OPERATING SYSTEM ASSIGNMENT 2

Question 1: 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: -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, pre-emptive Priority and Round Robin (time slice:20)). b. Compare the average waiting time, turnaround time of each process for the different scheduling algorithms.

APPROACH: Firstly, we will create a class Process to store all attributes of a process(eg- jobid, priority, bursts times etc.). Before applying any scheduling algorithm, data of processes will be collected from a file and stored in an array of Process objects.

FCFS:- For FCFS all available processes will be stored in a priority queue which in a pop() gives the process with minimum arrival time. Suppose, a process has been scheduled and is running on CPU. After the end of its CPU burst it goes for IO(if present). The trick is that, after returning from IO operation it again enters in the priority queue with a new arrival time. As FCFS is non-pre-emptive, any process after getting CPU must finish the current CPU burst before going to IO or terminating.

PREEMPTIVE-PRIORITY: In this algorithm, we need to remember that, at any time instance if any new process with higher priority arrives, then it will be given the CPU after pre-empting the current process. The process which is being pre-empted must maintain remaining CPU burst for later continuation. Here also, after IO process re-enters within the priority queue, but in this case, priority is based on process-priority. The processes which are returning back from IO are temporarily stored in queue with less-arrival-time first. Then according to time-counter processes are inserted into priority queue based on process priority. System works in this way.

ROUND-ROBIN: It is purely pre-emptive. We will maintain a time quantum. Processes will be pushed in a simple queue. One by one processes will be given a chance to execute for the quantum duration. Any process pre-empted will enter the queue again in the rear. Process returning from IO will be inserted into the queue again. One temporary queue is needed for maintaining the IO-returned processes. They will be inserted in the main queue according to their returning time from the IO operation.

