Course	CS350 - Operating Systems
Sections	01 (11:30), 02 (16:00), 03 (8:30)
Instructor	Ashraf Aboulnaga & Borzoo Bonakdarpour
Date of Exam	October 25, 2011
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: (10 marks)	Question 2: (12 marks)	Question 3: (13 marks)
Question 4: (10 marks)	Question 5: (15 marks)	
Total: (60 marks)		

1. **(10 marks)**
Write **T** or **F** next to each of the following statements, to indicate whether it is True or False.

a. (1 mark) Threads of a process share the same address space.

b. (1 mark) External fragmentation cannot happen when using paging to manage virtual memory.

c. (1 mark) The behaviour of a binary semaphore is identical to the behaviour of a lock.

d. (1 mark) For a deadlock to occur, there must be a cycle of processes waiting for each other.

e. (1 mark) The purpose of the TLB is to place the contents of the most frequently accessed page frames in the L2 cache of the CPU.

f. (1 mark) When an interrupt happens, it is handled by a special thread in the kernel that is dedicated to interrupt handling.

g. (1 mark) When a TLB fault exception happens and is successfully handled, the instruction that caused the exception is restarted after exception handling completes.

h. (1 mark) Operating systems typically use only non-preemptive scheduling.

i. (1 mark) The need for synchronizing threads that run concurrently and access shared resources only arises if the machine has multiple CPUs.

j. (1 mark) The set of system calls implemented by an operating system can be changed by a user program.

2. (12 marks)

a. (3 marks)
List three hardware features that are used by general purpose operating systems.

1.

2.

3.

b. (3 marks)
List three steps involved in a context switch.

1.

2.

3.

c. (3 marks)
The following are three tasks that are required for managing virtual memory. Some of these tasks are performed by the kernel and some are performed by the MMU. Write K next to the task if it is performed by the kernel, and write M next to the task if it is performed by the MMU.

- Checking for violations of read-only page protection.
- Deciding which frame of physical memory a page should be mapped to.
- Setting the valid bit in a page table entry.

d. (3 marks)

The following is a possible implementation of the P() operation on a semaphore in OS/161.

```
void
P(struct semaphore* sem)
{
  int spl;
  spl = splhigh();
  while (sem->count==0) {
    thread_sleep(sem);
  }
  sem->count--;
  splx(spl);
}
```

Why is the condition `(sem->count==0)` checked in a while loop and not using an if statement?

3. (13 marks)

Consider a virtual memory system that uses segmentation combined with paging. Virtual addresses in this case are of the form (seg #, page #, offset). In this system, virtual and physical addresses are both 32 bits long, a process may have 16 segments, and the page size is 4KB (2^{12} bytes). A process P has three segments, and the following page tables are associated with its 3 segments. Frame numbers are given in hexadecimal:

Segment 0	
Page #	Frame #
0	0x00078
1	0x00024
2	0x00023

Segment 1	
Page #	Frame #
0	0x00088
1	0x00049
2	0x0003f
3	0x000ce
4	0x000cd

Segment 2	
Page #	Frame #
0	0x00079
1	0x00029
2	0x0002f
3	0x000ae

a. (8 marks)

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

- `0x20001a60`

- `0x000052ef`

- `0x10004ab3`

- `0xa00003c9`

b. (3 marks)

Explain how the virtual memory system would enable another process Q to share Segment 2 with process P.

c. (2 marks)

Give one reason that would make it useful to share segments between processes as in part (b) of this question.

4. (10 marks)

The **dining philosophers** problem is specified as follows.

- Five philosophers live in a house, where a round table is laid for them to eat (see the figure below).
- Each philosopher has an assigned place at the table.
- The life of each philosopher consists of alternating between *thinking* and *eating*.
- Each philosopher requires two forks to eat.
- On the round table, there are 5 plates, one for each philosopher, and 5 forks between the plates (see the figure below).
- A philosopher wishing to eat goes to his assigned place at the table, picks up the two forks on the either side of the plate, and eats. When the philosopher is done, he lays down the two forks that he used for eating.

Since two philosophers who sit in adjacent seats share a fork, and since multiple philosophers can eat at the same time, we need an algorithm to synchronize how the philosophers use their forks. Our synchronization algorithm must allow a philosopher to pick up two forks simultaneously, so that the philosophers can eat. The algorithm must ensure that each fork is used by at most one philosopher at any one time (mutual exclusion). The algorithm must also avoid starvation and deadlock.

