Section 1: MCQ (2 marks each)

MCQ 1 and 2 based on the following code fragment:

```
int main( int argc, char** argv )
{
    int N, i;
    char newArgStr[12];

    printf("Oh no!");

    N = atoi(argv[1]);  //convert cmd line arg to number
    sprintf(newArgStr, "%d", N-1); //convert N-1 to string

    for (i = 0; i < N; i++){
        fork();
        execl("./a.out", "./a.out", newArgStr, NULL);
    }

    return 0;
}
```

If the program above is compiled as "**`a.out`**", answer the following two MCQs:

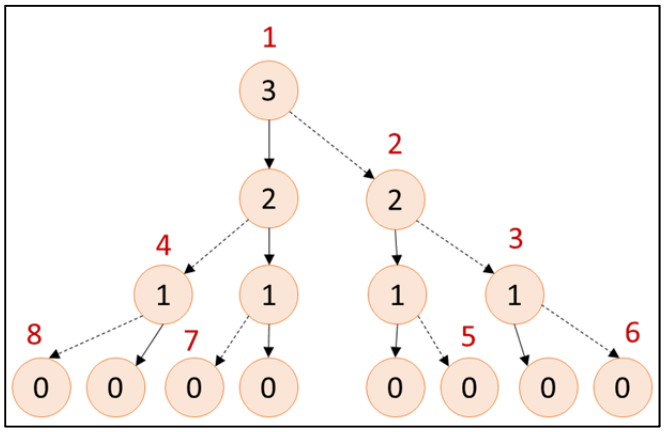
1. How many **processes** in total will be created if we execute "**`a.out 3`**"? Remember to count the first "**`a.out`**".

    a. 3
    b. 8
    c. 15
    d. 31
    e. None of the above

2. How many "**`Oh no!`**" messages are printed in total if we execute "**`a.out 3`**"?

    a. 3
    b. 8
    c. 15
    d. 31
    e. None of the above

ANS:



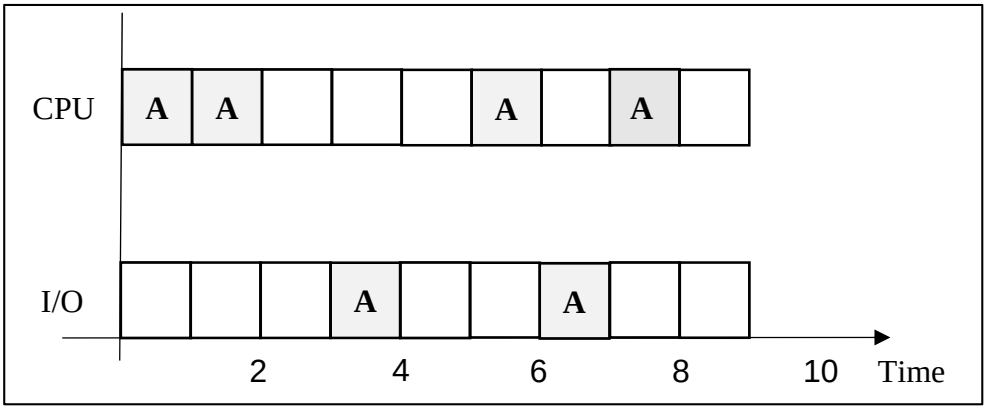
The diagram summarize the answers for both Q1 and Q2. The solid lines represent the execl() (i.e. the process transformed into another executable) and the dotted lines represent new processes created. The value in the circle represents the N received by the a.out as command line argument.

Q1. ANS = (b). As can be seen, a total of 8 processes are created.
Testing: The understanding of fork().
Difficulty: HARD
Stats: a = 8, **b = 45**, c = 46, d = 15, e = 25

Q2. ANS = (c). Every time "a.out" executes, a message will be printed. So, the number of circles represent the number of messages.
Testing: the interplay between fork() and execl().
Difficulty: Medium.
Stats: a = 10, b = 15, **c = 72**, d= 13, e = 29

Fun fact: I tried to think of a "fresh" fork + excel question for a long time to no avail. It is easy just to throw tons of loops together and call a messy question "hard". I wanted a question that is "clean" but with interesting behavior. Eventually, this question came to me in my dream (no joking!). I woke up one morning and have this question clear in my mind. It is actually very clean (the two lines doing integer ←→ string conversion are the only ugly part) and give different answers to Q1 and Q2! I guessed correctly that many will choose the same answer for both ☺.