Code Snippet:

```
#include <bits/stdc++.h>
using namespace std;

class Process
{
public:
    int job_id;
    int priority;
    int arrival_time;
```

```
    int secondary_arrtime;
    vector<int> cpu_bursts;
    vector<int> io_bursts;
    int ci;

    Process(int jid, int p, int at)
    {
        job_id = jid;
        priority = p;
        arrival_time = at;
        secondary_arrtime = at;
        ci = 0;
    }

    void print()
    {
        cout << "PID:" << job_id << "\t" << priority << "\t" << arrival_time <<
endl;
    }
};

vector<Process> plist;
unordered_map<int, Process *> mp;

struct comp
{
    bool operator()(Process *const &p1, Process *const &p2)
    {
        // return "true" if "p1" is ordered
        // before "p2", for example:
        if (p1->priority < p2->priority)
        {
            return false;
        }
        else if (p1->priority > p2->priority)
        {
            return true;
        }
        else
            return p1->secondary_arrtime > p2->secondary_arrtime;
    }
};

void inputAllProcess(string filename)
{
    ifstream f;
    f.open(filename);
    if (!f)
    {
        cout << "File Error!" << endl;
        return;
    }

    int jobid, prio, arrtime;

    while (!f.eof())
    {
```

```
        f >> jobid;
        f >> prio;
        f >> arrtime;
        Process p(jobid, prio, arrtime);
        int k, cnt = 0;

        do
        {
            f >> k;
            if (k == -1)
                break;
            if (cnt == 0)
            {
                p.cpu_bursts.push_back(k);
            }
            else
            {
                p.io_bursts.push_back(k);
            }
            cnt = (cnt + 1) % 2;
        } while (true);
        plist.push_back(p);
    }

    f.close();
}

void fcfs()
{
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> minHeap;

    for (int i = 0; i < plist.size(); i++)
    {
        minHeap.push({plist[i].arrival_time, plist[i].job_id});
        mp[plist[i].job_id] = &plist[i];
    }

    int curtime = 0;
    int i = 0;
    while (!minHeap.empty())
    {

        auto it = minHeap.top();
        minHeap.pop();
        Process &p = *mp[it.second];

        if (p.secondary_arrtime > curtime)
        {
            curtime = p.secondary_arrtime;
        }

        cout << "-------------------------------------" << endl;
        cout << "Time: " << curtime << endl;
        p.print();
```

```
        cout << "Executes from " << curtime << " to " << (curtime +
p.cpu_bursts[p.ci]) << endl;
        curtime += p.cpu_bursts[p.ci];
        p.secondary_arrtime = curtime;

        if (p.ci != p.io_bursts.size())
        {
            cout << "Process going for IO" << endl;
            p.secondary_arrtime += p.io_bursts[p.ci];
        }

        if (p.ci == p.cpu_bursts.size() - 1)
        {
            cout << "Process Finished!!" << endl;
        }
        else
        {
            minHeap.push({p.secondary_arrtime, p.job_id});
        }

        p.ci++;
    }
}

void round_robin()
{
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> minHeap;
    queue<pair<int, int>> waitingQue;
    int QUANTUM = 20; // TIME QUANTUM FOR RR SCHE..

    for (int i = 0; i < plist.size(); i++)
    {
        minHeap.push({plist[i].arrival_time, plist[i].job_id});
        mp[plist[i].job_id] = &plist[i];
    }

    int curtime = 0;
    int i = 0;

    while (!minHeap.empty() || !waitingQue.empty())
    {
        while (!minHeap.empty())
        {
            auto it = minHeap.top();

            if (waitingQue.empty())
            {
                curtime = it.first;
            }
            if (it.first > curtime)
                break;
            minHeap.pop();
            Process &p = *mp[it.second];

            waitingQue.push(it);
        }
```

```
        auto it = waitingQue.front();
        Process &p = *mp[it.second];
        waitingQue.pop();

        cout << "-------------------------------------" << endl;
        cout << "Time: " << curtime << endl;
        p.print();

        if (p.cpu_bursts[p.ci] > QUANTUM)
        {
            p.cpu_bursts[p.ci] -= QUANTUM;
            cout << "Executed From " << curtime << " to " << (curtime +
QUANTUM) << endl;
            curtime += QUANTUM;
            waitingQue.push({curtime, it.second});
        }
        else
        {
            cout << "Executed From " << curtime << " to " << (curtime +
p.cpu_bursts[p.ci]) << endl;
            curtime += p.cpu_bursts[p.ci];
            if (p.ci == p.io_bursts.size())
            {
                cout << "Process Finished" << endl;
            }
            else
            {
                cout << "Going for IO" << endl;
                if (p.ci + 1 == p.cpu_bursts.size())
                    cout << "Process Finished!!" << endl;
                else
                    minHeap.push({curtime + p.io_bursts[p.ci], it.second});
                p.ci++;
            }
        }
    }
}

void preemptive_priority()
{
    priority_queue<Process *, vector<Process *>, comp> waiting_queue;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> minHeap;

    for (int i = 0; i < plist.size(); i++)
    {
        minHeap.push({plist[i].arrival_time, plist[i].job_id});
        mp[plist[i].job_id] = &plist[i];
    }

    int curtime = 0;

    while (!waiting_queue.empty() || !minHeap.empty())
    {
        if (waiting_queue.empty())
        {
```

```
        curtime = minHeap.top().first;
    }

    while (true)
    {
        if (minHeap.empty())
            break;
        auto it = minHeap.top();
        int ttt = it.first;
        if (ttt > curtime)
            break;
        minHeap.pop();
        Process *p = mp[it.second];
        waiting_queue.push(p);
    }

    Process *pp = waiting_queue.top();
    waiting_queue.pop();

    cout << "-----------------------------------------------\n";
    cout << "Time: " << curtime << endl;
    pp->print();

    int endtime = curtime + pp->cpu_bursts[pp->ci];
    int counter = curtime;
    while (!minHeap.empty() && counter < endtime)
    {
        auto it = minHeap.top();
        Process *temp = mp[it.second];
        if (counter == temp->secondary_arrtime)
        {
            minHeap.pop();
            waiting_queue.push(temp);
            if (temp->priority < pp->priority)
            {
                endtime = counter;
                break;
            }
        }
        else
            counter++;
    }

    cout << "Executes from " << curtime << " to " << endtime << endl;

    if (endtime == curtime + pp->cpu_bursts[pp->ci])
    {
        if (pp->ci < pp->io_bursts.size())
        {
            cout << "Going for IO..\n";
        }
        else
        {
            curtime = endtime;
            cout << "Process Finished..\n";
            continue;
        }
```

```
            pp->secondary_arrtime = endtime + pp->io_bursts[pp->ci];
            pp->ci++;
            if (pp->ci == pp->cpu_bursts.size())
            {
                curtime = endtime;
                cout << "Process Finished..\n";
                continue;
            }
            minHeap.push({pp->secondary_arrtime, pp->job_id});
        }
        else
        {
            cout << "Process Pre-empted..\n";
            pp->secondary_arrtime = endtime;
            pp->cpu_bursts[pp->ci] = (-endtime + curtime + pp->cpu_bursts[pp-
>ci]);
            waiting_queue.push(pp);
        }

        curtime = endtime;
    }
}

int main()
{
    string filename;
    cin >> filename;
    inputAllProcess(filename);

    // fcfs();
    // round_robin();
    preemptive_priority();

    return 0;
}
```

