Section 1: MCQ (2 marks each)

1. Which of the following statement(s) regarding the difference between library calls and system calls is/are **TRUE**?

   i. Library calls are programming language dependent, while system calls are dependent only on the operating system.
   ii. System call may be invoked during a library call.
   iii. System calls are only invoked to handle system level I/O functionalities.

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

   i. (TRUE) System calls are provided by OS, and libraries calls are provided individual languages. Some library calls wrap around one or more system calls to provide the functionality to user program.
   ii. (TRUE) see above.
   iii. (FALSE) For example: creating a new process, terminate a process, create shared memory region, etc.

ANS: **b**
Stats: Number of students who chose a particular option: (a=4, **b=169**, c=7, d=14, e=3)

MCQ2 and MCQ 3 are based on the following scenario:

> Given a program P with two independent tasks A and B, let us examine the two implementations below:
>
> i. Sequential implementation: Single program that executes task A then task B.
> ii. Multithreaded implementation: A program with two threads working on task A and B concurrently.

2. Suppose both tasks are CPU-intensive, which of the following statement(s) is/are **TRUE** if the two implementations are executed on a **single CPU (single core)**?

   i. If the threads are implemented as **user threads**, then multithreaded implementation will finish execution in shorter amount of time compared to the sequential implementation.

   ii. If the threads are implemented as **kernel threads**, then multithreaded implementation will finish execution in shorter amount of time compared to the sequential implementation.

   a. (i) only.
   b. (i) and (ii).
   c. (ii) only.
   d. None of (i) and (ii).

With a single CPU, there is no way to speed up the execution. The two threads will still need to take turn to use CPU. Actually, adding threads may slow down the overall execution time due to the overhead!

ANS: **d**
Stats: (a=36, b=31, c=21, **d=109**)

3. Suppose task A is CPU-intensive and task B is I/O intensive (reading from a file), which of the following statement(s) is/are **TRUE** if the two implementations are executed on a **single CPU (single core)**?

   i. If the threads are implemented as **user threads**, then multithreaded implementation will finish execution in shorter amount of time compared to the sequential implementation.

   ii. If the threads are implemented as **kernel threads**, then multithreaded implementation will finish execution in shorter amount of time compared to the sequential implementation.

   a. (i) only.
   b. (i) and (ii).
   c. (ii) only.
   d. None of (i) and (ii).

This is essentially the scenario illustrated by the "Responsive" thread demo during the lecture. With an I/O bound thread and a CPU bound thread, if we can switch between the two, we may be able to do them in "parallel" (the I/O waits on file data while the CPU thread do some useful work in the mean time). So, the question become a simple test of user/kernel thread, i.e. which thread model allow both threads to be "schedulable" by OS.

ANS: **c**
Stats: (a=29, b=49, **c=93**, d=25, e=1)

Fun fact: Someone picked "e" ……

4. Which of the following statement(s) regarding the scheduling algorithms SJF (shortest job first) and SRT (shortest remaining time) is/are TRUE?

   i. If all tasks arrived at the beginning, then SRT will give the same schedule as SJF.

   ii. If the tasks arrive in the same order as their cpu time (i.e. shortest job first), then SJF will give a better schedule compared to SRT in terms of average turn around time.

   iii. If the tasks arrive in the opposite order as their cpu time (i.e. longest job first), then both SJF and SRT give the same schedule.

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

The only difference between SRT and SJF is that the former is **preemptive**. Preemptive scheduling only makes a difference **if the newly arrived task replaces the current running task**. This is not possible under non-preemptive scheduling, where a running task will just run until done/give up.

With the above guiding principle, you should be able to deduce (i) is TRUE (no new arrival), (ii) is FALSE (new arrival takes longer time ➔ no replacement), (iii) is also FALSE (SRT **will** preempt current running task in that scenario resulting in different schedule).

