Mid-term Exam 1

Student Name:

Note: The total score on this exam is 50 points. The maximum time allowed is 50 minutes. *This is a closed book exam. But you can bring a piece of note to the exam.*

Problem 1 (3 questions, 16 points)

1. **2 points**
List four different ways a process can be terminated. For each, write if it is a voluntary termination or an involuntary one.

Answer:

Processes may end either by:

- executing a **return** from the main function. Voluntary.
- calling the exit(int) function. Voluntary.
- calling the abort(void) function. Voluntary.
- being signaled. Involuntary.

2. **2 points**
For each of the following attributes, specify whether it is a per-process or a per-thread item: (a) registers, (b) program counter, (c) stack, (d) open files, (e) global variables, and (f) address space.

Answer:

Per-process items: (d), (e), and (f)
Per-thread items: (a), (b), and (c)

2. **12 points**
Mark each of the following statements as true or false.

a. Unsafe state means deadlock. FALSE

b. A user-level thread executing a blocking system call will only block the calling thread. FALSE

c. The fork() call returns 1 to the parent process and 0 to the child process. FALSE

d. With Pthread-mutexes, only the thread which currently holds the lock can unlock it. TRUE

e. If the thread is implemented in the kernel, each process keeps its own thread table. FALSE

f. (Blocking) mutex is a special case of semaphore where semaphore S is initialized to be 1. TRUE

g. “Disabling interrupt” is one of the approaches to solve mutual exclusion problem. It works for both single and multi-processor systems. FALSE

h. In the implementation of semaphore, there are two atomic operations: Up(S) and Down(S). Both can block the calling thread. FALSE

i. Up(S) and Down(S) operations on the same semaphore can be done by different threads, meaning that one thread does Down(s) and a different thread can do Up(S). TRUE

j. A solution to mutual exclusion can make assumption about the speed of CPUs. Sometimes, a fast CPU can allow some mutual exclusion solution to work. FALSE

k. In dining philosopher problem, grabbing both chopsticks at the same time would not cause deadlock. TRUE

l. Recovery from deadlock through preemption works. FALSE

Problem 2 (**8 questions**, **23 points**)

1. **2 points**
What is the maximum depth of the stack for this code (in term of the depth of the function call tree)? Be sure to count the first call to main.

```
int main() {
  f(10);
  return 0;
}

int f(int n) {
  if (n <= 0)
    return 0;
  else
    return f(n-4)*f(n-1)+f(n-2);
}
```

Answer: 12 (including the first call to main.
main→f(10) →f(9)→ f(8)→f(7)→f6)->f(5)->f(4)->f(3)->f(2)-f(1)->f(0)/f(-1)/f(-2)

2. **3 points**
A file whose file descriptor is fd contains the following sequence of bytes: 3, 1, 8, 5, 9, 2, 6, 5, 2, 5, 7, 9. The following systems are made:

```
lseek(fd, 3, SEEK_SET);
read(fd, buffer1, 6);
lseek(fd, -4, SEEK_CUR);
read(fd, buffer2, 3);
```

What do buffer1 and buffer2 contain after the read has completed?

Answer:

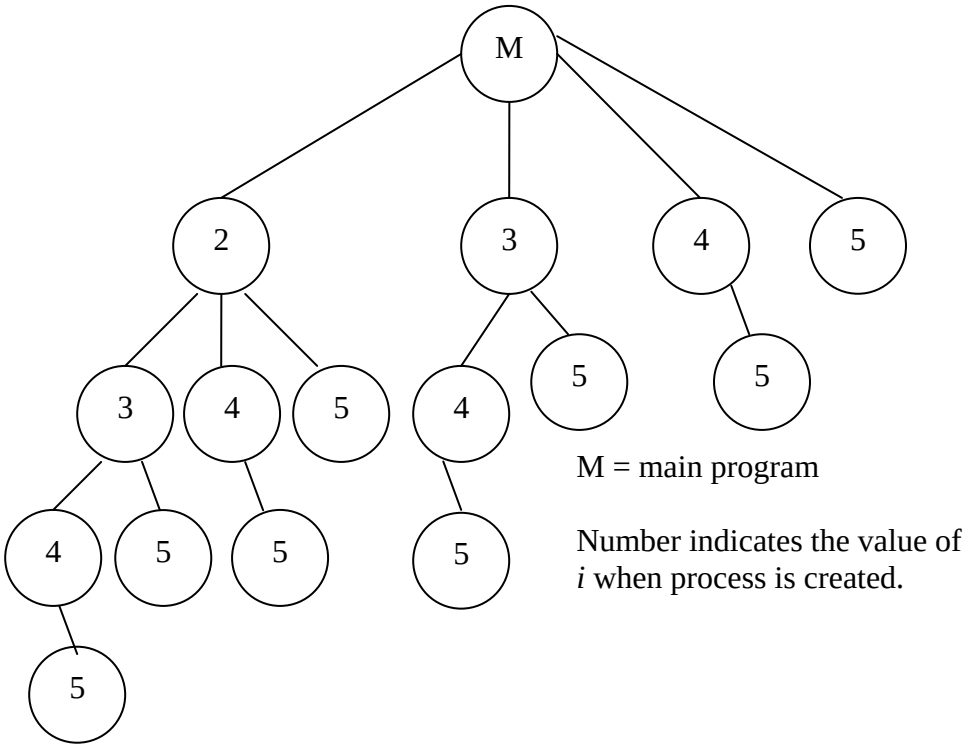
buffer1: 5, 9, 2, 6, 5, 2
buffer2: 2, 6, 5

