National Institute 0f Technology, Hamirpur

Department Of Computer Science and Engineering
July-Dec 2020

Subject Name: Advanced O.S Lab	Subject Code: CSD – 416
Course: Advanced O.S	Semester: 4th Year, 7th Semester
Submitted By: Student Name: Rishabh Katna Roll No: 17MI552	Submitted To: Dr. Siddhartha Chauhan
	Faculty Signature:

Experiment 1

Aim : To simulate various job scheduling algorithms.

Theory :

Job Scheduling algorithms:

These are the algorithms used by the CPU to schedule various job in efficient ways to achieve maximum performance and CPU utilization.

The Purpose of a Scheduling algorithm

1.Maximum CPU utilization

2.Fare allocation of CPU

3.Maximum throughput

4.Minimum turnaround time

5.Minimum waiting time

6.Minimum response time

Various job scheduling algorithms are used to achieve this purpose. Some of them are :

a. fcfs scheduling (First-Come-First-Serve Algorithm).

b. sjf scheduling (Shortest Job First Scheduling)

c. round robin scheduling

d. priority scheduling

The main initializer code used to setup the program is given below.

Initializer Code:

```cpp
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>
#include <iomanip>

using namespace std;

vector<string> namesOfProcesses = {"P1", "P2", "P3", "P4", "P5"};
vector<int> arrivalTimes = {};
vector<int> burstTimes = {};
vector<int> priorities = {2, 3, 4, 5, 5};


const int INFINITY = 100;
const int TIME_QUANTUM = 2;
```

```cpp
class Job {
public:
    string name;
    bool isServed = false, isPresentInReadyQueue = false;
    int arrivalTime, burstTime, waitingTime, completionTime, turnAroundTime,
originalBurstTime, priority;

    Job(string name, int arrivalTime, int burstTime, int priority){
        this->name = name;
        this->arrivalTime = arrivalTime;
        this->burstTime = burstTime;
        this->waitingTime  = 0;
        this->completionTime = 0;
        this->turnAroundTime = 0;
        this->isServed = false;
        this->originalBurstTime = this->burstTime;
        this->priority = priority;
    }

    void printStats(){
        cout
        << setw(20) << this->name
        << setw(5) << this->arrivalTime
        << setw(5) << this->originalBurstTime
        << setw(5) << this->waitingTime
        << setw(5) << this->completionTime
        << setw(5) << this->turnAroundTime
        << endl;
    }
};

void insertJobsInJobQueue(vector<Job> &jobQueue)
{
    int numberOfJobs = namesOfProcesses.size();

    generateRandomArrivalTimes();

    generateRandomBurstTimes();

    for(int i=0; i<numberOfJobs; i++){
        string nameOfCurJob = namesOfProcesses[i];
        int arrivalTimeOfCurJob = arrivalTimes[i];
        int burstTimeOfCurJob = burstTimes[i];
        int priorityOfCurJob = priorities[i];

        Job newJob(nameOfCurJob, arrivalTimeOfCurJob, burstTimeOfCurJob,
priorityOfCurJob);

        jobQueue.push_back(newJob);
    }


}

void insertJobsInReadyQueue(queue<Job*> &readyQueue, vector<Job> &jobQueue, int
timer){

    // check for all the jobs which are arrived before timer
    // passed in original jobQueue
    for(auto &job : jobQueue){
        // check if job has arrived and has not been isisServed yed
        if(job.arrivalTime <= timer && job.isServed == false &&
```

```cpp
job.isPresentInReadyQueue == false){
            Job* jobptr = &job;
            // push address of job in the ready queue
            jobptr->isPresentInReadyQueue = true;
            readyQueue.push(jobptr);
        }
    }

}



void printFinalInformation(vector<Job> &finishedJobs, int totalWaitingTime, int
totalTurnAroundTime){

    int totalNumberOfJobs = namesOfProcesses.size();
    double averageWaitingTime = (1.0 * totalWaitingTime) / (totalNumberOfJobs);
    double averageTurnAroundTime = (1.0 * totalTurnAroundTime) /
(totalNumberOfJobs);
    cout
    << setw(20) << "Name"
    << setw(5) << "A.T"
    << setw(5) << "B.T"
    << setw(5) << "W.T"
    << setw(5) << "C.T"
    << setw(5) << "T.A.T"
    << endl;



    for(auto job : finishedJobs) job.printStats();
    printf("Average turn around time in the system = %.2lf\n",
averageTurnAroundTime);
    printf("Average waiting time in the system = %.2lf\n", averageWaitingTime);
    cout <<
"................................................................." <<
endl;

}


/////////////////// generate random arrival times ////////////////////////
void generateRandomArrivalTimes()
{
    // change the arrival times
    int numberOfProcesses = namesOfProcesses.size();

    arrivalTimes.resize(numberOfProcesses);


    for(int i=0; i < numberOfProcesses; i++){
        int randomArrivalTime = 0;

        // generate random arrival time
        randomArrivalTime = (rand() % 19);

        arrivalTimes[i] = randomArrivalTime;
    }

}


/////////////////// generate random burst times ////////////////////////////
```

```
void generateRandomBurstTimes()
{
    // change the arrival times
    int numberOfProcesses = namesOfProcesses.size();

    burstTimes.resize(numberOfProcesses);


    for(int i=0; i < numberOfProcesses; i++){
        int randomBurstTime = 0;

        // generate random burst time
        randomBurstTime = (rand() % 10);

        burstTimes[i] = randomBurstTime;
    }

}
```

