QUESTION 1

Explain what happens in a computer when you press a key on the keyboard.

- When you press a key on the keyboard, an interrupt signal is sent to the computer which is specific to that key. The signal arrives at the interrupt controller in the OS which thus handles that signal.

QUESTION 2

There is a sleep system call in UNIX. It pauses the current process (who makes this system call) for a specified number of seconds. Explain what happens in the OS kernel after this sleep system call is invoked. (Hint: focus on process state transitions, wait queues, scheduling).

- When the sleep system call is invoked, the OS performs a context switch to the kernel which proceeds saves the user registers and creates a return value placed into a dedicated register. It then goes to 'sleep' for the specified number of seconds. During which the scheduler switches the kernel to another process in the queue after executing "return_from_interrupt".

QUESTION 3

Multi-level Feedback Queue (MFQ) algorithm can:

(1) give interactive jobs more priority.

(2) give CPU-bound jobs bigger time slice, and

(3) avoid starvation.

Explain how MFQ achieves each of these three goals. (Make sure to explain each one separately. Describe how MFQ works in general won't get any credit)

- Multi-level Feedback Queue algorithm can:
    1. It can give interactive jobs more priority as every job in MFQ follows a priority-based system. It starts from the top and switches between the queues based on the use of use of the quantum, if it isn't used it stays in the same queue else switches further down.
    2. Each job gets a time slice they need with the CPU-bound jobs starting at the top. They have higher priority in the queue and are reassigned in the queue if they require more additional time slice based on the necessity of the job.
    3. MFQ avoids starvation since the last queue is with Round Robin scheduling algorithm that feeds the queue back into itself to assist and avoid starvation of processes.

QUESTION 4

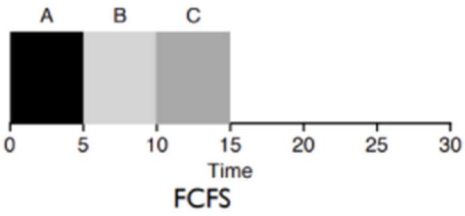
- Which of the following are true?
  - OS controls everything (software and hardware), it is constantly running.
  - OS is sleeping most of the time.
  - OS sits between applications and hardware, so applications do not run directly on the CPU.
  - Applications run directly on the CPU.
  - To access IO devices, an application must make a system call.
  - OS kernel runs in the most privileged mode, which is Ring 3
  - An application specifies which system call to make by calling its name (e.g., fork)
  - When an application executes a protected instruction, it will trigger a fault.

QUESTION 5

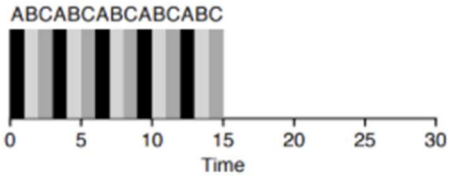
- Which of the following are true?
  - Thread is a unit for scheduling, and threads in the same process share one
    address space.
  - A user-level process can call arbitrary kernel functions by making system calls.
  - A thread can be implemented in both kernel and user level, but user-level thread is more lightweight to create, schedule and manage.
  - To switch the context from one thread to another within the same process, we just need to save the program counter and stack pointer of the current thread and load the program counter and stack pointer of the next thread.
  - If a process has multiple user-level threads and one user-level thread is blocked
    on an I/O request, other threads are blocked too.
  - A process with multiple threads has multiple stacks.

QUESTION 6

For CPU scheduling, consider the following two scenarios



FCFS



Round Robin every 1s

1). What are the average response times?

- Average Response Time:
  - FCFS: (5+5+5)/3 = 5
  - RR: (1x15)/15 = 1

2). What are the average turnaround times?

- Average Turnaround Time:
  - FCFS: (5+10+15)/3 = 10
  - RR: (13+14+15)/3 = 14

QUESTION 7

Consider the following program:

```
1 int main() {
2 int count = 0;
3 int pid;
4
5 if( !(pid = fork()) ) {
6      while((count < 2) && (pid = fork()) ) {
7          count++;
8          printf("%d", count)
9      }
10     if (count > 0) {
11         printf("%d", count);
12     }
13 }
14 if(pid) {
15     waitpid(pid, NULL, 0);
16     count = count << 1;
17     printf("%d", count)
18     }
19 }
```

1). How many processes are created during the execution of this program? Explain. (10 points)

- There are 4 processes that are created during the execution of the given program. For the first time the parent process creates a new process in the if statement but the statements in the first won't execute since there's! in the if statement. The newly created process will return 0 in the first fork since it's not created a new process, thus the while statements will be executed. In the while statements the initial count is 0 and first condition is less than 2 (< 2) so two subprocess are created. Thus, there's 1 parent, 1 child and 2 children _subprocessess.

2). List all the possible outputs of the program. (10 + 3 bonus points)

- The output is: 1 1 1 2 2 4 0

QUESTION 8

Consider the following program

```
int x;
int main(int argc, char *argv[]) {
  int y;
  int* z = malloc(sizeof(int)););
}
```

Please fill in the table below for possible memory segments (data, code, stack, heap)

Address	Location
x	data
main	code

y	stack
z	stack
*z	heap

QUESTION 9

Consider the following code snippet for a multi-threaded game engine called RealEngine

Thread 1:

```
loadAssets();
launchUI();
realEngineStarted = true;
cache_t * c = objCacheInit();
…
```

Thread 2:

```
while (!realEngineStarted) { }
objCacheInsert(c, new Sprite("mario"));
…
```

1). Is the code buggy? Explain why or why not.

- Yes, the code is buggy since Thread 2 can run CacheInsert before initializing c.

2). If there is a bug, explain how we might fix it.

- We can fix the bug by adding locks.

QUESTION 10

Three players are throwing a frisbee to one another. Player 1 first has the ball. He randomly (50% probability) decides which player (Player 2 or 3) to throw to. Then after the receiver catches the frisbee, he or she randomly throws another player too. An example printout is below.

Player 1 throws the frisbee to Player 2.

Player 2 throws the frisbee to Player 3.

Player 3 throws the frisbee to Player 2.

Please complete the following program to simulate this process. Each player runs in a separate thread, and he or she communicates with the other players using semaphores. (Note: Pseudo code or abbreviated code is fine. Thread creation logic is omitted. Only focus on the implementation of the thread for each player.) (10 points)

```
//To-be-completed: state your semaphore initializations
1).

//each player runs in its own thread
void Player (void *p)
{
    //i = 0,1, or 2, indicating which player is playing
    int i = (int)p;

    //To-be-completed: start your coding here
    2).

}
```

```
sem_init (&sem [0], 0, 1);
sem_init (&sem [1], 0, 0);
sem_init (&sem [2], 0. 0);
void Player (void *p)
{
  int i = (int)p.
  while (1)
  {
          r = get_random (0, 1);
          if (r < 0.5)
          {
                  sem_wait(sem[(i+1) % 3]).
                  printf ("Player %d throws the frisbee to Player %d.\n", i+1, ((i+1)
  % 3) + 1);
          }
          else
          {
                  sem_wait(sem[(i+2) % 3]).
                  printf ("Player %d throws the frisbee to Player %d.\n", i+1, ((i+2)
  % 3) + 1);
          }
          sem_post(sem[i]).
  }
}
```