ANS: **a**
Stats: (**a=134**, b=23, c=11, d=12, e=17)

5. Consider the following code:

```
while ( fork() != 0 )     {
    execl( "Hello.exe", "Hello.exe", NULL);
}
```

Which of the following statement is TRUE?

a. Only when “Hello.exe” is a valid executable, then the code is a “fork bomb”.
b. Only when “Hello.exe” is not a valid executable, then the code is a “fork bomb”.
c. Regardless of whether “Hello.exe” is a valid executable, the code is a “fork bomb”.
d. Regardless of whether “Hello.exe” is a valid executable, the code will not result in “fork bomb”.
e. None of the above

This question tests two concepts, the behavior of fork() and execl(). After the fork(), both parent and child processes will start at the same place, i.e. testing the condition of while loop. The child process (with a 0 return value) will fail the loop immediately. So, we only need to worry about the parent process in the loop body.

It is clear that if the parent process get to "loop" again, then it can create a fork bomb where infinite number of processes are created. Now, you just need to check whether the parent can "survive" the body of loop.

execl() (or any exec() calls) will **replace** the calling process with the executable image **if** the executable can be found. Hence, an invalid executable will cause the execl() to fail and **returns** normally ➔ parent process will then carry out another iteration!

ANS: b.
Stats: (a=21, **b=34**, c=78, d=63, e=1)

"Fun" fact: This is the only MCQ where the correct choice is not dominant. As lab 2 focuses on exec() quite heavily, I'm surprised that the behavior is not better known…. ☹

6. Consider the following code:

```
for( i = 0; i < 3; i++){
      p = fork( );
      if ( p == 0 )
           funcA();
      else
           funcB();
}
```

Which of the following statement is TRUE?

	funcA() executed	**funcB**() executed
a.	3 times	3 times
b.	4 times	4 times
c.	7 times	7 times
d.	8 times	8 times
e.	None of the above	

You can visualize the execution as a "tree":

Start
| \
A B
| \ | \
A B A B
|\ |\ |\ |\
ABABABAB

ANS: C
Stats: (a=35, b=10**, c=120**, d=11, e=21)

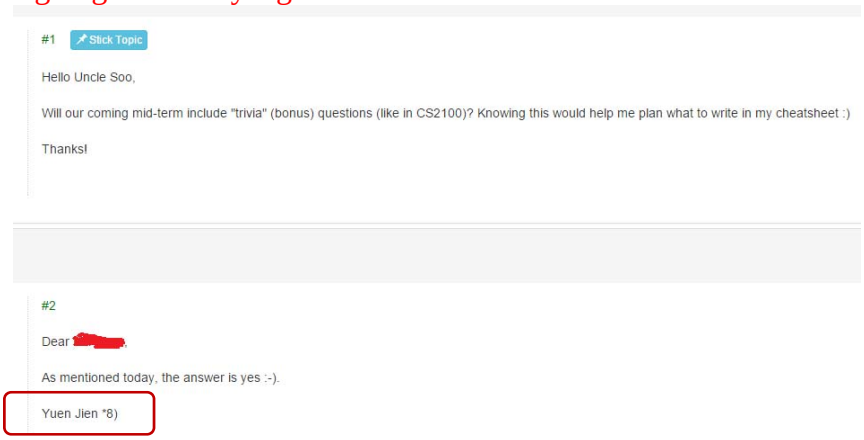
Section 2: Bonus Question (1 mark)

7. Uncle Soo always sign off his forum messages with:

a. Yuen Jien *8)
b. Yuen Jien *8(
c. Uncle Soo *8)
d. Uncle Soo *8(
e. Yuen Jien \^^/

ANS:a

Fun Fact 1:
This forum post (posted before the midterm): I couldn't stop laughing while answering and signing it with my signature…



Fun fact 2: Someone wrote "*8) ← is this a pointer?" under the answer ☺. Btw, the "*" is my short hair, "8" is the spectacle, and ")" is the smiling mouth.

Stats: (**a=135**, b=2, c=53, d=1, e=6) This is far from the hardest question ;-)

Section 3: Short Questions (28 marks)

Behind the scene: I intended to have a wide spread of difficulties in this section. The intended difficulty is (from easiest to hardest)

1. Question 9 (stack frame)
2. Question 8 (MLFQ)
3. Question 11 (Semaphore)
4. Question 10 (Synchronization – the "killer" question)

As you will discover later on, the actual result shows that (2) is somehow harder than (3).

