NAME - MAHFUJUL HAQUE	**CLASS ROLL - 002010501036**
CLASS - BCSE III	**SESSION - 2022-2023**
SEMESTER - FIRST	**ASSIGNMENT - 02**

QUESTION 01: Design a CPU scheduler for jobs whose execution profiles will be in a file that is to be read and appropriate scheduling algorithm to be chosen by the scheduler.
Format of the profile:
<Job id> <priority> <arrival time> <CPU burst(1) I/O burst(1) CPU burst(2) ……. >-1
(Each information is separated by blank space and each job profile ends with -1. Lesser priority number denotes higher priority process with priority number 1 being the process with highest priority.)
Example: 2 3 0 100 2 200 3 25 -1 1 1 4 60 10 ….. -1 etc.
Testing:
a. Create job profiles for 20 jobs and use three different scheduling algorithms (FCFS, preemptive Priority and Round Robin (time slice:20)).
b. Compare the average waiting time, turnaround time of each process for the different scheduling algorithms.

CODE:

FCFS

```
#include<stdio.h>
#include<stdlib.h>

typedef struct Queue {
      int front, rear, size;
      unsigned capacity;
      int* array;
} Queue;

Queue* createQueue(unsigned capacity) {
      Queue* queue = (Queue *)malloc(sizeof(Queue));
      queue->capacity = capacity;
      queue->front = queue->size = 0;

      queue->rear = capacity - 1;
      queue->array = (int *)malloc(queue->capacity*sizeof(int));
      return queue;
}

int isFull(Queue* queue) {
      return (queue->size == queue->capacity);
}

int isEmpty(Queue* queue) {
      return (queue->size == 0);
}

void enqueue(Queue* queue, int item) {
      if (isFull(queue)) return;
      queue->rear = (queue->rear + 1) % queue->capacity;
      queue->array[queue->rear] = item;
```

```
      queue->size = queue->size + 1;
}

int dequeue(Queue* queue) {
      if (isEmpty(queue))
            return -1;
      int item = queue->array[queue->front];
      queue->front = (queue->front + 1)
                        % queue->capacity;
      queue->size = queue->size - 1;
      return item;
}

int front(Queue* queue) {
      if (isEmpty(queue))
            return -1;
      return queue->array[queue->front];
}

int rear(Queue* queue) {
      if (isEmpty(queue))
            return -1;
      return queue->array[queue->rear];
}

typedef struct Process {
    int id;
    int priority;
    int AT;
    int bt[100];

    int ptr;
} Process;




int main(){

      int sum_wt = 0;
      int sum_tt = 0;

    freopen("input.txt", "r", stdin);
    freopen("output_fcfs.txt", "w", stdout);


    int no_of_processes;
    scanf("%d", &no_of_processes);

    Process ps[no_of_processes];
    for (int i = 0; i < no_of_processes; i++){
        scanf("%d", &ps[i].id);
        scanf("%d", &ps[i].priority);
        scanf("%d", &ps[i].AT);

        sum_tt -= ps[i].AT;

        int cnt = -1;
        do {
            cnt++;
            int temp;
            scanf("%d", &temp);
            ps[i].bt[cnt] = temp;
```

```
            sum_wt -= temp;

        } while (ps[i].bt[cnt] != -1);
        ps[i].ptr = 0;
    }


    Queue * FCFS = createQueue(no_of_processes);

    for (int i = 0; i < no_of_processes; i++){
        if (ps[i].AT == 0) {
            enqueue(FCFS, i);
        }
    }

    int completed = 0;
    int time = 0;
    while (completed != no_of_processes){

        if (isEmpty(FCFS)){
            printf("Time Stamp: %d: Idle CPU\n", time);
            time++;
            continue;
        }

        int pid = front(FCFS);
        ps[pid].bt[ps[pid].ptr] --;
        printf("Time Stamp: %d, executed: %d\n", time, pid);

        if (ps[pid].bt[ps[pid].ptr] == 0){
            dequeue(FCFS);
            if (ps[pid].bt[(ps[pid].ptr)+1] == -1 || ps[pid].bt[(ps[pid].ptr)
+2] == -1){
                completed++;
                printf("Completed %d\n", completed);
                sum_tt += time;
                ps[pid].AT = __INT_MAX__;
            } else {
                int io_time = ps[pid].bt[(ps[pid].ptr)+1];
                ps[pid].AT = time + io_time;
                ps[pid].ptr+=2;
            }
        }

        time++;

        for (int i = 0; i < no_of_processes; i++){
            if (ps[i].AT == time) {
                enqueue(FCFS, i);
            }
        }

    }


    printf("Average Waiting Time: %f\n",
((float)(sum_tt-sum_wt))/no_of_processes);
    printf("Average Turnaround Time: %f\n", ((float)sum_tt)/no_of_processes);
}
```

PREEMPTIVE PRIORITY

```
#include<stdio.h>
#include<stdlib.h>
```

```
typedef struct Process {
    int id;
    int priority;
    int AT;
    int bt[100];

    int ptr;
} Process;




int main(){

    int sum_wt = 0;
      int sum_tt = 0;

    freopen("input_pp.txt", "r", stdin);
    freopen("output_preemptive.txt", "w", stdout);

    int no_of_processes;
    scanf("%d", &no_of_processes);

    Process ps[no_of_processes];
    for (int i = 0; i < no_of_processes; i++){
        scanf("%d", &ps[i].id);
        scanf("%d", &ps[i].priority);
        scanf("%d", &ps[i].AT);

        sum_tt -= ps[i].AT;

        int cnt = -1;
        do {
            cnt++;
            int temp;
            scanf("%d", &temp);
            ps[i].bt[cnt] = temp;
            sum_wt -= temp;
        } while (ps[i].bt[cnt] != -1);
        ps[i].ptr = 0;
    }



    int completed = 0;
    int time = 0;
    while (completed != no_of_processes){
        int i = 0;
        for (; i < no_of_processes; i++){
            if (ps[i].AT <= time){
                ps[i].bt[ps[i].ptr] --;

                printf("Time Stamp: %d, executed: %d\n", time, ps[i].id);
                if (ps[i].bt[ps[i].ptr] == 0){
                    if (ps[i].bt[(ps[i].ptr)+1] == -1 || ps[i].bt[(ps[i].ptr)
+2] == -1){
                        completed++;
                        sum_tt += time;
                        ps[i].AT = __INT_MAX__;
                    } else {
                        int io_time = ps[i].bt[(ps[i].ptr)+1];
                        ps[i].AT = time + io_time;
```

```
                        ps[i].ptr += 2;
                    }
                }
                break;
            }
        }
        if (i == no_of_processes){
            printf("Time Stamp: %d: Idle CPU\n", time);
        }
        time++;
    }

    printf("Average Waiting Time: %f\n",
((float)(sum_tt-sum_wt))/no_of_processes);
    printf("Average Turnaround Time: %f\n", ((float)sum_tt)/no_of_processes);
}
```

ROUND ROBIN

```
#include<stdio.h>
#include<stdlib.h>

typedef struct Queue {
      int front, rear, size;
      unsigned capacity;
      int* array;
} Queue;

Queue* createQueue(unsigned capacity) {
      Queue* queue = (Queue *)malloc(sizeof(Queue));
      queue->capacity = capacity;
      queue->front = queue->size = 0;

      queue->rear = capacity - 1;
      queue->array = (int *)malloc(queue->capacity*sizeof(int));
      return queue;
}
int isFull(Queue* queue) {
      return (queue->size == queue->capacity);
}
int isEmpty(Queue* queue) {
      return (queue->size == 0);
}
void enqueue(Queue* queue, int item) {
      if (isFull(queue)) return;
      queue->rear = (queue->rear + 1) % queue->capacity;
      queue->array[queue->rear] = item;
      queue->size = queue->size + 1;
}
int dequeue(Queue* queue) {
      if (isEmpty(queue))
            return -1;
      int item = queue->array[queue->front];
      queue->front = (queue->front + 1)
                        % queue->capacity;
      queue->size = queue->size - 1;
      return item;
}
int front(Queue* queue) {
      if (isEmpty(queue))
            return -1;
      return queue->array[queue->front];
}
int rear(Queue* queue) {
```

```
      if (isEmpty(queue))
            return -1;
      return queue->array[queue->rear];
}

typedef struct Process {
    int id;
    int priority;
    int AT;
    int bt[100];

    int ptr;
} Process;

#define TIME_QUANTUM 20


int main(){

      int sum_wt = 0;
      int sum_tt = 0;

    freopen("input.txt", "r", stdin);
    freopen("output_roundrobin.txt", "w", stdout);

    int no_of_processes;
    scanf("%d", &no_of_processes);

    Process ps[no_of_processes];
    for (int i = 0; i < no_of_processes; i++){
        scanf("%d", &ps[i].id);
        scanf("%d", &ps[i].priority);
        scanf("%d", &ps[i].AT);
        sum_tt -= ps[i].AT;

        int cnt = -1;
        do {
            cnt++;
            int temp;
            scanf("%d", &temp);
            ps[i].bt[cnt] = temp;
            sum_wt -= temp;
        } while (ps[i].bt[cnt] != -1);
        ps[i].ptr = 0;
    }


    Queue * FCFS = createQueue(no_of_processes);

    for (int i = 0; i < no_of_processes; i++){
        if (ps[i].AT == 0) {
            enqueue(FCFS, i);
        }
    }

    int completed = 0;
    int time = 0;
    while (completed != no_of_processes){

        if (isEmpty(FCFS)){
            printf("Time Stamp: %d: Idle CPU\n", time);
            time++;
            for (int j = 0; j < no_of_processes; j++){
```

```
                if (ps[j].AT == time) {
                    enqueue(FCFS, j);
                }
            }
            continue;
        }

        int i = 0;
        for (; i < TIME_QUANTUM; i++){
            int pid = front(FCFS);
            ps[pid].bt[ps[pid].ptr] --;
            printf("Time Stamp: %d, executed: %d\n", time, ps[pid].id);
            if (ps[pid].bt[ps[pid].ptr] == 0){
                dequeue(FCFS);
                if (ps[pid].bt[(ps[pid].ptr)+1] == -1 ||
ps[pid].bt[(ps[pid].ptr)+2] == -1){
                    completed++;
                    sum_tt += time;
                    printf("Completed %d\n", completed);
                    ps[pid].AT = __INT_MAX__;
                } else {
                    int io_time = ps[pid].bt[(ps[pid].ptr)+1];
                    ps[pid].AT = time + io_time;
                    ps[pid].ptr+=2;
                }
                time++;
                for (int j = 0; j < no_of_processes; j++){
                    if (ps[j].AT == time) {
                        printf("added!");
                        enqueue(FCFS, j);
                    }
                }
                break;
            }
            time++;
            for (int j = 0; j < no_of_processes; j++){
                if (ps[j].AT == time) {
                    printf("added!");
                    enqueue(FCFS, j);
                }
            }
        }

        // if (time == 900) return 1;

        if (i == TIME_QUANTUM){
            int temp = dequeue(FCFS);
            enqueue(FCFS, temp);
        }
        getchar();
    }

    printf("Average Waiting Time: %f\n",
((float)(sum_tt-sum_wt))/no_of_processes);
    printf("Average Turnaround Time: %f\n", ((float)sum_tt)/no_of_processes);
}
```

