Section 1: MCQ (2 marks each)

MCQ 1 and 2 based on the following code fragment:

```
int main( )
{
    int cid[9]= {0}; //init the entire array to zeros

    for (i = 0; i < 9; i++){
        cid[i] = fork();
        <Point Alpha>
        if ( <Condition> ) {
            ...some statements...
            exit(0);
        }
    }

    return 0;
}
```

1. In loop **`i = 3`**, how many zeroes are there in the **`cid[]`** array for the **child process** at **`<point alpha>`**? The **child process** refers to the process just created by the **`fork()`** system call in that loop interation.

   a. 6
   b. 7
   c. 8
   d. 9
   e. None of the above

Q1. ANS = (a). For the main process, cid[0..2] contains non-zero value just before the fork() in i = 3 loop iteration. So, the child process will receive a zero in cid[3] ➔ cid[3..8] are zeroes ➔ 6 zeros.

Testing: fork() behavior and memory duplication.
Difficulty: EASY
Stats: **a = 104**, b = 17, c = 10, d = 19, e = 24

2. Which of the following condition(s) can be used for <Condition> if we want to create a total of **9 additional processes (i.e. not counting the original process)**?

   i. `cid[i] == 0`
   ii. `cid[i] != 0`
   iii. `cid[0] == 0`

   a. (i) only.
   b. (i) and (ii) only.
   c. (ii) and (iii) only.
   d. (i), (ii) and (iii).
   e. None of the above.

Q2. ANS = (b). Note the "exit()" in the if() block.

i. **Correct.** Child process enters if() ➔ main process will survive and loop to create new child ➔ 9 loops ➔ 9 processes created.
ii. **Correct.** Similarly, parent process enters if() ➔ child process will survive to create more processes.
iii. **Wrong.** Only child from the i=0 loop enters if() ➔ from i=1 onwards, both parent and child will survive to create processes ➔ $2^8 + 1$ processes are created.

Testing: fork() behavior.
Difficulty: EASY
Stats: a = 54, **b = 96**, c = 10, d = 4, e = 10

3. Which of the following statment(s) regarding **zombie** process is **TRUE**?

   i. Zombie process takes up a slot in the OS PCB table.
   ii. Zombie process is created so that `wait()` system call can be implemented properly.
   iii. A user command running in the **background** under a shell interpreter can become a zombie process.

   a. (i) only.
   b. (i) and (ii) only.
   c. (ii) and (iii) only.
   d. (i), (ii) and (iii).
   e. None of the above.

Q3. ANS = (d).

i. **Correct**. Zombie process retains PID and stores termination status and result at its PCB table.
ii. **Correct.** As dicussed in lecture, without the zombie state, we wont be able to return information about terminated child process to the parent process.
iii. **Correct.** This is essentially similar to (ii). A **foreground** process is waited by the shell interpreter directly. A **background** process, on the other hand, is not

waited. That's how the interpreter can continue to accept user input. So, if the background process terminated, it will turned into a a zombie process.

Testing: wait() behavior.
Difficulty: MEDIUM (option iii needs further understanding gained from lab)
Stats: a = 22, b = 29, c = 15, **d = 100**, e = 8

4. Which of the following statment(s) regarding **Unix Shared Memory** IPC is **TRUE**?
   i. Shared memory region created by program **P** can stay around after **P** exited.
   ii. Shared memory region is identified by a pointer (memory address).
   iii. Shared memory region can be accessed by any process (including process from other users).

   a. (i) only.
   b. (i) and (iii) only.
   c. (ii) and (iii) only.
   d. (i), (ii) and (iii).
   e. None of the above.

Q4. ANS = (b).

i. **Correct**. Without an explicit delete command (shmctl(..IPC_RMID…)), the shared memory region stays around.

ii. **Wrong.** Demonstrated by the "master-slave" and lab 3 exercies, the shared memory region is identified by a **number** (id). A memory address is generated only when you attempt to **attach** to the identified region.

iii. **Correct.** Permission bits can be set to allow other processes to use a particular shared memory region. If you use the most permissive setting, any process can access. (e.g. a permission bits of 0666).