a. F.C.F.S algorithm :

It is the simplest algorithm to implement. The process with the minimal arrival time will get the CPU first. The lesser the arrival time, the sooner will the process gets the CPU. It is the non-preemptive type of scheduling.

Code :

```
class FcfsScheduler{
    void serveJob(Job &jobToServe, int timer){

        jobToServe.waitingTime = (jobToServe.arrivalTime > timer) ? 0 : (timer
- jobToServe.arrivalTime);

        jobToServe.completionTime = jobToServe.arrivalTime +
jobToServe.waitingTime + jobToServe.burstTime;

        jobToServe.turnAroundTime = jobToServe.completionTime -
jobToServe.arrivalTime;




    }
    public:
        void scheduleJobs(vector<Job> jobQueue){
            int timer = 0;

    // stores the finished jobs
            vector<Job> finishedJobs;

            // calculate the total waiting time of the processes
            int totalWaitingTime = 0, totalTurnAroundTime = 0;

            // process jobs while there are jobs available in ready queue
            // delay of counter time
            while(!jobQueue.empty()){
                // get the first job from the readyQueue
```

```
                    Job jobToServe = jobQueue[0]; jobQueue.erase(jobQueue.begin());

                    // get the time at which job will leave the system
                    serveJob(jobToServe, timer);

                    totalTurnAroundTime += jobToServe.turnAroundTime;


                    totalWaitingTime += jobToServe.waitingTime;

                    // book a new time slot for the new job
                    timer = jobToServe.completionTime;

                    // since the job is completed, hence we can add the job in the
finishedJobs array
                    finishedJobs.push_back(jobToServe);
              }

          printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);
       }



};
```

b. SJF scheduling

The job with the shortest burst time will get the CPU first. The lesser the burst time, the sooner will the process get the CPU. It is the non-preemptive type of scheduling.

```cpp
class SjfScheduler {
    protected:
        const int inf = 100;
        vector<Job> finishedJobs = {};
        int totalWaitingTime = 0, totalTurnAroundTime = 0;
        void insertJobsInReadyQueue(vector<Job*> &readyQueue, vector<Job>
&jobQueue, int timer){
            for(auto &job : jobQueue){
                // check if job has arrived and has not been served yed
                if(job.arrivalTime <= timer && job.isServed == false){
                    Job* jobptr = &job;
                    // push address of job in the ready queue
                    readyQueue.push_back(jobptr);
                }
            }

        }

        static bool onBurstTime(Job* j1, Job* j2){
            return j1->burstTime <= j2->burstTime;
        }
};
```

i. SJF Preemptive scheduling

It is the preemptive form of SJF. In this algorithm, the OS schedules the Job according to the remaining time of the execution.

```cpp
class SjfPreemptiveScheduler : public SjfScheduler{
    void serveJob(Job* jobToServe, int timer){
        jobToServe->isServed = true;
        jobToServe->completionTime = timer + 1;
        jobToServe->turnAroundTime = jobToServe->completionTime - jobToServe->arrivalTime;
        jobToServe->waitingTime = jobToServe->turnAroundTime - jobToServe->originalBurstTime;
        // jobToServe->responseTime = jobToServe->waitingTime;
    }
    public:
        void scheduleJobs(vector<Job> jobQueue){
            int timer = 0;
            while(timer < inf){
                // address of all job objects
                vector<Job*> readyQueue;

                // this code is wrong
                insertJobsInReadyQueue(readyQueue, jobQueue, timer);

                if(readyQueue.empty()){
                    timer++;
                    continue;
                }

                sort(readyQueue.begin(), readyQueue.end(), onBurstTime);

                Job* jobToServe = readyQueue[0]; // address of the job to serve

                jobToServe->burstTime -= 1;

                if(jobToServe->burstTime == 0) {
                    serveJob(jobToServe, timer);
                    totalWaitingTime += jobToServe->waitingTime;
                    totalTurnAroundTime += jobToServe->turnAroundTime;


                    finishedJobs.push_back(*jobToServe);
                }

                timer ++;
            }
            printFinalInformation(finishedJobs, totalWaitingTime, totalTurnAroundTime);
        }
};
```

