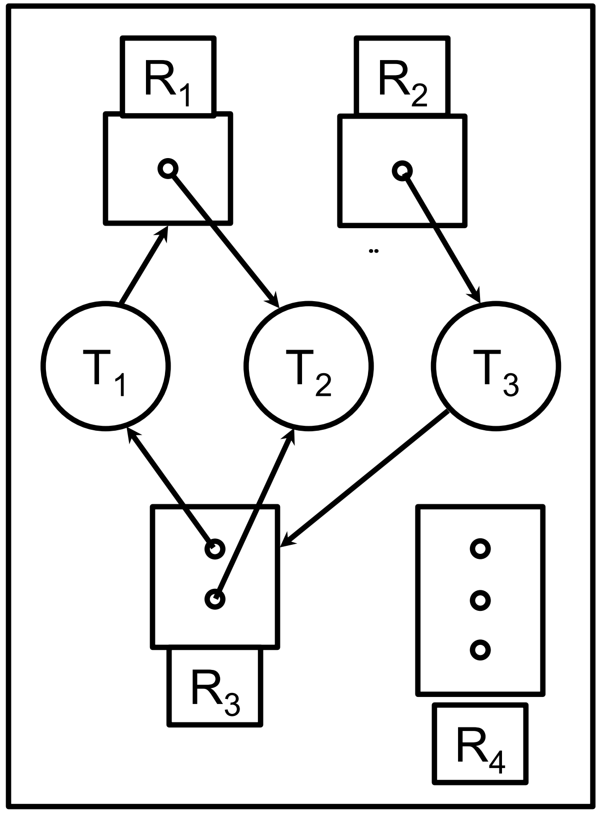
| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 1 | 0 | 1 | 0 |
| T2 | 1 | 0 | 1 | 0 |
| T3 | 0 | 1 | 1 | 0 |

(based on arrows from resource to process)
Allocation

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 0 | 0 | 1 | 0 |
| T2 | 1 | 0 | 1 | 0 |
| T3 | 0 | 1 | 0 | 0 |

(based on arrows from process to resource)
Need

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 1 | 0 | 0 | 0 |
| T2 | 0 | 0 | 0 | 0 |
| T3 | 0 | 0 | 1 | 0 |



Total

| R1 | R2 | R3 | R4 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |

Available

| R1 | R2 | R3 | R4 |
|---|---|---|---|
| 0 | 0 | 0 | 3 |

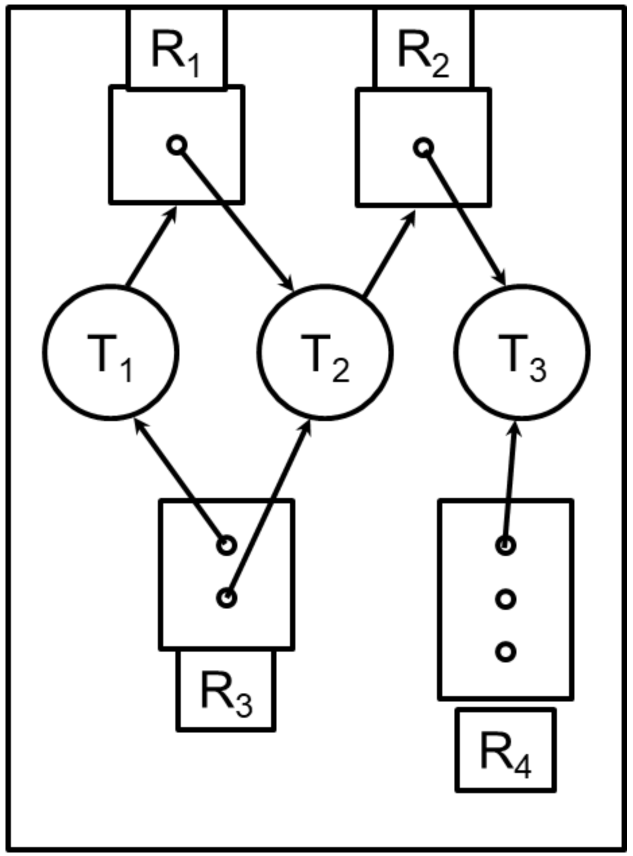
Available

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| | 0 | 0 | 0 | 3 |
| T2 | 1 | 0 | 1 | 3 |
| T3 | 1 | 1 | 1 | 3 |
| T1 | 1 | 1 | 2 | 3 |