OUTPUT:

```
FCFS
```

```
Time Stamp: 0, executed: 3
Time Stamp: 1, executed: 3
Time Stamp: 2, executed: 3
Time Stamp: 3, executed: 3
...
...
Time Stamp: 17924, executed: 7
Time Stamp: 17925, executed: 7
Completed 20
Average Waiting Time: 15029.650391
Average Turnaround Time: 13526.000000
```

```
PREEMPTIVE PRIORITY
```

```
Time Stamp: 0, executed: 16
Time Stamp: 1, executed: 16
Time Stamp: 2, executed: 16
Time Stamp: 3, executed: 16
...
...
Time Stamp: 18481, executed: 0
Time Stamp: 18482, executed: 0
Average Waiting Time: 11413.299805
Average Turnaround Time: 9909.650391
```

```
ROUND ROBIN
```

```
Time Stamp: 0, executed: 16
Time Stamp: 1, executed: 16
Time Stamp: 2, executed: 16
Time Stamp: 3, executed: 16
...
...
Time Stamp: 17924, executed: 17
Time Stamp: 17925, executed: 17
Completed 20
Average Waiting Time: 17141.150391
Average Turnaround Time: 15637.500000
```

QUESTION 02: Create child processes: X and Y.

a. Each child process performs 10 iterations. The child process displays its name/id and the current iteration number, and sleeps for some random amount of time. Adjust the sleeping duration of the processes to have different outputs (i.e. another interleaving of processes' traces).

b. Modify the program so that X is not allowed to start iteration i before process Y has terminated its own iteration i-1. Use semaphore to implement this synchronization.

c. Modify the program so that X and Y now perform in lockstep [both perform iteration I, then iteration i+1, and so on] with the condition mentioned in Q (2b) above.

d. Add another child process Z.

Perform the operations as mentioned in Q (2a) for all three children.

Then perform the operations as mentioned in Q (2c) [that is, 3 children in lockstep].

CODE:

A.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include <sys/wait.h>
// #include <semaphore.h>
// #include <fcntl.h>

#define SLEEP 0.1
#define NO_OF_PROCESS 3
int main(int argc, char const *argv[])
{

    int parent_pid = getpid();
    //2 processes

    pid_t *child = (pid_t *)malloc(NO_OF_PROCESS * sizeof(pid_t));

    for (int i = 0; i < NO_OF_PROCESS; i++){
        child[i] = fork();
        if (child[i] < 0) {
            //error
            printf("Forking failed\n");
            printf("Child no: %d", i);
            return 1;
        } else if (child[i] == 0){
            /**
             * @brief
             * child ->
             * do the loop here
             * 10 iterations
             */
            srand(getpid());
            for (int j = 0; j < 10; j++){
                // sleep(((float)rand() / (RAND_MAX)) * SLEEP);
                printf("Process PID: %d\tParent PID: %d\tIteration count: %d\
n", getpid(), getppid(), j);
                sleep(rand()%10);
            }

            break;


        } else {
            /**
             * @brief
             * parent ->
             * print details
             */
            printf("Child process: %d, pid: %d \n", i, child[i]);
        }
    }


    if (getpid() == parent_pid){
        int v;
        for (int i = 0; i < 2; i++){
            waitpid(child[i], &v, 0);
        }
        free(child);
```

```
    }
    return 0;
}
```

B.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <fcntl.h>

#define SLEEP 0.1
#define NO_OF_PROCESS 3
#define NO_OF_ITERATION 10

char * uitoa(int num){
    char * str;
    str = (char *)malloc(5 * sizeof(char));
    str[4] = 0;
    for (int i = 3; i >= 0; i--){
        str[i] = (num%10)+'0';
        num/=10;
    }
    return str;
}

int main(int argc, char const *argv[])
{

    int parent_pid = getpid();
    pid_t *child = (pid_t *)malloc(NO_OF_PROCESS * sizeof(pid_t));

    sem_t ***sem = (sem_t ***)malloc(NO_OF_PROCESS * sizeof(sem_t**));
    for (int i = 0; i < NO_OF_PROCESS-1; i++){
        sem[i] = (sem_t **)malloc(NO_OF_ITERATION * sizeof(sem_t *));
        for (int j = 0; j < NO_OF_ITERATION; j++){
            char * pcnt = uitoa(i);
            char * itcnt = uitoa(j);
            sem[i][j] = sem_open(strcat(pcnt, itcnt), O_CREAT, 0660, 0);
            free(pcnt);
            free(itcnt);
        }
    }


    for (int i = 0; i < NO_OF_PROCESS; i++){
        child[i] = fork();
        if (child[i] < 0) {
            //error
            printf("Forking failed\n");
            printf("Child no: %d", i);
            return 1;
        } else if (child[i] == 0){
            /**
             * @brief
             * child ->
             * do the loop here
             * 10 iterations
```

```
             */
            srand(getpid());
            for (int j = 0; j < 10; j++){
                if (i != 0){
                    sem_wait(sem[i-1][j]);
                }
                    sleep(rand()%10);
                printf("Process PID: %d\tParent PID: %d\tIteration count: %d\
n", getpid(), getppid(), j);
                if (i != NO_OF_PROCESS - 1){
                    sem_post(sem[i][j]);
                }
            }
            break;


        } else {
            /**
             * @brief
             * parent ->
             * print details
             */
            printf("Child process: %d, pid: %d \n", i, child[i]);
        }
    }


    if (getpid() == parent_pid){
        int v;
        for (int i = 0; i < 2; i++){
            waitpid(child[i], &v, 0);
        }
        free(child);

        for (int i = 0; i < NO_OF_PROCESS-1; i++){
            for (int j = 0; j < NO_OF_ITERATION; j++){
                char * pcnt = uitoa(i);
                char * itcnt = uitoa(j);
                sem_unlink(strcat(pcnt, itcnt));
                free(pcnt);
                free(itcnt);
            }
            free(sem[i]);
        }
    }
    return 0;
}
```

C.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <fcntl.h>

#define NO_OF_PROCESS 3
#define NO_OF_ITERATION 10
```

```
char * uitoa(int num){
    char * str;
    str = (char *)malloc(5 * sizeof(char));
    str[4] = 0;
    for (int i = 3; i >= 0; i--){
        str[i] = (num%10)+'0';
        num/=10;
    }
    return str;
}

int main(int argc, char const *argv[])
{

    int parent_pid = getpid();
    pid_t *child = (pid_t *)malloc(NO_OF_PROCESS * sizeof(pid_t));

    sem_t **sem = (sem_t **)malloc((NO_OF_PROCESS) * sizeof(sem_t *));
    for (int i = 0; i < NO_OF_PROCESS; i++){
        char * itcnt = uitoa(i);
        sem_unlink(itcnt);
        sem[i] = sem_open(itcnt, O_CREAT, 0660, 1);
        free(itcnt);
    }

    sem_unlink("mutex");
    sem_unlink("mutex2");
    sem_t *mutex = sem_open("mutex", O_CREAT, 0660, 1);
    sem_t *mutex2 = sem_open("mutex2", O_CREAT, 0660, 1);

    sem_unlink("count");
    sem_t *count = sem_open("count", O_CREAT, 0660, NO_OF_PROCESS);

    for (int i = 0; i < NO_OF_PROCESS; i++){
        child[i] = fork();
        if (child[i] < 0) {
            //error
            printf("Forking failed\n");
            printf("Child no: %d", i);
            return 1;
        } else if (child[i] == 0){
            /**
             * @brief
             * child ->
             * do the loop here
             * 10 iterations
             */
            pid_t temp = getpid();
            srand(temp);
            for (int j = 0; j < NO_OF_ITERATION; j++){
                // sem_wait(mutex2);
                sem_wait(sem[i]);
                sem_wait(count);
                printf("Process PID: %d\tIteration count: %d\n", i, j);
                sleep(sleep(rand()%10));
                sem_wait(mutex);
                int val;
                int t = sem_getvalue(count, &val);
                if (!val){
                    for (int k = 0; k < NO_OF_PROCESS; k++){
                        sem_post(sem[k]);
                        sem_post(count);
                    }
```

