CS 350 - Fall 2012 Midterm - Sample Solution

Course	CS350 - Operating Systems
Sections	01 (8:30), 02 (10:00), 03 (14:30)
Instructor	Dave Tompkins (Sec 1) & Ashraf Aboulnaga (Sec 2 and 3)
Date of Exam	October 30, 2012
Time Period	19:00-21:00
Duration of Exam	120 minutes
Number of Exam Pages (including this cover sheet)	11 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: (12 marks)	Question 2: (9 marks)	Question 3: (14 marks)
Question 4: (10 marks)	Question 5: (15 marks)	
Total: (60 marks)		

1. (12 marks)

a. (2 marks) Explain why spin locks are widely used on multiprocessors to enforce mutual exclusion.

Spin locks are widely used on multiprocessors because disabling interrupts is not effective in a multiprocessor environment.

OR

Spin locks are widely used on multiprocessors because threads often wait for a lock only a short amount of time so the overhead of spinning is lower than the overhead of blocking.

b. (4 marks) Some operating systems (including earlier versions of Linux) have a single kernel stack that is used by all threads when they enter the kernel.

(i) Explain how such an operating system would manage trap frames and thread contexts *differently* than OS/161.

An O/S with a single kernel stack could not store the thread context information and the trap frame on the stack. Instead, the O/S would have to store the thread context and trap information in the thread container (structure / object).

(ii) One advantage of using a single kernel stack is to conserve memory. Provide one significant disadvantage.

The O/S would not work on a multi-processor architecture.

OR

The O/S would not allow threads to execute concurrently in the kernel.

c. (2 marks) Briefly describe two of the methods (discussed in class) an operating system could use to prevent deadlocks.

Any two of:

- No Hold and Wait (prevent a thread from requesting more than one resource at a time)
- Preemption (take resources away from a thread and give them to another)
- Resource Ordering (number the resource types, and require requests to be in order)

d. (2 marks) You decide you want to decrease the *quantum* in OS/161 to half of its default value. Briefly describe an advantage and a disadvantage of such a change.

- Advantage: The O/S would become more responsive (or more fine-grain control of process scheduling)
- Disadvantage: The O/S would have greater overhead from more frequent context switches

e. (2 marks) Explain how a relocation register facilitates Dynamic Relocation and why the MMU might want to maintain an additional size register in addition to the relocation register.

In Dynamic Relocation, a relocation register is required to store the physical offset for the virtual memory for each process. The physical address is determined by adding the virtual memory address to the relocation register. A size register is useful to prevent processes from accessing physical memory that belongs to other processes (this would happen if the process accesses an address that exceeds its virtual address space).

2. (9 marks)

a. (5 marks) (i) Explain why in OS/161 the while loop on the left cannot be replaced with an `if` statement (as shown on the right). (ii) Briefly describe a sequence of events where the left (`while`) implementation would work, and the right (`if`) implementation would not:

```
void P(struct semaphore *sem)
{
  /* ...interrupts off... */
  while (sem->count==0) {
    thread_sleep(sem);
  }
  sem->count--;
  /* ...interrupts on... */
}
```

```
void P(struct semaphore *sem)
{
  /* ...interrupts off... */
  if (sem->count==0) {
    thread_sleep(sem);
  }
  sem->count--;
  /* ...interrupts on... */
}
```

i) [2 marks]
Because there could be more than one thread waiting on a P() (and/or because another thread could execute a P() before the waiting thread becomes active).

ii) Sequence of events: [3 marks]

1. Semaphore S is created and initialized to 0
2. Thread A performs P(S). Calls thread_sleep(S). Thread A is now blocked.
3. Thread B performs V(S). Calls thread_wakeup(S). Thread A is now ready.
4. Thread C performs P(S) before thread A becomes active.