Safe Sequence: T2, T3, T1
or T2, T1, T3

## Q4 Deadlocks (20 pts) Evening Section

- Consider the following Resource Allocation Graph with 3 processes and 4 resource types. Number of small circles in the box of resource Rj indicates the number of instances of resource Rj. An arrow from process Ti to resource Rj indicates that Ti requests 1 instance of Rj; an arrow from resource Rj to process Ti indicates that Ti is holding 1 instance of Rj. Run Banker's algorithm to check if the current state is safe, by writing out the matrices Max, Allocation and Need, and vectors Total and Available. If yes, give a safe sequence of process completions and fill in the table with the sequence of process completions without deadlock, and available resources after the completion of each process.



## Q4 Deadlocks (20 pts) Evening Section ANS

Max

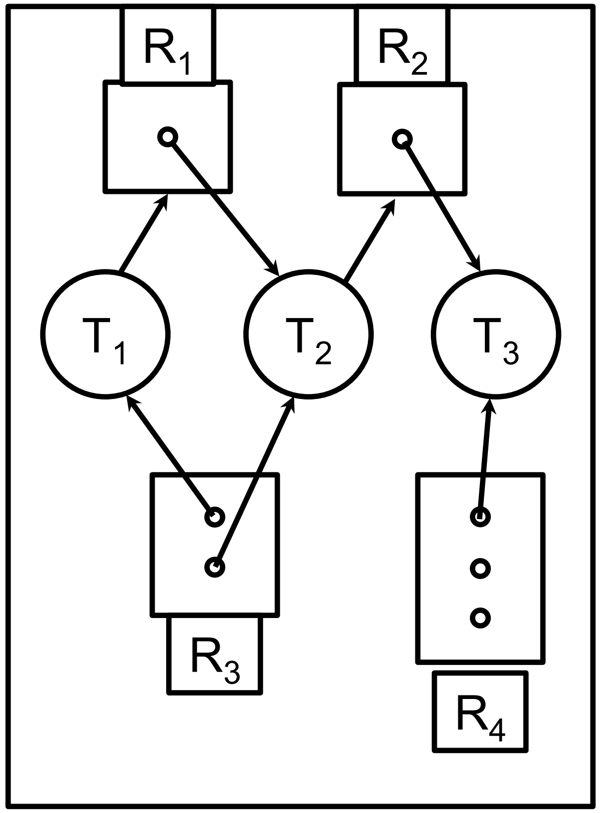
| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 1 | 0 | 1 | 0 |
| T2 | 1 | 1 | 1 | 0 |
| T3 | 0 | 1 | 0 | 1 |

Allocation

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 0 | 0 | 1 | 0 |
| T2 | 1 | 0 | 1 | 0 |
| T3 | 0 | 1 | 0 | 1 |

Need

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 1 | 0 | 0 | 0 |
| T2 | 0 | 1 | 0 | 0 |
| T3 | 0 | 0 | 0 | 0 |



Total

| R1 | R2 | R3 | R4 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |

Available

| R1 | R2 | R3 | R4 |
|---|---|---|---|
| 0 | 0 | 0 | 2 |

Available

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| | 0 | 0 | 0 | 2 |
| T3 | 0 | 1 | 0 | 3 |
| T2 | 1 | 1 | 1 | 3 |
| T1 | 1 | 1 | 2 | 3 |

Safe Sequence: T3, T2, T1

# Extra Exercise (Not in Exam)

Max

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 1 | 0 | 1 | 0 |
| T2 | 1 | 1 | 1 | 0 |
| T3 | 0 | 1 | 1 | 0 |

Allocation

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 0 | 0 | 1 | 0 |
| T2 | 1 | 0 | 1 | 0 |
| T3 | 0 | 1 | 0 | 0 |

