Section 1: MRQ (Each question = 0.5 x 4 = 2 marks)

Question 1 to 5 are **multiple response question (MRQ)**. You need to evaluate each suboption given in the question to either "**1**" (True) or "**0**" (False). Write **only "1" or "0"** in the answer sheet. **Each suboption is worth 0.5 marks.**

1. Each of the following cases insert zero or more statements at the Point α and β. Evaluate whether the described behavior is T (true, i.e. correct) or F (false, i.e. incorrect).

```
int main( )
{
     //This is process P
     if ( fork() == 0 ){
          //This is process Q
          if ( fork() == 0 ) {
               //This is process R
               ......
               return 0;
          }
          <Point α>
     }
     <Point β>

     return 0;
}
```

	Point α	Point β	Behavior
a	*nothing*	wait(NULL);	Process Q ***always*** terminate before P. Process R can terminate at any time w.r.t. P and Q. **[False: Q waits for R]**
b	wait(NULL);	nothing	Process Q ***always*** terminate before P. Process R can terminate at any time w.r.t. P and Q. **[False: Q waits for R and P don't wait]**
c	wait(NULL);	wait(NULL);	Process P **never terminates**. **[False: Although Q has an additioinal wait, the wait will return immediately as there is no child.]** **[This is unintentionally "hard" as I wrongly assumed that wait() can get stuck. As such, I**

			decided to "void" this question, i.e. you get 0.5 for this suboption regardless of your choice.]
d	execl(*valid executable….)*;	wait(NULL);	Process Q ***always*** terminate before P. Process R can terminate at any time w.r.t. P and Q. **[True: P wait for Q even though Q is now a "new" executable]**

2. Evaluate each of the following statements regarding **Process Control Block (PCB).**

a	The hardware context in the PCB always reflect the actual register values in the processor. **[False: HW context in PCB is updated only when process swap out]**
b	The memory context in the PCB is not the actual memory space used by the process. **[True: We mentioned that it is "points" to the real memory space. With updated understanding, you should know that PCB contains page table.]**
c	The PCBs themselves are stored in the memory. **[True: Part of the OS memory space]**
d	The OS context of a PCB may contains information used for scheduling, e.g. priority, time quantum allocated, etc. **[True]**

3. Evaluate each of the following statements regarding **Process Scheduling.**

a	The interval between timer interrupt is always an integer multiple of time quantum. **[False: Should be the other way round]**
b	The MLFQ algorithm favors (i.e. give higher priority to) IO Intensive process. **[True: Basic behavior of MLFQ]**
c	Non-premptive scheduling algorithm cannot cause starvation. **[False: Counter example = Non-premptive priority scheduling can still cause starvation]**
d	Given the same period of time, smaller interval between timer interrupt lengthen task turn-around time. **[True: Shorter ITI ➔ More Timer Interrupt ➔ less time spent on actual user process]**

4. Evaluate each of the following statements regarding **Thread** and **POSIX Thread (pthread).**

a	All pthreads must execute the same function when they start. **[False: Pthread can start on any function as long as the function signature is void* f(void*)]**
b	Multi-threaded program using pure user threads can never exploit multi-core processors. **[True: This is the main short coming of user thread as OS is not aware of the threads.]**
c	If we use a 1-to-1 binding in a hybrid thread model, the end result is the same as pure-user thread model. **[False: The result is a pure kernel-thread model as each user thread is bounded to a kernelt thread that can be scheduled.]**
d	On a single-core processor that support simultaneous multithreading, we can execute more than one thread at the **same time**. **[True: This is the main strength of SMT processors as discsued in lecture.]**

5. Evaluate each of the following statements regarding **Unix Signal.**

a	A process can install user-define handler for **multiple different signals.** **[True: We demoed this in the lecture.]**
b	We can install user-define handler for all signals. **[False: The "kill -9" i.e. SIGKIL is not captureable]**
c	A parent process can force the child processes to execute any part of their code by sending signal to them. **[False: Only the signal handler can be triggered.]**
d	The "kill" signal (sent by the "kill" command) is different from the "interrupt" signal (sent by pressing "ctrl-c"). **[True: We demoed this in the lecture.]**

Section 2: Short Questions (30 marks)

Question 6 (10 marks)

