CS3103 2122A midterm online solution

Operating Systems (City University of Hong Kong)

Student ID: _____

Name: _____

CITY UNIVERSITY OF HONG KONG

Course code & title : CS3103 Operating Systems

Session : Semester A 2021-2022

Time allowed : 110 minutes

There are **7** questions.

---

1. Answer ALL questions.
2. Write your answers in the space provided.

---

*This is a **close-book** examination.*

Candidates are allowed to use the following materials/aids:
Approved calculators.
One page double-sides handwritten A4 note.

Materials/aids other than those stated above are not permitted. Candidates will be subject to disciplinary action if any unauthorized materials or aids are found on them.

Academic Honesty

I pledge that the answers in this exam are my own and that I will not seek or obtain an unfair advantage in producing these answers. Specifically,

- *I will not plagiarize (copy without citation) from any source;*
- *I will not communicate or attempt to communicate with any other person during the exam; neither will I give or attempt to give assistance to another student taking the exam; and*
- *I will use only approved devices (e.g., calculators) and/or approved device models.*
- *I understand that any act of academic dishonesty can lead to disciplinary action.*

I pledge to follow the Rules on Academic Honesty and understand that violations may led to severe penalties.

Student ID: _____

Name: _____

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Total

Q1. [8 points] What is a thread pool? List two benefits of using thread pool?

1. A thread pool is a collection of threads, created at process startup, that sit and wait for work to be allocated to them.
2. Benefits:
    - Servicing a request with an existing thread is usually faster than waiting to create a thread.
    - A thread pool limits the number of threads that exist at any one point of time.

Q2. [8 points] What are the two communication models in inter-process communication? Discuss the benefits and drawbacks of each model.

1. Message passing and Shared memory.
2. message passing is better for small amount of data.
3. shared memory is better for large data but has the overhead of setting up shared memory area.
4. message passing has the overhead of involving kernel.

Q3. [22 points]

Today, to utilize our machine better, most of our devices use a time-sharing mechanism on CPU resources. That's why we need a good CPU scheduler to maximize our system performance. When comparing different CPU scheduler algorithms, we need some metrics that make sense in scheduling to measure their performance. Here we define two metrics: turnaround time and response time.

They are defined as follows:

$$T_{turnaround} = T_{completion} - T_{arrival}$$
$$T_{response} = T_{firstrun} - T_{arrival}$$

Please use the metrics defined above to answer the following four questions. Be sure to show your calculation by equations or diagrams.

a. [9 points] Suppose we have three jobs: A, B, C, and with different lengths: 30, 50, 20, and different $T_{arrival}$: 10, 0, 10, respectively. (Assume A arrived just a hair before C.) Please calculate each $T_{turnaround}$ and $T_{response}$ of them, and $T_{turnaround}$ and $T_{response}$ for average, using SJF (Shortest Job First, Non-preemptive) and STCF (Shortest Time-to-Completion First, also known as Shortest Job First, Preemptive) policy, respectively. And analysis which of these policies is better, please list your reasons.

	0	10	20	30	40	50	60	70	80	90	100
SJF	B	B	B	B	B	C	C	A	A	A	
STCF (PSJF)	B	C	C	A	A	A	B	B	B	B	

SJF:

$T_{turnaround}$ (A) = 100 – 10 = 90
$T_{turnaround}$ (B) = 50 – 0 = 50
$T_{turnaround}$ (C) = 70 – 10 = 60
$T_{turnaround}$ (average) = (90 + 50 + 60)/3 = 66.67

