Q. Processes

Part I. *Answer all the below:*

```
int n, x, status = 0;
while(n < 2)
{
        x = fork();
        if(x != 0)
        {
                waitpid(x, &status, 0);
        }
        else
        {
                printf("hi\n");
        }
        ++n;
}
return 5;
```

a. In the above code snippet how many times does "hi" print out?

b. How would you modify this code so that it creates the same number of Processes that do the same thing, but it finishes faster (wall clock time)?

Part II. *Answer **6** of the following **9**:*

a. What does fork() do?

b. What does exec() do?

c. What would happen if you called exec(), but did not fork()?

1. Threads

Part I. *Answer **all** the below:*

```
.. fu( .. ) {... }
void sna()
{
        int arg = 500;
        pthread_t tid = 0;

        pthread_create(&tid, NULL, fu, &arg);

        return;
}
```

a. Most of the time the snippet above Segfaults when fu() starts running. Why? (presume fu() is defined correctly)

b. List 2 ways you could fix the code snippet above.

c. Explain what would happen when you call each of the following inside a thread:
return, pthread_exit(), exit(), _exit()

Part II. *Answer **5** of the following **7**:*

a. When you create a new thread, where is its stack located?

b. Why would a Process with 100 user threads take longer to do the same job as a Process with 100 kernel threads?

c. Would a blocking call in a kernel thread on a single-core computer block other threads in the same Process? Explain why or why not.

d. Would a blocking call in a user thread on a multi-core computer block other threads in the same Process? Explain why or why not.

e. Describe two benefits of a user thread over kernel threads.

f. Describe two benefits of a kernel thread over user threads.

g. Describe two benefits of a kernel thread over a Process.

2. Synchronization

Part I. *Answer all the below:*

```
while(notdone)
{
        pthread_lock(&var0);

        if( sharedvar < 0)
        {
                pthread_lock(&var1);
                increment(sharedvar+incr);
        }
        else if( sharedvar > 0)
        {
                increment(sharedvar-incr)
                pthread_unlock(&var1);
        }
        else
        {
                notdone = sharedvar;
                pthread_unlock(&var1);
        }
        pthread_unlock(&var0);
```

a. Is the above code free from potential deadlocks? Explain why or why not.

Part II: *Answer 5 of the following 8:*

a. What is a critical section? Why is it important to identify critical sections in multithread programs

b. What are the four conditions for deadlock?

c. What two rules can you apply to avoid deadlock?

d. Why can't the user do synchronization without the help of special instructions?

e. Is it possible to recover from a deadlock between semaphores? Describe why or why not.

f. Is it possible to recover from a deadlock between mutexes? Describe why or why not.

d. Why are zombie Processes bad?

e. How does a Process become an orphan?

f. Can a Process be both a zombie and an orphan? Explain why or why not.

g. What happens to a parent Process if the child Segfaults?

h. What function does not return, except when there is an error?

i. What kind of function is not called, but is invoked on your behalf?