

NATIONAL UNIVERSITY OF SINGAPORE
SCHOOL OF COMPUTING

MIDTERM TEST - ANSWERS

AY2019/20 Semester 1

CS2106 – INTRODUCTION TO OPERATING SYSTEMS

October 2019

Time Allowed: **1 hour**

INSTRUCTIONS

1. This question paper contains **SEVENTEEN (17)** questions and comprises **SIXTEEN (16)** 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 a **CLOSED BOOK** test. However, a single-sheet double-sided A4 reference sheet is allowed.
5. Write your **STUDENT NUMBER** below with a pen.

A								
----------	--	--	--	--	--	--	--	--

Questions in Part B	Marks
13	/3
14	/3
15	/7
16	/5
17	/5
Total	/23

Part A: MCQ questions.

Questions 1-12 from this section should be answered using the MCQ bubble form provided. Answers given in the midterm paper will not be considered for grading.

1. [1 mark] What are the advantages of microkernel design over monolithic design in Operating Systems?
 - i. Kernel is generally more robust.
 - ii. Better performance.
 - iii. Kernel is extensible.
 - iv. Better isolation and protection between kernel and process management services.
 - A. i. and ii. only.
 - B. i. and ii. and iii. only.
 - C. ii. and iii. only.
 - D. i., iii. and iv. only.
 - E. iii. and iv. only.
2. [1 mark] Ubuntu 16.04 (on our lab machines) is using a microkernel.
 - A. True.
 - B. False.
 - C. It has elements of both microkernel and monolithic kernel.
 - D. I am not sure.
 - E. I forgot.
3. [1 mark] The modern operating system serves as an abstraction that:
 - i. Allows the user of a computer to directly interact with the hardware
 - ii. Manages and allocates hardware resources to ensure efficient and fair usage
 - iii. Controls the execution of processes
 - iv. Provides security and protection for different executing programs
 - A. ii. and iii. only.
 - B. i. only.
 - C. iii. and iv. only.
 - D. ii., iii. and iv. only.
 - E. None of the above. only.
4. [1 mark] A context switch is performed because process A was “picked” by the OS scheduler for execution. Considering the generic five-states process model, process A transits between the following states:
 - A. Blocked -> Ready
 - B. Blocked -> Running
 - C. Ready -> Running
 - D. Running -> Ready
 - E. None of the above

5. [1 mark] Process A performs the `exit()` system call. Considering the generic five-states process model, process A transits between the following states:
 - A. Running -> Blocked
 - B. Running -> Ready
 - C. Running -> Running
 - D. Ready -> Ready
 - E. None of the above

6. [1 mark] Assume a single processor (core) system and no simultaneous multi-threading (ignore this term if you do not know what it is). Consider the generic five-states process model. How many processes can be in READY state at the same time? How many processes can be in RUNNING state at the same time?
 - A. Exactly one process in READY state and one process in RUNNING state.
 - B. Any number of processes can be in READY state, but at most one process in RUNNING state.
 - C. The operating system decides on the fly how many processes are in READY and RUNNING state.
 - D. Exactly one process is in RUNNING state, while there might be no process in READY state.
 - E. Multiple processes can be in READY state and multiple processes can be in RUNNING state.

7. [2 marks] **Consider the following situations for using the stack pointer and the frame pointer.**

Assume that the stack frame layout is the same as the one given in the lecture. Assume there is an instruction named: `return <offset>(register)` (e.g. `return -4($fp)`) that returns to the address at `offset + <value in register>`.

