Assignment 3
Sumeet Maan

1. [2 points] What do multi-programming and time-sharing have in common?

   Ans: They both allow the processes to be in memory simultaneously, they also allow context switching if required.

   a. [4 points] How are they different?

      Ans: They are different in the following ways.

      i. In a time sharing, each process is executed on the CPU for a fixed time slice/quantum. When the quantum expires then the processor switches to a different process. Whereas in multi programming, the processes continue to run until they are blocked by an I/O request or a high priority request, etc.

      ii. The main aim of time sharing is to focus on user responsiveness and device utilization at the cost of efficiency, whereas it is vice versa in the case of multi-programming. In multi programming, efficiency is the main aim which comes at the cost of user responsiveness.

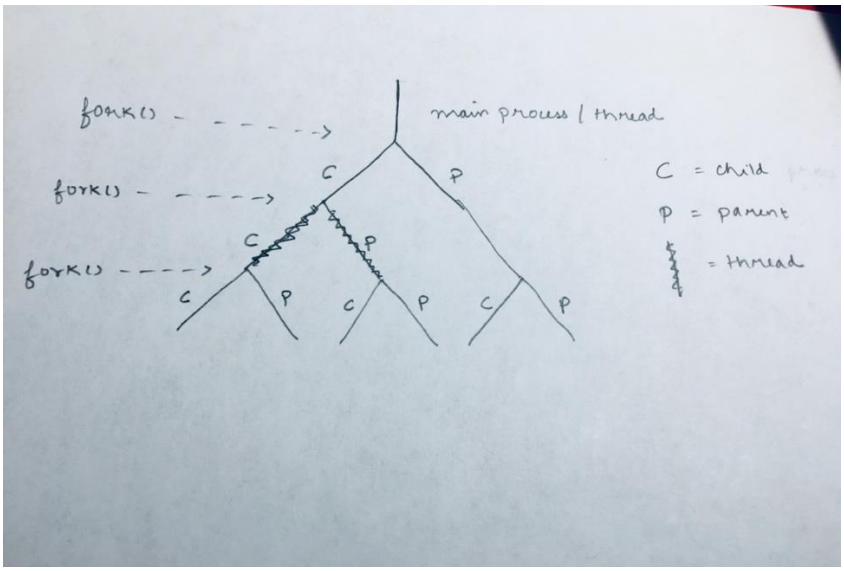
   b. [4 points] Given the two scenarios below, complete the following table

Scenario	Choose Multiprogramming or Time Sharing	Justify
Multi-user system like the CSE server	Time-Sharing	Here the main focus is user responsiveness.
Batch Processing System (does not interface with users)	Multiprogramming	Here the main focus is efficiency and device utilization.

2. [5 points] Multiprogramming is key to a modern operating system's operation. Explain why in your own words.
   Ans: Multiprogramming is the key due to the increased demand of user wanting to do multitasking even on a single processor. Multiprogramming gives the essence that the user is doing multitasking on a single processor by switching the processes between the CPU very fast. Multiprogramming allows the user to run multiple processes by increasing the resource utilization, device utilization, throughput with lower latency.

3. Consider the code segment

Ans: This can be explained with the help of tree diagram.



As we can see in the diagram there are 6 leave nodes which represent processes and there are 2 threads.

a. How many unique processes are created (including the parent process)?
Ans: There are 6 processes created
b. How many unique threads are created (other than the process main thread)?
Ans: There are 2 threads created

4. [14 points] The following state transition table is a simplified model of process management, with the labels representing transitions between states of READY, RUN, BLOCKED, Suspended, and Zombie.
    1. **Ready -> Run**: This can occur in the case where there is CPU availability, or if the scheduler decides to run the process on the CPU in case of an interrupt or premption basis.
    2. **Run -> Ready**: When the scheduler decides to run another process (waiting in the ready) queue on the CPU. Can happen in case of preemption or scheduler algorithm.
    3. **Run -> Blocked**: When a process wants to do an I/O while running, the CPU puts it in the blocked state.
    4. **Blocked -> Ready**: when the process finishes the I/O operations and there is space in the main memory, the process moves to Ready queue.
    5. **Ready -> Suspended:** When the ready queue is full, a process wants to do I/O the scheduler will send that process in the ready to the suspended queue/state. Also suspended state is in the disc and not in the main memory. The process here waits to complete its I/O. A second event could be that if a high priority process comes in, then the processor will select a low priority process from the ready queue to be sent to the suspended state to make space for the high priority process.
    6. **Blocked -> Suspended**: When a process is not done with the I/O and needs to be moved from main memory to the disc memory to make space in the main memory
    7. **Run -> Zombie**: When the parent process does not have wait system call on the child process, then in that event the child process will not be terminated and will have an entry in the process table.

5.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define MSGSIZE 16
#define READ_END 0
#define WRITE_END 1

void convertToUpperCase(char *s){
    for(int i=0; s[i]!= '\0';i++){
        if( s[i]>='a' && s[i]<='z'){
            s[i] = s[i]-32;
        }
    }
}
void leftRotateByHalf(char *s){
    int n = strlen(s);
    int pivot = n/2;
    if (n != 0) { //if 0, don't rotate
        reverse(s, 0, pivot);
        reverse(s, pivot, n);
        reverse(s, 0, n);
    }
}
int main( )
{
    char *msg1 = " How are you?";
    char inbuff[MSGSIZE];
    int p[2], pid;
    pipe(p);
    pid = fork();

    if (pid<0){
        fprintf(stderr, "Could not perform first fork");
    }
    if (pid == 0){
        //child process
        close(p[READ_END]);
```

```
        convertToUpperCase(msg1);
        write(p[1],msg1,MSGSIZE);
        close(p[WRITE_END]);
    }
    else
    {
        //parent process
        int fd[2], c_pid;
        pipe(fd);
        char inbuff2[MSGSIZE];
        c_pid = fork();
        if (c_pid<0){
            fprintf(stderr, "Could not perform second fork ");
        }
        if (c_pid == 0){
            close(p[WRITE_END]);
            read(p[READ_END], inbuff, MSGSIZE);
            printf("The message in Upper case is %s",inbuff);
            close(p[READ_END]);
            leftRotateByHalf(inbuff);

            close(fd[READ_END]);
            write(fd[WRITE_END],inbuff,MSGSIZE);
            close(fd[WRITE_END]);
        }
        else{
            if(pid==0){
                close(fd[WRITE_END]);
                read(fd[READ_END], inbuff2, MSGSIZE);
                printf("Rotated String: %s",inbuff2);
                close(fd[READ_END]);
            }
            wait(NULL);
        }
        wait(NULL);
    }
    exit(0);
}
```

6.