```
                }
                sem_post(mutex);
                // sem_post(mutex2);
            }
            break;


        } else {
            /**
             * @brief
             * parent ->
             * print details
             */
            printf("Child process: %d, pid: %d \n", i, child[i]);
        }
    }


    if (getpid() == parent_pid){
        int v;
        for (int i = 0; i < NO_OF_PROCESS; i++){
            waitpid(child[i], &v, 0);
        }
        free(child);
        for (int j = 0; j < NO_OF_PROCESS; j++){
            char * itcnt = uitoa(j);
            sem_unlink(itcnt);
            free(itcnt);
        }
        free(sem);
    }
    return 0;
}
```

OUTPUT:

A.

```
Child process: 0, pid: 4873
Process PID: 4873      Parent PID: 4872      Iteration count: 0
Child process: 1, pid: 4874
Process PID: 4873      Parent PID: 4872      Iteration count: 1
Child process: 2, pid: 4875
Process PID: 4874      Parent PID: 4872      Iteration count: 0
Process PID: 4875      Parent PID: 4872      Iteration count: 0
Process PID: 4875      Parent PID: 4872      Iteration count: 1
Process PID: 4875      Parent PID: 4872      Iteration count: 2
Process PID: 4873      Parent PID: 4872      Iteration count: 2
Process PID: 4873      Parent PID: 4872      Iteration count: 3
Process PID: 4874      Parent PID: 4872      Iteration count: 1
Process PID: 4874      Parent PID: 4872      Iteration count: 2
Process PID: 4875      Parent PID: 4872      Iteration count: 3
Process PID: 4874      Parent PID: 4872      Iteration count: 3
Process PID: 4873      Parent PID: 4872      Iteration count: 4
Process PID: 4875      Parent PID: 4872      Iteration count: 4
Process PID: 4873      Parent PID: 4872      Iteration count: 5
Process PID: 4875      Parent PID: 4872      Iteration count: 5
Process PID: 4874      Parent PID: 4872      Iteration count: 4
Process PID: 4873      Parent PID: 4872      Iteration count: 6
```

```
Process PID: 4874    Parent PID: 4872    Iteration count: 5
Process PID: 4874    Parent PID: 4872    Iteration count: 6
Process PID: 4873    Parent PID: 4872    Iteration count: 7
Process PID: 4873    Parent PID: 4872    Iteration count: 8
Process PID: 4875    Parent PID: 4872    Iteration count: 6
Process PID: 4873    Parent PID: 4872    Iteration count: 9
Process PID: 4874    Parent PID: 4872    Iteration count: 7
Process PID: 4875    Parent PID: 4872    Iteration count: 7
Process PID: 4874    Parent PID: 4872    Iteration count: 8
Process PID: 4875    Parent PID: 4872    Iteration count: 8
Process PID: 4874    Parent PID: 4872    Iteration count: 9
Process PID: 4875    Parent PID: 4872    Iteration count: 9
```

B.

```
Child process: 0, pid: 5512
Child process: 1, pid: 5513
Child process: 2, pid: 5514
Process PID: 5512    Parent PID: 5511    Iteration count: 0
Process PID: 5513    Parent PID: 5511    Iteration count: 0
Process PID: 5512    Parent PID: 5511    Iteration count: 1
Process PID: 5512    Parent PID: 5511    Iteration count: 2
Process PID: 5512    Parent PID: 5511    Iteration count: 3
Process PID: 5513    Parent PID: 5511    Iteration count: 1
Process PID: 5514    Parent PID: 5511    Iteration count: 0
Process PID: 5514    Parent PID: 5511    Iteration count: 1
Process PID: 5512    Parent PID: 5511    Iteration count: 4
Process PID: 5513    Parent PID: 5511    Iteration count: 2
Process PID: 5512    Parent PID: 5511    Iteration count: 5
Process PID: 5512    Parent PID: 5511    Iteration count: 6
Process PID: 5513    Parent PID: 5511    Iteration count: 3
Process PID: 5514    Parent PID: 5511    Iteration count: 2
Process PID: 5512    Parent PID: 5511    Iteration count: 7
Process PID: 5513    Parent PID: 5511    Iteration count: 4
Process PID: 5514    Parent PID: 5511    Iteration count: 3
Process PID: 5512    Parent PID: 5511    Iteration count: 8
Process PID: 5513    Parent PID: 5511    Iteration count: 5
Process PID: 5514    Parent PID: 5511    Iteration count: 4
Process PID: 5512    Parent PID: 5511    Iteration count: 9
Process PID: 5514    Parent PID: 5511    Iteration count: 5
Process PID: 5513    Parent PID: 5511    Iteration count: 6
Process PID: 5514    Parent PID: 5511    Iteration count: 6
Process PID: 5513    Parent PID: 5511    Iteration count: 7
Process PID: 5513    Parent PID: 5511    Iteration count: 8
Process PID: 5513    Parent PID: 5511    Iteration count: 9
Process PID: 5514    Parent PID: 1467    Iteration count: 7
Process PID: 5514    Parent PID: 1467    Iteration count: 8
Process PID: 5514    Parent PID: 1467    Iteration count: 9
```

C.

```
Child process: 0, pid: 5624
Child process: 1, pid: 5625
Child process: 2, pid: 5626
```