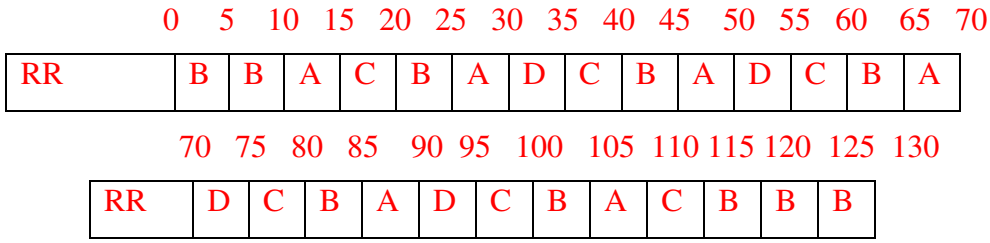
$T_{response}$ (A) = 70 – 10 = 60
$T_{response}$ (B) = 0 – 0 = 0
$T_{response}$ (C) = 50 – 10 = 40
$T_{response}$ (average) = (60 + 0 + 40)/3 = 33.34

PSJF/STCF:
$T_{turnaround}$ (A) = 60 – 10 = 50
$T_{turnaround}$ (B) = 100 – 0 = 100
$T_{turnaround}$ (C) = 30 – 10 = 20
$T_{turnaround}$ (average) = (50 + 100 + 20)/3 = 56.67

$T_{response}$ (A) = 30 – 10 = 20
$T_{response}$ (B) = 0 – 0 = 0
$T_{response}$ (C) = 10 – 10 = 0
$T_{response}$ (average) = (20 + 0 + 0)/3 = 6.67

STCF is better than SJF since it archives a good turnaround time and response time.

b. [5 points] Suppose we have another four jobs: A, B, C, and D, and with different lengths: 30, 50, 30, 20, and different T arrival: 10, 0, 10, 20 respectively. (Assume A arrived just a hair before C.) Please calculate each $T_{turnaround}$ and $T_{response}$ of them, and $T_{turnaround}$ and $T_{response}$ for average, using Round-robin (quantum = 5) scheduling policy. And analysis the pros and cons of this policy.



ANR1

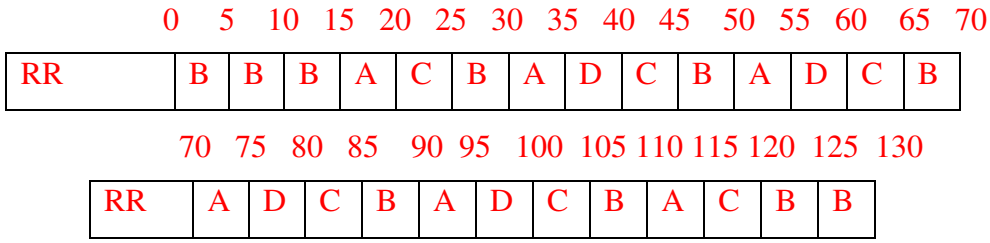
RR:

$T_{turnaround}$ (A) = 110 – 10 = 100
$T_{turnaround}$ (B) = 130 – 0 = 130
$T_{turnaround}$ (C) = 115 – 10 = 105
$T_{turnaround}$ (D) = 95 – 20 = 75
$T_{turnaround}$ (average) = (100 + 130 + 105 + 75)/4 = 102.5

$T_{response}$ (A) = 10 – 10 = 0
$T_{response}$ (B) = 0 – 0 = 0
$T_{response}$ (C) = 15 – 10 = 5
$T_{response}$ (D) = 30 – 20 = 10
$T_{response}$ (average) = (0 + 0 + 5 + 10)/4 = 3.75

RR is fair but performs poorly on metrics such as turnaround time.

ANR2



RR:

$T_{turnaround}$ (A) = 115– 10 = 105
$T_{turnaround}$ (B) = 130 – 0 = 130
$T_{turnaround}$ (C) = 120 – 10 = 110
$T_{turnaround}$ (D) = 100 – 20 = 80
$T_{turnaround}$ (average) = (105 + 130 + 110 + 80)/4 = 106.25

$T_{response}$ (A) = 15 – 10 = 5
$T_{response}$ (B) = 0 – 0 = 0
$T_{response}$ (C) = 20 – 10 = 10
$T_{response}$ (D) = 35 – 20 = 15
$T_{response}$ (average) = (5 + 0 + 10 + 15)/4 = 7.5

RR is fair but performs poorly on metrics such as turnaround time.

c. [3 points] There is a trade-off in Round-robin scheduling. If we consider the system that demands a better response time, how should we design the length of a time slice of Round-robin scheduling in (b)? Does this new design have any shortcomings? If so, please list some of them.

