

Question 1. [18 MARKS]

This question is about general knowledge on processes.

Part (a) [3 MARKS]

In an operating system processes may be in a number of different states during their lifetime. List three distinct states.

Solution:

New, ready, waiting, running, terminated.

Part (b) [2 MARKS]

Describe two distinct state transitions that may occur between these states and identify the events that cause each state transitions.

Solution:

new - created -> ready
ready - scheduled/dispatched -> running
running - interrupt ready
running - I/O, wait call, etc. -> wait
wait - I/O completion etc. -> ready
running - exit -> terminated

Part (c) [3 MARKS]

List three components of a process control block.

Solution:

process state, PID, program counter, registers, memory limits, open files etc. (other answers are possible, but would likely include a number of the above listed components.)

Part (d) [1 MARK]

List one difference between a program and a process.

Solution:

Note, may answers are correct:

a process is a program in execution
a process consists of the program, i.e., the text segment, but also contains other resources, such as registers, data segment, a stack, memory management, information, file descriptors, a process ID ...
a program is static, a process is dynamic
one program may be associated with many processes

Part (e) [1 MARK]

List one difference between a process and a thread.

Solution:

Note many answers may be correct, here are a few correct ones:

a process is heavy weight, it is associated with a context consisting of code, data, registers, stack, files

a thread is light weight, compared to a process its context is much smaller, simply consisting of registers, stack

context switching a process is much more expensive than ...

threads share a process's resources, such as files, code, global data

Part (f) [1 MARK]

To effectively use threads, is a multi-CPU machine required (Yes/No)?

Solution:

No

Part (g) [1 MARK]

What is the difference between a process and a process descriptor?

Solution: A process is the actual code, data and stack in memory; a process descriptor is a data structure that can hold the state of the process.

Part (h) [1 MARK]

What is the main advantage of the layered approach to operating system design?

Solution: Modularity.

Part (i) [1 MARK]

What is a context switch?

Solution: A change in execution from one process to another.

Part (j) [1 MARK]

What is a race condition?

Solution: It is a situation where multiple processes access that same data and the outcome of execution depends on the particular order in which the accesses take place.

Part (k) [1 MARK]

Consider the following implementation of a lock for two processes P_i and P_j :

```
Flag[i] = 1;
while (Flag[j]==1);
```

```
/* Critical Section */
```

```
Flag[i] = 0;
```

Which of the three requirements of the critical section problem (if any) does this implementation fail to satisfy?

Solution: Progress.

Part (l) [2 MARKS]

What are the other two requirements a solution to the critical section problem should satisfy?

Solution: mutual exclusion and bounded wait.

Question 2. [5 MARKS]

This question is about the management of process control blocks.
Consider the following C function:

```

struct LL{
    struct PD *head;
    ...
};

typedef int ProcessId ;
struct PD {
    struct PD *link;
    ...
    struct LL *inlist;
};

struct *PD DequeueTail(struct LL *list) {
    struct PD *curr = list->head, *prev = curr;
    if(curr == NULL) {
        return NULL;
    }
    while(curr->link != NULL) {
        prev = curr;
        curr = curr->link;
    }
    prev->link = NULL;
    return curr;
}

```

Part (a) [2 MARKS]

Clearly describe the case that this function for dequeuing the last element of a list does not handle.

Solution:

If the list contains one element then the head of the list is not updated.

Part (b) [3 MARKS]

Fix the code so that this case is handled correctly. Indicate clearly where any added code belongs in the function.

Solution

After the check for `curr == NULL`, add the following:

```

if(curr->link == NULL) {
    list->head == NULL;
    return curr;
}

```

Question 3. [8 MARKS]

This question is about inter-process communication. Consider the following pseudo-code:

<u>Process A</u>	<u>Process B</u>
send()	print "B1"
print "A1"	receive()
send()	print "B2"
print "A2"	receive()
receive()	send()
print "A3"	

Part (a) [5 MARKS]

If sends and receives are blocking, write down the output that would result from executing the above pseudo-code.

Solution: *B1 A1 B2 A2 A3*
B1 B2 A1 A2 A3

Part (b) [1 MARK]

Is this the only possible order? Circle the correct answer.

YES

NO

Solution: *NO, there are two possible orders*

Part (c) [4 MARKS]

If only the send calls are non-blocking, is it possible to get a different order of output?

YES

NO

If yes, write one of the possible orders of output.

Solution: *Yes.*

A1 B1 B A2 A3 (there are others)

Question 4. [10 MARKS]

This question is about semaphores. The following code fragment sketches the implementation of a semaphore.

Data structure:

```
typedef struct {
    int value;
    struct process *L;
}semaphore
```

Code fragment:

```
1 void wait(semaphore S){
2
3     S.value = S.value - 1
4
5     if (S.value < 0){
6         add this process to S.L
7         block(S);
8     }
9 }
10
11 void signal(semaphore S){
12
13     S.value = S.value + 1
14
15     if (S.value <= 0){
16         remove a process P from S.L;
17         wakeup(P);
18     }
19 }
```

Part (a) [1 MARK]

In the wait function, what does the condition “(S.value < 0)” in the if statement check for?

Solution: Checks that no process has already acquired the semaphore.

Part (b) [1 MARK]

In the signal function, what does the condition “(S.value <=0)” in the if statement check for?