```
Process PID: 0  Iteration count: 0
Process PID: 1  Iteration count: 0
Process PID: 2  Iteration count: 0
Process PID: 0  Iteration count: 1
Process PID: 1  Iteration count: 1
Process PID: 2  Iteration count: 1
Process PID: 1  Iteration count: 2
Process PID: 0  Iteration count: 2
Process PID: 2  Iteration count: 2
Process PID: 0  Iteration count: 3
Process PID: 1  Iteration count: 3
Process PID: 2  Iteration count: 3
Process PID: 2  Iteration count: 4
Process PID: 0  Iteration count: 4
Process PID: 1  Iteration count: 4
Process PID: 1  Iteration count: 5
Process PID: 0  Iteration count: 5
Process PID: 2  Iteration count: 5
Process PID: 2  Iteration count: 6
Process PID: 0  Iteration count: 6
Process PID: 1  Iteration count: 6
Process PID: 1  Iteration count: 7
Process PID: 0  Iteration count: 7
Process PID: 2  Iteration count: 7
Process PID: 2  Iteration count: 8
Process PID: 1  Iteration count: 8
Process PID: 0  Iteration count: 8
Process PID: 0  Iteration count: 9
Process PID: 2  Iteration count: 9
Process PID: 1  Iteration count: 9
```

QUESTION 03: Implement the following applications using different IPC mechanisms. Your choice is restricted to Pipe, FIFO:
a. Broadcasting weather information (one broadcasting process and more than one listeners)
b. Telephonic conversation (between a caller and a receiver)

APPROACH: I used FIFO using mkfifo instruction to implement the broadcasting program.

CODE:

A_BROADCASTER

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <sys/stat.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <fcntl.h>


#define BUFFER_SIZE 1024

typedef char bool;
bool true = 1;
bool false = 0;

void swap(char *c1, char *c2)
{
    char *tmp = c1;
    c1 = c2;
    c2 = tmp;
}

/* A utility function to reverse a string  */
void reverse(char str[], int length)
{
    int start = 0;
    int end = length -1;
    while (start < end)
    {
        swap((str+start), (str+end));
        start++;
        end--;
    }
}

// Implementation of itoa()
char* itoa(int num, char* str, int base)
{
    int i = 0;
    bool isNegative = false;

    /* Handle 0 explicitly, otherwise empty string is printed for 0 */
    if (num == 0)
    {
        str[i++] = '0';
        str[i] = '\0';
        return str;
    }

    // In standard itoa(), negative numbers are handled only with
    // base 10. Otherwise numbers are considered unsigned.
    if (num < 0 && base == 10)
    {
        isNegative = true;
        num = -num;
    }

    // Process individual digits
    while (num != 0)
    {
        int rem = num % base;
        str[i++] = (rem > 9)? (rem-10) + 'a' : rem + '0';
        num = num/base;
    }

    // If number is negative, append '-'
```

```
    if (isNegative)
        str[i++] = '-';

    str[i] = '\0'; // Append string terminator

    // Reverse the string
    reverse(str, i);

    return str;
}

int main()
{
    int fd, n, numBytes;
    char *myFifo = "/tmp/myFifo";
    mkfifo(myFifo, 0666);
    char buffer[BUFFER_SIZE];

    printf("Enter number of receivers: ");
    scanf("%d", &n);
    char tmpBuffer[33];


    // Semaphores
    sem_t *semFull[n];
    for(int i = 0; i < n; i++)
    {
        sem_unlink(itoa(i, tmpBuffer, 10));
        if((semFull[i] = sem_open(itoa(i, tmpBuffer, 10), O_CREAT, 0660, 0)) ==
SEM_FAILED){ printf("Sem Failed (%d) !!!\n", i); return 1; }
    }

    sem_t *semCnt, *semCheckNext;
    sem_unlink("cnt"), sem_unlink("chk");
    if((semCnt = sem_open("cnt", O_CREAT, 0660, 0)) == SEM_FAILED){ printf("Sem
Failed in cnt !!!\n"); return 1; }
    if((semCheckNext = sem_open("chk", O_CREAT, 0660, 0)) == SEM_FAILED)
{ printf("Sem Failed in chk !!!\n"); return 1; }




    printf("\nYou are the WEATHER MASTER !!!\n");
    printf("Let everyone know about the weather :)  ...\n");
    printf("Enter \"!end\" to end (without quotes) ...\n\n");

    int curIter = 0;
    do
    {
        curIter++;
        for(int i = 0; i < n; i++) sem_post(semFull[i]);

        printf("Enter your message [#%d]: ", curIter);
        // fflush(stdin);
        // scanf("%[^\n]s", buffer);
        scanf("%s", buffer);
        // getchar();  // to remove dummy newline

        fd = open(myFifo, O_WRONLY);  // write only mode
        // If I am not using the return of write function, fifo not properly
written or read | WHY ???
----------------------------------------------------------------------
        // for(int i = 0; i < n; i++) numBytes = write(fd, buffer,
```

```
strlen(buffer) + 1); /* printf("lol2 | %d\n", write(fd, buffer, strlen(buffer)
+ 1)); */ // +1 to account for '\0'
        for(int i = 0; i < n; i++) numBytes = write(fd, buffer,
BUFFER_SIZE); /* printf("lol2 | %d\n", write(fd, buffer, strlen(buffer) + 1));
*/ // +1 to account for '\0'
        close(fd);

        // printf("lol3\n");
        while(true)
        {
            // printf("lol4\n");
            sem_wait(semCheckNext);
            int val;
            sem_getvalue(semCnt, &val);
            if(val == curIter * n) break;
        }
    }
    while(strcmp(buffer, "!end"));

    printf("\n\nMay the weather be with you ...\n");
    printf("Program ended ...\n");

    sem_unlink("cnt"), sem_unlink("chk");  // printf("su\n");
    for(int i = 0; i < n; i++) sem_unlink(itoa(i, tmpBuffer, 10)),
sem_destroy(semFull[i]);
    sem_destroy(semCnt), sem_destroy(semCheckNext);  // printf("sd\n");

    return 0;
}
```

A_LISTENER

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <fcntl.h>


#define BUFFER_SIZE 1024

typedef char bool;
bool true = 1;
bool false = 0;

void swap(char *c1, char *c2)
{
    char *tmp = c1;
    c1 = c2;
    c2 = tmp;
}

/* A utility function to reverse a string  */
void reverse(char str[], int length)
{
    int start = 0;
    int end = length -1;
    while (start < end)
    {
```

```
        swap((str+start), (str+end));
        start++;
        end--;
    }
}

// Implementation of itoa()
char* itoa(int num, char* str, int base)
{
    int i = 0;
    bool isNegative = false;

    /* Handle 0 explicitly, otherwise empty string is printed for 0 */
    if (num == 0)
    {
        str[i++] = '0';
        str[i] = '\0';
        return str;
    }

    // In standard itoa(), negative numbers are handled only with
    // base 10. Otherwise numbers are considered unsigned.
    if (num < 0 && base == 10)
    {
        isNegative = true;
        num = -num;
    }

    // Process individual digits
    while (num != 0)
    {
        int rem = num % base;
        str[i++] = (rem > 9)? (rem-10) + 'a' : rem + '0';
        num = num/base;
    }

    // If number is negative, append '-'
    if (isNegative)
        str[i++] = '-';

    str[i] = '\0'; // Append string terminator

    // Reverse the string
    reverse(str, i);

    return str;
}

int main()
{
    int fd, n, numBytes;
    char *myFifo = "/tmp/myFifo";
    mkfifo(myFifo, 0666);
    char buffer[BUFFER_SIZE];

    printf("Enter number of receivers: ");
    scanf("%d", &n);
    char tmpBuffer[33];


    // Semaphores
    sem_t *semFull[n];
    for(int i = 0; i < n; i++)
```

```
    {
        sem_unlink(itoa(i, tmpBuffer, 10));
        if((semFull[i] = sem_open(itoa(i, tmpBuffer, 10), O_CREAT, 0660, 0)) ==
SEM_FAILED){ printf("Sem Failed (%d) !!!\n", i); return 1; }
    }

    sem_t *semCnt, *semCheckNext;
    sem_unlink("cnt"), sem_unlink("chk");
    if((semCnt = sem_open("cnt", O_CREAT, 0660, 0)) == SEM_FAILED){ printf("Sem
Failed in cnt !!!\n"); return 1; }
    if((semCheckNext = sem_open("chk", O_CREAT, 0660, 0)) == SEM_FAILED)
{ printf("Sem Failed in chk !!!\n"); return 1; }




    printf("\nYou are the WEATHER MASTER !!!\n");
    printf("Let everyone know about the weather :)  ...\n");
    printf("Enter \"!end\" to end (without quotes) ...\n\n");

    int curIter = 0;
    do
    {
        curIter++;
        for(int i = 0; i < n; i++) sem_post(semFull[i]);

        printf("Enter your message [#%d]: ", curIter);
        // fflush(stdin);
        // scanf("%[^\n]s", buffer);
        scanf("%s", buffer);
        // getchar();  // to remove dummy newline

        fd = open(myFifo, O_WRONLY);  // write only mode
        // If I am not using the return of write function, fifo not properly
written or read | WHY ???
-----------------------------------------------------------------------
        // for(int i = 0; i < n; i++) numBytes = write(fd, buffer,
strlen(buffer) + 1); /* printf("lol2 | %d\n", write(fd, buffer, strlen(buffer)
+ 1)); */ // +1 to account for '\0'
        for(int i = 0; i < n; i++) numBytes = write(fd, buffer,
BUFFER_SIZE); /* printf("lol2 | %d\n", write(fd, buffer, strlen(buffer) + 1));
*/ // +1 to account for '\0'
        close(fd);

        // printf("lol3\n");
        while(true)
        {
            // printf("lol4\n");
            sem_wait(semCheckNext);
            int val;
            sem_getvalue(semCnt, &val);
            if(val == curIter * n) break;
        }
    }
    while(strcmp(buffer, "!end"));

    printf("\n\nMay the weather be with you ...\n");
    printf("Program ended ...\n");

    sem_unlink("cnt"), sem_unlink("chk");  // printf("su\n");
    for(int i = 0; i < n; i++) sem_unlink(itoa(i, tmpBuffer, 10)),
sem_destroy(semFull[i]);
    sem_destroy(semCnt), sem_destroy(semCheckNext);  // printf("sd\n");
```

```
    return 0;
}
```

B_CALLER

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <sys/types.h>


#define BUFFER_SIZE 1024

typedef char bool;
bool true = 1;
bool false = 0;


int main()
{
    int fd;
    char *myFifo = "/tmp/myFifo";
    mkfifo(myFifo, 0666);
    char buffer[BUFFER_SIZE];
    bool callEndedBySelf = false;

    // Semaphores ...
    sem_unlink("cs"), sem_unlink("rs");
    sem_t *callerCanSend, *receiverCanSend;
    if((callerCanSend = sem_open("cs", O_CREAT, 0660, 0)) == SEM_FAILED)
{ printf("Sem failed callerCanSend !!!\n"); return 1; }
    if((receiverCanSend = sem_open("rs", O_CREAT, 0660, 0)) == SEM_FAILED)
{ printf("Sem failed callerCanSend !!!\n"); return 1; }


    printf("You are the CALLER !!!\n");
    printf("Enter \"!end\" to end (without quotes) ...\n\n");

    do
    {
        // SENDING ...
        fd = open(myFifo, O_WRONLY);  // write only mode

        sem_wait(callerCanSend);
        // printf("lol\n");
        printf("Enter your message: ");
        scanf("%[^\n]s", buffer);
        getchar();  // to remove dummy newline

        write(fd, buffer, strlen(buffer) + 1);  // +1 to account for '\n'
        close(fd);

        if(!strcmp(buffer, "!end")){ callEndedBySelf = true; break; }


        // RECEIVING ...
        fd = open(myFifo, O_RDONLY);  // read only mode
        sem_post(receiverCanSend);
```

```
        read(fd, buffer, BUFFER_SIZE);
        printf("Message received: %s\n", buffer);
        close(fd);
    }
    while(strcmp(buffer, "!end"));

    printf("\t--- CALL ENDED BY %s ---\n", (callEndedBySelf)? "CALLER (You)":
"RECEIVER");
    printf("Program ended ...\n");

    sem_destroy(receiverCanSend), sem_destroy(callerCanSend);
    sem_unlink("cs"), sem_unlink("rs");

    return 0;
}
```

