XOBSNATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM TEST

SOLUTIONS

AY2021/22 Semester 1

CS2106 – INTRODUCTION TO OPERATING SYSTEMS

September 2021 Time Allowed: **1 hour**

INSTRUCTIONS

1. This question paper contains **SEVENTEEN (17)** questions and comprises **TWELVE (12)** printed pages.
2. Maximum score is **40 marks** and counts towards 20% of CS2106 grade.
3. Write legibly with a pen or pencil. **MCQ form should be filled out using pencil.**
4. This is an **OPEN BOOK** test. You may consult any printed materials, but electronic devices are not permitted.
5. Write your **STUDENT NUMBER** below with a pen.

A								

Questions in Part B	**Marks**
15	**/4**
16	**/6**
17	**/2**
Total	**/12**

Part A: MCQ questions (2 marks each question)

1. Which of the statement(s) is/are TRUE regarding OS, user programs, and hardware?
   - (i) You need an OS to run any program.
   - (ii) An OS manages hardware resources for user programs.
   - (iii) An OS can protect a user program from other malicious applications.

   A. (i), (ii), and (iii)
   B. (i) and (ii)
   C. (ii) and (iii)
   D. (iii) only
   E. None of the above

Answer: C.

(i) is incorrect: you can write a program to run without an OS

2. 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 on the operating system.
   - (ii) A system call may be invoked during a library call.
   - (iii) If a system call and library call both execute approximately the same code, the library call is likely to take a longer amount of time to complete, as the system call is lower-level.

   A. (i) only.
   B. (i) and (ii) only.
   C. (ii) and (iii) only.
   D. (i), (ii) and (iii).
   E. None of the above.

Answer: **B** (iii incorrect, system call will take longer, context switch)

Tests library vs system call understanding. System calls are more expensive (all else being equal) due to context switching.

Note: modified from CS2106 15/16 S2 Midterm

3. Consider the following program:

```
Line#   Code
1       int main() {
2           f1(10);
3       }
4
5       void f1(int x) {
6           if (x == 0) {
7               /* How many stack frames? */
8               return;
9           }
10          return f2(x);
11      }
12
13      void f2(int x) {
14          f1(x/2);
15          return;
16      }
```

Given that before main() calls f1(10), there is one stack frame (for the main() function) on the stack, how many stack frames are on the stack at line 7?

A. 5
B. 6
C. 9
D. 10
E. None of the above

Answer: D

f1() is called 5 times, f2() is called 4 times, plus the stack frame for main(), so a total of 10 stack frames.

4. Consider the following code:

```
while (fork() != 0) {
   execl("a.out", "a.out", NULL);
   execl("b.out", "b.out", NULL);
}
```

Recall that a "fork bomb" occurs when more and more processes are uncontrollably created. Assume that the code compiles correctly, and assume if a.out and b.out are valid executables, they simply print a line and exit. Which statement is TRUE?

A. When a.out is a valid executable and b.out is invalid, this code is a "fork bomb"
B. When b.out is a valid executable and a.out is invalid, this code is a "fork bomb"
C. Regardless of whether a.out or b.out are valid executables, this code is a "fork bomb"
D. Regardless of whether a.out or b.out are valid executables, this code is NOT a "fork bomb"
E. None of the above

Answer: **D / E**

(only if a.out and b.out are BOTH INVALID executables, then this code is a fork bomb. This option is not stated.)

Further notes: definition of a fork bomb is somewhat ambiguous. So D/E both can be correct.

Note: modified from CS2106 15/16 S2 Midterm

5. Consider the following code:

```
int globalVar = 0;

for (int i = 0; i < 1000; i++) {
  if (fork() == 0) {
    globalVar++;
    break;
  } else {
    globalVar--;
  }
}
// Point α
```

Note that `break` exits the closest enclosing loop. What is/are possible final value(s) of `globalVar` at Point α?

A. 1000
B. -1000
C. 990
D. 0
E. All of the above

Answer: **nullify (full marks for everyone)**

Main realization should be that globalVar is duplicated across each process.

It can either be -1000 for the parent process, or 1 for any of the child processes (since they each break immediately after adding 1).