Testing: Shared memory mechanism and behavior
Difficulty: "EASY" but need self exploration on the IPC calls, e.g. read the given source code, attempted the lab exercises etc.
Stats: a = 17, **b = 31**, c = 19, d = 96, e = 11

Surprisingly, this is the ONLY question where the right answer did not receive majority. I realized that many mistook the return address of "shmat()" (attach function) as the identifier ☹.

Actually, all shared memory code we have shown so far print out the integer region id "Shared Memory Id = ***12345***" once the region is created successfully…

5. Given the following pseudo code:

Code A	Code B

`wait( S );`
`<do some work>`
`signal( S );`
	`<heavy computation>`

Which of the following setup can potentially cause **priority inversion**?

a. A high priority task running code B and a lower priority task running code A.
b. A high priority task running code A and a lower priority task running code B.
c. The highest and lowest priority tasks running code A and a middle priority task running task B.
d. The highest and lowest priority tasks running code B and a middle priority task running task A.
e. None of the above

Q5. ANS = (c).
Priority inversion:

- Assume priority based pre-emptive scheduling algorithm.
- A low priority task acquires a resource needed by high priority task.
- The low priority task is preempted by high priority task ➔ cannot release the resource ➔ blocks high priority task
- A medium priority task can execute despite the existence of a high priority task.

The code above construct a simple demonstration of this problem. If a high priority task **H** and a low priority task **L** run code A, then it is possible that the **L** enter the critical section but get pre-empted by **H** before finishing its work. So, **L** still "holds" the semaphore, preventing **H** from executing. At this time, if a medium task **M** enter the system to run code **B**, it will get picked over H despite having a lower priority.

Testing: Priority Inversion
Difficulty: HARD, requires understanding of the original problem and apply it in a new situation (semaphore).
Stats**:** a = 35, b = 40, **c = 78**, d = 6, e = 15

6. Ms. Raycond coded the following function:

```
int globalVar = 0;   //shared among all threads
void* doSum( void* arg)
{
    int i, localVar = 0;

    for (i = 0; i < 50000; i++){
         localVar++;
    }
    globalVar += localVar;
}
```

If we spawn **two threads** to work on the **doSum() function** and **wait for them to finish**, what is the **most accurate** description of the program behavior?

a. The program is now deterministic with the globalVar equal to 100000 for all runs.
b. The program still exhibits race condition. The globalVar value can be 0, 50000 or 100000.
c. The program still exhibits race condition. The globalVar value can be 50000 or 100000.
d. The program still exhibits race condition. The globalVar value can be any positive number.
e. None of the above

Q6. ANS = (c).
Revisit of our favourite "pet question" ☺. This version is much easier as each task is essentially doing:

globalSum = globalSum + 50000; // which is equivalent to the following assembly code

```
load r1, globalSum      #A
add r1, r1, 50000       #B
store r1, globalSum     #C
```

With two threads, there are only 2 possible outcomes:
1) Threads cooperate properly, i.e. one finished adding before another ➔ 100000 added to globalSum.
2) Threads interleaves before #C ➔ both reads a 0 into r1 ➔ both writes 50000 back to globalSum.

Testing: Interleaving
Difficulty: EASY.
Stats: a = 13, b = 6, **c = 123**, d = 29, e = 3

Section 2: Short Questions (28 marks)

Question 7 (7 marks)

Consider an array **A** of **N integer values**, a task can execute two operations: i) **IN**: read and remove one of the N values and ii) **OUT**: write into one of the N values. Below is an attempt to use semaphore to synchronize the tasks in operating on the array values:

```
Semaphore mutex = 1;   //binary semaphore
int A[N];   //shared array
```

```
int IN( int idx ) {
     int result;
     wait( mutex );
     result = A[idx];
     //"remove" value
     A[idx] = -1;
     signal( mutex );
     return result;
}
```

```
void OUT( int idx, int newValue ) {
     wait( mutex );
     A[idx] = newValue;
     signal( mutex );
}
```

a. [2 marks] Briefly describe one shortcoming of this implementation.

b. [5 marks] Give an implementation that solve the shortcoming in (a). Note that you can only:
   - Introduce / modify the semaphore declaration and initialization.
   - Add **only** wait / signal to the IN and OUT operation.

