NATIONAL UNIVERSITY OF SINGAPORE
SCHOOL OF COMPUTING

MIDTERM TEST

REFERENCE SOLUTION

AY2025/26 Semester 1

CS2106 – INTRODUCTION TO OPERATING SYSTEMS

October 2025 Time Allowed: **1 hour**

INSTRUCTIONS

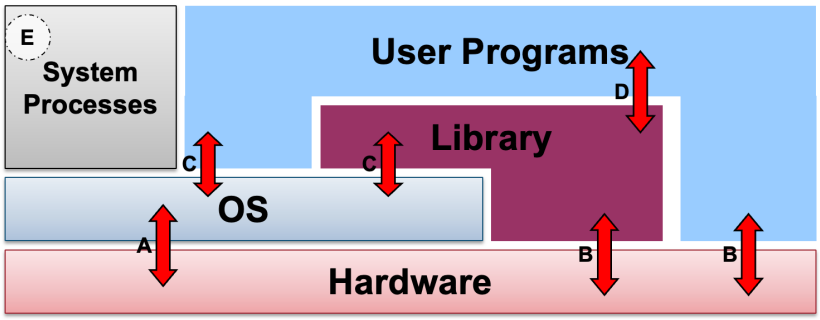
1. This question paper contains **TEN (10) MCQ** questions and **TWO (2) short answer** questions and comprises **THIRTEEN (13)** printed pages.
2. Maximum score is **40 marks** and counts towards 20% of CS2106 grade.
3. **MCQ form should be filled out using pencil.**
4. This is a **CLOSED-BOOK** test. You are allowed to bring one A4-sized cheat sheet.
5. Write your **STUDENT NUMBER** below with a pen.

A								

MCQ questions (2 marks each question)

1. In a microkernel, which functionality is typically kept inside the kernel rather than moved to user-space?
   A. File system
   B. Device drivers
   C. Interrupt handler
   D. Memory management
   E. None of the above (A-D)

2. Based on the following diagram. A C program calls fprintf() in the standard library to write to standard output (assuming the content actually prints on the screen). Which sequence best describes the interaction across layers? Note that the program needs to execute some instructions to prepare for the function invocation (e.g., setup the stack).



   A. B only
   B. B -> D -> B
   C. B -> C -> A
   D. B -> D -> B -> C -> A
   E. None of the above (A-D)

   The C program executes code on CPU hardware: B
   The program invokes fprintf(), which is a user space library: D
   fprintf() executes code on CPU hardware: B
   fprintf() invokes write() system call: C
   write() system call in kernel mode executes code on CPU hardware: A

3. Consider the following program:

Line#	Code
1	`int f1(int x) {`
2	`    if (x <= 1) {`
3	`        /* How many stack frames? */`
4	`        return 0;`
5	`    }`
6	`    f2(x);`
7	`    return f1(x - 2);`
8	`}`
9	
10	`void f2(int y) {`
11	`    if (y % 2 == 0) return;`
12	`    f1(y - 3);`
13	`}`
14	
15	`int main() {`
16	`   f1(5);`
17	`   return 0;`
18	`}`

Given that before main() calls f1(5), there is one stack frame (for the main() function) on the stack, how many stack frames are on the stack at line 3 (for the first time)?

A. 4
B. 5
C. 6
D. 7
E. None of the above (A-D)

4. During the lecture, we covered callee-saved registers. Many calling convention also includes caller-saved registers. When calling a function, the caller will save the value of all caller-saved registers on its stack frame. When the function returns, the caller will restore the caller-saved registers from its stack frame.

   For this question, suppose registers $t0, $t1 are caller-saved registers, $t2, $t3 are callee-saved registers. Consider a program in which main() calls F(), and F() calls G() (G does not call other functions). main() uses registers $t0, $t1, and $t2, F() uses registers $t1 and $t2, and G() uses registers $t0, $t2, and $t3. Suppose the calling convention saves both caller-saved and callee-saved registers. Which registers are saved on F() and G()'s stacks?