Need

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| T1 | 1 | 0 | 0 | 0 |
| T2 | 0 | 1 | 0 | 0 |
| T3 | 0 | 0 | 1 | 0 |

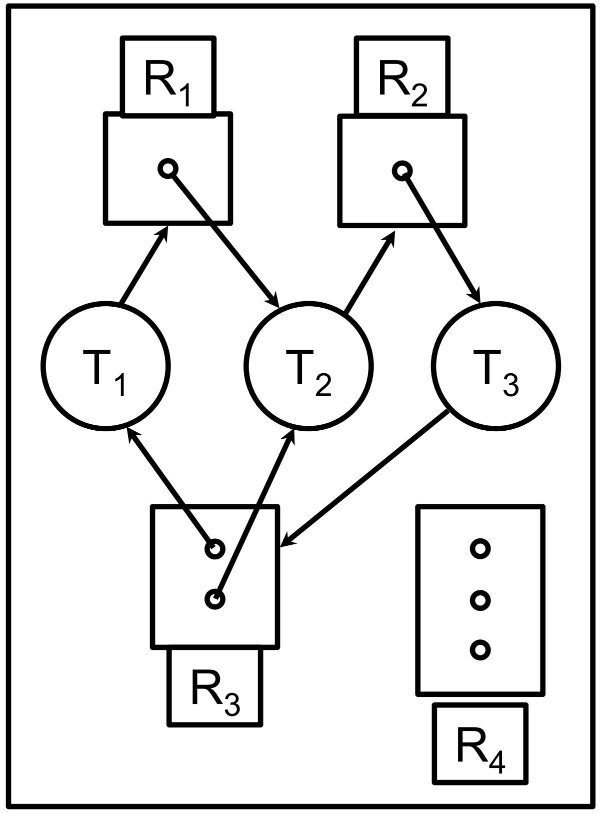
Total

| R1 | R2 | R3 | R4 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |

Available

| R1 | R2 | R3 | R4 |
|---|---|---|---|
| 0 | 0 | 0 | 3 |

Deadlock, no safe sequence



## Q5 Scheduling (30 pts)

- a) (10 pts) Consider the sequence of processes with CPU burst time in parentheses: P1(10ms), P2(2ms), P3(2ms) arriving at time 0 in the order of P1, P2, P3. Calculate the average response time under 1) First Come, First Served (FCFS). 2) Shortest Job First (SJF). 3) Shortest Remaining Time First (SRTF). 4) Round-Robin (RR) with time quantum 2. 5) Fixed-Priority scheduling with the priority ordering P3>P2>P1. (There is no need to draw the Gantt chart, but please show the response time of each process R1, R2, R3 and calculate R=(R1+R2+R3)/3.)
- ANS:
- FCFS: P1->P2->P3, R=(10+12+14)/3=12
- SJF: P2->P3->P1 or P3->P2->P1, R=(14+2+4)/3=6.7
- SRTF: P2->P3->P1 or P3->P2->P1, R=(14+2+4)/3=6.7
- RR: P1(2)->P2(2)->P3(2)->P1(8): R=(14+4+6)/3=8
- FP: P3->P2->P1, R=(14+4+2)/3=6.7

## Q5 Scheduling (30 pts) Morning Section

- b) (20 pts) Consider the set of 2 processes whose arrival time and CPU/IO burst times are given below. For each scheduling algorithm (FCFS, SJF, SRTF, RR, Fixed-Priority (FP)), draw the Gantt chart by filling in the table with the PID that runs in each time slot, and calculate the response time for each process, and the average response time. For RR scheduling, the time quantum is 1. For FP scheduling, assign P2 (PID 2) higher priority than P1 (PID 1). If a time slot is idle with no active process executing, then fill in X. (Except for any possible idle time slots at the end of schedule, leave them empty and do not fill in X.)

| PID | Arriv. time | CPU Burst | IO Burst | CPU Burst | FCFS Resp. Time | SJF Resp. Time | SRTF Resp. Time | RR Resp. Time | FP Resp. Time |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 3 | 2 | 4 | | | | | |
| 2 | 1 | 1 | 2 | 2 | | | | | |
| | | | | | Avg RT | Avg RT | Avg RT | Avg RT | Avg RT |