3. Suppose process A has the following execution schedule:



What is the **most accurate process state** changes for A according to the generic 5-state process model?

a. New→Ready→Running→Blocked→Ready→Running→ Blocked→ Running→Terminated
b. New→Ready→Running→Blocked→Running→Blocked→ Ready→Running→Terminated
c. New→Ready→Running→Blocked→Ready→Running→ Blocked→Ready→Running→Terminated
d. New→ Running→Blocked→Running→Blocked→Running→Terminated
e. None of the above

ANS: (c). The only testing point here is the understanding that a process goes through block→**ready** before being selected for running. It is impossible to go from block directly to running.

Diffulcty: Easy
Stat: a = 8, b = 3, **c = 116,** d = 5, e = 7

4. Suppose we have the following multi-threaded processes on a **hybrid thread** system:

Process	Number of User Threads	Number of Kernel Threads
P1	4	1
P2	1	1
P3	5	3

If all threads are CPU intensive that runs forever, what is the approximate proportion of CPU time utilized by the processes after a long time? You can assume a preemptive fair scheduler.

a. P1 = 20%; P2 = 20%; P3 = 60%
b. P1 = 33%; P2 = 33%; P3 = 33%
c. P1 = 40%; P2 = 10%; P3 = 50%
d. P1 = 25%; P2 = 10%; P3 = 65%
e. None of the above

ANS: (a). Testing the distinction between user thread and kernel thread. Only the latter is visible to OS (i.e. schedulable). So, regardless of the binding (1-1 1-N N-M) between user and kernel thread, it is the number of kernel threads that determine the CPU received.

Diffulcty: Easy
Stat: **a = 100,** b = 14, c = 7, d = 14, e = 4

5. OS provides **system call** interface for the following reason(s):
   i. Faster execution time for the user program.
   ii. Hide the low level hardware details from user program.
   iii. Protect system integrity from user programs.

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

ANS: (c).
i. FALSE, System call requires execution mode change and possibly context change. This introduces quite a bit of overhead.
ii. TRUE, (ii) and (iii) are the main reasons for providing system call.
iii. TRUE,

Diffulcty: Easy
Stat: a = 3, b = 8, **c = 114**, d = 13, e = 1

6. On an interactive processing enviroment with round-robin preemptive scheduler, shortening the interval between timer interrupt (i.e. more timer interrupts in the same time duration) can have the following effect(s):
    i. User programs can be more responsive.
    ii. User program can have a shorter turn around time.
    iii. User program can utilize the CPU more.

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

ANS: (a).
- i. TRUE, Time quantum is a multiple of Interval between Timer Interrupt (ITI). Shorter ITI → Time quantum (without changing the multiple) → process get its CPU faster → more responsive.
- ii. FALSE, If every thing stays unchanged, more interrupts ➔ scheduler runs more often ➔ net loss of user cpu time.
- iii. FALSE, see (ii)

Diffulcty: Medium
Stat: **a = 61**, b = 46, **c = 5**, d = 10, e = 17

Section 2: Bonus Question (1 mark)

7. Which of the following fantastical creatures have **not been** mentioned in the lectures so far?

    a. Vampire
    b. Zombie
    c. Man-eating Titan
    d. Undead
    e. All of the above have been mentioned in the course

ANS: (a).
Tally: Zombie is mentioned during process termination, Man-eating Titan is mentioned during dining philosopher, Undead is mentioned during signal handler (the process that refuses to die).
Stat: a = 93, b = 0, c = 15, d = 9, e = 21. (seems that zombie is the most well known monster in this course, just like in the real world ☺).