ii. SJF Non-Preemptive scheduling

he job with the shortest burst time will get the CPU first. The lesser the burst time, the sooner will the process get the CPU. It is the non-preemptive type of scheduling.

```cpp
class SjfNonPreemptiveScheduler : public SjfScheduler{
    void serveJob(Job* jobToServe, int timer)
    {
        jobToServe->isServed = true;

        jobToServe->waitingTime = timer - jobToServe->arrivalTime;

        jobToServe->completionTime = timer + jobToServe->burstTime;

        jobToServe->turnAroundTime = jobToServe->completionTime - jobToServe->arrivalTime;
    }
    public:
        void scheduleJobs(vector<Job> jobQueue){
            int timer = 0;
            while(timer < inf){
                // address of all job objects
                vector<Job*> readyQueue;

                insertJobsInReadyQueue(readyQueue, jobQueue, timer);

                // if no job has arrived, then don't do anything
                if(readyQueue.empty()){
                    timer++;
                    continue;
                }

                sort(readyQueue.begin(), readyQueue.end(), onBurstTime);

                Job* jobToServe = readyQueue[0]; // address of the job to serve

                serveJob(jobToServe, timer);

                timer = jobToServe->completionTime;


                totalWaitingTime += jobToServe->waitingTime;

                totalTurnAroundTime += jobToServe->turnAroundTime;

                finishedJobs.push_back(*jobToServe);

            }

            printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);
        }



};
```

c. Prioriy Scheduler

In this algorithm, the priority will be assigned to each of the processes. The higher the priority, the sooner will the process get the CPU. If the priority of the two processes is same then they will be scheduled according to their arrival time.

```cpp
class ComparyPriorities
{
    /* data */
    public:
        bool operator () (Job* j1, Job* j2){
            return j1->priority < j2->priority;
        }
};



class PriorityScheduler {
    vector<Job> finishedJobs = {};

    void insertJobsInReadyQueue(priority_queue<Job*, vector<Job*>,
ComparyPriorities> &readyQueue, vector<Job> &jobQueue, int timer){

    // check for all the jobs which are arrived before timer
    // passed in original jobQueue
        for(auto &job : jobQueue){
            // check if job has arrived and has not been isisServed yed
            if(job.arrivalTime <= timer && job.isServed == false &&
job.isPresentInReadyQueue == false){
                Job* jobptr = &job;
                // push address of job in the ready queue
                jobptr->isPresentInReadyQueue = true;
                readyQueue.push(jobptr);
            }
        }

    }

    void serveJob(Job* jobToServe, int timer){

        jobToServe->isServed = true;

        jobToServe->completionTime = timer;

        jobToServe->turnAroundTime = jobToServe->completionTime - jobToServe->arrivalTime;

        jobToServe->waitingTime = jobToServe->turnAroundTime - jobToServe->originalBurstTime;
    }


    public:

        void scheduleJobs(vector<Job> jobQueue){

            int timer = 0;

            priority_queue<Job*, vector<Job*>, ComparyPriorities> readyQueue;
// when something is entered, it gets to its appr location

            int totalWaitingTime = 0, totalTurnAroundTime = 0;
```

```cpp
            while(timer < INFINITY){

                // insert jobs in ready queue
                insertJobsInReadyQueue(readyQueue, jobQueue, timer);


                if(readyQueue.empty()) {
                    timer++;
                    continue;
                }



                // get the best job
                Job* jobToServe = readyQueue.top();


                jobToServe->burstTime--;


                timer++;



                if(jobToServe->burstTime == 0){
                    // done processing .... no need for more cpu allocation
                    // out of our readyQueue
                    readyQueue.pop();

                    serveJob(jobToServe, timer);

                    totalWaitingTime += jobToServe->waitingTime;

                    totalTurnAroundTime += jobToServe->turnAroundTime;

                    finishedJobs.push_back(*jobToServe);
                }

            }


        printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);
        }


};
```

d. Round-Robin Scheduler :

