

Duration: 120 minutes  
Aids Allowed: None

Student Number:

Last Name: SOLUTION

First Name: \_\_\_\_\_

Instructor: \_\_\_\_\_

---

*Do not turn this page until you have received the signal to start.  
Do not use PENCIL for writing down the answers to the questions.  
(In the meantime, please fill out the identification section above,  
and read the instructions below carefully.)*

---

MARKING GUIDE

# 1: \_\_\_\_\_/ 20

# 2: \_\_\_\_\_/ 18

# 3: \_\_\_\_\_/ 10

# 4: \_\_\_\_\_/ 12

# 5: \_\_\_\_\_/ 20

# 6: \_\_\_\_\_/ 20

TOTAL: \_\_\_\_\_/100

This midterm consists of 6 questions on 16 pages (including this one), printed on one side of the paper. *When you receive the signal to start, please make sure that your copy of the test is complete. If you need more space, use the reverse side of the page and indicate clearly the part of your work that should be marked.*

**Question 1.** [20 MARKS]

This question consists of **multiple small questions** about the material covered in the lectures and in the assignments. A short answer for each question will suffice.

**Part (a)** [1 MARK]

In OS/161, what is the purpose of the function "splhigh()"?

Solutions: Both answers are fine:

- set interrupt level to highest possible value
- disable interrupts

**Part (b)** [1 MARK]

In OS/161, is the cat-mouse problem implemented in kernel space or in user space?

Solutions:

- in kernel space (user space is really only implemented in A2)

**Part (c)** [1 MARK]

In OS/161, what function(s) handle(s) a context switch? Both answers are fine:

- either mi\_switch, md\_switch or mips\_switch
- fine if only one is listed

**Part (d)** [1 MARK]

The function to wakeup threads in OS/161 is given below (assertions and comments have been removed.) Describe a simple way to have this function wakeup only one thread?

```

/*
 * Wake up one or more threads who are sleeping on "sleep address" ADDR.
 */
void thread_wakeup(const void *addr){
    int i, result;

    for (i=0; i<array_getnum(sleepers); i++) {
        struct thread *t = array_getguy(sleepers, i);
        if (t->t_sleepaddr == addr) {
            array_remove(sleepers, i);
            i--;
            result = make_runnable(t);
        }
    }
}

```

Solution: Insert a break statement in the if statement, which would wakeup the first thread found only.

```

...
if (t->t_sleepaddr == addr) {
    array_remove(sleepers, i);
    result = make_runnable(t);
    break;
}
...

```

**Part (e)** [1 MARK]

Why is there a while loop around `thread_wakeup` in the semaphore implementation in OS/161?

```
// P(...) is like wait(...) in our textbook
void P(struct semaphore *sem){
    ...
    while (sem->count==0) {
        thread_sleep(sem);
    }
    ...
}
```

In V, `thread_wakeup` is called, which wakes up one or more threads waiting on the semaphore (see above question), but only one of the threads will successfully evaluate the condition (it would be false) in the while loop next, all others need to go back to sleep.

**Part (f)** [1 MARK]

What is a breakpoint in a debugger?

A breakpoint stops the execution of a program at the point defined by the breakpoint.

**Part (g)** [2 MARKS]

What is the proper sequence of cvs commands to include a new file, "new\_file.c," into your cvs repository?

- `cvs add new_file.c`
- `cvs commit` or `cvs commit new_file.c`

**Part (h)** [1 MARK]

Which environment variable points to the location of the cvs repository? `CVSROOT`

**Part (i)** [1 MARK]

You have executed "`cvs checkout module-name`" (the same as `cvs co` or `cvs checkout`) into an empty working directory. As you are working together with a partner, he or she has made modifications to the cvs repository. How do you synchronize your working directory with these changes made by your partner?  
`cvs update`

**Part (j)** [1 MARK]

What is a merge conflict? A merge conflict may occur during a cvs update, if changes in the cvs repository and working directory can not be resolved by cvs.

A line of code that some user has modified and committed to the repository has also been modified by the user updating his sources from the repository. In this situation, CVS can not determine what the correct line of code should be and flags the situation to the updating user by signaling a merge conflict, which has to be manually resolved by the updating user.

**Part (k)** [1 MARK]

How do you resolve a merge conflict? It has to be manually resolved.