Section 3: Short Questions (28 marks)

Question 8 (6 marks)

Consider the 5-threads global sum program:

5 threads execute the following code, where **`globalVar`** is shared
`for ( int i = 0; i < 50000; i++ )` `  globalVar++;`

a. [2 marks] What is the **smallest possible value** for **`globalVar`** at the end of execution (i.e. after all threads terminated)?

b. [2 marks] Briefly describe the interleaving pattern which gives you the answer in (a).

c. [2 marks] If the loop counter "**`i`**" is changed to a shared global variable among the 5 threads, what is the smallest and largest possible value for **`globalVar`** at the end of execution?

Ans:
Difficulty: EXTREME
This turn out to be the HARDEST question in this paper. If I marked strictly then 99% of the class gave the wrong answer ☺. I decided to change my marking to award marks for the incorrect answer but with the right understanding and award bonus mark for those went one step further and showed the deeper understanding. So, this is the only question that you may get 7-8 marks out of 6 ☺.

It is easier to discuss (a) and (b) together.

globalSum = 50000 is a very common answer which shows good understanding of race condition and execution interleaving. You can get 50,000 in several ways, here's one:

each task execute load; increment; before switching to another task and then perform the store. In this case, every 5 additions is registered as only 1 increment.

You get full marks for (a) = 50,000, 1 mark for tiny deviation e.g. 49999, 50001 etc. Full mark is only awarded for (b) if your description is correct and _complete_.

However, the smallest **globalSum is actually 2 (Two!).** You can pick any two threads and make the following observation:

Task A loaded globalsum into register (0) and swapped out.
Task B finished 49, 999 iterations and swapped out.
Task A STORES the incremented registers to globalSum and wiped out all the increments so far (globalSum = 1).
Task B wakes up, take the globalSum into register (1) and swapped out again.
Task A finishes the rest of the iterations.
Task B wakes up, continue with the increment and stores the result (2) back to globalSum.

You can easily extend the above to 5 threads (just "kill off" the other 3 threads and make sure that two threads follow the above pattern).

ps: Don't be demoralized. Only ONE student got this answer ☺

(c) Smallest globalSum is 1. Here's one way to get it:

Task A loaded globalsum into register (0) and swapped out.
Other tasks incremented i until it is larger than 49999. globalSum at that point is not important.
Task A wakes up, add 1 time, store globalSum (1).
Taks A reads i, check condition and terminated the loop.

(c) Largest globalSum is…… **HUGE** ☺, here's one way to see the craziness:

Task B added one time, and read the i (preparing to do the "i++). However, just before writing i, it was swapped out.
Task A did all the hardwork and incremented globalSum 49,999 times, but before the last check, it was swapped out.
Task B wakes up and write (1) to "i".
Task A wakes up and loaded i for checking and found that it has still close to 49,999 iterations……. ☹

Task B can do the above EVERYTIME Task A finishes near to 49,999 times.

BUT, that's not all, we can use Task C to restart Task B when it is *finally* near the end and restart the entire process for A and B. Don't forget that we still have task D and E…….

To award marks according to understanding. Here's what I did, if your answer is less than or equal to 50,000, 0 mark (didn't see the idea of interleaving). 1 mark if your answer is around 250,000 (at least saw the potential of explosion). 1 + 1 mark (i.e. 1 bonus mark) if your answer is way larger (you saw the idea of restarting).

Question 9 (6 marks)

Consider the standard 3 levels MLFQ scheduling algorithm with the following parameters:

- Time quantum for all priority levels is 2 time units (TUs).
- Interval between timer interrupt is 1 TU.

a. [4 marks] Give the **CPU schedule** for the following 2 tasks using notation similar to MCQ3. The first time unit has been filled as an example in the answer sheet.