The time slice should be shorter, which can achieve a better response time. Ye. The cost of context switching will dominate overall performance.

d. [5 points] If considering the system to achieve good performance for multiple jobs, including short jobs, long jobs, and interactive jobs, how would you design the scheduling algorithm? Please detail your design.

It will adopt the Multilevel (Feedback) Queue. Firstly, it should set multiple ready queues, each with a different priority, with the first queue having the highest priority, In those queue, the higher the priority queue, the smaller the running time slice of each process. MLFQ always performs the interactive job because it has high priority work. For other jobs with the same priority, those jobs will run in RR. MLFQ doesn't know whether a job will be a short job or a long-running job, it first assumes it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process. In this manner, MLFQ approximates SJF.

Q4 (15 points) Consider the following two threads to be run concurrently in a shared memory 2000. Assume that the threads run in a single-processor system. The value at shared memory 2000 is initialized to 0. We also assume that Thread 1 will get scheduled first.

Thread 1

```
.main            # the entry point of the thread
mov $3,%ax       # %ax is initialized to 3
.top             # a jump point
mov 2000,%dx     # get the value from the address to register %dx
add $1,%dx       # increment the value in register %dx by 1
mov %dx,2000     # store the value back to the address
sub $1,%ax       # decrement the value in register %ax by 1
test $0,%ax      # if the value in %ax is greater than 0
jgt .top         # jump back to .top if %ax = 0
halt             # stops running this thread
```

Thread 2

```
.main            # the entry point of the thread
mov $3,%ax       # %ax is initialized to 3
.top             # a jump point
mov 2000,%dx     # get the value from the address to register %dx
add $2,%dx       # increment the value in register %dx by 2
mov %dx,2000     # store the value back to the address
sub $1,%ax       # decrement the value in register %ax by 1
test $0,%ax      # if the value in %ax is greater than 0
jgt .top         # jump back to .top if %ax = 0
```

```
halt            # stops running this thread
```

a. [3 points] What is the final value at the shared memory 2000 if there are no interrupts?

b. [5 points] What is the function of the register %ax, and what is the final value of the register %ax if there are no interrupts?

c. [7 points] If interruption occurs, can we always get the correct value at the shared memory 2000? If yes, please explain the reason; if not, what part of instructions in the code block of Thread1 and Thread2 needs to be protected?

Answers

a) 9, for thread 0, 1 + 1 + 1 = 3, so the value at memory 2000 is equal to 3 after thread 0 halts. Then, thread 1 starts, and keep adding the value at memory 2000, so we get 3 + 2 + 2 + 2 = 9. The final result is 9.

b) %ax is used as a looper, iterator or counter, the final value is 0.

c) Not always.

```
Thread 0
mov 2000,%dx
add $1,%dx
mov %dx,2000


Thread 1
mov 2000,%dx
add $2,%dx
mov %dx,2000
```

Q5 (18 points) Consider an operating system with one CPU that has four states for each process: *RUNNING* (the process is using the CPU right now), *READY* (the process could be using the CPU right now, but some other process is using the CPU), *WAITING* (the process is waiting on I/O), and *DONE* (the process is finished executing). The system uses SWITCH_ON_END as the switching policy when a process issues an I/O request: the system will NOT switch to another process while doing I/O, instead of waiting until the process is finished.

Now suppose that we have two processes, and each of them consists of 4 requests. Each CPU request takes 1 time unit, and each I/O request takes 5 time units (1 time unit on CPU and 4 time units on I/O device) to complete.