In the Round Robin scheduling algorithm, the OS defines a time quantum (slice). All the processes will get executed in the cyclic way. Each of the process will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a preemptive type of scheduling.

```cpp
class RoundRobinScheduler {
    static bool onArrivalTime(Job j1, Job j2){
        return j1.arrivalTime < j2.arrivalTime;
    }



    void serveJob(Job* jobToServe, int timer){
        jobToServe->isServed = true;
            // mark isisServed as true
        // these are calculated accurately
        jobToServe->completionTime = timer;
    // calculate waiting time, execution time and tat
        jobToServe->turnAroundTime = jobToServe->completionTime - jobToServe->arrivalTime;

        jobToServe->waitingTime = jobToServe->turnAroundTime - jobToServe->originalBurstTime;
    }
public:



    void scheduleJobs(vector<Job> jobQueue) {

        sort(jobQueue.begin(), jobQueue.end(), onArrivalTime);


        vector<Job> finishedJobs = {};
        int timer = 0;

        // stores the addresses of the jobs

        queue<Job*> readyQueue;

        // store the job processes in previous step
        Job* prevJobProcessed = NULL;


        int totalWaitingTime = 0;
        int totalTurnAroundTime = 0;

        while(timer < INFINITY){

            insertJobsInReadyQueue(readyQueue, jobQueue, timer);

            // insert the prevJobProcesses
            if(prevJobProcessed != NULL) readyQueue.push(prevJobProcessed);

            prevJobProcessed = NULL;

            if(readyQueue.empty()){
                timer += 1;
                continue;
            }
```

```cpp
            /// if some jobs have arrived

            // take first job of the queue
            Job* jobToServe = readyQueue.front(); readyQueue.pop();
            // process the job for interval

            int processingInterval = min(TIME_QUANTUM, jobToServe->burstTime);

            jobToServe->burstTime -= processingInterval;

            timer += processingInterval;

            // check if job has done its processing

            // job will not be pushed in ready queue again
            if(jobToServe->burstTime != 0){
                prevJobProcessed = jobToServe;
                continue;
            }


            // burstTime == 0
            serveJob(jobToServe, timer);

            // cout << jobToServe->name << ' ' << jobToServe->burstTime <<
endl;
            totalWaitingTime += jobToServe->waitingTime;

            totalTurnAroundTime += jobToServe->turnAroundTime;

            finishedJobs.push_back(*jobToServe);

        }

        printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);

    }



};
```

Output :

driver Code :

```cpp
int main()
{
    vector<Job> jobQueue = {};

    insertJobsInJobQueue(jobQueue);

    // start the processing

    cout << "Round Robin
Scheduler ........................................................." << endl;
    RoundRobinScheduler roundRobinScheduler;
    roundRobinScheduler.scheduleJobs(jobQueue);
    cout <<
"............................................................................" <<
endl;

    cout << "Sjf Preemptive
Scheduler ......................................................" << endl;
    SjfPreemptiveScheduler sjfPreemptiveScheduler;
    sjfPreemptiveScheduler.scheduleJobs(jobQueue);
    cout <<
"............................................................................." <<
endl;

    cout << "Sjf NonPreemptive
Scheduler ..................................................." << endl;
    SjfNonPreemptiveScheduler sjfNonPreemptiveScheduler;
    sjfNonPreemptiveScheduler.scheduleJobs(jobQueue);

    cout << "fcfs
scheduler ........................................................." << endl;
    FcfsScheduler fcfsScheduler;
    fcfsScheduler.scheduleJobs(jobQueue);
    cout <<
"............................................................................." <<
endl;

    cout << "Priority
Scheduler ...................................................." << endl;
    PriorityScheduler priorityScheduler;
    priorityScheduler.scheduleJobs(jobQueue);


}
```