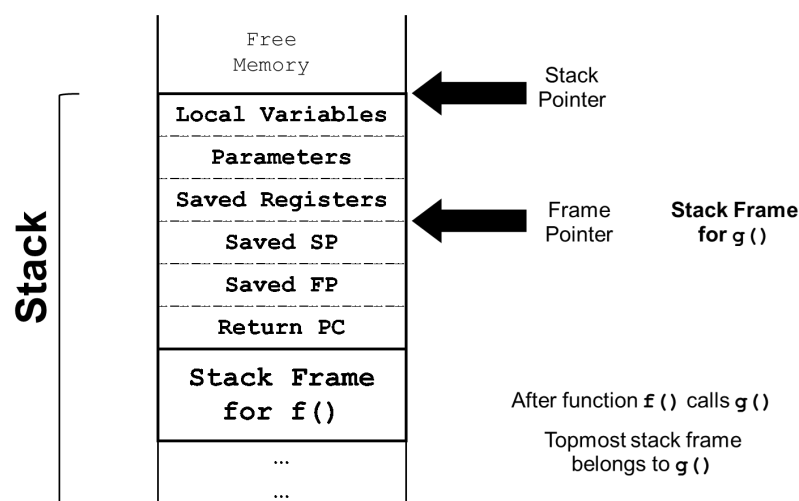


Figure 1: Stack frame layout from lecture notes.

- i. Allocating space for local variables in the middle of a function, e.g.:

Line#	Code Snippet
1	void f(char c, int x) {
2	int a = 5;
3	<some code>
4	int b = 7; // here
5	<more code...>
6	}

- ii. Accessing the arguments passed to the current function
 iii. Returning to the caller function after the stack and frame pointer is restored to the caller's original values.

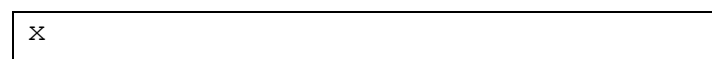
In what situations would we use the stack pointer instead of the frame pointer?

- A. i. only.
 B. ii. and iii. only.
 C. i. and iii. only.
 D. All of the above.
 E. None of the above.
8. [2 marks] C has a feature called variable-length arrays (VLAs). It allows programmers to write code as follows:

Line#	Code Snippet
1	int f(int input) {
2	int arr[input];
3	int x = 5; // Point A
4	return arr[0];
5	}
6	int main(void) {
7	int input;
8	scanf("%d", &input);
9	int retval = f(input);
10	printf("f returned %d\n", retval);
11	}

The highlighted code (line 2) is an array that has space allocated based on user input at runtime. In gcc, **arr** would be allocated **on the stack**.

Assume the stack frame of `f` (at Point A in line 3) looks like the image below given any value of `input`, and the stack grows upwards (towards the top of this page). Also assume that the stack pointer is currently right above `x`, and the frame pointer is currently at `input`.



arr[0]
arr[1]
... rest of arr ...
arr[input - 2]
arr[input - 1]
input
<saved fp, sp, return address, etc>
<main's stack frame>

Figure 2: Stack frame at Point A in line 3.

Consider the following statements for **Point A in line 3**:

- i. `f` can reference the variable `x` using a constant offset from the frame pointer, e.g. `offset($fp)`.
- ii. `f` can reference the value in `arr[0]` using a constant offset from the stack pointer, e.g. `offset($sp)`.
- iii. `f` can reference the variable `x` using a constant offset from the stack pointer, e.g. `offset($sp)`.

Which statements are **true** at Point A in line 3?

- A. i. only
- B. ii. and iii. only
- C. i. and iii. only
- D. All of the above
- E. None of the above

9. [2 marks] The creator of the Linux kernel (Linus Torvalds) has expressed **strong displeasure** about using VLAs in kernel code. What are some valid criticisms of VLAs based on your knowledge of VLAs from **Question 8**?

- i. VLAs are more inconvenient for C programmers to use compared to using `malloc` for the same purpose.
- ii. Using VLAs requires more code to calculate the addresses of locations on the stack frame than a fixed-size array declared at compile time.
- iii. The size of the stack frame changes based on the size of the VLA, which makes it impossible to locate certain variables.

- A. i. only.
- B. ii. only.
- C. ii. and iii. only.
- D. All of the above.
- E. None of the above.