b. (4 marks) Dilbert is writing some code for the OS/161 kernel. He wants to start a new thread (`foo`) and then wait until it has completed some initialization code before the original thread can continue. His implementation is below. His co-worker Wally thinks that this code won't always work as Dilbert expects. Do you agree with Wally? Briefly justify your answer. If you agree that there is a problem, briefly describe how you would change the code to fix this problem.

```
// global variables (assume they are initialized properly)
struct lock *lck;
struct cv *done;

void dilbert()
{
  thread_fork("dil",NULL,0,foo,NULL);
  lock_acquire(lck);
  cv_wait(done, lck);
  lock_release(lck);
  /* ... continue ... */
}
```

```
void foo (void *p, unsigned long n)
{
  /* ... initialize stuff ... */
  lock_acquire(lck);
  cv_signal(done, lck);
  lock_release(lck);
  /* ... continue ... */
}
```

[1 mark]
I agree with Wally – a problem could occur.

[2 marks]
In between the thread_fork() and the lock_acquire() in dilbert, a context switch could occur and the foo thread could become active. The foo thread could acquire the lock, cv_signal and then release the lock before the dilbert thread becomes active again. The dilbert thread would then acquire the lock and then wait (forever) for the signal that had already occurred.

[1 mark]
Several options to fix the problem, including:

- Move the lock_acquire() before the thread_fork()
- Add code inside dilbert for something like: while (! initialized) cv_wait, and then inside foo, set initialized to true before signalling.
- Use a semaphore instead of a lock.

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 hexadecimal notation (recall that each hexadecimal digit represents 4 bits).

	Frame Number
0	0x0014e
1	0x03b65
2	0x00351
3	0x00875
4	0x06a3f

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.

- `0x00003b65`

  0x00875b65

- `0x00006a3f`

  NO TRANSLATION

- `0x00000fe6`

  0x0014efe6

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.

- `0x00351fff`

  0x00002fff

- `0x03b65000`

  0x00001000

- `0x000e3000`

  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}).

Which of the following four addresses can be used for `as_vbase1` and which of the addresses cannot be used? If the address can be used, write `VALID` next to it. If the address cannot be used, write `INVALID` next to it and briefly explain why it cannot be used.

- `0x0046ae93`

  INVALID: as_vbase1 must be page-aligned. That is, must be an address divisible by 0x1000.

- `0x00405000`

  VALID

- `0x8fe40000`

  INVALID: address is in kernel space

- `0x7ffff000`

  INVALID: address is part of stack space

4. (10 marks)

In Assignment 1, you were required to synchronize the behaviour between cats and mice. In this question we add a third species (dogs) that can also eat from the same bowls. Just as cats can eat mice if they try to eat at the same time, dogs can eat cats or mice. All of the restrictions from Assignment 1 are the same (e.g.: no two animals can be eating at the same bowl at the same time).

