Name: **Mohith Ankem**

Student ID: **017400825**

CS259 Project 2

Question 1:

The key is ensuring that the system avoids a deadlock scenario, where all processes could be waiting for resources held by other processes, preventing any of them from progressing. If the system has 6 tape drives, and each of the N processes may need up to 2 drives, then to guarantee that deadlock doesn't happen, we must ensure that at least one process can always get enough resources to finish its work. This means ensuring that processes cannot indefinitely hold resources while waiting for additional ones. For deadlock to occur, each process would need to hold 1 tape drive and wait for another. The system is deadlock-free if $N \leq 4$. Each process could grab 1 tape drive and wait for another, leading to a situation where no process can get the second drive it needs to complete its task. This is a potential deadlock situation.

Question 2:

a) No, deadlock would not occur. At step (o), process A holds resource R and is waiting for S, and process B holds S and is waiting for T. If process C were to request S instead of R at this step, process C would block, since S is already held by process B. However, there would be no circular waiting chain, as process C would not hold any resources at the time of the request. This breaks the circular wait condition necessary for deadlock. Since process A can still release resource R, and no circular chain of dependencies exists, the system avoids deadlock. In essence, there is no scenario where each process holds a resource while waiting for another that is held by a different process, which is the condition for deadlock.

b) Yes, deadlock would **occur**. If process C requests both R and S at step (o), it would effectively hold both resources (R and S), which would create a circular dependency. Here's the situation:

Process A is holding R and waiting for S.

Process B is holding S and waiting for T.

Process C would hold both R and S, meaning processes A and B would be unable to acquire the resources they are waiting for.

This situation forms a **circular wait** where:

- A is waiting for S (held by C),
- B is waiting for T (and already holding S),
- C is waiting for something else or holding both R and S, leading to deadlock because the resources needed by each process are held by the others in the cycle.

Thus, requesting both R and S by process C at step (o) would indeed result in a deadlock.

Question 3:

a) The "all-encompassing danger region" is the shaded area bounded by instructions I_2 to I_3 (for process A on the x-axis) and I_6 to I_7 (for process B on the y-axis). If both processes enter this region, they will be waiting on each other's resources, leading to a deadlock.

b) No, it is impossible for the execution trajectory to reach the intersection of I_3 and I_7. To do so, the system would need to cross through the shaded danger region, which would result in deadlock due to mutual exclusion. In this case, the processes would be holding resources while waiting for the other, thus preventing the system from progressing.

c) Two possible completion paths that will allow reaching point u safely from point t are:

t -> I_6 -> I_7 -> I_8: Process B finishes by releasing the plotter, and then Process A completes.
t -> I_2 -> I_3: Process A releases the plotter first, and then Process B finishes.

Both paths avoid deadlock and allow the system to safely reach point u.

d) A diagonal execution trajectory indicates that both processes are progressing simultaneously. This would typically happen on a multiprocessor or parallel processing platform, where both processes can execute in parallel and acquire resources at the same time without waiting for one another.

Question 4:

Flowchart (a) Both processes can set their flags (P1 = True and P2 = True) simultaneously and then check the other's flag. In this case, both will wait indefinitely, as each will find the other's flag set, resulting in deadlock. Neither process will proceed into the critical section.

Lack of Mutual Exclusion: If the scheduler happens to switch processes after setting P1 or P2 but before checking the other process's flag, both processes could find the other's flag unset and both enter the critical section. This scenario violates the mutual exclusion condition. Flowchart (a) does not provide effective mutual exclusion and can result in deadlock or simultaneous entry into the critical section.

Flowchart (b) Because each process checks the other process's flag before setting its own, they cannot both set their flags and enter the critical section at the same time. If one process sees that the other is already in the critical section, it will wait.

No Deadlock: There's no chance of deadlock in this version. Even if both processes try to enter the critical section at the same time, one of them will see the other's flag as True and wait, ensuring that only one process enters the critical section at a time. Flowchart (b) provides effective mutual exclusion without deadlock, as it ensures only one process can access the critical section at a time by checking the other process's flag before setting its own flag.