10. [2 marks] Consider two threads executing once the following C code, and accessing shared variables `a`, `b` and `c`:

Line#	Code Snippets
1	Initialization
2	<code>int a = 4;</code>
3	<code>int b = 0;</code>
4	<code>int c = 0;</code>
5	Thread 1
6	<code>b = 10;</code>
7	<code>a = -3;</code>
8	Thread 2
9	<code>if (a < 0) {</code>
10	<code>c = b - a;</code>
11	<code>} else {</code>
12	<code>c = b + a;</code>
13	<code>}</code>

You can assume that each statement can be written using store and load operations (and these operations are not interrupted while running; they are atomic). Furthermore, the order of execution of statements within each thread is preserved by the C compiler and the hardware so it matches the code above.

What are the possible values for variable `c` after both threads complete?

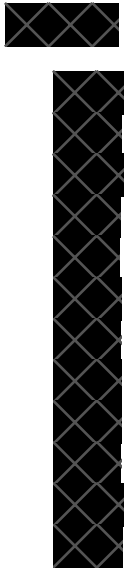
- A. -4, 0, 4.
 - B. -3, -4, 7, 13, 14.
 - C. ~~-3~~, 4, ~~7~~, 13, 14.
 - D. 4, 7, 13, 14.
 - E. None of the above.
11. [1 mark] In Question 10, `c` might have multiple values when both threads complete because the code has:
- A. Critical section.
 - B. Mutual exclusion.
 - C. Deadlock.
 - D. Race condition.
 - E. Exception.

12. [2 marks] In cooperating concurrent tasks, sometimes we need to ensure that all N tasks have reached a certain point in the code before proceeding. This specific synchronization mechanism is commonly known as a **barrier**. Assume that these tasks are executing using threads. Consider the barrier we implemented in Tutorial 5:

Line#	Code Snippets
1	Initialization:
2	int arrived = 0; //shared variable
3	Semaphore mutex = 1;
4	Semaphore waitQ = 0;
5	Barrier(N) {
6	wait(mutex);
7	arrived ++;
8	signal(mutex);
9	
10	if (arrived == N)
11	signal(waitQ);
12	wait(waitQ);
13	signal(waitQ);
14	}

Choose the statement that is **false** about this barrier:

- A. waitQ will always have a value of 1 at the end of the execution of function Barrier (line 13).
- B. This barrier cannot be reused to synchronize a set of N threads because waitQ will not be 0 at the end of the execution.
- C. This barrier cannot be reused to synchronize a set of N threads because arrived will not be 0 at the end of the execution.
- D. We have a race condition because arrived is accessed outside the critical section.
- E. The barrier implementation is correct, and no changes are needed.








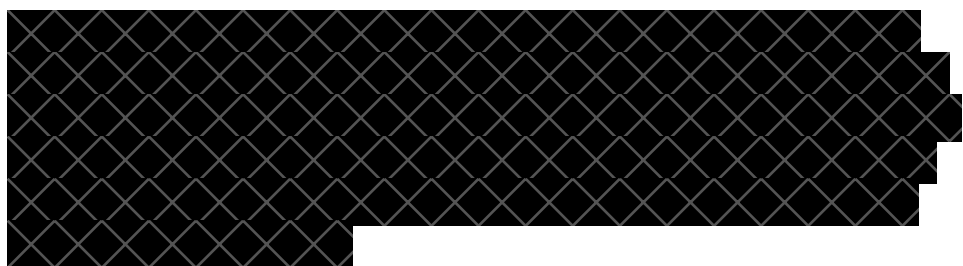




Part B. Short Questions

13. [3 mark] Assume you want to write a reusable barrier: N threads can use the same function `Reusable_barrier()` to block multiple times during their execution and wait for all the other threads to reach the same point. You may assume that all threads use this barrier synchronization in for loop. The code executed by each thread, and the attempt to implement the barrier follow:

Line#	Code Snippets
1	A thread executes the following statements:
2	<code>while (true) {</code>
3	<code> Compute();</code>
4	<code> Reusable_barrier(N);</code>
5	<code> Critical_code_after_barrier();</code>
6	<code>}</code>
7	Initialization for barrier:
8	<code> int arrived = 0; //shared variable</code>
9	<code> Semaphore mutex = 1;</code>
10	<code> Semaphore waitQ = 1; //NOTE the change</code>
11	
12	Reusable_barrier(N) {
13	<code> wait(mutex);</code>
14	<code> if (arrived == 0) {</code>
15	<code> wait(waitQ);</code>
16	<code> }</code>
17	<code> arrived++;</code>
18	<code> if (arrived == N) {</code>
19	<code> arrived = 0;</code>
20	<code> signal(waitQ);</code>
21	<code> }</code>
22	<code> signal(mutex);</code>
23	
24	<code> wait(waitQ);</code>
25	<code> signal(waitQ);</code>
26	<code>}</code>

For each of the following statements state if it is true or false and explain your decision.

True/False	Statement
	a. Exactly one thread will <code>signal (waitQ)</code> in line 20 at each successful use of the barrier.
	Explanation for a.:  
	b. <code>waitQ</code> will have a value of 0 after all threads successfully call the barrier once.
	Explanation for b.:  
	c. Barrier will work properly in the given scenario.
	Explanation for c.:   

14. [3 mark] Assume a similar scenario to Question 13 (but with a different implementation below). The following code has been written to implement the reusable barrier. N threads can use the same function `Reusable_barrier_block()` to block multiple times during their execution and wait for all the other threads to reach the same point. After using `Reusable_barrier_block`, the critical code can be executed, followed by a call to `Reusable_barrier_reset()`.

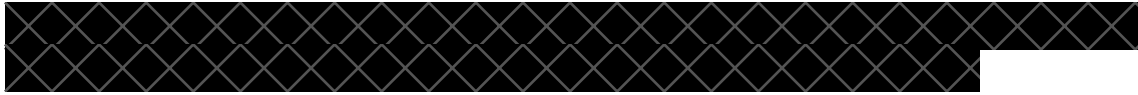
Line#	Code Snippets
1	A thread executes the following statements:
2	while (true) {
3	Compute();
4	Reusable_barrier_block (N);
5	Critical_code_after_barrier();
6	Reusable_barrier_reset (N);
6	}
7	Initialization:
8	int arrived = 0;
9	Semaphore mutex = 1;
10	Semaphore waitQ = 0;
11	Reusable_barrier_block (N) {
12	wait(mutex);
13	arrived ++;
14	if (arrived == N)
15	signal(waitQ);
16	signal(mutex);
17	wait(waitQ);
18	signal(waitQ);
19	}
20	Reusable_barrier_reset(N) {
21	wait(mutex);
22	arrived --;
23	if (arrived == 0)
24	wait (waitQ);
25	signal(mutex);
26	}

For each of the following statements state if it is true or false and explain your decision in one-two sentences.

True/False	Statement
<input checked="" type="checkbox"/>	<p>a. Exactly one thread will <code>wait(waitQ)</code> in line 24 at each successful use of the barrier.</p> <p>Explanation for a.:</p> <p>[REDACTED]</p> <p>[REDACTED]</p>
<input checked="" type="checkbox"/>	<p>b. Some threads might deadlock in <code>Reusable_barrier_reset()</code>.</p> <p>Explanation for b.:</p> <p>[REDACTED]</p> <p>[REDACTED]</p>
<input checked="" type="checkbox"/>	<p>c. Barrier will work properly in the given scenario.</p> <p>Explanation for c.:</p> <p>[REDACTED]</p> <p>[REDACTED]</p> <p>[REDACTED]</p>

15. Answer each of the following questions briefly (in one or two sentences). State your assumptions, if any.

a. [1 mark] Under what conditions does FIFO scheduling result in the shortest possible average response time?