<u>**Question 2:**</u> **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.**

<u>APPROACH</u>: a) This part is really straightforward. Using fork, we will create two child processes X and Y. Each of them will iterate for 10 iterations with some sleep duration in between. This sleep is important

to make the iterations noticeable. Parent must wait for both child processes to finish their executions.

b) To implement this one, there will be some additional features to the previous code. As given, X must wait before iteration I until Y finishes iteration I-1. This can be implemented by introducing a semaphore ‘sem’ with initial value 1. X must wait on ‘sem’ before doing a particular iteration. And Y must signal on ‘sem’ after finishing any iteration. It assures that X will do i^{th} iteration when already Y has finished its $(i-1)^{th}$ iteration.

c) This one is some more advancement of the previous one. Here we will have two semaphores ‘s1’ and ‘s2’. s1 initialized with 1 and s2 with 0. X must wait on s1 before executing and signal on s2 after an iteration. Y must wait on s2 before executing and signal on s1 after executing. This conditions ensures that both X and Y will be locked state, means both iterate i^{th} then both $(i+1)^{th}$.

Code Snippet:

Q2a:

```
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

using namespace std;

int main()
{
    int id1 = fork(), id2;
    if (id1)
    {
        id2 = fork();
    }

    if (id1 == 0)
    {
        for (int i = 0; i < 10; i++)
        {
            cout << "Process X - ITERATION: " << i << endl;
            sleep(1);
        }
    }
    else if (id2 == 0)
    {
        for (int i = 0; i < 10; i++)
        {
            cout << "Process Y - ITERATION: " << i << endl;
            sleep(2);
        }
    }
    else
    {
        int wpid, status = 0;
```

```
        while ((wpid = wait(&status)) > 0);
    }
}
```

Q2b:

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

using namespace std;

int main(){
    unsigned int sem_value = 1;
    sem_t *sem = sem_open("semaphore", O_CREAT | O_EXCL, 0644, sem_value);

    int id1 = fork(),id2;
    if(id1){
     id2 = fork();
    }

    if(id1==0){
        for(int i=0; i<10; i++){
            sem_wait(sem);
            cout << "Process X - ITERATION: " << i << endl;
            srand(time(NULL));
            sleep(rand()%4+1);
        }
    }else if(id2==0){
        for(int i=0; i<10; i++){
            cout << "Process Y - ITERATION: " << i << endl;
            srand(time(NULL));
            sleep(rand()%2+1);
            sem_post(sem);
        }
    }else{
        int wpid, status=0;
        while((wpid=wait(&status))>0);
        sem_unlink("semaphore");
        sem_close(sem);
    }

}
```

Q2c:

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

using namespace std;

int main(){
    sem_t *s1 = sem_open("semaphore1", O_CREAT | O_EXCL, 0644, 1);
    sem_t *s2 = sem_open("semaphore2", O_CREAT | O_EXCL, 0644, 0);

    int id1 = fork(),id2;
    if(id1){
     id2 = fork();
    }

    if(id1==0){
        for(int i=0; i<10; i++){
            sem_wait(s1);
            cout << "Process X - ITERATION: " << i << endl;
            srand(time(NULL));
            sleep(rand()%4+1);
            sem_post(s2);
        }
    }else if(id2==0){
        for(int i=0; i<10; i++){
            sem_wait(s2);
            cout << "Process Y - ITERATION: " << i << endl;
            srand(time(NULL));
            sleep(rand()%2+1);
            sem_post(s1);
        }
    }else{
        int wpid, status=0;
        while((wpid=wait(&status))>0);
        sem_unlink("semaphore1");
        sem_unlink("semaphore2");
        sem_close(s1);
        sem_close(s2);
    }

}
```