Most questions were designed to push you to think beyond what is taught and/or beyond a single topic (i.e. how each topic links to another).

Question 8 (10 marks)

Consider the standard MLFQ scheduling algorithm with 3 levels (0 = Lowest priority, 2 = Highest priority). The tasks are given different time quantum according to their priority (Priority 0 = 8 time units, priority 1 = 4 time units, priority 2 = 2 time units).

a) (**8 marks**) Give the pseudocode for the **scheduler function.** For simplicity, you can assume that all tasks are CPU intensive that runs forever (i.e. there is no need to consider the cases where the task blocks / give up CPU). Note that this function is invoked by timer interrupt that triggers once every time unit.

Please use the following variables and function in your pseudocode. You are not allowed to declare additional variables.

Variable / Data type declarations
Process **PCB** contains: **{ PID, TQLeft, … }** // TQ = Time Quantum, other PCB info irrelevant. **ReadyQ[3]** is an array of 3 PCB queues according to the priority level. The queues supports standard operations like **isEmpty()**, **insertBack()** and **removeFront()**. **RunningTask** is the PCB of the current task running on the CPU. **TimeQuatum[3]** is an array of 3 values { 8, 4, 2} corresponding to the time quantum given to task at a particular priority level. **TempTask** is an empty PCB, provided to facilitate context switching.
"Pseudo" Function declarations
`SwitchContext( PCBout, PCBin );` Save the context of the running task in ***`PCBout`***, then setup the new running

environment with the PCB of ***PCBin***, i.e. vacating ***PCBout*** and preparing for ***PCBin*** to run on the CPU.

b) (**2 marks**) The pseudocode does not handle the case when a task blocks prematurely on I/O. If we want to handle those cases, should the handling code be a part of the code in (a)? Briefly explain.

ANS:

a. There are a few key points that you should have in your answer:

1. RunningTask.TQleft--
2. If (RunningTask.TQLeft > 0) return //no need to schedule
3. If (RunningTask.priority > 0)
   RunningTask.priority--
4. ReadyQ[RunningTask.priority]. insertBack(RunningTask)
5. For(i= 2 down to 0) //same logic can be expressed as a 3-level "if-else"
6. if !ReadyQ[i].empty()
7. TempTask = ReadyQ[i]. removeFront()
8. TempTask.priority = i
9. TempTask.TQleft = TimeQuantum[i]
10. switchContext(TempTask, Running Task);

Common mistakes:

- Started the scheduler with a "while (TRUE)" or similar. This highlight a rather serious misunderstanding of the working of scheduler. Remember that scheduler is just a piece of code ➔ it requires CPU to run too! If it is running forever, there is no way the "user task" can run!
- Missed the fact that the time quantum bookkeeping is part of the scheduler job. The user task is just a normal program, which do not "track its own CPU usage and decrease the time quantum automatically". The only way we can enforce time quantum is to make the scheduler goes off every timer interrupt **and** reduce the quantum left.
- Missed the fact of the priority reduction (line 3-4 above) in MLFQ.

Tips: Inserting the running task back to the correct ReadyQ before making selection makes the code much simpler!

Fun fact: Someone managed to shrink the whole algorithm into ~4 lines of ***very neat*** and easily understandable pseudo-code!

b.

This is a "sneaky" question ☺. What we were testing is not really on scheduling but whether you understand the roles of system call and interrupt. The scheduler triggers only once every timer interrupt, while the I/O calls are always issued in between (the task need to be running to execute the I/O calls!). So, checking for I/O call in this scheduler is either a) wasteful of CPU time (as the whole time tick may be lost!) if you check once every time tick or b) not possible if you want to check "right away" as I/O calls do not trigger scheduler automatically.

To extend on (b), note that I/O are all system calls as OS controls the access to all devices ➔ during I/O system call handling, the OS can then block and re-schedule the tasks.

Note that I do not award marks on simply the "Yes/No", instead your explanation must make sense to earn any mark!

Stats (for whole question): Marks average = 5.33, Std-dev= 2.8

Question 9 (5 marks)

In this question, we will examine various aspects of a **stack frame**.