Below is a program **P** sketch:

```
int main() {
   int i, result[100000];   //100,000 results

   for (i = 0; i < 100000; i++)
       result[i] = compute(i);   //compute takes ~1ms per call
   <Write the result[] array into a file "secret.txt">
}
```

Suppose **P** is executed on an OS with **standard 3 levels MLFQ scheduler** with a **100ms time quantum**. For consistency, let's assume **priority 2 is the highest** and priority 0 is the lowest.

a. Briefly describe the priority change of **P** overtime on this system. **[1 mark]**

b. Briefly describe with **pseudo-code** on how to maximize processor usage **unfairly** on this system to complete the computation as fast as possible. **[3 marks]**

c. As discussed in tutorial, exploits like (b) can be mitigated by keep track of processor usage **across timeslices**. By using **only topics learned in CS2106 so far,** briefly describe with **pseudo-code** on how you are still able to maximize processor usage **unfairly**. You need to state all relevant assumptions and explain how the ***result[]*** array is maintain properly (contains all result, in the same order as the original code) through the "cheat". **[4 marks]**

d. Suggest one simple scheduler fix for your exploit in (c). **[2 marks]**

Ans:

a. **P starts with priority 2 and drops to 0 over time.**
b. **Main idea: give up before time quantum lapses.**
   Code:

```
for (i = 0; i < 100000; i++){
    result[i] = compute(i);
    if (i+1 % 99 == 0)  sleepMillisecond(1);  //or any similar blocking call
}
```

Rationale: The idea for this was discussed in both lecture and tutorial, so it is an intended easy question. Asking for a "pseudo code" allows me to check whether you understand

the key ideas (What's time quantum, How do "give up" etc) and their application in real situation.

c. **Assumption:**
- **Child process do not inherit parent's usage statistic**

Main ideas:
- **Spawn child to continue to run. Child is a new process == highest priority**
- **Parent should exit to avoid clogging up the system and do redundant work**

Code:

```
for (i = 0; i < 100000; i++){
   if (i+1 % 99 == 0){
      if (fork()!=0) exit();
   }
   result[i] = compute(i);
}
```

Rationale: This is moderately challeging as it tests your understanding of "fork()" as well as the property of scheduling. The above solution is more like a "relay", the parent process work near to the TQ, then allow the direct child process to continue. It is not really parallel computing as there is only one actual working process at any time. Also, due to the property of fork(), the result array computed so far are passed to the child by the duplication. So, no effort is needed to maintain nor synchronize.

Common "wrong" solution:
- **Trying to solve this as parallel problem by spawning ~1000 processes or threads in one go and let them work. Not only this creates huge stress on the system resources (process / thread is not free), it also requries additional synchronization effort (wait, thread join etc). Hence, these solutions only get partial credit.**

d. **Forces child process to inherit fully / partly the parent's cpu statistic.**

Rationale: By inheriting the CPU usage stats from parent, the child effectively are continuing its parent's "life" and will be caught cheating in this manner.
Answer that are close but too impratical / hard to implement get partial credit. For example, couting the usage for "process group", sharing the time quantum between parent and child etc.

Fun fact: Best quote from students, "Parent Criminal = Child Criminal ☺" while refering to the inheritance idea.

Question 7 (8 marks)

In many programming languages, function parameter can be **passed by reference**. Consider this fictional C-like language example:

```
void change( int<Ref> i ) { //i is a pass-by-reference parameter
   i = 1234;   //this changes main's variable myInt in this case
}

int main() {
   int myInt = 0;
   change( myInt );      //myInt become 1234 after the function call
   ......  //other variable declarations and code
}
```

Mr. Holdabeer feels that he has the perfect solution **that works for this example** by relying on **stack pointer and frame pointer.** The key idea is to load main's local variable "`myInt`" whenever the variable "`i`" is used in the change() function.