Question 3: **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: a))) We will have one broadcaster process (say the parent) and multiple listeners(say child processes). We will use a FIFO mechanism to communicate between processes.

A shared and synchronized variable reqcnt will keep count of the number of read requests. Each listener intending to read will increment it. Broadcaster will only broadcast if the value of this reqcnt is equal to number of listeners.

Before reading each listener must wait on sems[i] where sems is an array of semaphores. After keeping the required broadcast value into FIFO, the broadcaster will signal all the listeners through signal(sems[i]). Also broadcaster makes the reqcnt = 0.

Code Snippet:

a>> Broadcasting:

```
#include <iostream>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <semaphore.h>
#include <sys/shm.h>
#include <errno.h>
#include <sys/wait.h>

using namespace std;

int main(){
      int pid;

      key_t k = ftok("/dev",77);
      int sid = shmget(k, sizeof(int), 0644 | IPC_CREAT);
      int * reqcnt = (int *) shmat(sid,NULL,0);
      *reqcnt = 0;

      int n, id=88,fd, val;
      cout << "Enter number of listeners: ";
      cin >> n;

      sem_t **sems;
      if(!(sems = (sem_t**) malloc(n*sizeof(sem_t*)))) {cout << "mem error";
return 0;}
      sem_t *mutex;
```

```
	for(int i=0; i<n; i++){
		string ss = "";
		ss.push_back('a'+i);
		cout << ss;
		sems[i] = sem_open((ss).c_str(), O_CREAT | O_EXCL, 0644, 0);
		cout << sems[i];
	}

	mutex = sem_open("bb", O_CREAT | O_EXCL, 0644, 1);

	for(int i=0; i<n; i++){
		pid = fork();
		id = i;
		if(pid==0) break;
	}


	if(pid==0){ //This is the listeners
		while(true){
			sem_wait(mutex);
			*reqcnt += 1;
			sem_post(mutex);
			sem_wait(sems[id]);
			fd = open("shared2",O_RDONLY);
			if(fd==-1){
				cout << "Listener " << id << " Fails to read the FIFO" <<
endl;
				return 2;
			}
			read(fd,&val,sizeof(val));
			cout << "Listener " << id << " finds " << val << endl;
			close(fd);
		}

	}else{ // This is broadcaster
		srand(time(NULL));
		int nn=5;
		while(nn--){
				while(true){
					sem_wait(mutex);
					if(*reqcnt==n){sem_post(mutex); break;}
					sem_post(mutex);
				}
				fd = open("shared2",O_WRONLY);
				if(fd==-1){
					cout << "Broadcaster Fails to write into the FIFO"
<< endl;
					return 2;
				}

				val = rand()%100;
				write(fd,&val,sizeof(val));
				cout << "Written " << val << endl;
				close(fd);

				sem_wait(mutex);
				*reqcnt = 0;
```

```
                sem_post(mutex);
                for(int i=0; i<n; i++){
                     sem_post(sems[i]);
                }
                sleep(6);
        }

        int wpid, status=0;
        while((wpid=wait(&status))>0);


        for(int i=0; i<n; i++){
             string ss = "";
             ss.push_back('a'+i);
             sem_unlink((ss).c_str());
             sem_close(sems[i]);
        }
        sem_unlink("bb");
        sem_close(mutex);
        shmdt(reqcnt);
    }

}
```

Question 4: 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

APPROACH: This is a classic synchronization problem with some modifications. Here multiple producers and consumers are present. Whatever is the case we will have a circular queue as a buffer. Producers will put new items into buffer and consumers will consume from there. For the solution we will have, -

Global shared(obviously synced) variables – itemsConsumed, rear, total, the queue.