Output :


```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                          1: bash          ∨   +  ⊡  🗑  ∨  ×

rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/course_work/lab_practical/advanced_os/practical1$ g++ 17MI552_job_scheduling.cpp
rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/course_work/lab_practical/advanced_os/practical1$ ./a.out
Round Robin Scheduler ...............................................
            Name  A.T  B.T  W.T  C.TT.A.T
            P3    3    2    1    6     3
            P5    10   1    2    13    3
            P1    2    5    7    14    12
            P4    3    9    11   23    20
            P2    15   6    4    25    10
Average turn around time in the system = 9.60
Average waiting time in the system = 5.00
...............................................................
...............................................................
Sjf Preemptive Scheduler ...............................................
            Name  A.T  B.T  W.T  C.TT.A.T
            P3    3    2    0    5     2
            P1    2    5    2    9     7
            P5    10   1    0    11    1
            P4    3    9    7    19    16
            P2    15   6    4    25    10
Average turn around time in the system = 7.20
Average waiting time in the system = 2.60
...............................................................
...............................................................
Sjf NonPreemptive Scheduler ...............................................
            Name  A.T  B.T  W.T  C.TT.A.T
            P1    2    5    0    7     5
            P3    3    2    4    9     6
            P4    3    9    6    18    15
            P5    10   1    8    19    9
            P2    15   6    4    25    10
Average turn around time in the system = 9.00
Average waiting time in the system = 4.40
...............................................................
```

```
...............................................................
fcfs scheduler ...............................................
            Name  A.T  B.T  W.T  C.TT.A.T
            P1    2    5    0    7     5
            P2    15   6    0    21    6
            P3    3    2    18   23    20
            P4    3    9    20   32    29
            P5    10   1    22   33    23
Average turn around time in the system = 16.60
Average waiting time in the system = 12.00
...............................................................
...............................................................
Priority Scheduler ...............................................
            Name  A.T  B.T  W.T  C.TT.A.T
            P4    3    9    0    12    9
            P5    10   1    2    13    3
            P3    3    2    10   15    12
            P2    15   6    0    21    6
            P1    2    5    18   25    23
Average turn around time in the system = 10.60
Average waiting time in the system = 6.00
...............................................................
```

Experiment 2

Aim : To simulate FIFO and LRU page replacement algorithm.

Theory:

Page Replacement algorithms:

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

a. FIFO (First-In First-Out Algorithm):

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

b. LRU(Least Recently Used Algorithm):

In this algorithm the new page will be replaced which is least recently used.

Code:

nitializer code:

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iomanip>
#include <unordered_map>
#include <climits>


using namespace std;


void printFrameStats(vector<int> frame){

    for(auto fr : frame){

        cout << setw(5) << fr << ' ';

    }

    cout << endl;
}



void printFinalStats(int numMisses, vector<int> referenceString, string
nameScheduler){

    int numHits = referenceString.size() - numMisses;

    string twentyDots(20, '.');

    cout << twentyDots << nameScheduler <<" Scheduler " << twentyDots << endl;

    cout <<"Total misses in cache \t = \t"  << numMisses << endl;

    cout <<"Total hits  in  cache \t = \t"  << numHits << endl;

    cout <<"Hit ratio of the system  = \t" << (1.0 * numHits /

referenceString.size()) << endl;


    cout << endl;

    cout << endl;




}
```

FIFO

```cpp
class FIFO {

    unordered_map<int, bool> isPresentInCache;

    int numMisses = 0, numFrames;

    vector<int> frame;

    public :
        int pageReplacement(vector<int> referenceString, int numFrames){

            frame.resize(numFrames, -1);

            int insertIndex = 0;

            for(int page : referenceString){
                // print current situation of frames
                printFrameStats(frame);

                if(!isPresentInCache[page]){

                    if(frame[insertIndex] != -1){
                        // replace the existing from cache
                        isPresentInCache[frame[insertIndex]] = false;
                    }

                    // insert into cache
                    frame[insertIndex] = page;


                    isPresentInCache[page] = true;

                    // remove the entry from cache

                    insertIndex = (insertIndex + 1) % numFrames;

                    numMisses++;


                }

            }

            return numMisses;

        }


};
```

LRU

```cpp
class LRU {

    unordered_map<int, int> indexOf, lastOccurenceOf;

    vector<int> frame;

    int numFrames = 0, numMisses = 0;

    int getAppropriateIndex(){

        for(int i=0; i<numFrames; i++){

            if(frame[i] == -1) return i;

        }

        // replace the frame
        int locationLeastUsedPage = INT_MAX;

        int leastUsedPage;

        for(auto occurence : lastOccurenceOf){

            if(occurence.second < locationLeastUsedPage) {

                leastUsedPage = occurence.first;

                locationLeastUsedPage = occurence.second;

            }

        }


        return indexOf[leastUsedPage] - 1;

    }


    public:

        int pageReplacement(vector<int> referenceString, int numFrames){
            this->numFrames = numFrames;

            frame.resize(numFrames, -1);

            for(int i = 0; i < referenceString.size(); i++){

                int currPage = referenceString[i];

                printFrameStats(frame);


                if(indexOf[currPage] > 0){ // currentPage is present in cache
                    lastOccurenceOf[currPage] = i;
                    continue;
                }