**Part (l)** [2 MARKS]

Here are code samples for two threads that use binary semaphores. P(.) corresponds to wait(.) and V(.) to signal(.)

```
semaphore *mutex, *data;
void me() {
    P(mutex);
    /* do something */
    P(data);
    /* do something else */
    V(mutex);
    /* clean up */
    V(data);
}

void you() {
    P(data)
    P(mutex);
    /* do something */
    V(data);
    V(mutex);
}
```

Give a sequence of execution and context switches in which these two threads can deadlock. Use the notation, **function-name** : **operation**, to describe the execution sequence.

me: P(mutex); you: P(data); me: P(data); (blocks) you: P(mutex); (blocks)

At this point, **I** am blocking on data and **you** are blocking on mutex, and we never advance.

**Part (m)** [2 MARKS]

Propose a change to one or both of the threads from the previous sub-questions that makes deadlock impossible. You may simply describe the change, no need to write down the full code.

To fix the deadlock, make both threads acquire both semaphores in the same order.

```
semaphore *mutex, *data;
void me() {
    P(mutex);
    /* do something */
    P(data);
    /* do something else */
    V(mutex);
    /* clean up */
    V(data);
}

void you() {
    P(mutex)
    P(data);
    /* do something */
    V(data);
    V(mutex);
}
```

**Part (n)** [1 MARK]

In a user-level thread package, a thread issues a blocking system call. Would the operating system try to schedule another thread from the same process?

A priori **no**, since the OS does not know about the existence of other threads in user space

**Part (o)** [1 MARK]

A lock operation for a user-level thread implementation may look as follows:

lock:

```
TSL Register, Mutex
CMP Register, #0
JZE ok
CALL thread_yield
JMP lock
```

ok: RET

TSL is the atomic test-and-set instruction.

What is the purpose of the "thread\_yield" call? Both answers are fine:

- give up control to thread scheduler in user space
- avoid busy waiting and monopolizing the CPU by giving up control to the thread scheduler/thread runtime/thread library threads in user space

**Part (p)** [1 MARK]

You want to enforce the sequential execution of two processes  $P$  and  $Q$ .  $P$  and  $Q$  share the following variable semaphore `flag` and `flag` is initialized to 1. Is the following code correct?

```
Q:          P:
wait{flag)  P's code block
Q's code block  signal(flag)
```

Not correct. (Flag should be initialized to 0.)

**Part (q)** [1 MARK]

List five possible process states.

Any five of the below (others may make sense too): started (a.k.a. admitted to the system/created), running, ready, wait (a.k.a. blocked), terminated, zombie.

**Question 2.** [18 MARKS]

The **three-state process model** refers to a process or thread state model for multi-tasking systems that does not explicitly maintain a state for freshly created processes or for completed processes.

**Part (a)** [6 MARKS]

What are the **three** essential states of this process model and what do they **signify**, briefly explain?

The three states are: **Running**, the process is currently being executed on the CPU;

**Ready**, the process is ready to execute, but has not yet been selected for execution by the dispatcher; and

**Blocked** (a.k.a. **waiting**) where the process is not runnable as it is waiting for some event prior to continuing execution.

**Part (b)** [4 MARKS]

What transitions are possible between each of the states, you may list all transitions or draw a diagram with labeled states?

Possible **transitions** are (1) Running to Ready, (2) Ready to Running, (3) Running to Blocked, and (4) Blocked to Ready.

**Part (c)** [8 MARKS]

What causes a process (or thread) to undertake such a transition? Clearly indicate the cause and the corresponding transition.

**Events** that cause transitions:

It is fine if one of the correct events is listed with the right transitions.

- Running to Ready: time-slice expired, yield, or higher priority process becomes ready.
- Ready to Running: Dispatcher chose the next thread to run.
- Running to Blocked: A requested resource (file, disk block, printer, mutex) is unavailable, so the process is blocked waiting for the resource to become available.
- Blocked to Ready: a resource has become available, so all processes blocked waiting for the resource now become ready to continue execution.

**Question 3.** [10 MARKS]

Briefly describe a plausible sequence of activities that occurs when a timer interrupt results in a context switch.

Only the bolded items should be listed. The sequence must be correct.

