# CS 350 - Winter 2011 Midterm - Sample Solution

| | |
|---|---|
| Course | CS350 - Operating Systems |
| Sections | 01 (11:30), 02 (16:00) |
| Instructor | Ashraf Aboulnaga |
| Date of Exam | February 15, 2011 |
| Time Period | 19:00-21:00 |
| Duration of Exam | 120 minutes |
| Number of Exam Pages (including this cover sheet) | 10 pages |
| Exam Type | Closed Book |
| Additional Materials Allowed | None |

**Please make your answers as concise as possible. You do not need to fill the whole space provided for answers.**

| | | |
|---|---|---|
| Question 1: (10 marks) | Question 2: (12 marks) | Question 3: (14 marks) |
| Question 4: (12 marks) | Question 5: (12 marks) | |
| Total: (60 marks) | | |

**1. (10 total marks)**

Write **T** or **F** next to each of the following statements, to indicate whether it is True or False.

**a. (1 mark)** An interrupt is ignored if it occurs while another interrupt is being handled by the kernel.

F

**b. (1 mark)** Two devices can each generate interrupts at the same time.

T

**c. (1 mark)** Division by zero causes an interrupt.

F

**d. (1 mark)** A thread can belong to more than one process.

F

**e. (1 mark)** A thread executing in the kernel cannot be preempted.

F

**f. (1 mark)** Disabling interrupts on all processors in a multi-processor system can guarantee mutual exclusion.

F

**g. (1 mark)** A deadlock can occur with only one resource type.

T

**h. (1 mark)** The MMU can do a TLB lookup for a running thread only if this thread is in privileged (i.e., kernel) mode.

F

**i. (1 mark)** A software controlled TLB requires a smaller page table than a hardware controlled TLB.

F

**j. (1 mark)** The number of pages in a program's address space is determined only by information that is in the program's ELF file.

F

**2. (12 total marks)**

**a. (4 marks)**

When a thread is preempted in round robin scheduling, two thread contexts are saved on that thread's kernel stack. Explain why two thread contexts are saved and not just one.

- The first context is for:

  The values of the registers when the timer interrupt occurs.

- The second context is for:

  The values of the registers when the thread scheduler decides to yield the current thread.

**b. (4 marks)**

Consider a virtual memory system that uses segmentation combined with paging, and has the following parameters:

- Each process has 8 segments.
- Each segment can be up to 8MB in size.
- Each page is 8KB in size.

Draw a diagram showing the different fields of a virtual address in this system. Make sure to label each field and indicate how many bits it contains. (Some useful information: $1\text{K} = 2^{10}$ and $1\text{M} = 2^{20}$).

**c. (2 marks)**

When a process makes a system call in OS/161, how does the kernel know which system call is being requested?

| |
|---|
| The process places the system call number in register v0. |

**d. (2 marks)**

In a program with multiple threads (such as the cat-mouse simulation in Assignment 1), different executions of the program with the same inputs can result in different outputs. Why?

| |
|---|
| The thread scheduler can interleave the execution of the threads in different ways in different executions of the program, resulting in different outputs for the same inputs. |

**3. (14 marks)**

Consider a virtual memory system that uses paging. Virtual and physical addresses are both 32 bits long, and the page size is 4KB = $2^{12}$ bytes. A process $P_1$ has the following page table. Frame numbers are given in notation (recall that each hexadecimal digit represents 4 bits).

| | Frame Number |
|---|---|
| 0 | 0x00788 |
| 1 | 0x00249 |
| 2 | 0x0023f |
| 3 | 0x00ace |
| 4 | 0x00bcd |

**a. (3 marks)**

For each of the following virtual addresses, indicate the physical address to which it maps. If the virtual address is not part of the address space of $P_1$, write `NO TRANSLATION` instead. Use hexadecimal notation for the physical addresses.

- `0x00001a60`

  `0x00249a60`

- `0x000051ff`

  `NO TRANSLATION`

- `0x00000fb5`

  `0x00788fb5`

**b. (3 marks)**

For each of the following physical addresses, indicate the virtual address that maps to it. If the physical address is not part of the physical memory assigned to $P_1$, write `NO TRANSLATION` instead. Use hexadecimal notation for the virtual addresses.

- `0x00aceff6`

  `0x00003ff6`

- `0x00249000`

  `0x00001000`

- `0x00000887`

  `NO TRANSLATION`

**c. (8 marks)**

Below is the `addrspace` structure used by OS/161 when using the `dumbvm` virtual memory implementation:

```
struct addrspace {
  vaddr_t as_vbase1; /* base virtual address of code segment */
  paddr_t as_pbase1; /* base physical address of code segment */
  size_t as_npages1; /* size (in pages) of code segment */
  vaddr_t as_vbase2; /* base virtual address of data segment */
  paddr_t as_pbase2; /* base physical address of data segment */
  size_t as_npages2; /* size (in pages) of data segment */
  paddr_t as_stackpbase; /* base physical address of stack */
};
```

Recall that a page in OS/161 is 4KB ($2^{12}$).

Below are four addresses that should not be used for `as_vbase1` or `as_vbase2`. For each of these addresses, give a reason why it should not be used.

- `0x0044f892`

  This address does not represent a page boundary (i.e., not divisible by $2^{12}$ = 0x1000).

- `0x00000000`

  Address 0x0 represent a null pointer. A null pointer should not represent a valid address.

- `0x83410000`

  This address is in the upper half of the address space, which is reserved for the kernel.

- `0x7ffff000`

  This address is in the range of virtual addresses used for the stack.