                // cout << "This part of code running ..." << endl;
                // 2 =>
```

```
                // replacement needs to carry out
                int insertIndex = getAppropriateIndex();
                // cout << insertIndex << endl;
                //

                // galat ho ra hai
                indexOf[frame[insertIndex]] = -1;
                lastOccurenceOf[frame[insertIndex]] = INT_MAX;

                // insert new page
                frame[insertIndex] = currPage;
                // insertIndex ==> 0
                indexOf[currPage] = (insertIndex + 1);
                lastOccurenceOf[currPage] = i;
                numMisses++;
            }

            return numMisses;
        }


};
```

DRIVER Program

```
int main()
{
    vector<int> referenceString = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3};

    int numFrames = 4;

    vector<int> frame(numFrames, -1);

    FIFO fifo;
    LRU lru;

    int numMissesFIFO = fifo.pageReplacement(referenceString, numFrames);

    int numMissesLRU = lru.pageReplacement(referenceString, numFrames);

    printFinalStats(numMissesFIFO, referenceString, "FIFO");

    printFinalStats(numMissesLRU, referenceString, "LRU");

    return 0;

}
```

Output:
FIFO Scheduler:

```
PROBLEMS  5    OUTPUT   DEBUG CONSOLE   TERMINAL          1: bash

d_os/practical2$ g++ pageReplacement.cpp
rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/co
d_os/practical2$ ./a.out
    -1    -1    -1    -1
     7    -1    -1    -1
     7     0    -1    -1
     7     0     1    -1
     7     0     1     2
     7     0     1     2
     3     0     1     2
     3     0     1     2
     3     4     1     2
     3     4     1     2
     3     4     1     2
     3     4     0     2
     3     4     0     2
     3     4     0     2
...................FIFO Scheduler ....................
Total misses in cache    =      7
Total hits  in  cache    =      7
Hit ratio of the system  =      0.5
```

LRU Scheduler:

```
rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/cou
d_os/practical2$ ./a.out
    -1      -1      -1      -1
     7      -1      -1      -1
     7       0      -1      -1
     7       0       1      -1
     7       0       1       2
     7       0       1       2
     3       0       1       2
     3       0       1       2
     3       0       4       2
     3       0       4       2
     3       0       4       2
     3       0       4       2
     3       0       4       2
     3       0       4       2
...................LRU Scheduler ....................
Total misses in cache    =        6
Total hits  in  cache    =        8
Hit ratio of the system  =        0.571429
```

Experiment 3

Aim : To Implement a distributed file server.

Theory:

DFS:

A Distributed File System (DFS) as the name suggests, is a file system that is distributed on multiple file servers or multiple locations. It allows programs to access or store isolated files as they do with the local ones, allowing programmers to access files from any network or computer.

The main purpose of the Distributed File System (DFS) is to allows users of physically distributed systems to share their data and resources by using a Common File System. A collection of workstations and mainframes connected by a Local Area Network (LAN) is a configuration on Distributed File System. A DFS is executed as a part of the operating system. In DFS, a namespace is created and this process is transparent for the clients.

DFS has two components:

- •Location Transparency: Location Transparency achieves through the namespace component.
- •Redundancy: Redundancy is done through a file replication component.

Our DFS is divided into two modules:

a. Client module

b. Server module

a. Client module:

Client module runs on the client machine. It is responsible for the communication between client and the server. Our client module is divided into following components.

    i. Cache: Cache module is further divided into two parts:

        .File storage:

            This is the location where the cached files are stored in our system.

        .Cache record:

            This is a file which keeps record of the files currently stored in the cache and the timestamps of storage.

    Ii. Client side code :

        The program which sends the request to the server and   caches the recieved file from the server.

b. Server module:

Server module is further divided into two parts :

i. File storage:

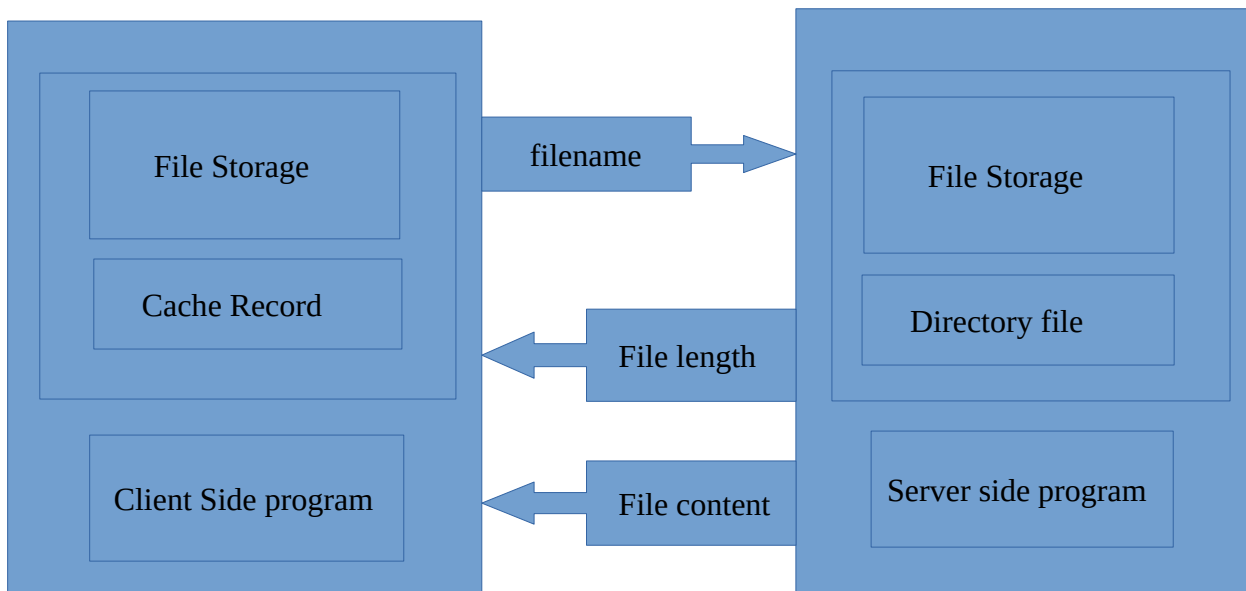This module contains all the files of the server.

ii. Directory:

This file is used to keep track of the last update timestamps of the files stored in the server storage. This is used to prevent the problem of cache coherence.

Iii. Server side code :

The program which automates the communication between client and the server. It accepts a filename from the server and return the conntents of the file to the client. The client caches the file and then stores it in its own cache.

DIAGRAM:

File Storage	filename →	File Storage
Cache Record	← File length	Directory file
Client Side program	← File content	Server side program

CLIENT MODULE

Code:

Client Module:

```cpp
#include <fstream>
#include <iostream>
#include <string>

#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <time.h>

#include <netinet/in.h>
#include <unistd.h>

using namespace std;

const int PORT = 5400;

/*
    @name   --> print_str
    @params --> string str
    @return --> void
    @desc   --> prints the string str on the console
*/
void print_str(string str)
{
    for (char ch : str)
        printf("%c", ch);
    printf("\n");
}

/*
    @name   --> is_in_cache
    @params --> string file_name
    @return --> true/false
    @desc   --> looks in the cache_record.txt file to check whether the file is
in the cache
*/
bool is_in_cache(char *file_name)
{
    // ifstream fin("./cache/cache_record.txt", "r");
    ifstream fin("./cache/cache_record.txt");

    string filename_str(file_name);

    while (!fin.eof())
    {
        string cur_filename;
        time_t timestamp;
        fin >> cur_filename >> timestamp;

        if (cur_filename == file_name)
        {

            cout << "--------------------------------------" << endl;

            cout << "file " << filename_str << " found in cache" << endl;
            cout << "--------------------------------------" << endl;
```

```cpp
            return true;
        }
    }

    return false;

    // return false;
}

/*
    @name   --> extract_dirname
    @params --> string path
    @return --> string dirname
    @desc   --> extracts the name of the dir from the given path
*/
string extract_dirname(const string &path)
{
    string dirname = path;

    int loc_last_slash = 0;
    for (int i = 0; i < dirname.size(); i++)
    {
        if (dirname[i] == '/')
            loc_last_slash = i;
    }
    dirname = dirname.substr(0, loc_last_slash);

    return dirname;
}

/*
    @name   --> is_path_exits
    @params --> string dirname
    @return --> true/false
    @desc   --> Checks whether the path exists.. if it does, return true else
false
*/
bool is_path_exits(string path)
{
    struct stat buffer;
    return (stat(path.c_str(), &buffer) == 0);
}

/*
    @name   -> cache file
    @params -> string filename, string filecontent
    @return -> void
    @desc   -> create a new file and writes the filecontent in the file in
cache folder
*/
void cache_file(string filename, const string &filecontent)
{
    // extract dir name
    string dirname = extract_dirname(filename);