B_RECEIVER

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <sys/types.h>


#define BUFFER_SIZE 1024

typedef char bool;
bool true = 1;
bool false = 0;


int main()
{
    int fd;
    char *myFifo = "/tmp/myFifo";
    mkfifo(myFifo, 0666);
    char buffer[BUFFER_SIZE];
    bool callEndedBySelf = true;

    // Semaphores ...
    sem_t *callerCanSend, *receiverCanSend;
    if((callerCanSend = sem_open("cs", O_CREAT, 0660, 0)) == SEM_FAILED)
{ printf("Sem failed callerCanSend !!!\n"); return 1; }
    if((receiverCanSend = sem_open("rs", O_CREAT, 0660, 0)) == SEM_FAILED)
{ printf("Sem failed callerCanSend !!!\n"); return 1; }


    printf("You are the RECEIVER !!!\n");
    printf("Enter \"!end\" to end (without quotes) ...\n\n");

    do
    {
        // RECEIVING ...
        fd = open(myFifo, O_RDONLY);  // read only mode
        sem_post(callerCanSend);
        read(fd, buffer, BUFFER_SIZE);
        printf("Message received: %s\n", buffer);
        close(fd);
```

```
        if(!strcmp(buffer, "!end")){ callEndedBySelf = false; break; }


        // SENDING ...
        fd = open(myFifo, O_WRONLY);  // write only mode

        sem_wait(receiverCanSend);
        printf("Enter your message: ");
        scanf("%[^\n]s", buffer);
        getchar();  // to remove dummy newline

        write(fd, buffer, strlen(buffer) + 1);  // +1 to account for '\n'
        close(fd);
    }
    while(strcmp(buffer, "!end"));

    printf("\n\t--- CALL ENDED BY %s ---\n", (!callEndedBySelf)? "CALLER":
"RECEIVER (You)");
    printf("Program ended ...\n");

    sem_destroy(receiverCanSend), sem_destroy(callerCanSend);

    return 0;
}
```

OUTPUT:

A_BROADCASTER

```
Enter number of receivers: 1

You are the WEATHER MASTER !!!
Let everyone know about the weather :)  ...
Enter "!end" to end (without quotes) ...

Enter your message [#1]: Nice weather...
Enter your message [#2]: Enter your message [#3]: Bad
Enter your message [#4]: !end


May the weather be with you ...
Program ended ...
```

A_LISTENER

```
Enter receiver number: 0
We listen to the WEATHER MASTER !!!
Let us everyone know about the weather :)  ...

Message received: Nice
Message received: weather...
Message received: Bad
Message received: !end


May the weather be with you ...
        --- END OF TRANSMISSION ---
Program ended ...
```

B_CALLER

```
You are the CALLER !!!
```

```
Enter "!end" to end (without quotes) ...

Enter your message: Hello
Message received: Hi
Enter your message: How are you
Message received: Fine
Enter your message: ok Bye
Message received: !end
        --- CALL ENDED BY RECEIVER ---
Program ended ...
```

B_RECEIVER

```
You are the RECEIVER !!!
Enter "!end" to end (without quotes) ...

Message received: Hello
Enter your message: Hi
Message received: How are you
Enter your message: Fine
Message received: ok Bye
Enter your message: !end

        --- CALL ENDED BY RECEIVER (You) ---
Program ended ...
```

QUESTION 04: Write a program for p-producer c-consumer problem, p, c >= 1. A shared circular buffer that can hold 25 items is to be used. Each producer process stores any numbers between 1 to 80 (along with the producer id) in the buffer one by one and then exits. Each consumer process reads the numbers from the buffer and adds them to a shared variable TOTAL (initialized to 0). Though any consumer process can read any of the numbers in the buffer, the only constraint being that every number written by some producer should be read exactly once by exactly one of the consumers.
(a) The program reads in the value of p and c from the user, and forks p producers and c consumers.
(b) After all the producers and consumers have finished (the consumers exit after all the data produced by
all producers have been read), the parent process prints the value of TOTAL.
Test the program with different values of p and c.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <semaphore.h>
#include <fcntl.h>

#include <sys/mman.h>
#include <sys/wait.h>

typedef char bool;
bool true = 1;
bool false = 0;
```

```
#define MAX_BUFFER_SIZE 1024


typedef struct __Data
{
    int producerId;
    int data;
} Data;

// Data INVALID_DATA;
// INVALID_DATA.producerId = INVALID_DATA.data = -1;

typedef struct __CircularBuffer
{
    // Data *buffer;
    Data buffer[MAX_BUFFER_SIZE];
    int size, maxSize;
    int front, back;
} CircularBuffer;

void createCircularBuffer(CircularBuffer *cb, int size)
{
    cb->maxSize = size;
    // cb->buffer = (Data *)malloc(cb->maxSize * sizeof(Data));
    cb->size = cb->front = cb->back = 0;
}

void push(CircularBuffer *cb, Data d)
{
    if(cb->size >= cb->maxSize) return;

    cb->buffer[(cb->front++) % cb->maxSize] = d;
    cb->size++;
}

Data pop(CircularBuffer *cb)
{
    Data INVALID_DATA;
    INVALID_DATA.producerId = INVALID_DATA.data = -1;
    if(cb->size <= 0) return INVALID_DATA;

    cb->size--;
    return cb->buffer[(cb->back++) % cb->maxSize];
}


int main()
{
    int p, c, n;
    // Need shared memory for total
    int *TOTAL = (int *)mmap(NULL , sizeof(int), PROT_READ | PROT_WRITE ,
MAP_SHARED | MAP_ANONYMOUS , -1, 0);
    *TOTAL = 0;

    printf("Enter #producers: ");
    scanf("%d", &p);
    printf("Enter #consumers: ");
    scanf("%d", &c);
    printf("Enter size of circular buffer: ");
    scanf("%d", &n);

    // Semaphores ...
    sem_unlink("mutex"), sem_unlink("empty"), sem_unlink("full"),
```

```
sem_unlink("prodCntLeft");
    sem_t *mutex, *empty, *full, *prodCntLeft;
    if((mutex = sem_open("mutex", O_CREAT, 0660, 1)) == SEM_FAILED)
{ printf("Sem failed mutex !!!\n"); return 1; }
    if((empty = sem_open("empty", O_CREAT, 0660, n)) == SEM_FAILED)
{ printf("Sem failed empty !!!\n"); return 1; }
    if((full = sem_open("full", O_CREAT, 0660, 0)) == SEM_FAILED){ printf("Sem
failed full !!!\n"); return 1; }
    if((prodCntLeft = sem_open("prodCntLeft", O_CREAT, 0660, p)) == SEM_FAILED)
{ printf("Sem failed prodCntLeft !!!\n"); return 1; }

    // Need shared memory for circular buffer
    CircularBuffer *buffer = (CircularBuffer *)mmap(NULL ,
sizeof(CircularBuffer), PROT_READ | PROT_WRITE , MAP_SHARED | MAP_ANONYMOUS , -
1, 0);
    createCircularBuffer(buffer, n);
    pid_t producer[p], consumer[c];

    for(int i = 0; i < p; i++)
    {
        producer[i] = fork();

        if(producer[i] < 0)
        {
            printf("Fork producer #%d failed, exiting ...\n", i);
            return 1;
        }
        else if(producer[i] == 0)
        {
            // Producer Subprocess
            bool allProducerDone = false;
            int val;
            Data d;

            sem_wait(empty);
            sem_wait(mutex);

            d.data = d.producerId = i;
            push(buffer, d);
            printf("Producer #%d produces data = %d  [BUFFER SIZE = %d] ...\n",
i, i, buffer->size);
            sem_wait(prodCntLeft);
            sem_getvalue(prodCntLeft, &val);
            if(!val) allProducerDone = true;

            sem_post(mutex);
            sem_post(full);

            if(allProducerDone)
            {
                printf("\t\t___ ALL PRODUCERS DONE !!! ___\n");
                for(int j = 0; j < c; j++) sem_post(full);  // When consumers
see buffer is empty, they will know all producers are dead and exit ...
            }
            // sem_post_multiple(full, c);  // doesnot work

            return 0;
        }
        else
        {
            // returns pid of child to parent process
            printf("\t\t---------- Producer Process #%d created (pid: %d)
----------\n", i, producer[i]);
```

```
        }
    }

    for(int i = 0; i < c; i++)
    {
        consumer[i] = fork();

        if(consumer[i] < 0)
        {
            printf("Fork consumer #%d failed, exiting ...\n", i);
            return 1;
        }
        else if(consumer[i] == 0)
        {
            // Consumer Subprocess
            while(true)
            {
                // Every consumer keeps on consuming till producers can produce
...
                Data d;

                sem_wait(full);
                sem_wait(mutex);

                // printf("Debug consumer --> %d\n", buffer->size);
                if(buffer->size <= 0)
                {
                    printf("\t--- Buffer EMPTY , consumer #%d exitting ---\n",
i);
                    sem_post(mutex);
                    return 0;
                }
                d = pop(buffer);
                printf("Consumer #%d gets data = %d from producer #%d\n", i,
d.data, d.producerId);
                *TOTAL = *TOTAL + d.data;

                sem_post(mutex);
                sem_post(empty);
            }
        }
        else
        {
            // returns pid of child to parent process
            printf("\t\t---------- Consumer Process #%d created (pid: %d)
----------\n", i, consumer[i]);
        }
    }

    int stat;
    // printf("lol\n");
    for(int i = 0; i < c; i++) waitpid(consumer[i], &stat, 0);  //
printf("lol2\n");
    for(int i = 0; i < p; i++) waitpid(producer[i], &stat, 0);  //
printf("lol1\n");


    sem_unlink("mutex"), sem_unlink("empty"), sem_unlink("full"),
sem_unlink("prodCntLeft");
    sem_destroy(mutex), sem_destroy(empty), sem_destroy(full),
sem_destroy(prodCntLeft);

    printf("\n\nFinal Total Value = %d\n", *TOTAL);
```

```
    return 0;
}
```

