2223midterm - CS3103

Operating Systems (City University of Hong Kong)

Student ID:_____

Name: _____

CITY UNIVERSITY OF HONG KONG

Midterm

Course code & title:	CS 3103 Operating Systems
Session:	Semester B 2022-2023
Time allowed:	110 minutes

There are **7** questions.

---

1. Answer ALL questions.
2. Handwrite your answer on the exam paper.

---

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

Candidates are allowed to use the following materials/aids:
Approved calculators.

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.

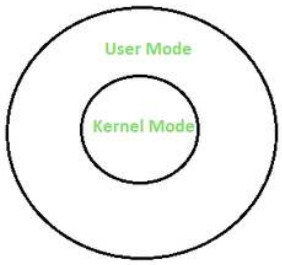
Academic Honesty

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.*

Q1. (**5 points**)
Please state the difference between the kernel mode and user mode.

1. The mode bit is **1 for the user mode** and **0 for the kernel mode.**
2. The user mode is **for running a user application** while the kernel mode **starts when the system boots.**
3. Some **privileged** instructions could only be executed in kernel mode.
4. Processes run in kernel mode, they have **unrestricted access to the hardware** while a process runs in user mode only has **limited access to the CPU and the memory**.
5. **System crash** in kernel mode is more severe than that in user mode. The system crash in user mode could be recovered by simply resuming the session.
6. The kernel address space is strictly mapped into the **address space,** while in the user mode **gets their own address space.**



Q2. (**10 points**)
Please write the five process states that describe the current activity of a process with detailed explanation.
The five states that are being used in this process model are as follows:
– **new:** The process is being created
– **ready:** The process is waiting to be assigned to CPU
– **running:** Instructions are being executed
– **waiting:** The process is waiting for some event to occur
– **terminated:** The process has finished execution

(If they have different names, e.g., DONE. It is also ok, as long as the meanings are the same)

Q3. **(10 points)**

Suppose the system currently has two processes P0 and P1.

(1) Let S and Q be two semaphores, which are both initialized to 1. The pseudo-codes of P0 and P1 are shown as follows. Is it possible for the two processes to have a deadlock? Why? **[5 points]**

P0	P1

wait (S)
wait (Q)
// critical section
signal(S)
signal(Q)
	wait (S)
	wait (Q)
	//critical section
	signal(Q)
	signal(S)

There is no deadlock here.

S and Q are initialized to 1. So, when process P0 or P1 starts execution, S value get initialized to 0, then another process will not get chance to move further.

For example:

Assume p0 started executing--->S value made to 0.
Then P1, if it starts executing then wait(S) will fail. Hence no further movement takes place.
P1 will get the chance of execution only after the completion of signal(S) in p0.
Same case will be there, if P0 starts execution.

Hence there is no deadlock.

(2) Let S, T and Q be three semaphores, where S and T are both initialized to 1, and Q is initialized to 3. The pseudo-codes of P0 and P1 are shown as follows. Is it possible for the two processes to have a deadlock? Why? **[5 points]**

P0	P1

wait (S)
wait (Q)
wait (T)
// critical section
signal(T)
signal(S)
signal(Q)
	wait (S)
	wait (T)
	wait (Q)
	//critical section
	signal(Q)
	signal(T)
	signal(S)

There is no deadlock here.

Given S and T are initialized to 1. So, when process P0 or P1 starts execution, S value get initialized to 0, then another process will not get chance to move further.

For example:
Assume p0 started executing--->S value made to 0.
Then P1, if it starts executing then wait(S) will fail. Hence no further movement takes place.
P1 will get the chance of execution only after the completion of p0.
Same case will be there, if P0 starts execution.
Hence there is no deadlock.

Q4. (**20 points**)

Consider the following processes, with the arrival and processing time as given in the table.

Process number	Arrival time	Processing time
1	0	9
2	1	1
3	4	5
4	5	3

Note:

The *turnaround time* of a task is the time distance between the arrival time and completion time of a task, where

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

The *response time* of a task is the time distance between the arrival time of a task and when it is executed for the first time, where

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

Please answer the following questions.

(a) Calculate the turnaround time and response time of each process for each of the execution of these processes using Round Robin (RR, time quantum = 1), Preemptive Shortest Job First (SJF) and FIFO scheduling. Please write your calculation procedures. **[12 points]**

RR

Process	Turnaround time	Response time
1	18	0
2	1	0
3	12	0
4	7	0

SJF

Process	Turnaround time	Response time
1	18	0

2	1	0
3	8	0
4	3	0

FIFO

Process	Turnaround time	Response time
1	9	0
2	9	8
3	11	6
4	13	10

(b) Discuss the tradeoff factors to consider when deciding the length of a time quantum in Round Robin scheduling. **[4 points]**

The time quantum is a crucial parameter in the Round Robin scheduling algorithm. Its length determines how often the CPU will switch between executing processes. The choice of a suitable time quantum depends on several factors, including:

1. Overhead: A shorter time quantum leads to more frequent context switches, which can increase overhead and reduce overall system performance.
2. Response time: A shorter time quantum results in shorter response time for interactive processes, improving their perceived performance. However, it may also cause the system to spend more time on context switching rather than executing useful work.
3. Throughput: A longer time quantum allows processes to execute for a longer duration, which can improve overall system throughput.
4. Fairness: A shorter time quantum can ensure that processes are given a fair share of CPU time, preventing long-running processes from monopolizing the CPU.