1. The timer interrupt generates an interrupt (exception or trap) which transfers control to the kernel-mode handler. **Transfer of control to OS/ interrupt handler.**
2. The handler switches to the kernel stack base for the current process.
3. It saves the user-level state (registers, sp, ip) on the stack. **(interrupted processes context/state is saved.)**
4. **The interrupt is handled, i.e., the interrupt service routine executes.**
5. **The dispatcher (scheduler) is called and a new process is selected.**
6. The scheduler chooses a new process to run.
7. The current in-kernel context is saved on the kernel stack, the sp stored in the PCB (or TCB).
8. The new process's sp is loaded; the new kernel stack base is substituted for the old processes' stack base. **(new processes state is loaded.)**
9. The new processes' kernel context is restored.
10. It returns to the assembly routine to restore the user-level state.
11. The user-state of the new process (as opposed to the old) is restored.

**Question 4.** [12 MARKS]

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

<pre> <u>Process A</u> send-to(B) send-to(B) print "A1" receive-from(B) </pre>	<pre> <u>Process B</u> print "B1" receive-from(A) print "B2" receive-from(A) send-to(A) print "B3" </pre>
--	---

**Part (a)** [3 MARKS]

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

Solution: *B1 B2 A1 B3*

**Part (b)** [3 MARKS]

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

Solution: A1, B1, B2, B3  
 B1, A1, B2, B3  
 B1, B2, A1, B3  
 B1, B2, B3, A1

**Part (c)** [3 MARKS]

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

Solution: B1, B2, A1, B3

**Part (d)** [3 MARKS]

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

Solution: A1, B1, B2, B3  
 B1, A1, B2, B3  
 B1, B2, A1, B3  
 B1, B2, B3, A1

**Question 5.** [20 MARKS]

Below, an attempt has been made to solve the **Readers-Writers Problem** with the monitor construct.

```

reader(){
  while(TRUE){
    ...
    startRead();
    ... read the resource
    finishRead();
    ...
  }
}

writer(){
  while(TRUE){
    ...
    startRead();
    ... write the resource
    finishRead();
    ...
  }
}

main(){
  reader();
  writer();
  reader();
  writer();
}

```

The corresponding monitor implementation is given below.

```

monitor readerWriter{

  int numberOfReaders;
  int numberOfWriters;
  boolean busy;

  startRead(){
    while(numberOfWriters != 0);
    numberOfReaders++;
  }

  finishRead(){
    numberOfReaders--;
  }

  startWrite(){
    numberOfWriters++;
    while( (busy==TRUE) OR (numberOfReaders > 0) );
    busy=TRUE;
  }

  finishWrite(){
    numberOfWriters--;
    busy=FALSE;
  }

  initialization{
    numberOfReaders = 0;
    numberOfWriters = 0;
    busy = false;
  }
}

```

**Part (a)** [3 MARKS]

This solution to the Readers-Writers Problem **fails**. Explain **what** happens and **why** it fails.

The solution leads to **deadlock** or gets **stuck**.

Assume startWrite executes first. The monitor operation “startWrite” terminates and “writing the resource” takes place.

One of the below may occur.

1. either: next, another startWrite takes place (another writer). It will busy-wait on its while-loop, as “busy==TRUE” is true (set by first/previous writer). No other process/thread can enter the monitor, thus the system is stuck (deadlocked). (Note it does not relinquish control of the monitor, the actual problem in this solution.)
2. or: next, a startRead takes place (a new reader). It will busy-wait on its while-loop, as “numberOfWriters != 0” (it is 1). ... the system is stuck. No other process can enter the monitor.

**Part (b)** [1 MARK]

How do you suggest to solve the problem? Briefly describe your idea.

Relinquish control of monitor, rather than busy wait. Use condition variables instead. (That’s what they are for.)

**Part (c)** [14 MARKS]

The **First Readers-Writers Problem** gives priority to readers. That means no reader should ever be kept waiting, unless a writer is already writing. This problem formulation does not worry about starvation.