Now, consider the solution to this problem shown on the next page. This solution has the following characteristics.

- The five philosopher are numbered 0 to 4.
- Each philosopher is represented by a thread that executes the function `philosopher(i)`, where `i` is the number of that philosopher.
- A philosopher can be in one of three predefined states: `HUNGRY` (waiting for a fork), `EATING` (has 2 forks and is eating), or `THINKING`.
- The solution uses a shared array `state` and a semaphore `mutex` to ensure mutual exclusion in accessing this array.
- The solution also uses an array of semaphores `s`.
- There are predefined functions for thinking and eating that can take any amount of time to complete.

```
// Global variables.  Shared among threads.
int state[5];      // Initially state[i]==THINKING for all i.
semaphore mutex;  // Initially set to 1.
semaphore s[5];    // Initially s[i] is set to 0 for all i.
```

```
void philosopher(int i) {
  while(TRUE){
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}

void take_forks(int i) {
  P(mutex);
  state[i] = HUNGRY;
  test(i);
  V(mutex);
  P(s[i]);
}

void put_forks(int i) {
  P(mutex);
  state[i] = THINKING;
  test(left(i));
  test(right(i));
  V(mutex);
}
```

```
int left(int i) {
// Philosopher to the left of i.
// % is the mod operator.
  return (i + 4) % 5;
}

int right(int i) {
// Philosopher to the right of i.
  return (i + 1) % 5;
}

void test(int i) {
  if (state[i] == HUNGRY &&
        state[left(i)] != EATING &&
        state[right(i)] != EATING) {
      state[i] = EATING;
      V(s[i]);
  }
}
```

a. (3 marks)

Describe the role of semaphore `s[i]` in this solution.

b. (2 marks)

Describe the role of the function `test(i)` in this solution.

c. (5 marks)

This solution suffers from starvation. Describe a concrete scenario in which starvation occurs.

5. (15 marks)

Consider the following variation on the producer/consumer problem with a bounded buffer. This problem may appear, for example, in a file sharing application. A partially implemented solution to this problem is given on the next page.

- One producer thread produces files that are to be copied by several consumer threads. The producer thread runs the function `producer()`.
- When the producer produces a file, it adds the name of that file to a buffer `buf` that has space for `N` file names. The value of `N` is predefined, and there is a predefined data type `bounded_buffer`. There are also predefined functions `produce_file()` and `add_to_buffer()` that respectively implement producing a file and adding an item to a buffer.
- If the buffer is full, the producer needs to wait until an item is removed from the buffer before adding the next file name.
- There are `K` consumer threads. The value of `K` is predefined.
- Each consumer threads runs the function `consumer()`.
- A consumer gets a copy of the first file name from the buffer `buf`, then it obtains a copy of the actual file. There are predefined functions `copy_first_item_from_buffer()` and `copy_file()` that respectively implement copying the first item of a buffer and copying a file with a given file name.
- When all `K` consumers have copied the file, the name of that file is removed from the buffer `buf`. There is a predefined function `remove_first_item_from_buffer()` that removes the first item from a buffer.
- If the buffer is empty, a consumer needs to wait until the producer adds a file name to the buffer.
- The buffer is managed as a First-In-First-Out (FIFO) queue. Items are added to the end of this queue and removed from the beginning of the queue. This FIFO property is already ensured by the predefined functions that add, copy, and remove items from the buffer.
- The buffer `buf` should not be accessed concurrently by more than one thread at any point in time (mutual exclusion).
- The consumer threads must all copy a given file before any consumer thread copies the next file. That is, after a consumer thread copies a file, it waits until all `K` consumer threads copy that file before it gets the next file name from `buf`.
- Your solution can use `K` and `N` if needed.

The solution shown on the following page calls the correct functions, but it does not implement any of the required synchronization rules. Add code to this solution to implement these rules. 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 consumers and number of items in the buffer are defined.
#define K ...
#define N ...

// Global variables.  Shared among threads.
// Add the declarations of any global variables that you need here.
bounded_buffer buf;
```

```
void producer() {
  // Declare local variables if needed.
  char *filename;

  while(TRUE){

    filename = produce_file();

    add_to_buffer(filename, buf);

  }
}
```

```
void consumer() {
  char *filename;

  while(TRUE){

    filename = copy_first_item_from_buffer(buf);

    copy_file(filename);

    remove_first_item_from_buffer(buf);

  }
}
```