Solution: Checks if some process is blocked waiting for the semaphore to be released.

Part (c) [1 MARK]

Does the implementation of the semaphore contain critical sections (yes/no)?

Solution: yes.

Part (d) [4 MARKS]

If your answer to the previous question is “yes”, insert, as necessary, synchronization mechanisms to protect the critical section; if your answer is “no”, clearly explain your reasoning.

Solution: yes there are CS

```

1 void wait(semaphore S){
2   lock(mutex);
3   S.value = S.value - 1
4
5   if (S.value < 0){
6     add this process to S.L
7     unlock(mutex);
8     block(S); }
9   else
10    unlock(mutex);
11 }
12
13 void signal(semaphore S){
14   lock(mutex);
15   S.value = S.value + 1
16
17   if (S.value <= 0){
18     remove a process P from S.L;
19     unlock(mutex);
20     wakeup(P); }
21   else
22     unlock(mutex);
23 }

```

Part (e) [3 MARKS]

Does this implementation of a semaphore avoid “busy waiting”? Explain your answer.

Solution:

Yes, it avoids busy waiting at the application level, as compared to other implementations of the semaphore that explicitly cause a process to spin in a while loop until the while-condition is negated by some other process.

No, it does not avoid busy waiting, since the implementation of the semaphore itself contains critical sections that must be protected. They may be protected with synchronization mechanisms that do require busy waiting. Although these critical sections in the implementation of the semaphore operations are very short.

However, in the implementation of the semaphore synchronization

mechanisms, such as disabling interrupts/atomic operations may be used, which would avoid busy waiting even in the implementation of the semaphore.

Question 5. [10 MARKS]

(Tarek's question)

This question is about baboons crossing valley.

```
begin_cross_east() {
    if ((WW > 0) || (CW > 0)) {
        EW = EW + 1;
        Wait(CanCrossE);
        EW = EW - 1;
        CE = CE + 1;
        Signal(CanCrossE);
    } else
        CE = CE + 1;
}

end_cross_east() {
    CE = CE - 1;
    if (CE == 0) {
        Signal(CanCrossW);
    }
}

begin_cross_west() {
    if ((EW > 0) || (CE > 0)) {
        WW = WW + 1;
        Wait(CanCrossW);
        WW = WW - 1;
        CW = CW + 1;
        Signal(CanCrossW);
    } else
        CW = CW + 1;
}

end_cross_west() {
    CW = CW - 1;
    if (CW == 0) {
        Signal(CanCrossE);
    }
}
```

Question 6. [8 MARKS]

This question is about synchronization primitives.

The `fetch-and-add` atomic instruction is defined as follows:

```
int fetch-and-add (int *var, int value) {
    int temp;

    temp = *var;
    *var = *var + value;
    return (temp);
}
```

Part (a) [5 MARKS]

Design “`enter critical-section`” (i.e., the entry section) and “`exit critical-section`” (i.e., the exit section) primitives using the `fetch-and-add` atomic instruction. What should the initial value of a lock variable be?

Solution:

```
<enter CS>:  while (fetch-and-add (lock,1) > 0);
<exit CS>:   lock = 0;
```

The initial value of lock should be 0.

Part (b) [3 MARKS]

Does your implementation satisfy each of the three requirements for the critical section problem? It suffice to list the three requirements and indicate whether each one is satisfied or not.

Solution:

The above implementation **does enforce mutual exclusion**. The `fetch-and-add` is atomic and a process enters the critical section only when the value of lock is 0 and atomically increments the value lock, preventing other processes from entering their critical sections.

The above implementation **allows progress**. If no process is in its critical section, the value of the lock is 0, and any process that wishes to enter its critical section may enter.

The above implementation depends on a hardware atomic instruction and spins. Similar to the TAS there can be no guarantee on how long a process will wait to enter its critical section. Hence, the implementation is not fair (i.e., there is **no bounded wait**).

Question 7. [15 MARKS]

This question is about mutual exclusion.

Consider the following definition of a linked-list node and a routine that can be used to add a node at the end of an existing list. Assume that two processes that share the linked-list execute this routine.

```

struct node{
    int data;
    struct node *next;
}
struct node *head;

add_node (struct node *p) {
    struct node *temp;

    if (head == NULL) {
        head = p;
        return;
    }
    temp = head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = p;
    return;
}

```

Part (a) [5 MARKS]

Assume that each process is executing on a separate processor. Describe a scenario that illustrates what can go wrong when the two processes add nodes to the list concurrently.

Solution:

If the list is initially empty and two processes attempt to add a node concurrently, both may read the value of head as NULL, and both may assign their respective values of p to head, which results in an incorrect list.

Part (b) [5 MARKS]

Can this problem occur if the two processes are executing on a single processor system? If yes, describe a scenario that shows the problem. If not, explain why.

Solution:

Yes, the problem can occur. In this case a timer interrupt after the first process reads the value of head as NULL will cause the same problem described as in part a.

Part (c) [5 MARKS]

Show how the problem in part a. can be corrected.

Solution:

```
add_node (struct node *p) {
    struct node *temp;

    /*****/
    lock(mutex);
    if (head == NULL) {
        head = p;
    }
    /*****/
    unlock(mutex);
    return;
}
temp = head;
while (temp->next != NULL) temp = temp->next;
temp->next = p;
/*****/
unlock(mutex);
return;
}
```