# University of Waterloo
# Midterm Examination Model Solution

## Fall, 2008

**1. (10 total marks)**

The OS/161 `addrspace` structure shown below describes the address space of a running OS/161 process. The virtual page size is ~~4096 bytes (0x1000 bytes)~~ 64K bytes ($2^{16}$ bytes). Assume that OS/161 gives each process a stack segment consisting of `16` pages, ending at virtual address `0x7fffffff`.

**Note:** On the original exam, the page size was given as 4K bytes. With that page size, the answer for each virtual address in part (a) is `INVALID VIRTUAL ADDRESS` and the answer for each physical address in part (b) is `UNMAPPED PHYSICAL ADDRESS`. The answers shown below are for a page size of $64K$ bytes.

```
struct addrspace {
  /* text segment */
  vaddr_t as_vbase1 = 0x05800000; /* virtual base address */
  paddr_t as_pbase1 = 0x10400000; /* physical base address */
  size_t as_npages1 = 0x20; /* number of pages */

  /* data segment */
  vaddr_t as_vbase2 = 0x20000000; /* virtual base address */
  paddr_t as_pbase2 = 0x00400000; /* physical base address */
  size_t as_npages2 = 0x1a0; /* number of pages */

  /* stack segment */
  paddr_t as_stackpbase = 0x10800000; /* physical base address */
};
```

**a. (5 marks)**

For each of the following virtual addresses, indicate which physical address the virtual address maps to. Also indicate whether the virtual address falls into the text segment, the data segment, or the stack segment. If the virtual address is not part of any segment, write `INVALID VIRTUAL ADDRESS` instead. Use hexadecimal notation for physical addresses.

- `0x20ffaa60` → `0x013faa60 (data segment)`
- `0x05b20104` → `INVALID VIRTUAL ADDRESS`
- `0x05960006` → `0X1056006 (text segment)`

**b. (5 marks)**

For each of the following physical addresses, indicate which virtual address maps to that address. Also indicate whether that virtual address falls into the text segment, the data segment, or the stack segment. If no virtual address maps to the specified physical address, write `UNMAPPED PHYSICAL ADDRESS` instead. Use hexadecimal notation for virtual addresses.

- `0x1081f3f0` → `0x7ff1f3f0 (stack segment)`
- `0x01dd0890` → `0x219d0890 (data segment)`
- `0x10701114` → `UNMAPPED PHYSICAL ADDRESS`

**2. (10 marks)**

An OS/161-like operating system is being designed to use round robin scheduling of threads. The hardware includes a timer device which will generate interrupts at a rate of $T$ interrupts per second. The desired scheduling quantum is $Q$ seconds. You may assume that $1/T$ divides evenly into $Q$, i.e., that the product $QT$ is a positive integer.

Your task is to write two kernel functions that will form part of the round robin scheduler implementation. These two functions are:

`OnTimerInterrupt()`: This function will be invoked by the timer interrupt handler each time there is a timer interrupt. This function should decide whether the currently-running thread's quantum has expired and, if it has expired, this function should cause that thread to yield the processor, so that a new thread can begin its quantum.

`OnDispatch()`: This function will be invoked each time a thread is dispatched by the scheduler. It can be used to indicate that the new thread's quantum is beginning.

Write these two functions below in C-like pseudo-code. You may declare and use any global or local variables you need. You may also call any of the OS/161 thread functions (e.g., `thread_sleep()`, `thread_yield()`, `thread_exit()`) that you need. You may also refer to the constants $Q$ and $T$ in your code.

These functions need not be complicated, so keep your implementations as simple as possible. Overly complicated solutions may be penalized.

```
int RemainingTicks;  /* global.  Initial value unimportant. */

OnTimerInterrupt() {
  RemainingTicks = RemainingTicks - 1;
  if (RemainingTicks == 0) {
     thread_yield();
  }
}

OnDispatch() {
  RemainingTicks = Q*T;
}
```

**3. (10 total marks)**

Consider a concurrent program with many threads. There are two types of threads. Type A threads repeatedly call `ProcedureA`, shown below. Type B threads repeatedly call `ProcedureB`.

These two procedures are synchronized using three semaphores, `semX`, `semY`, and `semZ`. You should assume that `semX` and `semY` each have an initial value of `1`, and that `semZ` has an initial value of `2`.