3. **3 points** How many processes does this code create? Draw the process tree.

```
int main() {
  int i;
  for (i=2; i<6; i++)
    fork();
```

```
        return 1;
    }
```

Answer:



4. **3 points**
Two concurrent threads execute the statements x = x - 3 and x = x + 5 respectively. The user forgot to use a mutex to guarantee mutual exclusion. The initial value of x is 11. List the possible values for x after the two threads execute.

Answer: Possible values 8, 13, 16

5. **3 points**
Two processes, A and B, each need resources 1, 2, and 3. If A asks for them in the order 1, 2, 3, and B asks for them in each of the following orders, please answer whether it is deadlock-possible or deadlock-free (Fill DP/DF in the form). Note that each process can release all the resources only after it finishes, which means that the resources will be released all together at the end (not one by one).

Order 1	1 2 3	
Order 2	1 3 2	
Order 3	2 1 3	
Order 4	2 3 1	
Order 5	3 1 2	
Order 6	3 2 1	

Answer:

Order 1	1 2 3	DF
Order 2	1 3 2	DF

Order 3	2 1 3	DP
Order 4	2 3 1	DP
Order 5	3 1 2	DP
Order 6	3 2 1	DP

6. **3 points**
Consider a system with m processes each needing a maximum of n resources. There are a total of k resources available. Under what condition can this system be deadlock free?

If a process has n resources, it can finish and cannot be involved in a deadlock. Therefore, the worst case is where every process has n-1 resources and needs another one. If there is one resource left over, one process can finish and release all its resources, letting the rest finish too. Therefore the condition for avoiding deadlock is k >= m(n - 1) + 1.

7. **3 points**
There are 3 processes. Each process requires resources R1, R2, and R3. There are two instances each of the three resource types. Mark the following given state as safe or unsafe. Explain.

Allocations: P1 has R3; P2 has R1, R2; P3 has R1, R3

Answer: This is a **safe** state because at this point there is an instance of resource R2 which if assigned to P3 will ensure that P3 has all the resources it needs and can finish upon which it will release R1, R2 and R3. At this point, P2 can be assigned R3 which will ensure that it finishes and releases its resources after which P1 can be run to completion thus preventing a deadlock.

8. **3 points**
What are the possible outputs from running program A? Explain why.

Program A:
```
main() {
        int i=0;
        printf("%d ", i);
        if (fork() > -1)
                exec("B")
        else
                exec("C")
}
```

Program B:
```
main() {
        int i=1;
        printf("%d ",i);
        return;
}
```

Program C:
```
main() {
        int i=2;
        printf("%d ",i);
        return;
}
```

Answer:
Everything works -> 0 1 1
Fork() fails && exec("C") works -> 0 2
Fork() works, but one of the exec("B") fails -> 0 1
Fork() fails && exec("C") fails -> 0

Problem 3 (**3 points**)