For each of the following code fragments, give the **maximum number of stack frames of the function `f()`** that **exists at the same time** during the execution of **`main()`**.

a) 1 mark

```
void f()
{
   int i;
   i=1234;

}
```

```
void main()
{
     int i;

     for (i = 0; i < 1000; i++)
          f();
}
```

b) 1 mark

```
void f( int i )
{
    if (i == 0)
        return;
    else
        f(i-1);
}
```

```
void main()
{
        f( 1000 );
}
```

What is the output of the following code fragment? You can assume that memory locations contain random values at the beginning. **State your assumptions (if any).**

c) 3 marks

```
void f( )
{
    int i;
    printf( "%d\n", i);
    i=1234;
}

void g( int para )
{
     printf("%d\n", para)
}
```

```
void main()
{
     f();
     f();
     g( 5432 );
     f();
}
```

ANS:

a. 1

b. 1001 (if you wrote 1000, I have to mark you as wrong ☹ as I couldn't give ½ mark here)

c.

Random value

1234 (Stack space is not cleared as shown in lecture/tutorial)

5432

5432 (see below for alternative)

The last entry check whether you have noted the fact that without parameter, f()'s stack frame may look just like g()'s, with the local variable slot mapped to the parameter slot. If you pointed this out in the assumption (e.g. you assume the parameter slots are located differently), then alternative answers can be accepted.

Note that you cannot just "assume away" the question, e.g. "stack frame are allocated on new region every function call" is not a valid assumption!

Fun fact: Someone figured out a way to write the answer as "*1234* 1234 5432 5432" and I have to mark it as correct! He wrote in the assumption "A miracle occurs and the random value in the stack frame happens to be "1234" ☺.

Stats: Marks average = 3.5, std-dev = 1.07

Question 10 (7 marks)

Suppose there a number of tasks (>1) share the following variables:

- **sCurrent** = integer, initialized to 0, **sNext** = integer, initialized to 2.
- **sArray[0….50000]** = an array of Boolean variables. Elements [0], [1] are initialized to FALSE, all other elements **sArray[2…49999]** are initialized to TRUE.

For your convenience, all shared variables have a "s" prefix.

Each task executes the following code (in pseudo code):

```
1   while (TRUE)  {
2           while ( sCurrent is equal to sNext)
3                   sleep for 1 second;

4           if (sNext >= 50000) end task;

5           factor = sNext;
6           sCurrent = sNext;

7           for (j = factor*2; j < 50000; j = j + factor) //sieving through the array
8                 sArray[j] = FALSE

10          for (k = factor+1; k < 50000; k++)        //looking for the next prime
11                  if (sArray[k] is TRUE)
12                          if (k > sNext)
13                                  sNext = k;
14                          break loop;

15          if (k >= 50000)
16                  sNext = 50000+1;
17                end task;
18  }
```

The above is an attempt to parallelize the famous Sieve of Eratosthenes algorithm. Essentially, we eliminate all multiples of a prime number, e.g. use 2 to eliminate of 4, 8, 10, 12….. Then the smallest number > 2 that is not eliminated (e.g. 3) will be used as the next prime number. The process is then repeated until the whole range is exhausted (50,000 in this case). The algorithm is an efficient way to find out all primes in a given range.

a) (**2 marks**) Line 12 seems redundant (**sNext** is used as **factor**, so how can **k** > **sNext**?) Do you think it is needed? Briefly explain.

b) (**3 marks**) The code currently still exhibits race conditions, use two tasks to identify the race conditions. You only need to list down the line numbers that respective tasks executes that causes the race condition. Below is an example where two tasks collaborated sucessfully:

	Task 1	**Task 2**
t0	2-17	
t1		2-17
t2	2-17	

c) (**2 marks**) If you can use critical sections to solve the issues in (b), indicates the **absolute minimal** lines of code you will protect. Give the corresponding line numbers in the answer sheet.

Ans:

This is a very hard question. So, if you get this wrong, don't be too demoralized. Rather, take your time to rethink and understand the question better. ☺

a.
It is not redundant, as other task may have increased sNext to a much larger number.