`ProcedureA` and `ProcedureB` also make calls to `ProcedureX`, `ProcedureY`, and `ProcedureZ`. These procedures are not shown. However, they do not make any use of synchronization primitives.

```
ProcedureA() {
  P(semX);
  ProcedureX();
  P(semZ);
  V(semX);
  ProcedureZ();
  V(semZ);
}
```

```
ProcedureB() {
  P(semY);
  ProcedureY();
  P(semZ);
  V(semY);
  ProcedureZ();
  V(semZ);
}
```

**a. (3 marks)**
Is it possible that two threads will execute `ProcedureX()` concurrently? Answer yes or no. For full credit, briefly (one sentence) justify your answer.

NO. Mutual exclusion for `ProcedureX()` is enforced by `semX`.

**b. (2 marks)**
Is it possible that some thread will execute `ProcedureX()` at the same time that another thread is executing `ProcedureY()`? Answer yes or no. For full credit, briefly (one sentence) justify your answer.

YES. `ProcedureX()` and `ProcedureY()` are protected by different semaphores. A Type A thread can execute `ProcedureX()` at the same time that a Type B thread is executing `ProcedureY()`.

**c. (2 marks)**
Is it possible that some Type A thread will execute `ProcedureX()` at the same time that another Type A thread is executing `ProcedureZ()`? Answer yes or no. For full credit, briefly (one sentence) justify your answer.

YES. `V(semX)` occurs before the call to `ProcedureZ()`.

**d. (4 marks)**
Is it possible that a deadlock will occur among these threads? Answer yes or no. If deadlock is possible, give a concrete example of a sequence of events (i.e., procedure calls) that would result in a deadlock. Be sure to indicate which thread performs each event in your example. If deadlock is not possible, indicate which of the deadlock prevention techniques (if any) discussed in class is being used by `ProcedureA` and `ProcedureB` to prevent deadlocks.

Deadlock is not possible.
Deadlock is prevented by ordering resources. For example, the order `semX`, `semY`, `semZ` is consistent with the way the resources (semaphores) are acquired by these applications.

**4. (10 marks)**

Re-implement `ProcedureA` and `ProcedureB`, from the previous question, using locks and condition variables instead of semaphores. Your implementation should provide the same synchronization as the original implementation, but using only locks and condition variables - no semaphores. You may also define and used additional shared global variables, if necessary. List any locks, condition variables, and global variables that you use. Be sure to specify an initial value for every global variable that you use.

The original implementations of `ProcedureA` and `ProcedureB` are repeated here, for your convenience. Recall that `semX` and `semY` each have an initial value of `1` and that `semZ` has an initial value of `2`.

```
ProcedureA() {
  P(semX);
  ProcedureX();
  P(semZ);
  V(semX);
  ProcedureZ();
  V(semZ);
}
```

```
ProcedureB() {
  P(semY);
  ProcedureY();
  P(semZ);
  V(semY);
  ProcedureZ();
  V(semZ);
}
```

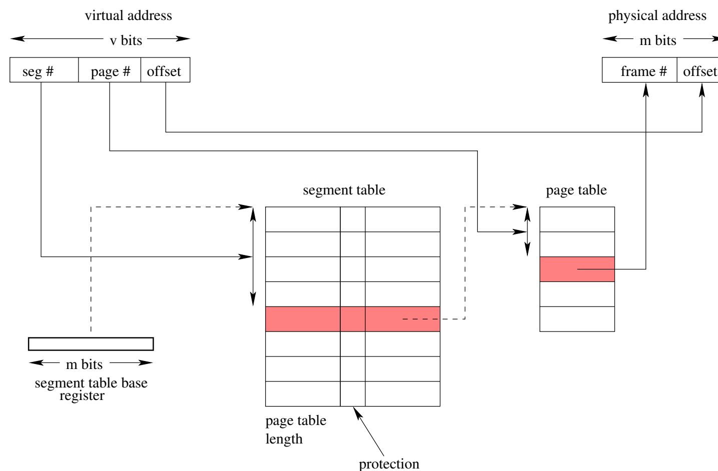
`Write your answer to Question 4 in the template below.`

```
List Lock, Condition Variable and Global Variable Declarations Here:

    locks: lockX, lockY, lockZ;
    condition variables: ZNotBusy;
    global integer variable: NumInZ, initially 0;
```

```
ProcedureA() {

  acquire(lockX);

    ProcedureX();

  acquire(lockZ);
  while (NumInZ >= 2) {
    wait(ZNotBusy);
  }
  NumInZ = NumInZ + 1;
  release(lockZ);
  release(lockX);

    ProcedureZ();

  acquire(lockZ);
  NumInZ = NumInZ - 1;
  if (NumInZ < 2) {
     signal(ZNotBusy);
  }
  release(lockZ);
}
```

```
ProcedureB() {

  acquire(lockY);

   ProcedureY();

  acquire(lockZ);
  while (NumInZ >= 2) {
    wait(ZNotBusy);
  }
  NumInZ = NumInZ + 1;
  release(lockZ);
  release(lockY);

   ProcedureZ();

  acquire(lockZ);
  NumInZ = NumInZ - 1;
  if (NumInZ < 2) {
     signal(ZNotBusy);
  }
  release(lockZ);
}
```