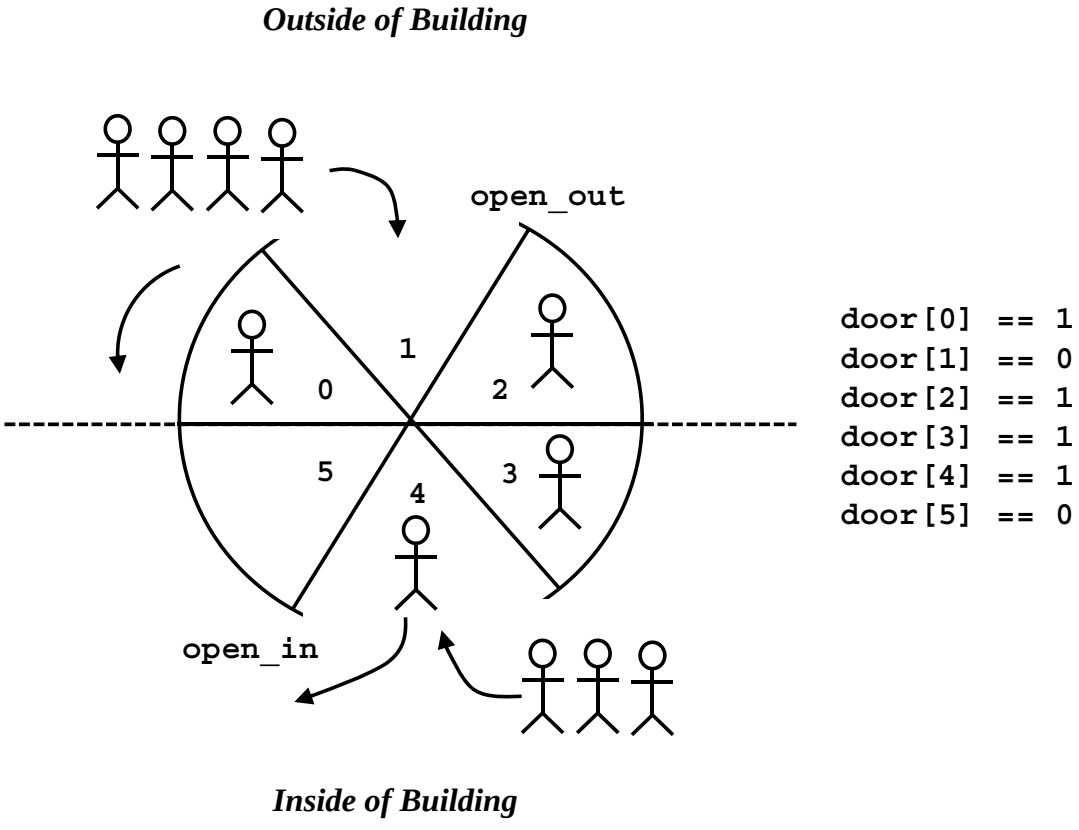
- **(7 marks)** List all of the synchronization primitives and shared variables that you would use to synchronize the cats, mice, and dogs and briefly explain how they would be used.
- **(3 marks)** Briefly justify why it is impossible for an animal to starve under your synchronization technique.

Note: You do not need to provide any code. You are only required to describe your design as you did in your Assignment 1 design document.

Similar to the cats and mice simulation in Assignment 1.

5. (15 marks)

For this question, you are required to write code that simulates several people entering and leaving a building through a revolving door. The details of the problem and the synchronization requirements are given below. A partially implemented solution is given on the last page of this exam. You are required to add code to this solution to ensure that it satisfies all the requirements of the problem. The following figure illustrates the variables used to represent the problem.



- The door has `N` compartments. Assume that N is an even number ≥ 4.
- Each compartment can hold at most 1 person.
- The compartments of the door are numbered `0` to `N-1`, and the door is represented by an array `int door[N]`. The condition `door[i] == 0` means that compartment `i` is free, while `door[i] == 1` means that compartment `i` is occupied.
- The compartment of the door that opens to the outside of the building (i.e., from which a person can now exit the building) is stored in the variable `int open_out`. The compartment of the door that opens to the inside of the building is stored in the variable `int open_in`. Initially, `open_out == 0` and `open_in == N/2`.
- The door revolves in one direction, say counterclockwise. You are provided with a function `rotate_door()` that simulates the rotation of the door by incrementing (modulo `N`) `open_out` and `open_in`. Do not modify this function.
- People who want to enter the building wait outside, and people who want to leave the building wait inside. To enter the building, a person must enter the revolving door from the compartment at `open_out` and exit the revolving door from the compartment at `open_in`. Vice versa for persons leaving the building.
- The simulation involves `K` people trying to enter and leave the building. You do not know how many of these `K` people are entering and how many are leaving. An initialization thread (not

shown) creates `K` threads, one to simulate each person, plus a thread to simulate the door. The initialization thread also initializes global variables – the ones given in the partial solution and any other ones that you add. A thread simulating a person entering the building runs the function `simulate_entering()`. A thread simulating a person leaving the building runs the function `simulate_leaving()`. The thread simulating the door runs the function `simulate_door()`. Your job is to provide properly synchronized implementations of these three functions.