   A. F(): $t1, $t2 G(): $t0, $t2, $t3
   B. F(): $t2 G(): $t2, $t3
   C. F(): $t0, $t1, $t2 G(): $t2, $t3
   D. F(): $t0, $t1 G(): $t0, $t1
   E. None of the above (A-D)

5. A process executes the following code:

Line#	Code
1	`void fn() {`
2	`    if (fork() > 0) {`
3	`        if (fork() == 0) {`
4	`        }`
5	`    } else {`
6	`        // ./worker is a different, valid program`
7	`        execl("./worker", "worker", NULL);`
8	`    }`
9	`}`
10	`int main() {`
11	`    if (fork() == 0) {`
12	`        fn();`
13	`    } else {`
14	`        if (fork() != 0) {`
15	`            fn();`
16	`        } else {`
17	`            // ./worker is a different, valid program`
18	`            execl("./worker", "worker", NULL);`
19	`        }`
20	`    }`
21	`    fork();`
22	`    // Point α`
23	`}`

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

A. 6
B. 8
C. 10
D. 12
E. None of the above (A-D)

6. Consider the following C code:

Line#	Code
1	`pid_t p, q, r;`
2	`int st, st1, st2;`
3	`printf("A ");`
4	`if ((p = fork()) == 0) {`
5	`    printf("B ");`
6	`    if ((q = fork()) == 0) {`
7	`        printf("C ");`
8	`        exit(3);`
9	`    } else {`
10	`        waitpid(q, &st, 0);`
11	`        printf("%d ", WEXITSTATUS(st) + 1);`
12	`        exit(2);`
13	`    }`
14	`} else {`
15	`    waitpid(p, &st1, 0);`
16	`    printf("%d ", WEXITSTATUS(st1));`
17	`    if ((r = fork()) == 0) {`
18	`        printf("D ");`
19	`        exit(1);`
20	`    } else {`
21	`        waitpid(r, &st2, 0);`
22	`        printf("%d\n", WEXITSTATUS(st2));`
23	`    }`
24	`}`

Similar to the wait() system call, waitpid(pid_t pid, int *status, int options) blocks the process until its child process terminates. The difference is that waitpid() blocks until the child process with the **specified pid** terminates (not any child). You can ignore the options parameter for this question. WEXITSTATUS is a macro that evaluates the exit status supplied in the corresponding exit() call.

Which of the following is a correct output of the above program? (\n is the new-line character that is invisible but causes printing to the next line)

A. A B 2 C 4 D 1\n
B. A B C 4 2 D 1\n
C. B A C 4 2 D 1\n
D. A C B 4 2 D 1\n
E. Output is non-deterministic

For questions 7 and 8, consider the following set of tasks:

Task	Arrival Time (TU)	Total CPU Time (TU)
A	0	8
B	1	4
C	2	9
D	3	5
E	6	2

7. We use the Shortest Remaining Time (SRT) scheduling algorithm. If there is any tie, we use FCFS to break the tie.

   What is the task completion order?

   A. B, E, D, A, C
   B. B, E, D, C, A
   C. A, B, D, E, C
   D. E, B, D, A, C
   E. None of the above (A-D)

8. We now switch to the **preemptive** version of MLFQ scheduling algorithm. Suppose there are three priority levels, and the scheduling Time Quantum is 4 TU. What is the turnaround time of task C?

   A. 23 TU
   B. 25 TU
   C. 26 TU
   D. 28 TU
   E. None of the above (A-D)

9. Consider the following C code:

Line#	Code
1	`int x = 0, y = 0;`
2	`Semaphore mutex = 1;`
3	`void T() {`
4	`    wait(mutex);`
5	`    x += 1;`
6	`    y += 2;`
7	`    signal(mutex);`
8	`    y += 1;`
9	`    wait(mutex);`
10	`    x += 2;`
11	`    signal(mutex);`
12	`}`
13	`void U() {`
14	`    y += 1;`
15	`    wait(mutex);`
16	`    x = 2 * x;`
17	`    signal(mutex);`
	`}`

The main function spawns two threads, one executing T() and the other executing U(). The main function then waits for both threads to finish. When both threads finish, which final x, y values are **NOT** possible?