OUTPUT:

```
Enter #producers: 30
Enter #consumers: 30
Enter size of circular buffer: 25
                ---------- Producer Process #0 created (pid: 8389) ----------
                ---------- Producer Process #1 created (pid: 8390) ----------
Producer #0 produces data = 0  [BUFFER SIZE = 1] ...
Producer #1 produces data = 1  [BUFFER SIZE = 2] ...
                ---------- Producer Process #2 created (pid: 8391) ----------
Producer #2 produces data = 2  [BUFFER SIZE = 3] ...
                ---------- Producer Process #3 created (pid: 8392) ----------
Producer #3 produces data = 3  [BUFFER SIZE = 4] ...
                ---------- Producer Process #4 created (pid: 8393) ----------
                ---------- Producer Process #5 created (pid: 8394) ----------
Producer #4 produces data = 4  [BUFFER SIZE = 5] ...
Producer #5 produces data = 5  [BUFFER SIZE = 6] ...
                ---------- Producer Process #6 created (pid: 8395) ----------
Producer #6 produces data = 6  [BUFFER SIZE = 7] ...
                ---------- Producer Process #7 created (pid: 8396) ----------
Producer #7 produces data = 7  [BUFFER SIZE = 8] ...
                ---------- Producer Process #8 created (pid: 8397) ----------
                ---------- Producer Process #9 created (pid: 8398) ----------
Producer #8 produces data = 8  [BUFFER SIZE = 9] ...
Producer #9 produces data = 9  [BUFFER SIZE = 10] ...
                ---------- Producer Process #10 created (pid: 8399) ----------
                ---------- Producer Process #11 created (pid: 8400) ----------
Producer #10 produces data = 10  [BUFFER SIZE = 11] ...
                ---------- Producer Process #12 created (pid: 8401) ----------
Producer #11 produces data = 11  [BUFFER SIZE = 12] ...
Producer #12 produces data = 12  [BUFFER SIZE = 13] ...
                ---------- Producer Process #13 created (pid: 8402) ----------
Producer #13 produces data = 13  [BUFFER SIZE = 14] ...
                ---------- Producer Process #14 created (pid: 8403) ----------
Producer #14 produces data = 14  [BUFFER SIZE = 15] ...
                ---------- Producer Process #15 created (pid: 8404) ----------
Producer #15 produces data = 15  [BUFFER SIZE = 16] ...
                ---------- Producer Process #16 created (pid: 8405) ----------
Producer #16 produces data = 16  [BUFFER SIZE = 17] ...
                ---------- Producer Process #17 created (pid: 8406) ----------
Producer #17 produces data = 17  [BUFFER SIZE = 18] ...
                ---------- Producer Process #18 created (pid: 8407) ----------
                ---------- Producer Process #19 created (pid: 8408) ----------
Producer #19 produces data = 19  [BUFFER SIZE = 19] ...
                ---------- Producer Process #20 created (pid: 8409) ----------
Producer #18 produces data = 18  [BUFFER SIZE = 20] ...
Producer #20 produces data = 20  [BUFFER SIZE = 21] ...
                ---------- Producer Process #21 created (pid: 8410) ----------
Producer #21 produces data = 21  [BUFFER SIZE = 22] ...
                ---------- Producer Process #22 created (pid: 8411) ----------
Producer #22 produces data = 22  [BUFFER SIZE = 23] ...
                ---------- Producer Process #23 created (pid: 8412) ----------
Producer #23 produces data = 23  [BUFFER SIZE = 24] ...
                ---------- Producer Process #24 created (pid: 8413) ----------
Producer #24 produces data = 24  [BUFFER SIZE = 25] ...
                ---------- Producer Process #25 created (pid: 8414) ----------
                ---------- Producer Process #26 created (pid: 8415) ----------
```

```
               ---------- Producer Process #27 created (pid: 8416) ----------
               ---------- Producer Process #28 created (pid: 8417) ----------
               ---------- Producer Process #29 created (pid: 8418) ----------
               ---------- Consumer Process #0 created (pid: 8419) ----------
Consumer #0 gets data = 0 from producer #0
Consumer #0 gets data = 1 from producer #1
               ---------- Consumer Process #1 created (pid: 8420) ----------
Producer #25 produces data = 25  [BUFFER SIZE = 24] ...
Producer #26 produces data = 26  [BUFFER SIZE = 25] ...
               ---------- Consumer Process #2 created (pid: 8421) ----------
Consumer #2 gets data = 2 from producer #2
Producer #27 produces data = 27  [BUFFER SIZE = 25] ...
               ---------- Consumer Process #3 created (pid: 8422) ----------
Consumer #1 gets data = 3 from producer #3
Producer #28 produces data = 28  [BUFFER SIZE = 25] ...
Consumer #2 gets data = 4 from producer #4
Consumer #2 gets data = 5 from producer #5
               ---------- Consumer Process #4 created (pid: 8423) ----------
Consumer #2 gets data = 6 from producer #6
Consumer #3 gets data = 7 from producer #7
               ---------- Consumer Process #5 created (pid: 8424) ----------
Producer #29 produces data = 29  [BUFFER SIZE = 22] ...
Consumer #2 gets data = 8 from producer #8
               ___ ALL PRODUCERS DONE !!! ___
Consumer #2 gets data = 9 from producer #9
Consumer #0 gets data = 10 from producer #10
Consumer #0 gets data = 11 from producer #11
Consumer #3 gets data = 12 from producer #12
               ---------- Consumer Process #6 created (pid: 8425) ----------
Consumer #5 gets data = 13 from producer #13
Consumer #5 gets data = 14 from producer #14
Consumer #0 gets data = 15 from producer #15
Consumer #0 gets data = 16 from producer #16
Consumer #3 gets data = 17 from producer #17
Consumer #0 gets data = 19 from producer #19
Consumer #0 gets data = 18 from producer #18
Consumer #3 gets data = 20 from producer #20
               ---------- Consumer Process #7 created (pid: 8426) ----------
Consumer #5 gets data = 21 from producer #21
Consumer #1 gets data = 22 from producer #22
Consumer #0 gets data = 23 from producer #23
Consumer #2 gets data = 24 from producer #24
Consumer #2 gets data = 25 from producer #25
Consumer #5 gets data = 26 from producer #26
Consumer #1 gets data = 27 from producer #27
Consumer #0 gets data = 28 from producer #28
Consumer #3 gets data = 29 from producer #29
        --- Buffer EMPTY , consumer #0 exitting ---
        --- Buffer EMPTY , consumer #5 exitting ---
        --- Buffer EMPTY , consumer #3 exitting ---
               ---------- Consumer Process #8 created (pid: 8427) ----------
        --- Buffer EMPTY , consumer #2 exitting ---
               ---------- Consumer Process #9 created (pid: 8428) ----------
        --- Buffer EMPTY , consumer #1 exitting ---
        --- Buffer EMPTY , consumer #8 exitting ---
               ---------- Consumer Process #10 created (pid: 8429) ----------
        --- Buffer EMPTY , consumer #10 exitting ---
               ---------- Consumer Process #11 created (pid: 8430) ----------
        --- Buffer EMPTY , consumer #11 exitting ---
               ---------- Consumer Process #12 created (pid: 8431) ----------
        --- Buffer EMPTY , consumer #12 exitting ---
               ---------- Consumer Process #13 created (pid: 8432) ----------
        --- Buffer EMPTY , consumer #13 exitting ---
```

```
                ---------- Consumer Process #14 created (pid: 8433) ----------
        --- Buffer EMPTY , consumer #14 exitting ---
                ---------- Consumer Process #15 created (pid: 8434) ----------
        --- Buffer EMPTY , consumer #15 exitting ---
                ---------- Consumer Process #16 created (pid: 8435) ----------
        --- Buffer EMPTY , consumer #16 exitting ---
                ---------- Consumer Process #17 created (pid: 8436) ----------
        --- Buffer EMPTY , consumer #17 exitting ---
                ---------- Consumer Process #18 created (pid: 8437) ----------
        --- Buffer EMPTY , consumer #18 exitting ---
                ---------- Consumer Process #19 created (pid: 8438) ----------
        --- Buffer EMPTY , consumer #19 exitting ---
                ---------- Consumer Process #20 created (pid: 8439) ----------
        --- Buffer EMPTY , consumer #20 exitting ---
                ---------- Consumer Process #21 created (pid: 8440) ----------
        --- Buffer EMPTY , consumer #21 exitting ---
                ---------- Consumer Process #22 created (pid: 8441) ----------
        --- Buffer EMPTY , consumer #22 exitting ---
                ---------- Consumer Process #23 created (pid: 8442) ----------
        --- Buffer EMPTY , consumer #23 exitting ---
                ---------- Consumer Process #24 created (pid: 8443) ----------
        --- Buffer EMPTY , consumer #4 exitting ---
                ---------- Consumer Process #25 created (pid: 8444) ----------
        --- Buffer EMPTY , consumer #25 exitting ---
                ---------- Consumer Process #26 created (pid: 8445) ----------
        --- Buffer EMPTY , consumer #6 exitting ---
                ---------- Consumer Process #27 created (pid: 8446) ----------
        --- Buffer EMPTY , consumer #26 exitting ---
                ---------- Consumer Process #28 created (pid: 8447) ----------
        --- Buffer EMPTY , consumer #27 exitting ---
        --- Buffer EMPTY , consumer #28 exitting ---
                ---------- Consumer Process #29 created (pid: 8448) ----------
        --- Buffer EMPTY , consumer #24 exitting ---
        --- Buffer EMPTY , consumer #9 exitting ---
        --- Buffer EMPTY , consumer #29 exitting ---
        --- Buffer EMPTY , consumer #7 exitting ---


Final Total Value = 435
```