Further notes: **0 is also a possible answer – parent does globalVar--, next fork() child does globalVar++ to get 0.**

6. Which of the following statement(s) about zombie and orphan processes is/are TRUE?
   (i) A process after exit() will always transition to zombie state.
   (ii) If the parent process calls wait() (and wait() returns successfully) after forking exactly one child, its child process will never become an orphan.
   (iii) A process can stay in zombie state forever.

A. (i), (ii), and (iii)
B. (i) only
C. (i) and (ii)
D. (ii) and (iii)
E. None of the above

Answer: A

(i) is obvious.

(ii): wait() will ensure that the parent will be blocked until the child terminates, so the child will not become an orphan.

(iii): if the parent never exits, the child process will stay in zombie state forever.

7. A process executes the following code:

Line#	Code
1	`fork();`
2	`if (fork() != 0) { exit(0); }`
3	`fork();`
4	`fork();`
5	
6	`// Point α`

How many processes will reach **Point α**?

A. 4
B. 8
C. 16
D. 32
E. None of the above

Answer: **B**

2 processes after the first line. Both run the second line, and both parent processes are killed, leaving two processes still. Both run fork(), resulting in 4 processes, and all 4 run fork() again, resulting in 8 processes. The intuition is that a fork() followed by all parents or all children exiting retains the same number of processes overall (as if the line didn't happen).

8. Which of the following statement(s) about system calls and interrupts is/are TRUE?
    (i) System calls and interrupts all require mode switches (user <-> kernel).
    (ii) System calls, like interrupts, can happen asynchronously.
    (iii) System calls and interrupt handlers use a different hardware context as user programs.

    A. (i), (ii), and (iii)
    B. (i) only
    C. (i) and (ii)
    D. (ii) and (iii)
    E. None of the above

Answer: B/E

(i) and (iii) are correct. (ii) system calls can only happen synchronously.

No credit for selecting (ii) as true (A, C, D).

9. Which of the following statement(s) about scheduling algorithms is/are TRUE?
    (i) Using a priority based scheduling algorithm allows a higher priority job to always pre-empt a lower priority job
    (ii) Shortest Remaining Time (SRT) can improve the average turnaround time over Shortest Job First (SJT)
    (iii) Round Robin (RR) can improve the average turnaround time over First-Come First-Serve (FCFS)

    A. (i), (ii), and (iii)
    B. (ii) only
    C. (i) and (ii)
    D. (ii) and (iii)
    E. None of the above

Answer: D

(i) is false because of priority inversion. (ii) and (iii) are true as pre-emptive scheduling algorithms can help shorter tasks that arrive later turn around faster instead of being blocked forever by a long task.

10. Which of the following statements is/are TRUE about changing the time quantum and ITI for a round-robin scheduler? Assume that the time quantum is always a multiple of the ITI.

A. Decreasing the ITI while keeping the total time quantum constant reduces the percentage of CPU time that the scheduler takes to run its own code.
B. Increasing the total time quantum while keeping the ITI constant is likely to reduce the response time for interactive processes.
C. A system with only long running CPU-bound tasks would prefer a short ITI to maximize throughput.
D. All of the above
E. None of the above

Answer: **E**

A: FALSE - decreasing ITI ⇒ scheduler runs more often ⇒ scheduler takes a larger percentage of the CPU time

B: FALSE - increasing time quantum ⇒ processes not currently running have to wait longer after I/O ⇒ increased response time

C: FALSE - a batch system would prefer that the ITI is long so that less CPU % is taken by the scheduler

Therefore the answer is **E**

11. A Shortest Remaining Time (SRT) scheduler is running on a batch-processing system, using estimated task durations through exponential averaging. New tasks can arrive at any time. Which of the statements below are TRUE in this scenario?

    (i) Tasks may starve
    (ii) The scheduler guarantees the smallest average waiting time
    (iii) Priority inversion is possible

    A. (i) only.
    B. (i) and (ii) only.
    C. (ii) and (iii) only.
    D. (i), (ii) and (iii).
    E. None of the above.

Answer: **A**

(i) TRUE: SRTF can starve longer jobs in favour of shorter ones

(ii) FALSE: there is no such guarantee, especially for a non-fixed set of tasks + estimated durations

(iii): FALSE: no priorities involved

12. Consider the following scenario.

    Assume that Process A is performing a long-running computation that generates one integer result periodically. Each time a result is generated, Process A wants to transmit a single integer to Process B, with a total of 10,000 integers over the lifetime of the program. Both processes know that 10,000 integers will be generated and transmitted.