Semaphores – mutex(binary) for mutual exclusion, full(counting) to keep full slots count, empty(counting) to keep empty slots count

Some random sleep in each producer and consumer

Each Producer: Must wait on '*empty*'. If it gets an empty slot then waits for the '*mutex*' to get access to the queue and other shared variables. If it gets access, then it creates any random number and stores in the queue's rear and adjust the 'rear' value. Then, quite obviously it signals on '*mutex*' and then on '*full*'. It exits thereafter.

Each Consumer: Runs an infinite while loop. In each iteration, waits on '*full*' and then on '*mutex*'. It thereafter checks the value of itemsConsumed == number of producers, if yes then there is nothing more to consume, consumer ends. Otherwise, consumer reads one item from queue front and adds it to the '*total*', increase the itemsConsumed by one, goes to next iteration.

Code Snippet:

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

#define BUF_SIZE 5

using namespace std;

int main(){

    // THE SEMAPHORES
    sem_t *mutex, *empty, *full;

    // SOME SHARED VARIABLES
    int *total , *itemsConsumed, *rear, *que, p, c;
    int type = 0, id;
    pid_t pid;

    // TAKING THE VALUE OF p AND c
    cin >> p >> c;

    /*============INITIALIZING THE SHARED VARIABLES===============*/
    key_t k1 = ftok("/dev",65), k2 = ftok("/dev", 64), k3=ftok("/dev",60), k4=8
93248628;

    int sid1 = shmget (k1, sizeof(int), 0644 | IPC_CREAT);
    int sid2 = shmget (k2, sizeof(int), 0644 | IPC_CREAT);
    int sid3 = shmget (k3, sizeof(int), 0644 | IPC_CREAT);
    int sid4 = shmget (k4, BUF_SIZE*sizeof(int), 0644 | IPC_CREAT);

    itemsConsumed = (int *)shmat(sid1, NULL, 0);
   total = (int *)shmat(sid2, NULL, 0);
   rear = (int *) shmat(sid3, NULL, 0);
   que = (int *) shmat(sid4, NULL, 0);

    *itemsConsumed = 0;
    *total = 0;
    *rear = 0;
```

```
/*==============================================================*/


/*======INITIALIZING THE SEMAPHORES FOR SYNCHRONIZATION=======*/
mutex = sem_open("mut", O_CREAT | O_EXCL, 0644, 1);
empty = sem_open("emp", O_CREAT | O_EXCL, 0644, BUF_SIZE);
full = sem_open("ful", O_CREAT | O_EXCL, 0644, 0);
/*============================================================*/

//FORKING ALL PRODUCERS
for(int i=0; i<p; i++){
    pid = fork();
    if(pid==0){
        type = 1;
        id = i;
        break;
    }
}


// FORKING ALL CONSUMERS
for(int i=0; i<c; i++){
    if(pid==0) break;
    pid = fork();
    if(pid==0){
        type = 2;
        id = i;
        break;
    }
}


if(type==1){ // IF IT IS PRODUCER--------------------------------
    srand(time(NULL));
    sleep(rand()%6+1);
    sem_wait(empty);
    sem_wait(mutex);

    /*====PRODUCING RANDOM VALUE STORE IN QUEUE=====*/
    srand(time(NULL));
    int temp = rand()%80+1;
    cout << "Producing.."  << temp << endl;
    que[*rear] = temp;
    *rear += 1;
    *rear %= BUF_SIZE;
    /*=============================================*/

    sleep(1);
    sem_post(mutex);
    sem_post(full);
    sleep(1);
}else if(type==2){ // IF A CONSUMER-------------------------------
    while(true){
        sem_wait(full);
        sem_wait(mutex);

        if((*itemsConsumed)==0) *total=0;
```

```
            *itemsConsumed += 1;
            if((*itemsConsumed)>p){
                sem_post(mutex);
                sem_post(full);
                break;
            }


