Midterm Questions

Multiple Choice

_______ provide(s) an interface to the services provided by an operating system.

- A. Shared memory
- B. System calls
- C. Simulators
- D. Communication

A race condition _______.

- A. results when several threads try to access the same data concurrently
- B. results when several threads try to access and modify the same data concurrently
- C. will result only if the outcome of execution does not depend on the order in which instructions are executed
- D. None of the above

A counting semaphore _______.

- A. is essentially an integer variable
- B. is accessed through only one standard operation
- C. can be modified simultaneously by multiple threads
- D. cannot be used to control access to a thread's critical sections

---

_______ occurs when a higher-priority process needs to access a data structure that is currently being accessed by a lower-priority process.

- A. Priority inversion
- B. Deadlock
- C. A race condition
- D. A critical section

Concept

> *One of the operating system goals is to manage system resources. Can you please name two hardware resources, and two software resources, respectively.*

- hardware: CPU, memory
- software: files, semaphores, locks

> *For a multi-threaded process, can you please name two items each thread within a process shares with others, and two items that are unique to each thread?*

- share: global variables, heap, files
- unique: stack, registers, PC

> *The CPU scheduling can be preemptive and non-preemptive, what is the main difference between them?*

- non-preemptive: A non-preemptive scheduling is evoked only when the current process running on the CPU gives up the CPU voluntarily either due to the termination of the process or the completion of its current CPU burst (for example waiting for I/O)
- preemptive: Otherwise, the scheduling is preemptive in nature.

> *Please briefly explain the three conditions that a solution to a Critical Section problem must hold.*

- mutual exclusion: there is at most one process that can be inside the critical section
- progress: one of the processes trying to enter will eventually get in
- bounded-waiting: a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

> *Suppose there are n instances of a resource in a system, each time one instance of resource can be acquired by a process and utilized exclusively. Explain how semaphores can be used to ensure the correct usage of the resource.*

- A counting semaphore `S` can be used, and is initialized to `n`

- A process executes `sem_wait(S)` before acquiring an instance of the resource, and executes `sem_post(S)` after finishing using it

- If all instances of the resource are used, subsequent request(s) calling `sem_wait(s)` will be blocked.

Consider the following code, what is the potential problem?

Let `S` and `Q` be two semaphores initialized to `1`.

P0	P1
wait(S);	wait(Q)
wait(Q);	wait(S);
…	…
Post(S);	Post(Q);
Post(Q);	Post(S);

- This can lead to a deadlock.
- Specifically, if P0 executes `wait(S)`, gets interrupted, and P1 executes `wait(Q)`, in that P0 and P1 each holds one semaphore waiting for another, this becomes a deadlock

---

Fork Counting

> *What is the total number of processes in the following code? Please elaborate.*

```
pid_t pid1, pid2;
pid1 = fork();
pid2 = fork();
if( pid1>0 && pid2==0){
    if(fork())
        fork();
}
```

There are 6 processes.

- After the first two `fork()`, there would be 4 processes.
- And only one process enters the `if` statements, within this part, `fork()` creates one more process, and the child process `fork()` creates will create another new process.

- Finally there are 6 processes

> *What is the total number of processes in the following code? Please elaborate.*

```
for(int i = 0; i < 5; i++){
    if(fork()){
        for(int j = 0; j < 3; j++){
            fork();
        }
    }
}
```

There are 9^5 processes.

- Any process entering the inner `for()` part will end up with 8 new processes.
- So after an outer `for()` loop, the number of processes will be multiplied by 9.
- And the total number of processes after 5 outer `for()` is 9^5

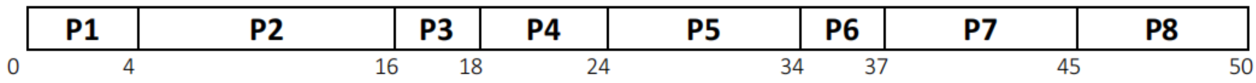
CPU Schedule

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

PROCESS	ARRIVAL TIME(MS)	BURST TIME(MS)
P1	0	4
P2	2	12
P3	5	2
P4	6	6
P5	8	10
P6	12	3
P7	15	8
P8	22	5

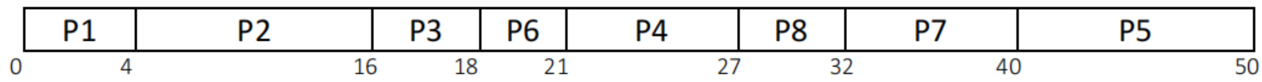
For each of the following scheduling algorithms, construct the *Gantt chart* depicting the sequence of process execution and calculate the ***average waiting time*** of each algorithm.

(a) FCFS



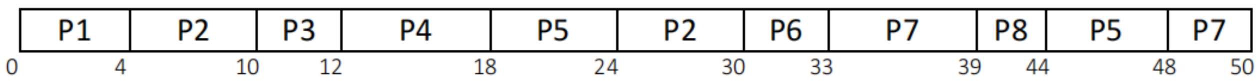
- Average waiting time = (0+2+11+12+16+22+22+23)/8 = 108/8 =13.5ms

(b) SJF non-preemptive



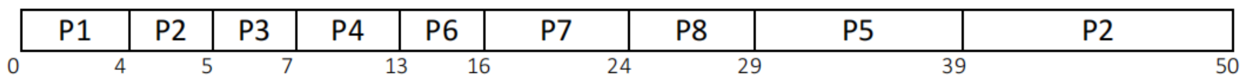
- Average waiting time = (0+2+11+15+32+6+17+5)/8 = 88/8 =11ms

(c) RR (quantum = 6 ms)



- Average waiting time = (0+16+5+6+30+18+27+17)/8 = 119/8 =14.875ms

(d) SJF preemptive



- Average waiting time = (0+36+0+1+21+1+1+2)/8 = 62/8 =7.75ms

---

Multi-Process Mutual Exclusion

The following code implements a solution to the critical section problem with n processes by using an atomic operation `test_and_set()`. Please illustrate how the three conditions of the solution are satisfied.

```
do{
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
```

```
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
}while(true);
```

Mutual Exclusion

- Initialize
    - The elements in the `waiting[n]` array (n processes) are initialized to `false`
    - The `lock` is initialized to `0`
- The first process `P[i]` executes the `test_and_set()` will find `key = false`
    - Since then, `lock` is modified to be `true` and the rest of the process will find `lock == true` an therefore they will return `key = true`, which will block them to the `while(waiting[i] && key)` cycle
- `P[i]` enters the critical section. Before then, it sets the `waiting[i] = false`. After finishes the critical section, it will perform the following two cases:
    - If there are other processes waiting for entering the critical section, `P[i]` will pick next process `p[j]` and set `waiting[j] = false` to let `P[j]` enter the critical section.
        - Then `p[j]` will execute the same process as `p[i]`
    - If there is no other process, `p[i]` will set `lock = false`
        - If later any other process comes, one of them will become the first to come and find the `lock = false` and hold the `key = true`.

Progress

- A process exiting the critical section either sets `key` to `false` or sets `waiting[j]` to `false`.
- Both allow a process that is waiting to enter its critical section to proceed

Bounded-waiting

- When a process leaves its critical section, it scans the array waiting in the cyclic ordering $(i+1, i+1, \ldots, n-1, 0, \ldots i-1)$.
- It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section
- Any process waiting to enter its critical section will thus do so within `n - 1` turns

---