    Process A may transmit *each* of the 10,000 integers by either:

    a) Placing it in a shared memory region that Process B can access.
    b) Using `send` in Process A and a blocking `receive` in Process B.

    Which of the statements below are TRUE?

    (i) It is possible for Process B to receive all 10,000 integers from Process A correctly over shared memory without using semaphores.
    (ii) After all 10,000 integers are transmitted, the total number of syscalls executed using message passing with send/receive is less than using shared memory.
    (iii) If Process B is waiting for an integer with the receive call, it is actively running on the CPU.

    A. (i) only.
    B. (i) and (ii) only.
    C. (ii) and (iii) only.
    D. (i), (ii) and (iii).

E. None of the above.

Answer: **A**

This is testing the concept of shared memory vs message passing + what it means to be blocked.

(i) TRUE: Possible to do this without semaphores: an inefficient solution would be to have one “control” value per integer, so have two arrays of 10,000 integers each. Array 1 is for the actual values, and Array 2 is the control values. Array2[i] == 1 ⇒ Array1[i] contains a transmitted value for Process B to read. We show a simple version of this in the IPC lecture, slide 10.

(ii) FALSE: send/receive are syscalls, whereas writing to shared memory does not require syscalls.

(iii) We mentioned a blocking receive, so students should know that BLOCKED != RUNNING.

13. Assume we have a single-core system that supports simultaneous multithreading. Process P executes 1 CPU intensive user thread that can be mapped to any of 25 kernel threads. Assume that process P is the only process in the system. How many threads can, at most, be in the RUNNING state at the same time?

A. 1
B. 2
C. 25
D. 50
E. None of the above

Answer: **A**

Since we have single-core with SMT, we can theoretically execute 2 threads simultaneously. However, we only have 1 user-thread worth of workload, even though it can be assigned any of 25 kernel threads. The maximum number of running threads is therefore just 1

14. Which of the following statement(s) about threads is/are **FALSE**?

(i) Unlike parent-child processes, for two threads in the same process, updates to function local variables from one thread are visible to the other thread.

(ii) In the pure user-thread model, when the OS interrupts a process (e.g., timer interrupt), the OS can pick a different (ready) thread in the process to run.

(iii) Using a hybrid thread model, if we bind N user threads to a kernel thread, when one of the N user threads is blocked on I/O, the remaining N-1 user threads can still run.

A. (i) only
B. (i) and (ii)
C. (ii) and (iii)
D. (i), (ii), and (iii)
E. None of the above

Answer: **C partial credit(1 mark) / D (2 marks)**.

(i): **ambiguous –** stacks are technically in same memory space, but lecture diagram shows them sort of separated?. (ii): the OS has no knowledge of user threads, and cannot pick another user thread to run. (iii): the OS can only schedule kernel threads. If one of the user threads is blocked, the OS will pick a different kernel thread to run, thus all N-1 remaining threads are also blocked.

Part B. Short Questions

15. Answer each of the following questions briefly. State your assumptions, if any

(i) [2 marks] If an OS using Shortest Job First (SJF) **cannot** predict task CPU time accurately, explain (or give a simple example) why SJF can result in longer average wait time than First-Come First-Serve (FCFS).

You should see a category letter (e.g. A, B1/2/3, C, etc.) on the top-left of the question in Softmark. This corresponds to the letters below.

There were two main arguments that were accepted:

- **A**: Faulty SJF leads to it scheduling longer jobs before shorter jobs. If the shorter jobs had arrived earlier than the longer jobs, FCFS would have scheduled them earlier, and this leads to SJF having a longer average wait time than FCFS.
  - A simple example that demonstrates this scenario is sufficient for full credit. E.g. two tasks A, B arrive in that order at about the same time. A requires 1 TU of CPU time, and B requires 20 TU. SJF mispredicts and selects B to run first.
    - There were occasional errors in waiting time calculations, for those answers that provided an example. As long as the example itself works, this was not penalised.
    - There were many unnecessarily complicated examples that involved 3 or more tasks and complicated arrival schedules.
    - If a partially faulty example is given (e.g. an arrival schedule that leads to only one possible schedule regardless of scheduler, but that would be correct if all the tasks had arrived at about the same time instead), partial credit is awarded (**B3**).
  - The answer **must** show how FCFS can lead to a shorter waiting time than SJF (e.g. “jobs arrive in order of job length”). If the answer simply talks about SJF without clearly showing how FCFS might lead to a shorter average waiting time, partial credit is awarded (**B1**).
  - If the answer talks about FCFS potentially having a shorter waiting time without clearly showing how SJF can lead to a longer waiting time, partial credit is awarded (**B2**).
- **D**: If new short jobs keep arriving, SJF would repeatedly pick the new short jobs over a waiting long job, leading to starvation. FCFS would serve the jobs in arrival order, and is guaranteed not to have starvation.
  - The implication is that SJF might lead to an unbounded wait time, whereas FCFS has a bounded wait time.
  - The answer must justify why SJF leads to starvation (e.g. new short jobs keep arriving). If the answer simply asserts that SJF can have starvation, partial credit is awarded (**E1**).

    - If the answer talks about SJF having starvation without relating it to FCFS, partial credit is awarded (**E2**).

The following answers were not awarded credit:

- **C**: Arguments based on SJF having higher overhead than FCFS.
    - A reasonable scheduler implementation will not have overheads that are appreciable in any way.
- **X**: Arguments based on wrong assumptions or facts.
    - The most common such argument was that SJF will pre-empt a currently running task if a shorter task arrives, leading to the currently running task having to wait.
        - SJF is not preemptive!
- **Y**: Arguments that could not be understood.
- **XX**: Blank.

(ii) [2 marks] There are 2 threads, and both threads need to acquire mutex A, mutex B, and mutex C before they can enter a critical section. How do you ensure the 2 threads never result in a deadlock?

All threads need to wait on the N mutexes in the same order.

Accepted solutions

(i) Solutions that can show an explicit example of a working solution for preventing deadlock are accepted. This includes wrapping the entire section with a new mutex, wrapping the mutex acquisition with a new mutex as well as busy waiting.

(ii) Solutions that prevent deadlock via busy waiting but also violate independence property are also accepted as the question did not state the requirement to be independent. This refers to solutions that uses a “Turn” variable. Solutions such as these require sufficient explanation or an explicit example to show how the “Turn” variable is implemented.

(iii) Solutions that do not have explicit examples require sufficient explanation with enough detail.

Rejected solutions

(i) Solutions that result in deadlock.

(ii) Solutions that describe the meaning of deadlock and why deadlocks happen.

(iii) Solutions that state that mutexes need to be released to prevent a deadlock.

(iv) Solutions that simply states to use a “turn variable” or “busy waiting” without sufficient explanation of its implementation. Deadlocks can still happen when the turn variable is implemented in the wrong way.

(v) Solutions that change the question requirements. Examples are solutions that

   i. Uses A, B, C as semaphores
   ii. Changes A, B, C to other types such as integers or Booleans
   iii. Sets A, B, C to non-binary values
   iv. Not all threads acquires mutex A, B, and C (Thread 1 acquire A and B, Thread 2 acquire C)
   v. Releases the mutex before finishing the critical section

(vi) Solutions that incorrectly use mutex functions and mutexes. Examples are

   i. While (wait(A)) {…}
   ii. While(A) {…}

16. Consider the following program consisting of a `main` function and `f`.

Line#	Code for part (a)	
1	`void f() {`	`void main() {`
2	`int i;`	`int i;`
3	`i = 1234;`	
4	`}`	`for (i = 0; i <= 1000; i++)`
5		`f();`
6		`}`

a. [2 marks] When we run `main()`, what is the **maximum number** of stack frames of function **`f()`** that are on the stack at the same time? Explain your answer.

Only 1. When f() returns each time, its stack frame is torn down.

Grading scheme:

1 mark for anything that indicated one frame of f() i.e., we accepted answers like "2 frames" if something like "1 frame of f() and 1 frame of main()" was mentioned.

1 mark for explaining in any way that the stack frame is torn down after each iteration.

Some students incorrectly believed that the lack of an explicit return meant that stack frames are not cleared up.