            /*======CONSUMING ITEM========*/
            int data = que[(*itemsConsumed-1)%BUF_SIZE];
            cout << "Consumer " << id << " consuming.." << data << endl;
            *total += data;
            *rear += 0;
            /*============================*/

            if((*itemsConsumed)==p){
                sem_post(mutex);
                sem_post(full);
                break;
            }

            sleep(1);
            sem_post(mutex);
            sem_post(empty);
            sleep(1);
        }
    }else{ // THE PARENT ---------------------------------------
    int status,wpid;
        while((wpid=wait(&status))>0);
        cout << "TOTAL: " << *total << endl;
        sem_unlink("mut");
        sem_close(mutex);

        sem_unlink("ful");
        sem_close(full);

        sem_unlink("emp");
        sem_close(empty);

    }//---------------------------------------------------------

    return 0;
}
```

<u>Question 5:</u> 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)

<u>APPROACH</u>: This is a classic synchronization problem. Writers write and Readers read. Some rule must be followed so that there is no discrepancy in the written or read data. A writer can only write when no other reader is reading and no writer is writing. But multiple readers can concurrently read provided no writer is writing.

To implement this scenario –

#wrt: Binary Semaphore that provides mutual exclusion in the data such that reading and writing doesn't happen together. Writer must wait on wrt before each intention to write and signal after writing on the same. But only the first reader must wait on wrt before reading, then other readers can read together, the last leaving reader must signal on wrt.

#readcount: Shared integer variable to keep track of the count of readers.

#mutex: Binary Semaphore to provide mutual exclusion on readcount. Any update to the readcount must be within wait and signal on mutex.

Code Snippet:

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

using namespace std;

sem_t *mutex, *wrt;
int *readcount;

void readnow(int id)
{
    sleep(rand() % (rand()%20) + 1);
    sem_wait(mutex);

    sleep(rand() % 2 + 1);

    *readcount += 1;
    if (*readcount == 1)
    {
        sem_wait(wrt);
    }

    cout << "READER " << id << " READING - CURRENT READERS COUNT - " <<
*readcount << endl;
    sem_post(mutex);

    sleep(rand() % 3 + 1);

    sem_wait(mutex);
    *readcount -= 1;
    cout << "READER " << id << " EXITS - READERS COUNT - " << *readcount <<
endl;
```

```
    if (*readcount == 0)
    {
        sem_post(wrt);
    }
    sem_post(mutex);
}

void writenow(int id)
{
    sleep(rand() % 19 + 1);
    sem_wait(wrt);

    sleep(rand() % 3 + 1);

    cout << "WRITER " << id << " WRITING ..." << endl;
    sleep(1);
    cout << "WRITER " << id << " EXITS ..." << endl;

    sem_post(wrt);
}

int main()
{
    key_t key = 9283749;
    int sid = shmget(key, sizeof(int), 0644 | IPC_CREAT);

    readcount = (int *)shmat(sid, NULL, 0);

    mutex = sem_open("mut", O_CREAT | O_EXCL, 0644, 1);
    wrt = sem_open("wrt", O_CREAT | O_EXCL, 0644, 1);

    cout << "Enter reader and writer count: ";
    int r, w;
    cin >> r >> w;

    pid_t pid;
    int type = 0;
    int id = 0;
    *readcount = 0;

    for (int i = 0; i < r + w; i++)
    {
        pid = fork();

        if (pid == 0)
        {
            if (i < r)
            {
                type = 1;
                id = i;
            }
            else
            {
                type = 2;
                id = i - r;
            }
```

```
            break;
        }
    }

    if (type == 1)
    {
    srand(time(NULL)+getpid());
    while(true)
        readnow(id);
    }
    else if (type == 2)
    {
    srand(time(NULL)+getpid());
    while(true)
        writenow(id);
    }
    else
    {
        int status, wpid;
        while ((wpid = wait(&status)) > 0)
            ;
        sem_unlink("mut");
        sem_close(mutex);

        sem_unlink("wrt");
        sem_close(wrt);
    }
    return 0;
}
```

Question 6: **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.**

APPROACH: This is a classic synchronization problem. This could be solved in various ways. Trying to solve this problem in normal ways using Semaphores can lead to deadlock. The most efficient way to solve this problem is to use monitors. We will have two files – monitor.h -> This is for creating the functionalities of monitor, main.cpp -> This is the main program where philosophers will think and eat.

Code Snippet:

monitor.h .