In general, a time quantum that balances these factors can be chosen based on the specific system requirements and workload.

(c) Suppose some processes are for real-time interactive applications and some are background processes. Please write your scheduling algorithm with explanation. **[4 points]**

One such algorithm is the Priority Round Robin scheduling algorithm, where each process is assigned a priority value, and the CPU is allocated to the highest-priority process that has arrived. Within each priority level, Round Robin scheduling is used to allocate CPU time. This way, the interactive applications can be assigned a higher priority value, ensuring that they are given preferential treatment and allowing them to respond to user input promptly.

Q5 **(15 points)**

Consider the following program.

```
int x = 6;

void *runner (void *arg) {
        x+=2;
    cout << "thread: " << x << endl;
    pthread_exit(NULL);
}

int main ()
{
    int pid;
    pthread_t tid;

    pid = fork();
        x++;
    if (pid == 0) {
        pthread_create(&tid, NULL, runner, NULL);
        pthread_join(tid, NULL);
        x+=3;
        cout << "process1: " << x << endl;
    }
    else {
                wait(NULL);
                x+=4;
        cout << "process2: " << x << endl;
    }
}
```

(a) List the outputs of this program. Please show the outputs in the order clearly. **[8 points]**

thread: 9

process1: 12

process2: 11

(b) Please briefly explain the reasons for the above outputs. **[7 points]**

The program starts with *int x = 6*.

It creates a new process using the *fork()*. Now there are two processes running: the parent process (process2) and the child process (process1).

Both processes increment *x* by 1 (*x++*), making *x* equal to 7 in their respective memory spaces.

The child process (process1) creates a new thread and executes the runner function. Inside the

runner function, *x* is incremented by 2 (*x+=2*), making *x* equal to 9 in the child process's memory space.

The "thread" output is printed: "*thread: 9*".

After the thread finishes, *x* is incremented by 3 (*x+=3*) in the child process's memory space, making *x* equal to 12.

The "process1" output is printed: "*process1: 12*".

Meanwhile, the parent process (process2) waits for the child process to finish using wait(NULL). Once the child process is finished, x is incremented by 4 (*x+=4*) in the parent process's memory space, making *x* equal to 11.

The "process2" output is printed: "*process2: 11*".

The given output sequence occurs when the parent process (process2) waits for the child process (process1) to finish before it proceeds with its own execution. This is why the "*thread*" and "*process1*" outputs are printed before the "*process2*" output.

Q6 **(18 points)**

Consider system running on one CPU where each process has four states: *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 scheduler has two options:

- **SWITCH_ON_IO:** the system switches to another process whenever a process issues an IO request and starts to wait for I/O completion.
- **SWITCH_ON_END:** the system can NOT switch to another process while one is doing I/O; instead, it can switch to another process when the current running process is completely finished.

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

Suppose that we have two processes: process 0 and process 1, while process 0 runs first. The request sequence of process 0 is: **I/O, CPU, I/O, I/O.** The request sequence of process 1 is: **I/O, CPU, CPU, CPU.**

When **SWITCH_ON_END is ON**, the states of each process (RUNNING: CPU, RUNNING: IO, WAITING, READY, DONE), the number of processes using CPU, the number of processes using I/O devices and CPU utilization are provided in the following table **as an example**.

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: CPU	READY	1	0
7	RUNNING: IO	READY	1	0
8	WAITING	READY	0	1
9	WAITING	READY	0	1
10	WAITING	READY	0	1
11	WAITING	READY	0	1
12	RUNNING: IO	READY	1	0
13	WAITING	READY	0	1
14	WAITING	READY	0	1
15	WAITING	READY	0	1
16	WAITING	READY	0	0
17	DONE	RUNNING: IO	1	0
18	DONE	WAITING	0	1
19	DONE	WAITING	0	1
20	DONE	WAITING	0	1
21	DONE	WAITING	0	1
22	DONE	RUNNING: CPU	1	0
23	DONE	RUNNING: CPU	1	0
24	DONE	RUNNING: CPU	1	0
25	DONE	DONE	0	0
26	DONE	DONE	0	0
CPU Utilization	8/ 24 = 33.3%			

Now suppose the system uses **SWITCH_ON_IO**, fill the following table and calculate **CPU utilization**. When you compute the CPU utilization, **DO NOT** include the time units that all the processes have finished. You do not need to draw the entire table in your answer and only need to ***answer the blanks of the following table***.