QUESTION 05: Write a program for the Reader-Writer process for the following situations:
a) Multiple readers and one writer: writer gets to write whenever it is ready (reader/s wait)
b) Multiple readers and multiple writers: any writer gets to write whenever it is ready, provided no other writer is currently writing (reader/s wait)

CODE:

B.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <semaphore.h>
#include <fcntl.h>
```

```
#include <sys/mman.h>
#include <sys/wait.h>

char * uitoa(int num){
    char * str;
    str = (char *)malloc(5 * sizeof(char));
    str[4] = 0;
    for (int i = 3; i >= 0; i--){
        str[i] = (num%10)+'0';
        num/=10;
    }
    return str;
}

int main(){
    int r_cnt = 10;
    int w_cnt = 10;
    int read_iteration_count = 20;

    int *buffer = (int *)mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE ,
MAP_SHARED | MAP_ANONYMOUS , -1, 0);
    int *reader_cnt = (int *)mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE ,
MAP_SHARED | MAP_ANONYMOUS , -1, 0);
    int *writer_cnt = (int *)mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE ,
MAP_SHARED | MAP_ANONYMOUS , -1, 0);

    *buffer =  -1; //init
    *reader_cnt = 0;
    *writer_cnt = 0;

    pid_t readers[r_cnt];
    pid_t writers[w_cnt];

    int t;

    //semaphores
    sem_unlink("wrt");
    sem_unlink("mutex");
    sem_unlink("w_priority");
    sem_unlink("start");

    sem_t *wrt = sem_open("wrt", O_CREAT, 0660, 1);
    sem_t *mutex = sem_open("mutex", O_CREAT, 0660, 1);
    sem_t *w_priority = sem_open("w_priority", O_CREAT, 0660, 1);
    sem_t *start = sem_open("start", O_CREAT, 0660, 0);
    sem_t *is_writer_done[r_cnt];
    for (int i = 0; i < r_cnt; i++){
        char * temp = uitoa(i);
        char * temp2 = strcat(temp, "is_writer_done");
        sem_unlink(temp2);
        is_writer_done[i] = sem_open(temp2, O_CREAT, 0660, 0);
    }




    //reader processes
    for (int i = 0; i < r_cnt; i++){
        readers[i] = fork();
        if (readers[i] < 0){
            perror("Forking failed\n");
            return 1;
```

```
        } else if (readers[i] == 0) {
            srand(getpid());
            sleep(sleep(rand()%10));
                sem_wait(start);

                //read and print this many times!!
                while (*writer_cnt > 0)
                    sem_wait(is_writer_done[i]);
                    //while since is_writer_done maybe > 0

                sem_wait(mutex);
                  *reader_cnt = *reader_cnt + 1;
                  if(*reader_cnt == 1) sem_wait(wrt);
                sem_post(mutex);

                printf("Reader cnt: %d\tValue: %d \n", i, *buffer);

                sem_wait(mutex);
                  *reader_cnt = *reader_cnt - 1;
                  if(*reader_cnt == 0) sem_post(wrt);
                sem_post(mutex);
            return 0;
        } else {
            printf("Reader No: %d created\n", i);
        }
    }

    //writer processes
    for (int i = 0; i < w_cnt; i++){
        writers[i] = fork();
        if (writers[i] < 0){
            perror("Forking failed\n");
            return 1;
        } else if (writers[i] == 0) {
            srand(getpid());
            sleep(sleep(rand()%10));

            sem_wait(start);

            sem_wait(w_priority);
            *writer_cnt = *writer_cnt + 1;
            sem_post(w_priority);

            sem_wait(wrt);
            *buffer = i;
            printf("Writer %d wrote data: %d\n", i, i);
            sem_post(wrt);

            sem_wait(w_priority);
            *writer_cnt = *writer_cnt - 1;
            sem_post(w_priority);

            for(int j = 0; j < r_cnt; j++) sem_post(is_writer_done[j]);

            return 0;
        } else {
            printf("Writer No: %d created\n", i);
        }
    }

    int v;
    for (int i = 0; i < r_cnt+w_cnt; i++){
        sem_post(start);
```

```
    }
    for (int i = 0; i < r_cnt; i++){
        waitpid(readers[i], &v, 0);
    }
    for (int i = 0; i < w_cnt; i++){
        waitpid(writers[i], &v, 0);
    }

    sem_unlink("wrt");
    sem_unlink("mutex");
    sem_unlink("start");
    sem_unlink("w_priority");
    for (int i = 0; i < r_cnt; i++){
        char * temp = uitoa(i);
        char * temp2 = strcat(temp, "is_writer_done");
        sem_unlink(temp2);
    }
    return 0;
}
```

OUPUT:

B.

```
Reader No: 0 created
Reader No: 1 created
Reader No: 2 created
Reader No: 3 created
Reader No: 4 created
Reader No: 5 created
Reader No: 6 created
Reader No: 7 created
Reader No: 8 created
Reader No: 9 created
Writer No: 0 created
Writer No: 1 created
Writer No: 2 created
Writer No: 3 created
Writer No: 4 created
Writer No: 5 created
Writer No: 6 created
Writer No: 7 created
Writer No: 8 created
Writer No: 9 created
Writer 5 wrote data: 5
Writer 9 wrote data: 9
Reader cnt: 8   Value: 9
Writer 2 wrote data: 2
Reader cnt: 6   Value: 2
Writer 0 wrote data: 0
Reader cnt: 7   Value: 0
Reader cnt: 2   Value: 0
Reader cnt: 3   Value: 0
Writer 1 wrote data: 1
Writer 4 wrote data: 4
Reader cnt: 0   Value: 4
Reader cnt: 1   Value: 4
Reader cnt: 4   Value: 4
```

