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.

Answers:

1. D
2. B
3. D
4. C
5. E
6. B
7. C
8. B
9. B
10. C
11. D
12. A

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	`while (true) {`
3	`    Compute();`
4	`    Reusable_barrier(N);`
5	`    Critical_code_after_barrier();`
6	`}`
7	**`Initialization for barrier:`**
8	`    int arrived = 0; //shared variable`
9	`    Semaphore mutex = 1;`
10	`    Semaphore waitQ = 1; //NOTE the change`
11	
12	**`Reusable_barrier(N)`** `{`
13	`    wait(mutex);`
14	`    if (arrived == 0) {`
15	`        wait(waitQ);`
16	`    }`
17	`    arrived++;`
18	`    if (arrived == N) {`
19	`        arrived = 0;`
20	`        signal(waitQ);`
21	`    }`
22	`    signal(mutex);`
23	
24	`    wait(waitQ);`
25	`    signal(waitQ);`
26	`}`

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

True/False	Statement
True	a. Exactly one thread will `signal(waitQ)` in line 20 at each successful use of the barrier.
	Explanation for a.: Line 20 is in in the critical section, so exactly one thread will signal. Almost all the students got this question correct.
False	b. `waitQ` will have a value of `0` after all threads successfully call the barrier once.
	Explanation for b.: Since waitQ is initialized to 1, and we have N+1 wait/signal pairs at a successful use of the barrier, the value at the end will be 1. After carefully reading the question, many of the students got this question right.
False	c. Barrier will work properly in the given scenario.
	Explanation for c.: All N threads reach lines 24/25. The synchronization there takes some time to complete. In the meantime, say, the first thread that passed line 25 goes on to execute very fast lines 5,3, and returns at the barrier. Since the other threads are still at lines 24/25, the first thread does not block in waiting at lines 15 or 24, and can pass the barrier together with the previous batch of threads. Only a few students got this part right, with the correct justification. Many students incorrectly justified that the barrier is not reusable because waitQ will not have value 0 at the end of the execution.

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()`.

Code Snippets

```
A thread executes the following statements:
while (true) {
     Compute();
     Reusable_barrier_block (N);
     Critical_code_after_barrier();
     Reusable_barrier_reset (N);
}
```

```
Initialization:
     int arrived = 0;
     Semaphore mutex = 1;
     Semaphore waitQ = 0;
Reusable_barrier_block (N) {
     wait(mutex);
     arrived ++;
     if (arrived == N)
          signal(waitQ);
     signal(mutex);
     wait(waitQ);
     signal(waitQ);
}
Reusable_barrier_reset(N) {
     wait(mutex);
     arrived --;
     if (arrived == 0)
          wait (waitQ);
     signal(mutex);
}
```

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

True/False	Statement
True	a. Exactly one thread will `wait(waitQ)` in line 24 at each successful use of the barrier.
	Explanation for a.: Line 24 is within the critical section, so exactly one thread will wait at each successful use of the barrier. Many students got this question right.
False	b. Some threads might deadlock in `Reusable_barrier_reset()`.
	Explanation for b.: There is no deadlock in reset(). Line 24 will not block because there is an extra signal (waitQ) from the use of barrier block(). About half of the students got this question right.
False	c. Barrier will work properly in the given scenario.
	Explanation for c.: All N threads reach lines 18/19. The synchronization there takes some time to complete. In the meantime, say, the first thread that passed line 19 goes on to execute very fast lines 4,5,3, and returns at the barrier block(). Since the other threads are still at lines 18/19, the first thread can pass the barrier together with the previous batch of threads. To correct this barrier, the reset part has to be a barrier by itself: add one more Semaphore resetQ and use it in the same way with the waitQ semaphore, but in reset function. Only a few students got this part right, with the correct justification. But more students noticed the problem compared to Q13, c. Many students incorrectly justified the barrier does not work because of the deadlock in part b.

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?

Answer:

Condition: if the jobs happen to arrive in the ready queue with the shortest completion times first (or, as a special case, if all jobs have the same completion time).

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

Answer:

Condition: if the job lengths are no longer than the length of the time slice.

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

Answer:

Condition: if the job lengths are all the same, and much greater than the time slice length.

a, b, c - These were quite straightforward - most answers given by students were accepted if what they argued made sense based on the criteria they were evaluating for (turnaround time, response time, etc).

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?

Answer:

If turnaround time is the priority - reducing time slice increases the % time the OS is doing work as a fraction of CPU time. Hence we get less efficient usage of CPU time.

Students who prioritized initial response time would have argued the opposite case.

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

Answer:

SJF was a surprising answer, but was considered to be possible. We expected that MLFQ would be a straightforward answer.

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.

Answer:

Two main trains of thought were present in the answers

1. Do not give CPU bound processes priority in I/O: we want our system to be responsible (prioritizing response time)

2: Give CPU processes I/O priority so that they can go back to waiting for CPU / using CPU - decreasing overall turnaround time at the expense of response time.

Both were accepted.

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`.

```
/* Define a pipe-based lock */
struct pipelock {
     int fd[2];

};
```

```
/* Initialize lock */
void lock_init(struct pipelock *lock) {

This was the most difficult question in the paper. It
seems that students did not even read the question.
The question will be discussed in tutorial.

}
```

```
/* Function used to acquire lock */
void lock_acquire(struct pipelock *lock) {

}
```

```
/* Release lock */
void lock_release(struct pipelock * lock) {

}
```

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

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

ProgramB is executed by the original process (at termination). ProgramA was a common wrong answer. Some people did not understand the question and gave us line numbers instead of a program name.

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.`	2
`programA`	2
`programB`	4

Answers from students included: 112 (half of all values in the actual answer - likely missed a fork()), 422 (reverse of the answer), and 824 were common wrong answers.
Students were supposed to draw the process tree to figure out how many processes are there and what code they execute.