Given that the stack frame arrangement shown **independently** as follows:

For `Main()`

		← $SP
…	…	
myInt	-12	
Saved SP	-8	← $FP
Saved FP	-4	
Saved PC	0	

For `change()`

		← $SP
Saved SP	-8	← $FP
Saved FP	-4	
Saved PC	0	

a. Suppose the `main()`'s and change()'s stack frame has been properly setup, and `change()` is now executing, show how to store the value "1234" into the right location. You only need pseudo-instructions like below. **[3 marks]**

- `Register_D ← Load Offset( Register_S )`
  Load the value at memory location `[Register_S] + Offset` and put into `Register_D`
  e.g. `$R1 ← Load -4($FP)`
- `Offset(Register_S) ← Store Value`
  Put the `value` into memory location `[Register_S] + Offset`,
  e.g. `-4($FP) ← Store 1234`

b. Briefly describe another usage scenario for pass-by-reference parameter that **will not work with** this approach. **[2 marks]**

c. Briefly describe a better, universal approach to handle pass-by-reference parameter on stack frame. Sketch the stack frame for the `change()` function to illustrate your idea. **[3 marks]**

ANS:

Rationale for this question: I believe most (or majority) of the class will use memory address to implement pass-by-reference if asked directly. It is actually much harder to analyze a solution that "almost work", hence this question. ☺ Turns out to be the hardest question in the paper, with class average on ~4 marks out of 8.

a. $R1 ← Load 4($FP) //get saved FP, i.e. main's FP
-4($R1) ← 1234 //don't forget the offset

Rationale: This is actually a sneaky checks on your understadning of the use of FP or SP pointer. In this case, FP is the correct use as we are not sure about the offset from SP. Secondly, you need to access the main()'s FP via the "saved FP" location. Note that we are not particular about the syntax, you can even write: -4(4($FP)) ← 1234 as answer and still get full marks. ☺

Common mistakes:

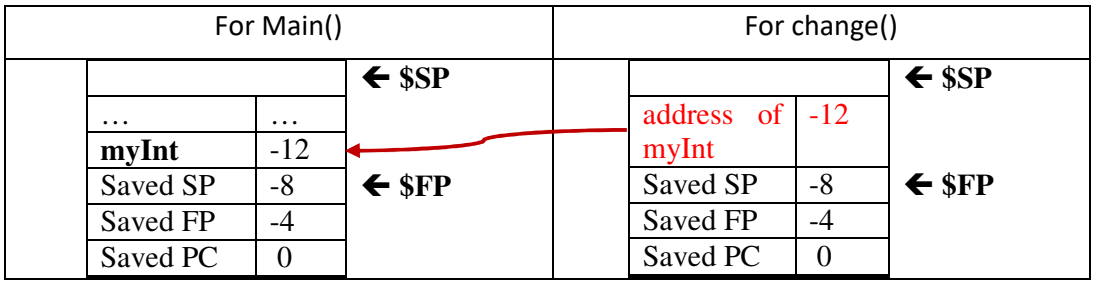
- **Use "saved FP" directly**
- **No offset**
- **Use FP/SP as the intermediates, e.g.**
  $SP ← load -4($FP) ….

b. If the reference is passed to another function as referece, then the scheme breaks down. (As saved FP only point back to the caller)

Rationale: As (a) only works with *direct caller*, it'll break down once you go beyond that.
Common mistakes:

- **Say something like "use change() *repeatedly*", which is actually fine under this scheme. Need to mention "use change() in a chain" or "if change() is a recursive function" etc.**
- **Use non-local variable as argument, e.g. heap data. This is not correct as heap data are pointed by a _local variable_ (a pointer), if you pass that pointer into change(), then this scheme still work.**
- **Use global variable as argument. Since global variable access will be compiled differently (they are accessed via direct address as laid out by compiler). So, this schme is not even applicable.**

c. 2: Place the actual address of the value. Change the instruction to load/store from that address



Question 8 (12 marks)

The code used by this question is on the next page so that you can **tear it out (up?)** to facilitate your answering.

For (a) and (b), we will study the program behavior of **multiple threads.** If the described behavior is possible, give the execution pattern using t<*thread number*>.<*line number*>-<*line number*> ,e.g. **t1.23-25** (thread 1 executing line 23 to 25), **t2.23-24** (thread 2 then execute line 23 and 24. Otherwise, give your answer as "impossible".

a. Suppose we have initialized the global linked list to 2→1 (1 is the last node). Is the following output **possible** if we execute **2 threads on the** printList() function?