- A partial implementation of `simulate_door()` is given to you. You should complete this implementation, not write a new version of the function from scratch. The implementation of `simulate_door()` continuously rotates the door, with no synchronization. You will need to synchronize the rotation of the door with persons entering and leaving the building.
- The door should not rotate while a person is entering or exiting it.
- People entering and leaving the building must be able to use the revolving door simultaneously. That is, if there are people waiting to enter and people waiting to leave, your solution should allow two persons to enter the door, one from each side, every time the door stops.
- A person who wants to use the door must wait until the compartment at which they are entering (`open_out` or `open_in`) is empty. When the door stops, if there is a person in compartment `open_out` or `open_in`, they exit the door. After that person exits (or if there was nobody in the compartment) a person waiting to enter the compartment can do so. Your solution must allow one person to leave the door and one person to enter in their place every time the door stops.
- There is no requirement that people should use the door in first-come-first-serve order. Any order is fine as long as all people eventually enter or leave.
- The `simulate_entering()` or `simulate_leaving()` function must return when the person being simulated successfully enters or leaves the building.
- The `simulate_door()` function must return when all `K` people have successfully entered and left the building.
- Global variables should not be accessed by more than one thread at any point in time (mutual exclusion).
- Your solution can use `N` and `K` if needed.
- For synchronization, you may use shared global variables, semaphores, locks, and condition variables as they are provided in OS/161. You may not use lower level primitives such as calling `thread_sleep()` or disabling interrupts.

```
// Number of compartments and number of persons are defined.
#define N ...
#define K ...

// Global variables.  Shared among threads.
// Add the declarations of any other global variables that you need.
// State the assumed initial value of any global variables that you add.
volatile int door[N];   // Initially all 0.
volatile int open_out;  // Initially 0;
volatile int open_in;   // Initially N/2;
volatile int num_entering, num_leaving, num_done; // Initially all 0;
struct lock *mutex;
struct cv *open_out_free, open_in_free;
struct cv *exit_door[N];
struct semaphore *ready_to_rotate; // Initially 0;
```

```
void rotate_door() {
  // % is the mod operator.
  open_out = (open_out + 1) % N;
  open_in = (open_in + 1) % N;
}

void simulate_door() {
  int i, num_this_turn;
  lock_acquire(mutex);
  while(TRUE){
    num_this_turn = 0;
    if (door[open_out] == 1) {
      num_this_turn++;
      cv_signal(exit_door[open_out], mutex);
    } else {
      cv_signal(open_out_free, mutex);
    }
    if (door[open_in] == 1) {
      num_this_turn++;
      cv_signal(exit_door[open_in], mutex);
    } else {
      cv_signal(open_in_free, mutex);
    }
    if (num_entering > 0) {
      num_this_turn++;
    }
    if (num_leaving > 0) {
      num_this_turn++;
    }
    lock_release(mutex);
    for (i = 0; i < num_this_turn; i++) {
      P(ready_to_rotate);
    }
    lock_acquire(mutex);
    if (num_done == K) {
      lock_release(mutex);
      return;
    }
    rotate_door();
  }
}
```

```
void simulate_entering() {
  int my_compartment;
  lock_acquire(mutex);
  num_entering++;
  cv_wait(open_out_free, mutex);
  my_compartment = open_out;
  door[my_compartment] = 1;
  V(ready_to_rotate);
  cv_wait(exit_door[my_compartment], mutex);
  door[my_compartment] = 0;
  cv_signal(open_in_free, mutex);
  num_entering--;
  num_done++;
  V(ready_to_rotate);
  lock_release(mutex);
}

void simulate_leaving() {
  int my_compartment;
  lock_acquire(mutex);
  num_leaving++;
  cv_wait(open_in_free, mutex);
  my_compartment = open_in;
  door[my_compartment] = 1;
  V(ready_to_rotate);
  cv_wait(exit_door[my_compartment], mutex);
  door[my_compartment] = 0;
  cv_signal(open_out_free, mutex);
  num_leaving--;
  num_done++;
  V(ready_to_rotate);
  lock_release(mutex);
}
```