ANS:

Difficulty: MEDIUM-ish?

a. There is nothing wrong with the current implementation ☺, given the very loose requirement. For example, there is no indication that "-1" is an invalid number, i.e. it is perfectly fine for multiple tasks to IN the same location (i.e. they will get a -1 from that location).

   Once you realized this, you can see that this code is essentially trying to share an array among many tasks.

   So, the only issue is that the mutex effectively "sequentialize" the access to the array **even when the index** is different, e.g. IN(5) should not blocks IN(3) or OUT(9).

   ("fun" fact: I carefully use the word "shortcoming" instead of "mistakes / incorrectness" to hint that the problem does not lie with correctness ☺).

b. Hence, we can

```
Semaphore mutex[N] = 1;   //N binary semaphore
int A[N];   //shared array
```

```
int IN( int idx ) {
     int result;
     wait( mutex[idx] );
     result = A[idx];
     //"remove" value
     A[idx] = -1;
     signal( mutex[idx] );
     return result;
}
```

```
void OUT( int idx, int newValue )
{
     wait( mutex[idx]  );
     A[idx] = newValue;
     signal( mutex[idx] );
}
```

Some copied the solution from producer-consumer for this question >.<. Hopefully you see that it is not the same problem (e.g. the array items are not consumed/produced in any order) despite superficial similarity.

It is also not a reader-writer problem as the writer does not change the entire array.

The code is purposely written to bear similarity to these classical problems as a way to test your understanding ☺. In general, do not wield known solutions as a hammer and look for nail.

Question 8 (6 marks)

The **responsiveness** of a scheduling algorithm refers to how soon can a newly created task receives its **first share of CPU time**. The following questions focus on a newly created task T_{new} added into an environment where **there are N (N > 0) ready to run tasks**. Restrict your answer to scheduling algorithms discussed in the course so far.

a. [4 marks] Give **two** algorithms that can be responsive. Briefly explain / describe how the algorithms enable responsiveness.

b. [2 marks] Give **one** algorithm that is irresponsive. Similarly explain / describe how the algorithm prohibits responsiveness.

ANS:

Difficulty: EASY-MEDIUM
This question tests your ability to compare and contrast between all scheduling algorithms. In a sense, you should give the most sensible answer from all algorithms learned. Also, if you simply repeat the properties of the algorithm without explaining how T_{new} can be responsive, only partial marks will be awarded.

a. Best Answers: Lottery, MLFQ, Priority. These algorithms have built in features to enable Tnew to receive its **first share** of CPU time asap.
[Full marks if explanation is correct]

   OK answers: SRT, SJF. These algorithms can allow Tnew to get ahead **only if Tnew possess certain property (e.g. short CPU time)** which is harder to arrange (i.e. you need to write Tnew in a certain way). Also, these algorithms **predict** the CPU usage of a new task, which means using a default value which you have no control over (as new task has no execution history to rely on).
   [Partial marks if explanation is correct, especially need to point out how to make Tnew to have shorter CPU time]

   Incorrect asnwer: Others, most notably **Round Robin**
   Don't confuse between **bounded waiting time** (i.e. guaranteed no starvation) and **short waiting time** (i.e. the task gets CPU quickly).
   Given an environment with N tasks, Tnew will be queued at the end and only receive cpu time after n * time_quantum, i.e. much slower than other algorithms discussed above.
   ["fun" fact: some of you probably know this is not the right answer and go so far to suggest "reduce the time_quantum / shorten timer interval" on the machine! ☺]

b. Best answers: FCFS followed by Round Robin

   Acceptable answers: Priority (set ultra low priority), SRT/SJK (make Tnew CPU intensive, note the same issue mentioned in (a)).