1) [9 points] Suppose that the 4 requests of process 0 and process 1 are **I/O, I/O, CPU, CPU**, and **I/O, I/O, I/O, CPU**, respectively, and process 0 runs first**.**

Fill the **states of each process**(RUNNING: CPU, RUNNING: IO, WAITING, READY, DONE), **the number of processes using CPU**, and **the number of processes using I/O devices** in the following tables and calculate **CPU utilization**. When you compute the CPU utilization, **DO NOT** include the time units that all the processes have done.

Time unit	*PID: 0*	*PID: 1*	*CPU (0, 1, or 2)*	*I/O Devices (0, 1, or 2)*
1	RUNNING: IO	READY	1	0
2	WAITING	READY	0	1
3	WAITING	READY	0	1
4	WAITING	READY	0	1
5	WAITING	READY	0	1
6	RUNNING: IO	READY	1	0
7	WAITING	READY	0	1
8	WAITING	READY	0	1
9	WAITING	READY	0	1
10	WAITING	READY	0	1
11	RUNNING: CPU	READY	1	0
12	RUNNING: CPU	READY	1	0
13	DONE	RUNNING: IO	1	0
14	DONE	WAITING	0	1
15	DONE	WAITING	0	1
16	DONE	WAITING	0	1
17	DONE	WAITING	0	1
18	DONE	RUNNING: IO	1	0
19	DONE	WAITING	0	1
20	DONE	WAITING	0	1
21	DONE	WAITING	0	1
22	DONE	WAITING	0	1
23	DONE	RUNNING: IO	1	0
24	DONE	WAITING	0	1
25	DONE	WAITING	0	1
26	DONE	WAITING	0	1
27	DONE	WAITING	0	1
28	DONE	RUNNING: CPU	1	0
29	DONE	DONE	0	0
30	DONE	DONE	0	0
CPU Utilization	8 / 28 = 28.57%			

NOTE: This table may have extra lines.

2) [9 points] Are system resources being effectively utilized? If yes, please explain the reason; If not, what is the problem? And describe any ideas you have to improve the CPU utilization and why they are good ideas.

Answer:
NO.

The system will not switch to another process when waiting for one process doing I/O.
We can use SWITCH_ON_IO and IO_RUN_IMMEDIATE to improve the CPU utilization.
SWITCH_ON_IO can utilize the CPU when the current process is waiting for I/O；
IO_RUN_IMMEDIATE will immediately switch to the process that will issue an I/O request. It is a good idea because after giving the I/O operation, other processes can be scheduled to run simultaneously. We can utilize the waiting time as far as possible by issuing the I/O first.

Q6 (15 points):

Here we have fork() function, which is a popular way to create a subprocess. Following code demonstrate the usage of fork() function.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   int i;
   for (i = 0; i < 10; i++)
      fork();
   magic();
}
```

1. Which space does the fork() function get executed? (Kernal or User)Why? (5 points)

   Kernel space, since it will handle many privilieged requests.

2. In the demo code, the for loop will quit after 10 times execution, describe what will happen if we change the for loop into an infinite loop? (5 points)

   A program that calls the fork() system call in an endless loop is called a "fork() bomb" since the number of processes will increase exponentially. The consequence of trying the fork() bomb on the actual machine is that the machine will not crash, but it cannot create a new program since there is no resource to create a new PCB (process control block).

3. According to the original code, how many times will the function magic() be executed. (5 points)

   1024 times since there will be 1023 children processes.

Q7 (14 points)

In this question, we will give out a pseudo-code implementation of a multi-thread problem to solve the following situation. Please read the question description and the pseudo-code below and point out the implement problems of the code.

The simulation of the Instant Noodles Window.

There are instant noodles for sale in the A window of the AC1 cafeteria of our school. The customer can use an order to show the type of noodle he wants. A window will have several workers working together to provide the corresponding noddle, one is responsible for cooking the noodles and putting it in a bowl (the back kitchen), and one will pick the cooked noodle, add the side dish according to the order and deliver it to the customer (the producer). Since the cooking time of noodle is much longer than simply adding the side dishes, the back kitchen will keep adding the cooked noodles in a bowl buffer.

The pseudo-code to simulate this process:

```
// shared data
Semaphore customer=0;
Semaphore bowl_buffer_full=N; // max number of bowls
Semaphore bowl_buffer_empty=0;
Semaphore producer=0;
Semaphore mutex = 1;

// order is the order delivered from the customer, there are NUM_NOODLES_TYPES kind of noodles.
int order = 0; // set to 0 means there is no customer or the order have been received by the producer

// noodle represents the cooked noodle bowls,
int noodle = 0; // set to 0 means the bowl buffer is empty
```

```
// code for customer
void customer {
  while (true) {
      order = rand()%NUM_NOODLE_TYPES +1; // pick a random type of noodle
      signal(producer); // asking the producer for the order
  }
}
```

```
// code for back_kitchen
void back_kitchen{
  while (true) {
      wait(bowl_buffer_empty);
      // cook the noodle
      noodle++; // and serve in a bowl
      signal (bowl_buffer_full);
  }
}
```

```
// code for producer
void producer {
  while (true) {
      wait(producer); // wait the order
      wait(bowl_buffer_full); // grab the bowl from back kitchen
      wait(mutex);
      // Read the order
      noodle--; // and side dish in the bowl
```

```
        order = 0; // set the order to 0, representing the order has been received
        signal(mutex);
        signal(bowl_buffer_empty);
        signal(customer); // deliver the bowl to the customer
    }
}
```

a) Please find out 2 errors in the implementation, please describe them and ***explain*** how to solve them. (8 points)

   a. Lack of the lock for the noodle variable.

   b. Lack of the wait(customer) in the customer code.

   c. Semaphore bowl_buffer_full=0;
      Semaphore bowl_buffer_empty=N;

   Any two of them get points.

b) Why don't we need to use the mutex to guard the **order** variable in customer? (3 points)
There's only one order, and there's only one thread modifying the order variable at a time.

c) To provide better performance, how can we change ***the code***, describe one of the possible ways you can think. No need for coding. (3 points)

Every solution that make sense should get points for that.

Any solution mentioned about the hardware environment or OS support, for example, the context switching overhead, won't get any point for that.

```
// shared data
Semaphore customer=0;
Semaphore reporter=0;
Semaphore bowl_buffer_full=0;
Semaphore bowl_buffer_empty=N; // max number of bowls
Semaphore producer=0;
Semaphore mutex = 1;
// order is the order delivered from the customer, there are NUM_NOODLES_TYPES kind of
dishes.
int order = 0; // set to 0 means the order have been received by the reporter

// noodle represents the cooked noodle bowls,
int noodle = 0; // set to 0 means the noodle bowl is empty
```

```
// code for customer
void customer {
  while (true) {
       order = rand()%NUM_NOODLE_TYPES +1; // pick a random type of noodle
       signal(reporter); // asking the reporter for the order
    wait(customer); // wait the producer to deliver the food
  }
}
```

```
// code for reporter
void reporter {
  while (true) {
       wait(reporter); // wait for the incoming order
       // Read the order
       signal(producer); // Send a message to the producer
  }
}
```

```
void back_kitchen{
  while (true) {
       wait(bowl_buffer_empty);
        wait(mutex);
       // cook the noodle
       noodle++; // and serve in a bowl
       signal(mutex);
       signal (bowl_buffer_full);
  }
}
```

```
void producer {
  while (true) {
       wait(producer); // wait the back_kitchen to cook the noodles
       wait(bowl_buffer_full); // grab the bowl from back kitchen
       wait(mutex);
       order = 0; // set the order to 0, representing the order has been receive
       noodle--; // and side dish in the bowl
       signal(mutex);
       signal(bowl_buffer_empty);
       signal(customer); // deliver the bowl to the customer
  }
}
```