Fun fact 1: Someone wrote "k = sNext+1 and k is monotonously increasing ➔ line 12 is useless! Q.E.D ", the best wrong proof ;-) .

Fun fact 2: This is actually a hint for part (b) and (c). It was added for both "easy" marks and to nudge you into the right direction.

Again, note that I do not award marks on simply the "Yes/No", instead your explanation must make sense to earn any mark!

b.
The code is "carefully written badly" such that there is only one point of failure. If you think through it carefully, there are many possibilities of *wasted computation* (e.g. re-sieve the array with a known prime, two or more task used the same prime to sieve etc), but they will not result in incorrect execution. So, to properly shows a race condition, you need to pinpoint that single point of failure.

The point of failure is in line 12-13. Part (a) is actually a ***failed attempt*** to correct the error! Many of you recognize the fact that line 12 is essential, but didn't take it one step further, i.e. what if a task T was stopped *after* line 12 and swapped out? As the "if (k > sNext)" condition has been checked, the swapped out task T will perform the "sNext = k" when it resumed! Now, imagine what happened if sNext is already very large (or better yet, imagine it is already at 50,001), task T resumes and assigned a much smaller k as sNext ➔ cause re-

sieving of the array. If this happens repeatedly, you will get an extremely slow execution due to redundan work!

So, an acceptable execution timeline must have an interleave pattern between line 12 and 13, e.g.

	Task 1	Task 2
t0	5-12	
t1		5-17
t2		2-17
t3	13(!)-17	

c.
From the above discussion, the minimal answer is "line 12, 13". Note that this essentially "sequentialize" the tasks. Alternative answer: "line 5,6 and line 12,13" is acceptable (though "5,6" is not critical).

Note that if your answer covers more than 5 lines, e.g. "line 2 to 17", "line 5 to line 13" etc, you'll be marked wrong as the question stressed that you should provide only "the **absolute minimal**" lines of code to protect.

Fun fact: Someone saw through the whole question, solved it perfectly and noted that "the corrected version will still be very slow, meh!". This is almost the exact question I original designed as part d, i.e. "Does this code executes faster than sequential verison?". However, I decided to pull out the part (d) as I think (b) and (c) are crazy enough. Btw, the answer to the imaginary part (d) is "No!". Sieve of Erathonese is quite hard to parellelize properly.

Stats: Marks average =2.49, Std-dev = 2.08.

Question 11 (6 marks)

(**2 marks each for 3 functions**) Suppose we have a single **Police** task and a larger number of **Citizen** tasks running simultaneously. The **Citizen** tasks, from time to time, will need to pass through a **checkpoint()** function. This **checkpoint()** function will normally release the task that enters it immediately. However, when there is a need, the **Police** task will perform a **lockdown()** function which will blocks all **Citizen** tasks reaching the **checkpoint()** function. The **Citizen** tasks remains blocked until the police task execute the **release()** function.

Note: you only need to ensure the citizen tasks will be blocked **after the police task successfully returns from the lockdown() function.**

Use **semaphore** to implement the 3 functions described above. The functions should contains **only wait() / signal()**. **Full marks will be given only to implementations that uses 1 semaphore**.

ANS:

Simplest answer:

Semaphore S(1);

//Note: No declaration or incorrect declaration is penalized heavily (50% of marks). This may seems harsh, but any semaphore solution will be rendered useless unless the correct initial semaphore values are given. For e.g. if the semaphore is initialized to 2 in this case, the answer will not work anymore!

checkpoint()	lockdown()	release()
wait(S);	wait(S);	signal(S);
signal(S);		

Fun fact 1: Someone wrote "You shall not pass!!" as a comment in "lockdown()".

Fun fact 2: After sending the paper for print, I realized that, to my horror, this question can be solved by the exact same approach used in tutorial 5 – Writer-Reader problem! As I set the tutorial much earlier than the midterm, I actually forgot that I have a similar question in both =.=. After banging my head on the wall for a while, I decided to let it stay as a way to tone down the overall difficulty of the paper.

Stats: Marks average= 4.02, Std-Dev = 2.26.