Question 5:

This protocol will not be effective in circumventing deadlock and could actually lead to deadlock under certain conditions. Upon arriving at the river bank, a person checks if someone on the opposite side either wants to cross or is currently crossing. If so, the person waits until the other is done before proceeding. Imagine two people, one on each side of the river, arrive at the river bank simultaneously. Both will follow the protocol and check whether someone on the opposite side wants to cross or is currently crossing. Since both want to cross, they will both see that the other person is also waiting and thus will wait for the other person to cross first. The result is that both will wait indefinitely, leading to a deadlock. Neither person will cross, as both are waiting for the other to finish. This protocol is ineffective at preventing deadlock. It introduces a waiting condition based on the status of the other person, which can lead to both parties waiting for each other indefinitely.

Question 6:

A. Mutual Exclusion: Only one process at a time can use a particular resource. If another process requests that same resource, the process must wait until the resource is released by the first process.

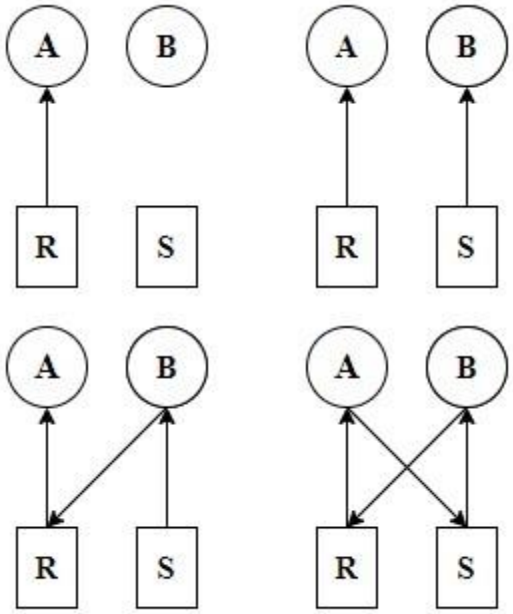
B. Hold and Wait: Processes holding resources granted earlier can request new resources. If the process must wait for a new additional resource, it continues to hold onto the resources it currently has.

C. No Preemption: Resources previously granted to a process cannot be forcibly taken away from that process. Resources can only be released voluntarily by the processes that are holding them.

D. Circular Wait: A circular chain of two or more processes must exist. Each is waiting for a resource held by the next member of the chain.

Question 7:

Below is the resource allocation graph for the sequence of instructions:



A deadlock occurs when there is a circular wait, which means a set of processes are waiting for resources held by each other in a circular manner, and none can proceed.

In this graph:

Process A is waiting for Resource S, which is requested by Process B.

Process B is waiting for Resource R, which is requested by Process A.

This forms a circular wait. Since neither process can proceed until the other releases the resource, the system is in a deadlocked state. Yes, the system is in a deadlocked situation after the instruction sequence. The circular wait between Processes A and B leads to deadlock because both processes are waiting for resources held by each other.

Question 8:

8A)

eigtha.c 1 | eigthb.c

C: > Users > Checkout > Desktop > APP > hw2 > eigtha.c > main()

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Sleep #1\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #2\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #3\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #4\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #5\n");
    fflush(stdout);
    sleep(1);
    printf("Program exiting\n");
    exit(0);
}
```

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ gcc eigtha.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Sleep #1
Sleep #2
Sleep #3
Sleep #4
Sleep #5
Program exiting
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ gcc eigtha.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Sleep #1
Sleep #2
Sleep #3
Sleep #4
Sleep #5
Program exiting
```

8B)

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int main() {
    signal(SIGINT, SIG_IGN);
    printf("Sleep #1\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #2\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #3\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #4\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #5\n");
    fflush(stdout);
    sleep(1);
    printf("Program exiting\n");
    exit(0);
}
```

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ gcc eigthb.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Sleep #1
Sleep #2
Sleep #3
Sleep #4
Sleep #5
Program exiting
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$
```