**4. (12 marks)**

Consider two threads running in the OS/161 kernel. One thread runs the function `thread1_fn` (listed below) and the other runs the function `thread2_fn`. The functions use two semaphores `s` and `m`, which are shared between the two threads.

```
void thread1_fn (void) {
  for (;;) {
    P(s);
    P(s);
    P(m);
    kprintf("a");
    V(m);
  }
}

void thread2_fn (void) {
  for (;;) {
    P(m);
    kprintf("b");
    V(m);
    V(s);
  }
}
```

**a. (2 marks)**

If the initial values of the semaphores are `s` = 0 and `m` = 1, list the first 10 characters of one possible output produced by the concurrent execution of these two threads.
If no output will be produced, explain why.

*bbabbbbaab*
Any string that starts with 2 *b*'s, and in which every *a* is preceded (not necessarily immediately) by 2 *b*'s.

**b. (2 marks)**

If the initial values of the semaphores are `s` = 2 and `m` = 1, list the first 10 characters of one possible output produced by the concurrent execution of these two threads, **different from the output that you listed in part (a)**.
If no output will be produced, explain why.

*abbaabbabb*
Any string in which the number of *a*'s not preceded by 2 *b*'s is at most 1.

**c. (2 marks)**

If the initial values of the semaphores are `s` = 2 and `m` = 0, list the first 10 characters of one possible output produced by the concurrent execution of these two threads, **different from the outputs that you listed in parts (a) and (b)**.
If no output will be produced, explain why.

No output will be produced since both threads will be blocked in the `P(m)` call and will never be unblocked.

**d. (6 marks)**

For the initial values of the semaphores in parts (a), (b), and (c), are the characters that you listed the only possible characters that can be produced at the start of the output? If yes, explain why. If no, list another 10 characters that can appear as the first characters of the output, and explain what determines whether one or the other of the character sequences appear.

- part (a)

  Other outputs are possible. For example, *bbbbaabbba*.
  The output produced depends on how the thread scheduler interleaves the execution of the two threads.

- part (b)

  Other outputs are possible. For example, *bbbbaaaabb*.
  The output produced depends on how the thread scheduler interleaves the execution of the two threads.

- part (c)

  No output will ever be produced since the threads will always be blocked.

**5. (12 total marks)**

**(Space for answer to Question 5. Question is on next page.)**

List shared locks, semaphores, condition variables, or other shared global variables here (and explain the purpose of each one):

```
  struct semaphore *two_active; /* To ensure that at most two threads are active.
                                   Initial value is 2. */
  volatile int num_active; /* Count of active threads to decide which of the two
                              functions to call. Initial value is 0. */
  struct lock *active_lock; /* To ensure mutual exclusion in accessing num_active. */
  struct lock *cs_lock; /* To ensure that the two functions i_am_first and
                           i_am_second are in the same critical section. */

  /* The following lines are for initialization. They did not need to be included
     in the answer to this question. */
  two_active = sem_create("two_active", 2);
  active_lock = lock_create("active_lock");
  cs_lock = lock_create("cs_lock");
  num_active = 0;
```

```
void thread_fn (void) {
```

```
  int i;
  int one_active;
  for (i = 0; i < 5; i++) {
    P(two_active);

    lock_acquire(active_lock);
    assert(num_active < 2);
    one_active = num_active;  /* one_active will be 0 or 1. */
    num_active++;
    lock_release(active_lock);

    lock_acquire(cs_lock);
    if (one_active) {  /* one_active == 1 */
      i_am_second();
    } else {
      i_am_first();
    }
    lock_release(cs_lock);

    lock_acquire(active_lock);
    num_active--;
    lock_release(active_lock);

    V(two_active);
  }
}
```

**(You can tear off this page of the exam if you want.)**

Consider a simulation that runs in the OS/161 kernel, somewhat similar to the cat-mouse simulation of Assignment 1. In this simulation, there are $N$ threads, and they are all of the same type. Each thread runs the function `void thread_fn(void)`. The rules of the simulation are as follows:

- There can be at most 2 concurrently running threads at any time. We will call these the *active* threads. If $N > 2$, there should be 2 active threads, and the $N - 2$ remaining threads should be waiting to become active.
- When a thread is first created, it tries to become active. If there are already two active threads, then the newly created thread waits.
- When a thread $t$ becomes active, it calls one of two functions. If $t$ is the only active thread, it calls the function `i_am_first()`. If there is already an active thread when $t$ becomes active, $t$ calls the function `i_am_second()`. Obviously, there can be no more than one thread already active when $t$ become active, since we can have at most two active threads.
- The two functions `i_am_first()` and `i_am_second()` are part of the same critical section. These functions do not take arguments or return values, and you can assume that they are already defined.
- After the function `i_am_first()` or `i_am_second()` returns, the active thread $t$ that called the function gives one of the waiting threads a chance to become active, and thread $t$ waits for its next chance to become active.
- After a thread as become active 5 times (and therefore called `i_am_first()` or `i_am_second()` a total of 5 times), the function `thread_fn` exits.

On the previous page, fill in the body of the function `thread_fn` so that the above rules are followed. List the synchronization primitives and shared global variables used and explain the purpose of each one. You may use locks, condition variables, or semaphores for synchronization, but you may not use any other synchronization mechanisms (for example, you may not disable interrupts).

There are three synchronization tasks in this problem:

1. Deciding when a thread becomes active.
2. After a thread becomes active, deciding which of the two functions to call.
3. Ensuring mutual exclusion for `i_am_first()` and `i_am_second()`.

The solution on the previous page uses a semaphore for task 1, a shared variable protected by a lock for task 2, and a lock for task 3. Solutions that use other synchronization primitives are also possible.