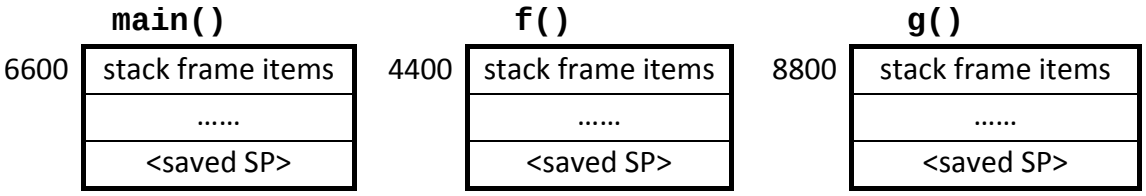
   Incorrect answer: Lottery (arguing the task is "cursed" by bad luck is a bit of a stretch, eh? ☺), MLFQ (suggest that the new task somehow get a special treatment and put at the lowest priority?)

Question 9 (7 marks)

Instead of using the stack memory, Mr. S. Penn suggested the following alternative to support function invocation:

- During compilation, all functions will be allocated a **predetermined** memory location to store their stack frame. (Similar to how global variable has a fixed memory location). There is no change to the stack frame structure.
- The stack pointer (SP) / frame pointer (FP) can simply points to these predetermined locations during the function execution.

For example, suppose there is a program with three functions: **`main()`**, **`f()`** and **`g()`**. The compiler allocated the following locations for their respective stack frame:



a. [1 mark] Suppose **`main()`** calls **`g()`**, show the value(s) of the **`<saved SP>`** in the relevant stack frame(s) when **`g()`** is still executing. Put a "**`---`**" for irrelevant **`<saved SP>`**. If you think it is not possible, please **cross out the stack frames** as an indication.

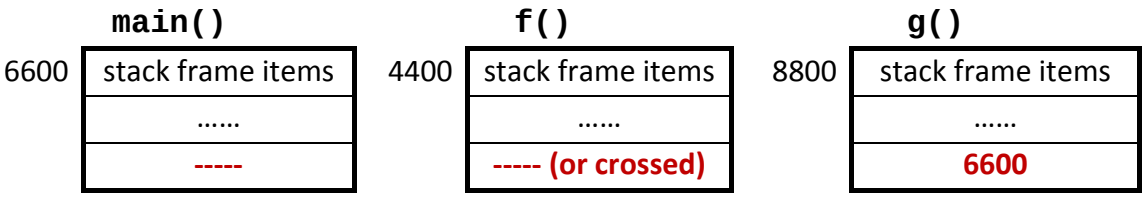
b. [2 marks] Suppose **`main()`** calls **`f()`** which calls **`g()`**, show the value(s) of the **`<saved SP>`** in the relevant stack frame(s) when **`g()`** is still executing. Put a "**`---`**" for irrelevant **`<saved SP>`**. If you think it is not possible, please **cross out the stack frames** as an indication.

c. [2 marks] What are the conditions for this implementation scheme to work?

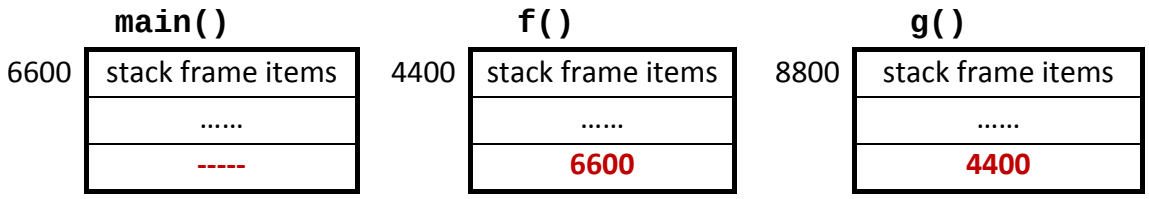
d. [2 marks] Give one example where this implementation scheme fails?

Ans:
Diffulcty = EASY.

a.



b.



c.