8C)

Output:

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ gcc eigthc.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Sleep #1
^CJumped to Interrupt handler
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Sleep #1
Sleep #2
Sleep #3
^CJumped to Interrupt handler
Sleep #4
Sleep #5
Program exiting
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$
```

When Ctrl+C is pressed for the first time, we see the message printed “Jumped to interrupt handler”. A second Ctrl+C exits the program.

8D)

eigtha.c | eigthb.c | eigthc.c | eigthd.c

C: > Users > Checkout > Desktop > APP > hw2 > C eigthd.c > main()

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void handleInterrupt(int sig)
{
    signal(SIGINT,handleInterrupt);
    printf("Jumped to interrupt handler\n");
    fflush(stdout);
}
int main() {


    signal(SIGINT,handleInterrupt);
    printf("Sleep #1\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #2\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #3\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #4\n");
    fflush(stdout);
    sleep(1);
    printf("Sleep #5\n");
    fflush(stdout);
    sleep(1);
    printf("Program exiting\n");
    exit(0);
}

```

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ gcc eigthd.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Sleep #1
Sleep #2
^CJumped to interrupt handler
^CJumped to interrupt handler
Sleep #3
^CJumped to interrupt handler
Sleep #4
^CJumped to interrupt handler
Sleep #5
Program exiting
```

Now the program can catch multiple Ctrl+C statements.

9)

eigtha.c | eigthb.c | eigthc.c | eigthd.c 1 | ninth.c 1

C: > Users > Checkout > Desktop > APP > hw2 > C ninth.c > main()

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void handleAlarm()
{
    printf("\n Timeout reached. Exciting program. \n");
    exit(0);
}

int main()
{
    signal(SIGALRM, handleAlarm);

    printf("Enter input - Time Limit: 10 seconds\n");
    fflush(stdout);
    alarm(10);

    char input[100];
    gets(input);
    alarm(0);

    printf("User entered: %s\n", input);
    exit(0);
}
```

Output:

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Enter input - Time Limit: 10 seconds
Hello There
User entered: Hello There
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Enter input - Time Limit: 10 seconds

 Timeout reached. Exciting program.
```

10-a)

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define MSGSIZE 16

int main() {
    int pfd[2];
    int retval;

    char msg[MSGSIZE];
    char buffer[MSGSIZE];

    retval = pipe(pfd);
    if (retval == -1) {
        printf("Failed to create pipe\n");
        exit(1);
    }

    for (int i = 1; i < 5; i++) {
        snprintf(msg, MSGSIZE, "Message %d", i);
        printf("Writing %s\n", msg);
        write(pfd[1], msg, MSGSIZE);
    }

    for (int i = 1; i < 5; i++) {
        read(pfd[0], buffer, MSGSIZE);
        printf("Read %s\n", buffer);
    }

    exit(0);
}
```

Output:

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ touch tenth.c
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ gcc tenth.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
Writing Message 1
Writing Message 2
Writing Message 3
Writing Message 4
Read Message 1
Read Message 2
Read Message 3
Read Message 4
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$
```

10-b)

eigtha.c | eigthb.c | eigthc.c | eigthd.c 1 | ninth.c 1 | tenth.c | tenthb.c 9

C: > Users > Checkout > Desktop > APP > hw2 > C tenthb.c > MSGSIZE

```c
#include <stdio.h>
#include <unistd.h>

#define MSGSIZE 16
int main() {
    int pfd[2];
    int retval;
    char msg[MSGSIZE];
    char buffer[MSGSIZE];
    pid_t pid;