Task A
Behavior: **CPU 3TUs**, *I/O **1TU***, **CPU 3TUs**

Task B
Behavior: **CPU 1TU**, *I/O **1TU***, **CPU 1TU**, *I/O **1TU***, **CPU 1TU**

Note that we only ask for the CPU schedule, you will have to keep track of the priority level of the tasks separately on your own.

b. [2 marks] Using the tasks A and B above to illustrate one advantage and one disadvantage for the MLFQ scheduling algorithm.

Ans:
Difficulty: Easy – Medium

(a) answer:

CPU	A	A	B	A	B	A	A	B	A					
TU	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Testing point: understanding of time quantum and ITI (not the same thing). Once a task is scheduled, it stays for the _time quantum_ unless it is stalled by I/O. If you have blanks in the schedule ➔ incomplete understanding ➔ penalized.

(b) the question emphasized "Use task A and B to illustrate…..". Memorizing the pros / cons of scheduling algorithms is not sufficient, you need to be able to apply it in actual context. So, if your answer doesn't touch on task A or B in any way, no marks were awarded.

Sample answers (many possibilities):

Pros: MLFQ can learn the behavior of task, e.g. Task A is a relatively CPU intensive task which get penalized in priority as MLFQ is bias towards I/O intensive tasks.

Cons: Pure MLFQ can be gamed, e.g. if task B is engineered to have the shown behavior (blocking just before time quantum expiration), it can hogged the CPU.

Question 10 (6 marks)

Clever compiler can transform a recursive function into an equivalent **tail-recursive** function. The key characteristic of a tail recursive function: the partially computed result is passed to the callee so that the last callee (i.e. the base case) has the final result without further calculation. Below is an example of factorial (what else ☺) and its tail-recursive version:

Factorial – Recursive Example call: `fac(5)`	**Factorial – Tail Recursive** Example call: `facTail(5, 1)`

`int fac( int N )`
`{`
`  if (N == 0) return 1;`
`  return N * fac(N-1);`
`}`
	`int facTail( int N, int Res )`
	`{`
	`  if (N == 0) return Res;`
	`  return facTail(N-1, Res * N);`
	`}`

a. [2 marks] Assuming no special optimization (i.e. treating `facTail()` as a normal recursive function), what is the maximum number of stack frames of `facTail()` function during the function call `facTail(5, 1)`?

b. [2 marks] Using your understanding of function call and stack frame, briefly describe how we can exploit the behavior of tail-recursive function call to **reduce the usage of stack memory**.

c. [2 marks] If your optimization in (b) is used, what is the maximum number of stack frames of `facTail()` during the function call `facTail(13, 1)`?

Ans:

(a) Difficulty = Easy (to cushion you for the harder (b) and (c)).
ft(5) → ft(4) → … → ft(0), a total of **six** stack frames at the maximum point.

(b) Difficulty = Medium. Stack frame for a function only needs to be maintained if it is going to continue execution. In tail recursion, once a function make a recursive call, it ceased to be useful (on the "return" leg of the journey, a function simply pass along the return result without doing any further computation). So, the best way to exploit tail recursion is simply allow the recursive call to **overwrite the current stack frame** instead of allocating a new frame.

(c) As seen in (b), you only need one stack frame of ft() regardless of the parameter N.

Question 11 (10 marks)

Below is the blocking solution for producer-consumer problem:

```
while (TRUE) {
        Produce Item;

        wait( notFull );
        wait( mutex );
        buffer[in] = item;
        in = (in+1) % 5;
        count++;
        signal( mutex );
        signal( notEmpty );
}
```

Producer Process

```
while (TRUE) {

        wait( notEmpty );
        wait( mutex );
        item = buffer[out];
        out = (out+1) % 5;
        count--;
        signal( mutex );
        signal( notFull );

        Consume Item;
}
```

Consumer Process

Suppose we initialized the semaphores as follows (note that the size of the shared buffer is **5** and pay attention to the two incorrect initializations):

- **`notFull = Semaphore(3);`** //incorrect!
- **`notEmpty = Semaphore(1);`** //incorrect!
- **`mutex = Semaphore(1);`** //correct

a. [2 marks] Briefly describe the effect of the incorrect **`notFull`** semaphore initialization. You should ignore the effect of the wrong **`notEmpty`** in this question.