Each function is used at most once **in a call chain**. Note that "each function is used at most once" is not correct, this scheme can support multiple calls to the same function:

```
for ( i = 0 to 10)
    f()
```

Just that more than one instance of f() (or g()) cannot exist at the same time.

d.

Any example that involves a recursion, multiple f()/g() existing at the same time will do.

e.g.

main() → f() → f()

Note: Some of you confuse (c) and (d)? (c) is the conditions, (d) is the actual example.

Question 10 (8 marks)

Given the following two multi-threaded processes with their respective execution behavior:

P_1	**T_1**	C1, IO1, C1, IO1, C1, ….. repeats
	T_2	C20
P_2	**T_1**	C1, IO1, C1, IO1, C1, ….. repeats
	T_2	C20

If we use **the standard 3-level MLFQ** scheduling with a time quantum of **2 time units**, answer the following. Note that whenever there is a need to order the processes / threads, you can assume P_1 is ordered before P_2 and the respective T_1 is ordered before T_2.

a. [4 marks] Suppose the threads are implemented as **kernel threads**, give the first 8 time units of the CPU schedule. Remember to indicate both the process number and thread number, e.g. P_2T_1.

b. [4 marks] Suppose the threads are implemented as **user threads**, give the first 8 time units of the CPU schedule. Remember to indicate both the process number and thread number, e.g. P_2T_1. You should try to give fair CPU share to the threads within the same process as much as possible.

For both questions, give **important assumptions you have made (if any)**. You may not receive any mark if key assumption (any assumption not stated in question) is missing.

ANS:

a. Key idea: Kernel threads ➔ OS visible ➔ OS schedule them as four separate tasks.

Only one answer given the setup.

P_1T_1	P_1T_2	P_1T_2	P_2T_1	P_2T_2	P_2T_2	P_1T_1	P_2T_1

Note: Priority for P1T2 and P2T2 decreased after first schedule ➔ P1T1 and P2T1 has higher priority at time unit 7 / 8.

There is no need for any assumption. Wrong assumptions or assumptions in direct conflict with the questions are penalized.

b. This is a sneaky question that actually test your understanding of a user thread rather than scheduling. So, it is important to give the **right assumptions** which indicates your understanding.

I gave:

a. [2 marks] for correct OS view. For the OS, there are only two processes, P1 and P2, so it will just schedule them accord to the MLFQ algorithm with 2 time quantum. Hence, if any part of the scheduling violate this, e.g. P1 receives more than 2 time quantum consecutively, you will be penalized.
b. [2 marks] for best shared use of CPU between threads.

Example of good answer [Full mark if assumptions are stated correctly]:

P_1T_1	P_1T_2	P_2T_1	P_2T_2	P_1T_1	P_1T_2	P_2T_1	P_2T_2

Key assumptions: I/O calls can be intercepted by the user library OR thread T1 cooperatively give up CPU time before its I/O call.

Example of good answer [3 marks if assumptions are stated correctly]:

P_1T_1	P_2T_1	P_1T_2	P_1T_2	P_2T_2	P_2T_2	P_1T_1	P_2T_1

Key assumptions: User library swap thread upon receiving a new CPU time slice.

Note that if you did not state the relevant assumption / wrong assumption, you will not receive full mark.

Section 3: Bonus Question (1 mark)

11. "Know what you don't know": Predict your score for this assessment (excluding this bonus question). If your prediction is with ±2 marks of your actual score, you will get a bonus "true understanding" 1 mark. ☺

ANS:

This is the 2nd time I use this bonus question (the 1st time was in another course). So, my apology if you got hit by this question twice. It is very fascinating to see the various predictions. Although I decided to moderate the result in the end and did not apply the bonus score, I collected the following statistics:

- Average ($Mark_{actual} - Mark_{Predict}$) = -2.39, i.e. most predict ~2.39 marks more than the actual result. In general, about 1/3 students predict **exactly** the score they get.
- Min = -21, Max = 32 (a bit skewed as I assume you predicted a 0 if you didn't fill in the box).