    retval = pipe(pfd);
    if (retval == -1) {
        printf("Failed to create pipe\n");
        exit(1);
    }
    pid = fork();
    if (pid < 0) {
        printf("Failed to fork process\n");
        exit(1);
    } else if (pid == 0) {
        for (int i = 1; i <= 4; i++) {
            read(pfd[0], buffer, MSGSIZE);
            printf("This is the child process (process ID %d). Reading message
            number %d from the pipe. Content is: %s\n", getpid(), i, buffer);
        }
    } else {
        for (int i = 1; i <= 4; i++) {
            printf("This is the parent process (process ID %d). Writing message
            number %d to the pipe.\n", getpid(), i);
            snprintf(msg, MSGSIZE, "Message %d", i);
            write(pfd[1], msg, MSGSIZE);
        }
    }
    exit(0);
}
```

Output:

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ gcc tenthb.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
This is the parent process (process ID 7371). Writing message number 1 to the pipe.
This is the parent process (process ID 7371). Writing message number 2 to the pipe.
This is the parent process (process ID 7371). Writing message number 3 to the pipe.
This is the parent process (process ID 7371). Writing message number 4 to the pipe.
This is the child process (process ID 7374). Reading message number 1 from the pipe. Content is: Message 1
This is the child process (process ID 7374). Reading message number 2 from the pipe. Content is: Message 2
This is the child process (process ID 7374). Reading message number 3 from the pipe. Content is: Message 3
This is the child process (process ID 7374). Reading message number 4 from the pipe. Content is: Message 4
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$
```

10-c)

eigtha.c | eigthb.c | eigthc.c | eigthd.c 1 | ninth.c 1 | tenth.c | tenthb.c 9 | tenthc.c 1

C: > Users > Checkout > Desktop > APP > hw2 > C tenthc.c > main()

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>

#define MSGSIZE 1
int count = 0;
void alarm_handler(int sig) {
    printf("Write blocked after %d characters.\n", count);
    exit(0);
}
int main() {
    int pfd[2], retval;
    retval = pipe(pfd);
    if (retval == -1) {
        printf("Failed to create pipe\n");
        exit(1);
    }
    signal(SIGALRM, alarm_handler);
    while (1) {
        char msg[MSGSIZE] = "a";
        alarm(3);
        int written = write(pfd[1], msg, MSGSIZE);
        if (written != -1) {
            count++;
        }
        alarm(0);

        if (count % 1024 == 0) {
            printf("%d characters in pipe\n", count);
            fflush(stdout);
        }
    }
    exit(0);
}
```

Output:

```
mohithankem@117Latitude3420: /mnt/c/Users/Checkout/Desktop/APP/hw2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ gcc tenthc.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw2$ ./a.out
1024 characters in pipe
2048 characters in pipe
3072 characters in pipe
4096 characters in pipe
5120 characters in pipe
6144 characters in pipe
7168 characters in pipe
8192 characters in pipe
9216 characters in pipe
10240 characters in pipe
11264 characters in pipe
12288 characters in pipe
13312 characters in pipe
14336 characters in pipe
15360 characters in pipe
16384 characters in pipe
17408 characters in pipe
18432 characters in pipe
19456 characters in pipe
20480 characters in pipe
21504 characters in pipe
22528 characters in pipe
23552 characters in pipe
24576 characters in pipe
25600 characters in pipe
26624 characters in pipe
27648 characters in pipe
28672 characters in pipe
29696 characters in pipe
30720 characters in pipe
31744 characters in pipe
32768 characters in pipe
33792 characters in pipe
34816 characters in pipe
35840 characters in pipe
36864 characters in pipe
37888 characters in pipe
38912 characters in pipe
39936 characters in pipe
40960 characters in pipe
41984 characters in pipe
43008 characters in pipe
44032 characters in pipe
45056 characters in pipe
46080 characters in pipe
47104 characters in pipe
48128 characters in pipe
49152 characters in pipe
50176 characters in pipe
51200 characters in pipe
52224 characters in pipe
53248 characters in pipe
54272 characters in pipe
55296 characters in pipe
56320 characters in pipe
57344 characters in pipe
58368 characters in pipe
59392 characters in pipe
60416 characters in pipe
61440 characters in pipe
62464 characters in pipe
63488 characters in pipe
64512 characters in pipe
65536 characters in pipe
Write blocked after 65536 characters.
```