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	RUNNING: IO	1	1
3	WAITING	WAITING	0	2
4	WAITING	WAITING	0	2
5	WAITING	WAITING	0	2
6	RUNNING: CPU	WAITING	1	1
7	RUNNING: IO	READY	1	0

8	WAITING	RUNNING: CPU	1	1
9	WAITING	RUNNING: CPU	1	1
10	WAITING	RUNNING: CPU	1	1
11	WAITING	DONE	0	1
12	RUNNING: IO	DONE	1	0
13	WAITING	DONE	0	1
14	WAITING	DONE	0	1
15	WAITING	DONE	0	1
16	WAITING	DONE	0	1
17	DONE	DONE	0	0
18	DONE	DONE	0	0
19	DONE	DONE	0	0
CPU Utilization	8/16=0.5			

Note that some rows might be **redundant**. In the given example, process 0 and 1 have both DONE states at time units 25 and 26.

Q7 **(22 points)**

Compare_and_swap (CAS) is an atomic operation commonly used in computer science to implement synchronization algorithms and ensure mutual exclusion in multi-threaded applications. It is a hardware instruction that allows a processor to compare the value of a memory location with an expected value, and only update the memory location if the value matches the expected value. Here is the semantic of the compare_and_swap instruction.

compare_and_swap

int compare_and_swap(int *value, int expected, int new value) {
int temp = *value;
if (*value == expected)
*value = new value;
return temp;
}

The following is the pseudocode for implementing mutual-exclusion via compare_and_swap() instruction.

Line	Pseudo Program
1	do{
2	while (compare_and_swap(&lock, 0, 1) != 0) ;
3	/* critical section */
4	**lock = ??;**
5	/* remainder section */
6	} while (true);

Please answer the following questions.
(a) Please state the meaning of 'atomic' [**4 points]**

atomic refers to an operation that is indivisible or unbreakable.

(b) State the initialization value of lock (when the system has zero processes) with reasons. Assuming the system only has one process now. Explain how the program works and state the value of lock when the process is in the critical section. **[10 points]**

Initialization value of 0

This is because the value of the lock variable determines whether the critical section is currently being executed by a process or not. When the system has zero processes, the critical section is not being executed by any process, so the value of the lock variable should be 0 to indicate that the critical section is available for a process to enter.

Assuming the system only has one process now, the program works as follows:

1. The process executes a do-while loop indefinitely, indicating that it will repeatedly attempt to enter the critical section.
2. The process first executes a while loop that repeatedly executes the compare_and_swap() instruction until it successfully sets the value of the lock variable to 1. This loop ensures that

the process can only enter the critical section when the lock variable is available (i.e., its value is 0).

3. Once the process has acquired the lock (i.e., the value of the lock variable is set to 1), it enters the critical section and executes the code within it.
4. After executing the critical section, the process sets the value of the lock variable back to 0 to indicate that the critical section is now available for other processes to enter.
5. The process then executes the remainder section of the code outside the critical section.

When the process is in the critical section, the value of the lock variable is 1. This indicates that the critical section is currently being executed by the process and is not available for other processes to enter. This ensures mutual exclusion and prevents race conditions in the critical section.

(c) Assuming that the first process in question (b) is still in the critical section (line 3), while the second process arrives. State how compare_and_swap provides mutual exclusion. **[4 points]**

If the value of the lock is 0 (i.e., the lock is not currently held by any process), the compare_and_swap() instruction will set the value of the lock to 1 (i.e., the second process has acquired the lock) and return 0.

However, if the value of the lock is 1 (i.e., the lock is currently held by the first process), the compare_and_swap() instruction will not set the value of the lock to 1 and will return 1. This indicates that the second process has failed to acquire the lock and must wait until the first process releases the lock.

(d) When the first process finishes execution and leaves the critical section, the value of lock should be set to what number at line 4. State your answer with explanation. **[4 points]**

lock =0
When the first process finishes execution and leaves the critical section, the value of the lock should be set to 0 at line 4.

Setting the value of the lock to 0 indicates that the lock is now available for other processes to acquire, and any other processes that are waiting to acquire the lock can now do so.

(e) Could this algorithm satisfy the bounded waiting requirement? Explain your answer. **[4 points]**

No. This algorithm does not satisfy the bounded waiting, since a process could wait for indefinite time.

An example with bounded waiting and hardware supported instructions:

```
boolean test_and_set(boolean *target){
  boolean rv = *target;
  *target = true;
  return rv;
}
do{
  waiting[i] = true;
  key = true;
  while (waiting[i] && key)
    key = test and set(&lock);
  waiting[i] = false;
  /* critical section */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = false;
  else
    waiting[j] = false;
  /* remainder section */
}
while (true);
```

You could see that the waiting is in the cyclic order. Any process will wait at most n-1 turns, which is a bound.

In contrast, the midterm program does not provide such a bound.

--END—