# Q5 Scheduling (30 pts) Morning Section ANS

| PID | Arriv. time | CPU Burst | IO Burst | CPU Burst | FCFS Resp. Time | SJF Resp. Time | SRTF Resp. Time | RR Resp. Time | FP Resp. Time |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 3 | 2 | 4 | 9 | 9 | 10 | 10 | 10 |
| 2 | 1 | 1 | 2 | 2 | 10 | 10 | 5 | 5 | 5 |
| | | | | | Avg RT 9.5 | Avg RT 9.5 | Avg RT 7.5 | Avg RT 7.5 | Avg RT 7.5 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FCFS | 1 | 1 | 1 | 2 | X | 1 | 1 | 1 | 1 | 2 | 2 | |
| SJF | 1 | 1 | 1 | 2 | X | 1 | 1 | 1 | 1 | 2 | 2 | |
| SRTF | 1 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | | |
| RR | 1 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | | |
| FP | 1 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | | |

Time 0 1 2 3 4 5 6 7 8 9 10 11 12

P2 arrival

P1 arrival

Gantt Chart

With SRTF, P2 runs whenever it is ready (at times 1 and 4) since it has shorter remaining time than P1; With FP, P2 runs whenever it is ready (at times 1 and 4) since it has higher priority than P1; with FCFS and SJF, P2 runs when P1 has finished each of its CPU burst due to non-preemptive scheduling

# Q5 Scheduling (30 pts) Evening Section

- b) (20 pts) Consider the set of 2 processes whose arrival time and CPU/IO burst times are given below. For each scheduling algorithm (FCFS, SJF, SRTF, RR, Fixed-Priority (FP)), draw the Gantt chart by filling in the table with the PID that runs in each time slot, and calculate the response time for each process, and the average response time. For RR scheduling, the time quantum is 1. For FP scheduling, assign P2 (PID 2) higher priority than P1 (PID 1). If a time slot is idle with no active process executing, then fill in X. (Except for any possible idle time slots at the end of schedule, leave them empty and do not fill in X.)

| PID | Arriv. time | CPU Burst | IO Burst | CPU Burst | FCFS Resp. Time | SJF Resp. Time | SRTF Resp. Time | RR Resp. Time | FP Resp. Time |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 3 | 2 | 4 | | | | | |
| 2 | 1 | 1 | 3 | 3 | | | | | |
| | | | | | Avg RT | Avg RT | Avg RT | Avg RT | Avg RT |

# Q5 Scheduling (30 pts) Evening Section ANS

| PID | Arriv. time | CPU Burst | IO Burst | CPU Burst | FCFS Resp. Time | SJF Resp. Time | SRTF Resp. Time | RR Resp. Time | FP Resp. Time |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 3 | 2 | 4 | 9 | 9 | 12 | 12 | 12 |
| 2 | 1 | 1 | 3 | 3 | 11 | 11 | 7 | 9 | 7 |
| | | | | | Avg RT 10 | Avg RT 10 | Avg RT 9.5 | Avg RT 10.5 | Avg RT 9.5 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FCFS | 1 | 1 | 1 | 2 | X | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| SJF | 1 | 1 | 1 | 2 | X | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| SRTF | 1 | 2 | 1 | 1 | X | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
| RR | 1 | 2 | 1 | 1 | X | 2 | 1 | 2 | 1 | 2 | 1 | 1 |
| FP | 1 | 2 | 1 | 1 | X | 2 | 2 | 2 | 1 | 1 | 1 | 1 |

Time 0 1 2 3 4 5 6 7 8 9 10 11 12

P2 arrival

P1 arrival

Gantt Chart

With SRTF, P2 runs whenever it is ready (at times 1 and 4) since it has shorter remaining time than P1; With FP, P2 runs whenever it is ready (at times 1 and 4) since it has higher priority than P1; with FCFS and SJF, P2 runs when P1 has finished each of its CPU burst due to non-preemptive scheduling