b. [2 marks] Briefly describe the effect of the incorrect **`notEmpty`** semaphore initialization. You should ignore the effect of the wrong **`notFull`** in this question.

c. [3 marks] Suppose we have 5 producers tasks (P1 to P5) and 3 consumers tasks (C1 to C3) using the incorrect initialization as stated. Give a snapshot of the semaphores (the value and the blocked tasks on them) at **any point during the execution**. **The only restriction is that the status of all 8 tasks must be clearly stated or implied in the snapshot.**

d. [3 marks] If you were to mark your friend's answer on part (c) above, give the rules that you use to determine correctness. Your rule should be concise and has clear outcome (i.e. if *XXXX*, then it is **wrong / correct** or similar). You can also consider using formula to express relationship.

Ans:

(a) Difficulty = Medium. Key point, the wrong semaphore init value limits the number of concurrent producer task (or the total useful value in buffer). Note that I was (very) strict for this part. Your answer should show that you understand all 5 locations of the buffer can still be used (e.g. after the first 3 production, consumer consumed 3, then the subsequent production will utilize buffer[3] and buffer[4] and wraps around etc). So, if you simply say "two slots will not be used" (or similar), that is INCORRECT.

(b) Difficulty = Easy. Key point, one consumer can start consuming without the data being placed in the buffer.

(c) Difficulty = Medium. The key phrase in the question is "All tasks status should be clearly stated or deducible". The "clearly stated" part refers to tasks shown explicitly on the semaphore queue, i.e. you know they are blocked. The "deducible" part can have a few example, e.g. if one task is missing from the picture and mutex is 0 ➔ that task is in the critical section.

So, the simplest answer is to just follow up on the original setup:

Mutex = 0, Queue = P2, P3, C1

notFull = 0, Queue = P4, P5

notEmpty = 0, Queue = C2, C3

The missing P1 is clearly in the critical section.

Note that if your answer need to write "P1, P2 is doing …., P3 is ….." explicitly outside of the semaphore ➔ incorrect.

There are (way too) many alternatives ☺. Feel free to explore a few other….

(d) Difficulty = Hard. This question intends to test your "meta understanding", i.e. not just knowing one possible execution scenario, but go deeper and understand how ALL correct scenario should look like. Although I personally like this question a lot, the marking nearly fried my brain ☹. In the end, I again award marks based on your **level of understanding:**

i) If you gave generic rules that are just reiterating semaphore property, e.g. "value cannot be negative", "value = init – wait + signal", etc. **NO MARK**.

ii) If you gave simple rules that are applicable to independent semaphore in this question, e.g. "If value of notfull > 4 == incorrect". 1-2 marks.

iii) If you gave rules that link the semaphores together, e.g. "notFull + notEmpty <= 4". 3 marks.

Note: If you gave any incorrect rules, penalty is applied. The most common incorrect rule is "notFull cannot be > 3" or "notEmpty cannot be > 1". Both shows incomplete understanding: the consumer will signal notFull at the end of consumption. i.e. if there is one consumer task that can consume at the beginning (as in this case), then it can definitely signal the notFull semaphore and bring it up to 4 before any producer starts. Similarly, once the 4 producers produces, each of them will signal notEmpty, again bring it up to 4 at the maximum. So, it is wrong to use the number "3" as the upperbound.

Note 2: Your answer should be about part (c). Any generic description that cannot be observed from (c) are ignored, e.g. "If there are two processes in the critical section", "if there are 2 producers in between of the wait(notFull) and wait(mutex)", etc.

Fun fact: Best answer is "Any answer given by me is considered as correct ☺" (smiley included).

Final Tally: The class average is ~24, with highest at 40. This is rather healthy by CS2106 standard (we only get an average of ~24 last year **after heavy moderation**). Although I'd like to think that my craziness in question setting has abated ;-), it is far more likely this cohort is stronger☺. So, good job and keep up the good performance!