Line#	Code for part (b)	
1	`void f(int i) {`	`void main() {`
2	`if (i == 0)`	`int i;`
3	`return 1;`	
4	`else`	`for (i = 0; i <= 1000; i++)`
5	`return i * f(i-1);`	`f(i);`
6	`}`	`}`

b. [2 marks] Assume we modify `f()` and `main()` as above. When we run `main()`, what is the **maximum number** of stack frames of function **`f()`** that are on the stack at the same time? Explain your answer.

1001. The largest call of f(i) is f(1000), which has 1001 stack frames right before the return is called. As an example, imagine f(1), which must call f(0) before returning, so it's (i+1) stack frames at the same time.

Grading scheme:

1 mark for anything that indicated 1001 frames of f() i.e., we accepted answers like "1002 frames" if something like "1001 frame of f() and 1 frame of main()" was mentioned.

1 mark for explaining in any way that recursion increases the number of stack frames.

For both (a) and (b) we gave marks even if students had *diagrams* that indicated such understanding.

Line#	Code for part (c)	
1	`void factorial(int i, int acc){`	`void main() {`
2	`  if (i == 0)`	`  int i;`
3	`    return acc;`	`  factorial(1000, 1);`
4	`  else`	`}`
5	`    factorial(i - 1, acc * i);`	
6	`}`	

c. [2 marks] Assume we create a factorial function by modifying f as above. We claim that a smart compiler can create and reuse only one stack frame for all invocations of **`factorial`**, e.g., if we call **`factorial(1000, ...)`**, **`factorial(999, ...)`** will reuse the stack frame that **`factorial(1000, ...)`** used.

Why can a compiler do this for the code in part (c) but not part (b)?

This is a slightly challenging question to test if students can explain tail-call optimization. Answer: for c), no information about the caller must be saved for the callee, as **acc** contains all the information needed to return at i==0, so we can reuse the stack frame while just replacing the values of i and acc.

Grading scheme:

1 mark: Answers that mentioned tail call optimization but did not sufficiently explain after that, 2 marks for a complete explanation involving TCO.

OR

Broadly: 1 mark: Accumulator/results are stored completely in params, 1 mark: no need to store previous results

There are many wordings, so the priority was on logical explanations (i.e., not just keywords).

17. [2 marks] Dr. Gru has invented a new atomic function:

```
int fetch_and_increment(int *addr);
```

The function increments the integer stored at the memory location and returns the value before the increment, in an **atomic** step. For example, if *a= 5, `fetch_and_increment(a)` will return 5 and increment *a to 6 atomically. You can assume the function is implemented correctly.

With this new atomic function, Dr. Gru came up with a new lock implementation:

Dr. Gru's new lock implementation

```
struct lock {
    int ticket;
    int turn;
}

void lock_init(struct lock *l) {
    l->ticket = 0;
    l->turn = 0;

}


void lock_acquire(struct lock *l) {
    int my_ticket = fetch_and_increment(&l->ticket);


    while (my_ticket != l->turn) {

    }

}


void lock_release(struct lock *l) {
    l->turn++;
}
```

Does Dr. Gru's lock implementation ensure mutual exclusion? If yes, explain how it enforces mutual exclusion. If not, give an example of how mutual exclusion can be violated.

Accepted answers (2 marks):

(i) Yes. **Atomicity** of fetch_and_increment guarantees that every thread that tries to acquire a lock, even when concurrently, will get a **unique** tick number. Consequently, line 11-12 ensures that at most one thread can enter the critical section at a time, and only after that thread releases the lock, will the next thread holding the next ticket number (at most one thread can hold that ticket number) be able to acquire the lock.

(ii) No. If there are more than 2^32 threads calling lock_acquire at the same time, more than one of them will receive the same ticket number due to integer overflow. Threads holding the same ticket number can enter the critical section at the same time, violating mutual exclusion.

Partial credit (1 mark):

(i) Only explain why the first thread can enter the critical section while all other threads are being blocked. Does not justify why only one of the blocked threads can enter the critical section after the lock is released.

(ii) Only explain mutual exclusion for 2 threads, not the general case.

(iii) Only mention fetch_and_increment is atomic, but did not elaborate why atomicity guarantees mutual exclusion.

Not accepted answers (0 mark):

(i) "yes" or "no" without any justification.

(ii) Wrong counter example for mutual exclusion violation.

(iii) Only mentioned threads will be busy-waiting in the while loop.

(iv) Reasoning for mutual exclusion is incorrect.

--End of paper--