[3 marks]

```
2 Nodes
2 Nodes
2 2 1 1
```

b. Suppose we start with an **empty** global linked list, we then execute **2 threads** on the *insertNumbers*(2) function. Can the global linked list contain only 2→1 (i.e. only 2 values) after both threads finish? **[3 marks]**

For (c) and (d), use general semaphore to ensure the indicated program behavior. You should i) **use the least number of semaphore(s)** and ii) protect the **least number of lines of code**. Show the semaphore declarations clearly then use the line number to indicate where you want to insert the semaphore(s) operations (i.e. wait(…) and signal(…)).

c. Suppose we run **10 threads** on the *insertNumbers*(4) function. Use sempahore to ensure that the global linked list has 40 nodes and the global node count = 40 after all threads finish. **[3 marks]**

d. Suppose we run multiple threads on the *insertNumbers*() function and multiple threads on the *printList*() function. Ensure the printList() function:
   - The printed number of nodes (i.e. line 24) always matches the total number of node values printed. **[3 marks]**

Ans:

a.

Key points: Interleave the line 24, 25-26 (2 times). So, simplest answer is:

```
t1.24
t2.24
t1.25-26
t2.25-26
t1.25-26
t2.25-26
```

b.

Key points: Interleave between line 12 and 13. (Note: there is no need to ensure *gTotalNode* is 2 according to the question). So, simplest answer is:

```
t1.18-19, 10-12
t2.18-19, 10-12
t1.13-14, 18-19, 10-12
t2.13-14, 18-19, 10-12
```

c.

Best answer:

```
wait( mutex )
line 12-14
signal(mutex)
```

Rationale: The above protects the minimal number of lines.
Common mistakes:

- **Protect the entire function (less severe), Protect the entire for-loop (severe)**

d.

Key point: This question requires the answer from (c).
Best answer:

```
wait(mutex)
line 12-14
signal(mutex)

AND

wait(mutex)
24-27
signal(mutex)
```

Rationale: **It is not obvious that the insertion can cause problem. However, note the requirement says "The total number of values printed must match the "total nodes value" printed". This essentially restrits that there is no insertion during printing.**

Common mistakes:

- **Protect only one of the two functions**
- **Introduce additional behavior NOT specified, e.g. ordering between printing and insertion, deadlock, full parallelism (i.e. turn this into reader / writer problem).**

Section 3: Bonus Question (1 mark)

9. "***This is a massacre!":*** Predict the **class average score of this paper** (excluding this bonus question). If your prediction is with ±1 mark of actual average, you will get a bonus "*I can see the future*" 1 mark. ☺

ANS:

The paper is actually not a massacre. Class average is at ~23.2, which is "normal" or "quite good!" if you use past year yardsticks ☺. As promised, I awarded 1 bonus mark for anyone who wrote 22-24 (inclusive) in the bonus box.

Tons of Statistics:

- **Raw score: Average (23.11), Median (23.5), Std.Dev (6.8), Max(37.5)**
- **Time taken: Q6 (5 hours), Q7 (3.5 hours), Q8(4 hours), MRQs(2.5 hours), Post-processing (5 hours) = 20 hours T.T**

Code for Question 8 (You can tear out this page for ease of answering)

Below is a modified code from the "linked list lab". The main differences are:

- The linked list is now maintained by a **global head pointer gHead** (line 6).
- There is a global count of number of nodes, **`gTotalNode`** (line 7).
- The insertion always insert at head position (i.e. first position) (line 9 to 15).
- There is a helper function to insert a series of number 1…N at the head of linked list (line 17 to 21).

```
1   typedef struct NODE {
2       int data;
3       struct NODE* next;
4   } node;
5
6   node* gHead = NULL;
7   int gTotalNode = 0;
8
9   void insertHead( int newData ) {
10      node* newNode = (node*)malloc(sizeof(node));
11      newNode->data = newData;
12      newNode->next = gHead;
13      gHead = newNode;
14      gTotalNode++;
15  }
16
17  void insertNumbers( int N ) {
18     for (int i = 1; i <= N; i++){
19        insertHead(i);
20     }
21  }
22
23  void printList() {
24      printf("%d Nodes\n", gTotalNode);
25      for ( node* ptr = gHead; ptr != NULL; ptr = ptr->next)
26          printf("%i ", ptr->data);
27      printf("\n");
28  }
29
```