In the spirit of monitors, by using **all aspects** a monitor has to offer, implement a solution to the **First Readers-Writers Problem**. Use the code template below. **Only** use the variables given to you in the code template and **do not forget to initialize them**. **Note, the variable numberOfWriters is not required in the solution.**

```
monitor readerWriter{
    int numberOfReaders;
    boolean busy;
    condition okToRead, okToWrite;

    startRead(){
        //your code goes here

    }

    finishRead(){
        //your code goes here

    }

    startWrite(){
        //your code goes here

    }

    finishWrite(){
        //your code goes here

    }

    initialization{
        //your initialization code goes here

    }
}
```

```
monitor readerWriter{

    int numberOfReaders;
    boolean busy;

    condition okToRead, okToWrite;

    startRead(){
        numberOfReaders++;
        if (busy==TRUE)
            okToRead.wait();
        okToRead.signal();
    }

    finishRead(){
        numberOfReaders--;
        //last reader finishing
        if (numberOfReaders == 0)
            okToWrite.signal();
    }

    startWrite(){
        if ( (numberOfReaders != 0) OR (busy==TRUE) )
            okToWrite.wait();
        busy=TRUE;
    }

    finishWrite(){
        busy=FALSE;
        if (numberOfReaders != 0)
            okToRead.signal();
        else
            okToWrite.signal();
    }
}
```

**Part (d)** [2 MARKS]

We said that the first readers-writers problem may lead to starvation. Explain **which type of processes may be starved** and **when this may occur**.

Readers may starve writers.

If there is a stream of readers, no writer will ever be let access the resource, since `startReader` does not check, if there are willing writers. It only checks if there are any writers writing.

**Question 6.** [20 MARKS]

This question is about **implementing a monitor** with **semaphores**.

Below, the general monitor construct is depicted, as discussed in the lecture. The overall objective of this question is to work toward an implementation of the monitor construct based on the below code templates and the use of **semaphores**.

**For the questions below, assume that the monitor's signal operation, if present, is always the last piece of code in any of the monitor's operations.**

```
monitor monitor-name {  
  
    variable declarations  
  
    condition x, y, z;  
  
    procedure body P1 (...){  
        ...  
    }  
  
    procedure body P2 (...){  
        ...  
    }  
  
    ...  
  
    procedure body Pn (...){  
        ...  
    }  
  
    { initialization code }  
  
}
```

**Partial support code for below questions:**

Below you are given a partial code template that illustrates the support code that must be placed around a monitor operation to ensure the monitor's semantic. (**Remark:** The support code may be generated by a compiler for a language that supports monitors.)

```
wrapper-for-monitor-operation-Pi(...){
    wait(mutex);

    body of Pi

    signal(mutex);
}
```

**Part (a)** [2 MARKS]

The variable, `mutex`, is a semaphore, shared by all procedures using the monitor operations. Note, `signal(...)` and `wait(...)` in the support code are operations on semaphores not the monitor's operations on condition variables. What is the initialization value of `mutex`?

```
semaphore mutex = 1;
```

**Part (b)** [2 MARKS]

What does the support code ensure?

- mutually exclusive access of monitor operations

**Part (c)** [2 MARKS]

For each condition variable, `x`, a semaphore and an integer variable are defined as follows:

```
semaphore x_sem = .....
```

```
int x_count = .....
```

How must they be initialized? **Hint:** A partial code fragment of the signal operation on condition variables is provided in a subquestion below.

```
semaphore x_sem = 0; int x_count = 0;
```

**Part (d)** [4 MARKS]

The code template below constitutes a fragment of the implementation of the signal operation on condition variable, `x`. That is, it implements `x.signal()`. Complete the implementation. Dotted lines indicate that **code may be missing**. **Remember the assumption that if `x.signal` is used in any monitor operation, `Pi`, it is always the last piece of code in that operation.** Exploit this assumption in your implementation.

Monitor's `x.signal()` implementation:

```
if (x_count.....){
    ...
    signal(.....);
```

```

    ...
}
...

```

Monitor's `x.signal()` implementation:

```

if (x_count > 0){
    x_count--;
    signal(x_sem);
}

```

**Part (e)** [6 MARKS]

Based on the above code templates, semaphore definitions, and variable definitions, design the corresponding support code for the wait operation on condition variable, `x`. That is, implement the `x.wait()` operation.

```

//x.wait()

x_count++;
signal(mutex);
wait(x_sem);
wait(mutex);

```

**Part (f)** [2 MARKS]

Briefly describe the purpose of `x_sem` in the above implementations of the monitor's signal and wait operations on the condition variable `x`.

`x_sem` implements waiting (busy or queued) for the condition variable `x`.

**Part (g)** [2 MARKS]

Briefly describe the purpose of `x_count` in the above implementations of the monitor's signal and wait operations on the condition variable `x`.

`x_count` counts the number of processes/threads that have called `x.wait` on the condition variable `x`. It is used to decide whether the monitor's signal operation should have an effect or not.