Given below is some incomplete code for two threads, *odd* and *even*. The odd thread executes the **printOdd** function and the even thread the **printEven** function. The two threads should synchronize so that the output is: 1, 2, 3, ..., 30 in order. Your job is to fill in the blanks in the code using the appropriate operations on the given binary semaphores.

```
//initialize the following binary semaphores appropriately
binarySemaphore binSemOdd = _________;
binarySemaphore binSemEven = _________;
void printOdd( )
{
        int i;
        for (i = 1; i < 30; i = i+2)
        {
                down(_____________________);
                printf("%d", i);
                up(_____________________);
        }
}
void printEven( )
{
        int i;
        for (i = 2; i <= 30; i = i+2)
        {
                down(_____________________);
                printf("%d", i);
                up(_____________________);
        }
}
```

Answer:

```
//initialize the following binary semaphores appropriately
binarySemaphore binSemOdd = __1_____;
binarySemaphore binSemEven = ___0_____;
void printOdd( )
{
        int i;
        for (i = 1; i < 30; i = i+2)
        {
                down(__binSemOdd _____);
                printf("%d", i);
                up(_binSemEven _____);
        }
}
void printEven( )
{
        int i;
```

```
        for (i = 2; i <= 30; i = i+2)
        {
                down(_binSemEven ___);
                printf("%d", i);
                up(___binSemOdd ____);
        }
}
```

Problem 4 (**5 points**)

Write a program using two threads that can always produce “Iowa State University (ISU)”.

- Thread 1 prints “Iowa ”
- Thread 2 prints “State ”
- Thread 1 prints “University ”
- Thread 2 prints “(ISU)”

Hint: Use condition variable.

Answer:

```
int    thread1_done = 0;
int    thread2_done = 0;
pthread_cond_t    cv_done_t1, cv_done_t2;
pthread_mutex_t  mutex;
```

Thread 1:

```
printf(“Iowa “);

pthread_mutex_lock(mutex);
thread1_done = 1;
pthread_cond_signal(cv_done_t1);
pthread_mutex_unlock(mutex);

……

pthread_mutex_lock(mutex);

while (thread2_done == 0) {
        pthread_cond_wait(cv_done_t2, mutex);
}

printf(“University \n“);
pthread_mutex_unlock(mutex);

pthread_mutex_lock(mutex);
thread1_done = 2;
pthread_cond_signal(cv_done_t1);
pthread_mutex_unlock(mutex);
```

Thread 2:

```
pthread_mutex_lock(mutex);

while (thread1_done == 0) {
        pthread_cond_wait(cv_done_t1, mutex);
```

```
}

printf("State \n");
pthread_mutex_unlock(mutex);

pthread_mutex_lock(mutex);
thread2_done = 1;
pthread_cond_signal(cv_done_t2);
pthread_mutex_unlock(mutex);

pthread_mutex_lock(mutex);

while (thread1_done == 1) {
        pthread_cond_wait(cv_done_t1, mutex);
}

printf("(ISU) \n");
pthread_mutex_unlock(mutex);
```

Problem 5 (**3 points**)

We have studied the Readers-Writers Problem. When there are multiple threads reading/writing a database, we allow that (1) many threads can read simultaneously, but (2) only one can be writing at any time (When a writer is executing, nobody else can read or write). The following is a solution to this problem.

Two semaphores: database and protector

Initial: protector=1; database =1; rc =0

READER:

```
While (1) {
        down(protector);
        rc++;
        if (rc == 1)  //first reader
                down(database);
        up(protector);

        read();

        down(protector);
        rc--;
        If (rc == 0) then // last one
                up(database);
        up(protector);
        ….
}
```

WRITER:

```
While (1) {
        generate_data();
        down(database);
        write();
        up(database);
}
```

Please answer the following two questions:

1. (1 **points**) is there any problem with this solution? If yes, please clearly explain what this problem is.
2. (**1 point**) What is the purpose of the semaphore – protector? Can we replace down(protector) and up(protector) with mutex-lock and mutex-unlock operations?

3. (**1 point**) What is the purpose of the semaphore – database?

Answer:

Q5.1. The solution above has the writer starvation problem. Readers might continuously enter, while a writer has to wait, though the writer may have arrived earlier than some of those readers. It is a reader-preference solution.

Q5.2. The purpose of semaphore – *protector* is to provide mutual exclusion between reader-reader conflicting/interfering operations. Yes, we can replace down(protector) and up(protector) with mutex-lock and mutex-unlock operations.

Q5.3. The purpose of semaphore – *database* is to provide mutual exclusion between reader-writer conflicting/interfering operations.