**5. (10 total marks)**




A virtual memory system uses paged segmentation. Translation of virtual addresses to physical addresses is described by the translation diagram shown above, which is from the course notes. The size of a virtual address in this system is 4 GB ($2^{32}$ bytes). The size of a physical address is also 4 GB ($2^{32}$ bytes). The maximum size of a single segment is 512 MB ($2^{29}$ bytes). The page size is 8 KB ($2^{13}$ bytes). Answer the following questions about this virtual memory system.

**a. (3 marks)**
Virtual addresses are broken into components (segment number, page number, and offset) as shown in the translation diagram. Give the size, in bits, of each of the three components.

**segment number:** → `3 bits`

**page number:** → `16 bits`

**offset:** → `13 bits`

**b. (2 marks)**
Physical addresses are broken into components (frame number, offset) as shown in the translation diagram. Give the size, in bits, of each of these components.

**frame number:** → `19 bits`

**offset:** → `13 bits`

**c. (2 marks)**
What is the maximum possible number of segment table entries for any process?

8 entries

**d. (1 marks)**
What is the maximum possible number of page table entries in any segment's page table?

$2^{16}$ entries

**e. (2 marks)**
Suppose that a particular process has 4 segments: text, data, heap and stack. The sizes of these segments are as follows: 64 KB ($2^{16}$ bytes) of text, 1 MB ($2^{20}$ bytes) of data, 1 MB ($2^{20}$ bytes) of heap, and 16 MB ($2^{24}$ bytes) of stack. What is the total number of valid page table entries that the kernel will maintain for this process? You may give your answer as a sum of terms.

$2^{16}/2^{13} + 2^{20}/2^{13} + 2^{20}/2^{13} + 2^{24}/2^{13} = 2^3 + 2^7 + 2^7 + 2^{11}$

**6. (10 total marks)**

**a. (6 marks)**

Consider a system with two processes, `ProcA` and `ProcB`. `ProcA` is running and `ProcB` is ready. Suppose that `ProcA` causes an arithmetic overflow exception while it is running, and that the kernel's response to this exception is to terminate `ProcA` and allow `ProcB` to run.

List the sequence of events that will occur when `ProcA` causes the arithmetic overflow exception. Create your list by choosing from the events shown below, using the event numbers to identify events. For example, a valid answer might be "7,5,4,8,5,2". Note that you may include an event more than one time in the sequence if it occurs more than once.

1. a `syscall` instruction occurs
2. privileged (kernel) execution mode is entered
3. unprivileged execution mode is entered
4. `ProcA`'s application state is saved into a trap frame
5. `ProcB`'s application state is saved into a trap frame
6. `ProcA`'s application state is restored from a trap frame
7. `ProcB`'s application state is restored from a trap frame
8. context switch to `ProcB`'s thread
9. context switch to `ProcA`'s thread
10. `ProcB`'s application code starts executing
11. the kernel determines which type of exception occurred

Write your answer here:

| 2,4,11,8,7,3,10 |
|---|

**b. (2 marks)**

On the MIPS, how does an application cause a system call to occur? How does the kernel know which system call the application is requesting?

| An application causes a system call by executing a `syscall` instruction. The kernel knows which system call is being requested by looking in register `v0` for a system call code, which is put there by the application just before it makes the system call. |
|---|

**c. (2 marks)**

What is the purpose of the OS/161 kernel function `splhigh()`?

| `splhigh()` disables interrupts by raising the processor's interrupt level. |
|---|