```
Reader cnt: 5   Value: 4
Writer 8 wrote data: 8
Reader cnt: 9   Value: 8
Writer 3 wrote data: 3
Writer 6 wrote data: 6
Writer 7 wrote data: 7
```

QUESTION 06: Implement Dining Philosophers' problem using Monitor. Test the program with (a) 5 philosophers and 5 chopsticks, (b) 6 philosophers and 6 chopsticks, and (c) 7 philosophers and 7 chopsticks

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/wait.h>

#define NOP 5 // number of philosophers

#define THINKING 0
#define HUNGRY 1
#define EATING 2

char * uitoa(int num){
    char * str;
    str = (char *)malloc(5 * sizeof(char));
    str[4] = 0;
    for (int i = 3; i >= 0; i--){
        str[i] = (num%10)+'0';
        num/=10;
    }
    return str;
}

typedef struct condition {
    sem_t *t;
    int count;
} Condition;

typedef struct DP {
    sem_t * mutex;
    sem_t * next;
    int next_count;
    int state[NOP];
    Condition x[NOP];
    // int turn[NOP];
} DP;

void cond_wait(DP * dp, int i){
```

```
    dp->x[i].count++;
    if (dp->next_count > 0){
        sem_post(dp->next);
    } else {
        sem_post(dp->mutex);
    }
    sem_wait(dp->x[i].t);
    dp->x[i].count--;
}

void cond_signal (DP * dp, int i){
    if (dp->x[i].count > 0){
            dp->next_count++;
            sem_post(dp->x[i].t);
            sem_wait(dp->next);
            dp->next_count--;
    }
}

void test (DP * dp, int i){
    if (dp->state[(i+NOP-1)%NOP] != EATING && dp->state[(i+1)% NOP] != EATING
&& dp->state[i] == HUNGRY){ // && dp->turn[i] == i && dp->turn[(i+NOP-1)%NOP]
== i){
        dp->state[i] = EATING;
        cond_signal(dp, i);
    }
}

void pickup(DP * dp, int i){

    sem_wait(dp->mutex);
        dp->state[i] = HUNGRY;
        test(dp, i);
        if (dp->state[i] == HUNGRY){
            cond_wait(dp, i);
        }
        printf("%d got the chopsticks\n", i);
    if (dp->next_count > 0){
        sem_post(dp->next);
    } else {
        sem_post(dp->mutex);
    }
}

void putdown(DP * dp, int i){

    sem_wait(dp->mutex);
        dp->state[i] = THINKING;
        // dp->turn[i] = (i+1)%NOP;
          // dp->turn[(i+NOP-1)%NOP] = (i+NOP-1)%NOP;
        printf("%d put the chopsticks down\n", i);
        test(dp, (i+NOP-1)%NOP);
        test(dp, (i+1)%NOP);
    if (dp->next_count > 0){
        sem_post(dp->next);
    } else {
        sem_post(dp->mutex);
    }
}
```

```
void init (DP * dp){

    /**
     * sem_t * mutex;
     * sem_t * next;
     * int next_count;
     * int state[NOP];
     * Condition x[NOP];
     */

    sem_unlink("mutex");
    dp->mutex = sem_open("mutex", O_CREAT, 0660, 1);
    sem_unlink("next");
    dp->next = sem_open("next", O_CREAT, 0660, 0);

    dp->next_count = 0;

    for (int i = 0; i < NOP; i++){
        dp->state[i] = THINKING;
        //condition
        dp->x[i].count = 0;
        char * temp = uitoa(i);
        char * temp2 = strcat(temp, "cond");
        sem_unlink(temp2);
        dp->x[i].t = sem_open(temp2, O_CREAT, 0660, 0);
        //turn
        // dp->turn[i] = ((i/2)*2 + 2)%NOP;
    }
}

int main(){

    DP * dp = (DP *)mmap(NULL, sizeof(DP), PROT_READ | PROT_WRITE , MAP_SHARED
| MAP_ANONYMOUS , -1, 0);

    init (dp);
    pid_t child [NOP];
    for (int i = 0; i < NOP; i++){
        child[i] = fork();
        if (child[i] < 0){
            perror("Forking failed\n");
            return 1;
        } else if (child[i] == 0){
            for (int j = 0; j < 10; j++){
                srand(getpid());
                sleep(rand()%10);
                pickup(dp, i);
                sleep(rand()%5);
                putdown(dp, i);
            }
            return 0;
        } else {
            1;
        }
    }

    int v;
    for (int i = 0; i < NOP; i++){
        waitpid(child[i], &v, 0);
    }
}
```

OUTPUT:

```
4 got the chopsticks
1 got the chopsticks
1 put the chopsticks down
2 got the chopsticks
2 put the chopsticks down
4 put the chopsticks down
3 got the chopsticks
3 put the chopsticks down
1 got the chopsticks
1 put the chopsticks down
4 got the chopsticks
2 got the chopsticks
2 put the chopsticks down
4 put the chopsticks down
3 got the chopsticks
0 got the chopsticks
0 put the chopsticks down
3 put the chopsticks down
0 got the chopsticks
0 put the chopsticks down
```

QUESTION 07: Write a program that will find out whether a system is in safe state or not with following specifications:
Command line input: name of a file - The file contains the initial state of the system as given below:
#no of resources 4 #no of instances of each resource 2 4 5 3
#no of processes 3 #no of instances of each resource that each process needs in its lifetime 1 1 1 1, 2 3 1 2, 2 2 1 3
The program waits to accept a resource allocation request to be supplied by the user or read from another file:
For example: 0 1 0 1 1 indicates that p0 has requested allocation of 1 instance of R0, R2 and R3 each. Your program should declare the result:
(1) should this request be granted?
(2) if your answer is yes, print the safe sequence in which all remaining needs can be granted one by one and also grant the request. If the requesting process's need is NIL, the program internally releases all its resources. Go back to accept another request till all processes finish with all their needs.
Testing:
a. Generate possible request sequences of each process.
b. Each such sequence must satisfy the maximum requirements of the process.

CODE:

```
#include<stdio.h>
#include<stdlib.h>
int no_of_types_resource;
int no_of_processes;
int no_of_requests;

int * max_resource_count;
int ** process_requirement;
int ** orig;
```

```
int ** process_requests;
int * permutate;
int * freq;

int cnt = 0;

void swap(int a, int b){
    int temp = permutate[a];
    permutate[a] = permutate[b];
    permutate[b] = temp;
}

void allocate (int loc){
    int pid = process_requests[loc][0];

    freq[pid] --;

    for (int i = 1; i<=no_of_types_resource; i++){
        process_requirement[pid][i-1] -= process_requests[loc][i];
        max_resource_count[i-1] -= process_requests[loc][i];
    }

    if (freq[pid] == 0){
        for (int i = 0; i < no_of_types_resource; i++){
            max_resource_count[i] += orig[pid][i];
        }
    }
}

void deallocate (int loc){
    int pid = process_requests[loc][0];

    freq[pid] ++;

    for (int i = 1; i<=no_of_types_resource; i++){
        process_requirement[pid][i-1] += process_requests[loc][i];
        max_resource_count[i-1] += process_requests[loc][i];
    }

    if (freq[pid] == 1){
        for (int i = 0; i < no_of_types_resource; i++){
            max_resource_count[i] -= orig[pid][i];
        }
    }
}

void solve(int index){
    if (index == no_of_requests){
        for (int i = 0; i < no_of_requests; i++){
            printf("%d ", permutate[i]+1);
        } printf("\n");
        cnt++;
        return;
    }

    int j = 0;
    for (; j < no_of_processes; j++){
        int k = 0;
        for (; k < no_of_types_resource; k++){
            if (process_requirement[j][k] > max_resource_count[k]) break;
        }
        if (k == no_of_types_resource) break;
    }
```

```
    if (j == no_of_processes) return;


    for (int i = index; i < no_of_requests; i++){

        swap(index, i);

        int k;
        for (k = 0; k < no_of_types_resource; k++){
            if (process_requests[permutate[index]][k+1] > max_resource_count[k]
)break;
        }

        if (k == no_of_types_resource) {
            int t = permutate[index];
            allocate(t);
            solve (index+1);
            deallocate(t);
        }
        swap(index, i);
    }
}

int main(){
    FILE * fpreq = fopen("prerequisite.txt", "r");
    FILE * falloc = fopen("allocation.txt", "r");
    freopen("output.txt", "w", stdout);
    fscanf(fpreq, "%d", &no_of_types_resource);
    max_resource_count = (int *)malloc (sizeof(int) * no_of_types_resource);
    for (int i = 0; i < no_of_types_resource; i++){
        int temp;
        fscanf(fpreq, "%d", max_resource_count+i);
    }

    fscanf(fpreq, "%d", &no_of_processes);
    freq = (int *) malloc (sizeof(int) * no_of_processes);
    process_requirement = (int **)malloc(sizeof(int *) * no_of_processes);
    orig = (int **)malloc(sizeof(int *) * no_of_processes);
    for (int i = 0; i < no_of_processes; i++){
        freq[i] = 0;
        process_requirement[i] = (int *)malloc(sizeof(int) *
no_of_types_resource);
        orig[i] = (int *)malloc(sizeof(int) * no_of_types_resource);
        for (int j = 0; j < no_of_types_resource; j++){
            int temp;
            fscanf(fpreq, "%d", &temp);
            process_requirement[i][j] = temp;
            orig[i][j] = temp;
        }
    }


    fscanf(falloc, "%d", &no_of_requests);
    process_requests = (int **)malloc(sizeof(int*)*no_of_requests);
    for (int i = 0; i < no_of_requests; i++){
        process_requests[i] = (int
*)malloc(sizeof(int)*(no_of_types_resource+1));
        for (int j = 0; j < (no_of_types_resource+1); j++){
            int temp;
            fscanf(falloc, "%d", &temp);
            process_requests[i][j] = temp;
            if (j == 0){
                freq[temp] ++;
```

```
            }
        }
    }

    permutate = (int *)malloc(sizeof(int)*no_of_requests);
    for (int i = 0; i < no_of_requests; i++){
        permutate[i] = i;
    }

    fclose(fpreq);
    fclose(falloc);

    printf("Valid output sequences are listed below: \n");
    printf("----------------------------------------\n");

    solve (0);
    printf("Total number of solutions: %d\n\n", cnt);

    return 0;
}
```

INPUT:

PREREQUISITE	ALLOCATION
4 2 4 5 3 3 2 1 1 1 2 3 4 2 2 2 4 3	8 0 1 0 1 1 1 0 3 2 0 2 1 1 4 1 0 1 1 0 0 1 1 0 2 1 2 1 1 0 0 1 1 0 0 1 2 0 0 0 2

OUTPUT:

```
Valid output sequences are listed below:
----------------------------------------
1 2 4 5 7 6 3 8
1 2 4 5 7 6 8 3
1 2 4 5 7 3 6 8
1 2 4 5 7 3 8 6
...
...
8 1 4 6 3 7 5 2
8 1 4 6 3 7 2 5
8 1 4 6 3 2 7 5
8 1 4 6 3 2 5 7
Total number of solutions: 624
```