b. [1 mark] Under what conditions does round robin scheduling behave identically to FIFO?



c. [1 mark] Under what conditions does round robin scheduling perform poorly compared to FIFO?



d. [1 mark] In the situation you described in part c. above, does reducing the time slice for round-robin scheduling help or hurt its performance relative to FIFO? Why?

e. [1 mark] Which scheduling algorithm gives a higher priority to I/O-bound processes for the CPU?

f. [2 marks] Do you think that a CPU-bound (CPU intensive) process should be given a higher priority for I/O than an I/O-bound process? Justify your answer.

16. [5 marks] You are required to implement an intra-process mutual exclusion mechanism (a lock) **using Unix pipes**. Your implementation **should not use mutex** (`pthread_mutex`) or semaphore (`sem`), or any other synchronization construct.

Information to refresh your memory:

- In multithreaded processes, file descriptors are shared between all threads in a process. If multiple threads simultaneously call `read()` on a file descriptor, only one thread will be successful in reading any available data up to the buffer size provided, the others will remain blocked until more data is available to be read.
- The read end of a pipe is at index 0, the write end at index 1.
- System calls signatures for `read`, `write`, `open`, `close`, `pipe` (some might not be needed):

```
int pipe(int pipefd[2]);  
int open(const char *pathname, int flags, mode_t mode);  
int close(int fd);  
ssize_t read(int fd, void *buf, size_t count);  
ssize_t write(int fd, const void *buf, size_t count);
```
- Marks will not be deducted for minor syntax errors.

Write your lock implementation below in the spaces provided. Definition of the pipe-based lock (`struct pipelock`) should be complete, but feel free to add any other elements you might need. You need to write code for `lock_init`, `lock_acquire`, and `lock_release`.

Line#	Code
1	* Define a i e-based lock */
2	[REDACTED]
3	[REDACTED]
	[REDACTED]
11	Initialize lock *
12	[REDACTED]
	[REDACTED]
	[REDACTED]
	}
21	* Function used to acquire lock *
22	[REDACTED]
	[REDACTED]
	}
31	* Release lock *
32	[REDACTED]
	[REDACTED]
	}

17. Consider the following C code. The executables `programA` and `programB` are located in the system path (they are called successfully from the program), and they simply print a message indicating the pid of the process executing them and then terminate. Assume appropriate `wait()` calls are used at line 22.

Line#	Code Snippets
1	<code>int main() {</code>
2	<code> char* argv1[] = {"programA", NULL};</code>
3	<code> char* argv2[] = {"programB", NULL};</code>
4	<code> char* env[] = {NULL};</code>
5	<code> printf("%d is entering fork.\n", getpid());</code>
6	<code> int index = 0;</code>
7	
8	<code> int pid = fork();</code>
9	
10	<code> pid = fork();</code>
11	
12	<code> if (pid == 0 && index == 0) {</code>
13	<code> index++;</code>
14	<code> pid = fork();</code>
15	<code> if (pid == 0) {</code>
16	<code> execve("programA", NULL, NULL);</code>
17	<code> }</code>
18	<code> }</code>
19	<code> if (pid != 0 && index == 0) {</code>
20	<code> pid = fork();</code>
21	<code> execve("programB", NULL, NULL);</code>
22	<code> }</code>
23	<code> . . .// wait processes</code>
24	<code> printf("%d is exiting from fork.\n", getpid());</code>
25	<code> return 0;</code>
26	<code>}</code>

- a) [2 marks] Which program is the original process executing when that process terminates?

- b) [3 marks] How many processes terminate in each of the relevant programs?

Program	Number of processes that terminate there
Current program exiting at line 24.	<div style="background-color: black; width: 100%; height: 20px;"></div>
programA	<div style="background-color: black; width: 100%; height: 20px;"></div>
programB	<div style="background-color: black; width: 100%; height: 20px;"></div>