A. x = 6, y = 3
B. x = 4, y = 4
C. x = 4, y = 3
D. x = 3, y = 3
E. All of A-D are possible

10. Suppose there are two binary semaphores S and T. Both S and T are initialized to 0. Now consider running the following three threads (pseudocode shown):

```
Thread A:
print("1 ");
signal(S);
wait(T);
print("4 ");

Thread B:
wait(S);
print("2 ");
signal(S);
signal(T);

Thread C:
wait(S);
print("3 ");
```

Which statement about the possible outcomes is correct? Here, complete output means the three threads finish executing.

A. One possible complete output is 1 2 3 4 , but the program may also deadlock
B. One possible complete output is 1 2 4 3 , and the program cannot deadlock
C. One possible complete output is 1 3 2 4
D. The program may deadlock after printing 1 2
E. None of the above options (A-D) are correct

Short answer questions (10 marks each question)

1.

Line#	**Code for part (a)**
1	`void g(int n) {`
2	`    if (n > 0) {`
3	`        g(n - 1);`
4	`        g(n - 1);`
5	`    }`
6	`}`
7	
8	`void f(int n) {`
9	`    if (n > 0) {`
10	`        g(n);`
11	`        f(n - 1);`
12	`    }`
13	`}`
14	
15	`int main() {`
16	`    f(3);`
17	`    return 0;`
18	`}`

a. [5 marks] When we run `main()`, what is the **maximum number** of stack frames of function **`g()`** that are on the stack at the same time? Explain your answer.

4

main() -> f(3) -> g(3) -> g(2) -> g(1) -> g(0)

Line#	**Code for part (b)**
1	`void fn(int n, int *result) {`
2	`    if (n >= 2) {`
3	`        int temp = n;`
4	`        fn(n - 1, &temp);`
5	`        fn(n - 2, &temp);`
6	`        *result += temp;`
7	`    } else {`
8	`        *result += 1;`
9	`    }`
10	`}`
11	
12	`int main() {`
13	`    int result = 0;`
14	`    fn(3, &result);`
15	`    printf("Result: %d\n", result);`
16	`    return 0;`
17	`}`

b. [5 marks] When we run main(), what will be the output of the program? Explain your answer.

8

Each fn() call allocates a new temp variable on its stack. The *result pointer in a callee points to the temp stack variable on the caller's stack.

2. a. [7 marks] The main() function spawns three threads to execute the following three functions. Complete the code such that the program guarantees to output the exact sequence: A B C A B C A B C . The program should never deadlock and always print this sequence. You can declare and use semaphores, but you cannot use busy waiting.

```
// insert variable declaration and initialization here
Semaphore r = 1, s = 0, t = 0;
void ThreadA() {

  for (int i = 0; i < 3; i++) {
    wait(r);
    printf("A ");
    signal(s);
  }

}
void ThreadB() {

  for (int i = 0; i < 3; i++) {
    wait(s);
    printf("B ");
    signal(t);
  }

}
void ThreadC() {

  for (int i = 0; i < 3; i++) {
    wait(t);
    printf("C ");
    signal(r);
  }

}
```

b. [3 marks] We covered the consumer producer problem during the lecture. Below is the blocking version of the algorithm (no busy wait). Is it okay if we switch the order between wait(notFull) and wait(mutex), that is, in the producer code, we first do wait(mutex) and then do wait(notFull)? If yes, explain why. If no, give a execution sequence that can lead to a problem.

```
while (TRUE) {
        Produce Item;

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

Producer Process

```
while (TRUE) {

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

        Consume Item;
}
```

Consumer Process

No. When the buffer is full and a producer comes, it will acquire the mutex and also be blocked on notFull. All other consumers will be blocked on mutex, and can't signal notFull. The system is deadlocked.

--End of paper--