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>
#include<cstdlib>
#include <semaphore.h>

#define N 7
#define THINKING 0
#define HUNGRY 1
#define EATING 2
```

```
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N

//semaphores to implement monitor
sem_t mutex;
sem_t next;
//count varaible for philosophers waiting on semaphore next
int next_count = 0;

//implementing condition variable using semaphore
//semaphore and integer variable replacing condition variable
typedef struct
{
	sem_t sem;
	//count variable for philosophers waiting on condition semaphore sem
	int count;
}condition;
condition x[N];

//state of each philosopher(THINKING, HUNGRY or EATING)
int state[N];

//turn variable corresonding to each chopstick
//if philosopher i wants to each the turn[i] and turn[LEFT]must be set to i
int turn[N];

//wait on condition
void wait(int i)
{
	x[i].count++;
	if(next_count > 0)
	{
		//signal semaphore next
		sem_post(&next);
	}
	else
	{
		//signal semaphore mutex
		sem_post(&mutex);
	}
	sem_wait(&x[i].sem);
	x[i].count--;
	//	printf("\nX.count -> %d",x.count);
}

//signal on condition
void signal(int i)
{
	if(x[i].count > 0)
	{
		next_count++;
		//signal semaphore x[i].sem
		sem_post(&x[i].sem);
		//wait semeaphore next
		sem_wait(&next);
		next_count--;
	}
```

```
}
void test(int i)
{
      if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING
&& turn[i] == i && turn[LEFT] == i)
      {
            state[i] = EATING;
            //signal on condition
            signal(i);

      }
}

void take_chopsticks(int i)
{
      //wait semaphore mutex
      sem_wait(&mutex);
      state[i] = HUNGRY;
      test(i);
      while(state[i] == HUNGRY)
      {
            //wait on condition
            wait(i);
      }
      if(next_count > 0)
      {
            //signal semaphore next
            sem_post(&next);
      }
      else
      {
            //signal semaphore mutex
            sem_post(&mutex);
      }
}

void put_chopsticks(int i)
{
      //wait semaphore mutex
      sem_wait(&mutex);
      state[i] = THINKING;
      //set turn variable pointing to LEFT and RIGHT philosophers
      turn[i] = RIGHT;
      turn[LEFT] = LEFT;

      test(LEFT);
      test(RIGHT);

      if(next_count > 0)
      {
            //signal semaphore next
            sem_post(&next);
      }
      else
      {
            //signal semaphore mutex
            sem_post(&mutex);
```

```
    }
}

void initialization()
{
    int i;
    sem_init(&mutex,0,1);
    sem_init(&next,0,0);
    for(i = 0;i < N;i++)
    {
        state[i] = THINKING;
        sem_init(&x[i].sem,0,0);
        x[i].count = 0;
        turn[i] = i;
    }
    //setting turn variables such that Philosophers 0,2 or 4 can grab both
chopsticks initially
    turn[1] = 2;
    turn[3] = 4;
    turn[6] = 0;

}
```

Main.cpp .

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>
#include <string.h>

#include "monitor.h"

void *philosopher(void *i)
{
    while(1)
    {
        //variable representing philosopher
        int self = *(int *) i;
        int j,k;
        j = rand();
        j = j % 11;
        printf("\nPhilosopher %d is thinking for %d secs",self,j);
        sleep(j);
        //philosopher take chopsticks
        take_chopsticks(self);
        k = rand();
        k = k % 4;
        printf("\nPhilosopher %d is eating for %d secs",self,k);
        sleep(k);
        //philosopher release chopsticks
        put_chopsticks(self);
    }
}

int main()
{
```

```
    int i, pos[N];
    //one thread corresponding to each philosopher
    pthread_t thread[N];
    pthread_attr_t attr;

    //initilize semaphore and other variables
    initialization();

    pthread_attr_init(&attr);

    for (i = 0; i < N; i++)
    {
        pos[i] = i;
        //create thread corresponding to each philosopher
        pthread_create(&thread[i], NULL,philosopher, (int *) &pos[i]);
    }
    for (i = 0; i < N; i++)
    {
        pthread_join(thread[i], NULL);
    }

    return 0;
}
```