    // check if the folder exists
    // update dirname so that the file is stored in the cache
    string dirname_cache = "./cache" + dirname.substr(1, dirname.size());

    if (!is_path_exits(dirname_cache))
    {
        // if not then create the folder
        int status = mkdir(dirname_cache.c_str(), 0777);
```

```cpp
        if (status != -1)
        {
            cout << "---------------------------------------" << endl;
            cout << "Successfully Created a directory" << endl;
            cout << "---------------------------------------" << endl;
        }

        else
        {
            cout << "---------------------------------------" << endl;

            printf("Couldn't create a new directory.. please try again\n");
            cout << "---------------------------------------" << endl;
        }
    }

    // create a file in the folder
    if (is_path_exits(dirname_cache))
    {
        string filename_cache = "./cache" + filename.substr(1,
filename.size()));
        // create a new file
        ofstream main_file(filename_cache, ios::out);

        // open the filename in the cache
        if (!main_file.is_open())
        {
            cout << "---------------------------------------" << endl;

            printf("Couldn't open the file... Error occured");

            cout << "---------------------------------------" << endl;

            return;
        }

        // start writing filecontent to file
        main_file << filecontent << endl;

        // create an entry in cache
        ofstream cache_record_file("./cache/cache_record.txt", ios::app);

        /*
        @TODO -> make changes in the construction of a cache record
        @Fields -> filename timestamp
        */
        time_t cache_timestamp = time(NULL);

        cache_record_file << filename << ' ' << cache_timestamp << endl;

        main_file.close();

        cache_record_file.close();

        return;
    }

    printf("Couldn't find the folder to store the file\n");
}

/*
    @name   --> recieve_timestamp_from_server
    @params --> int client_socket, string filename
```

```cpp
    @return --> timestamp when the file was last updated on the server
    @desc    --> gets the last update time stamp of a file and returns it to the
main program
*/
time_t recieve_timestamp_from_server(int client_socket, const char *filename)
{
    // cout << "Func called" << endl;

    time_t last_update_timestamp = time(NULL);
    // header = '1'
    int filename_len = string(filename).size();

    int datagram_len = filename_len + 1;

    // construct header
    char header = '1';

    // datagram
    string datagram = header + string(filename);

    // final datagram
    const char *timestamp_datagram = datagram.c_str();

    printf("%s\n", timestamp_datagram);

    // send datagram
    if (send(client_socket, timestamp_datagram, datagram_len, 0) < 0)
    {
        cout << "-------------------------------------" << endl;

        cerr << "Couldn't send the timestamp datagram" << endl;

        cout << "-------------------------------------" << endl;

        return time(NULL);
    }

    cout << "sent timestamp_datagram" << endl;

    // recv ts
    recv(client_socket, &last_update_timestamp, sizeof(last_update_timestamp),
0);

    // recieve time_t

    return last_update_timestamp;
}

/*
    @name    --> recieve_file
    @params --> int client_socket, int file_size, int num_blocks
    @return --> string file_content
    @desc    --> recieves num_blocks from server and store the blocks in a
string...
*/
string recieve_file(int client_socket, int file_size, int num_blocks)
{
    string file_content;
    int max_block_size = 1024;
    for (int i = 0; i < num_blocks; i++)
    {
        // fix bug in this code
```

```cpp
        int cur_block_size = min(max_block_size, (file_size - (i *
max_block_size)));

        char buffer[cur_block_size];

        recv(client_socket, &buffer, sizeof(buffer), 0);

        // printf("%s\n", buffer);

        string buffer_str(buffer);
        file_content += buffer_str;

        // printf("Block number %d recieved by the client\n", i);

        // printf("%s\n", buffer);
    }

    return file_content;
}
/*
    @name    --> get_file_cache_ts
    @params --> string filename
    @return --> timestamp when the file was cached on our client module
    @desc    --> searches for filename in client record, notes the last record
timestamp and returns it
*/
time_t get_file_cache_ts(const char *filename)
{

    // look into cache_dir
    string filename_str(filename);

    ifstream fin_cache("./cache/cache_record.txt");

    time_t cache_file_ts;
    while (!fin_cache.eof())
    {

        string cur_filename;
        time_t cur_ts;

        fin_cache >> cur_filename >> cur_ts;

        if (cur_filename == filename_str)
        {
            cout << "--------------------------------------" << endl;

            cout << cur_filename << '\n'
                 << cur_ts << endl;

            cout << "--------------------------------------" << endl;

            return cur_ts;
        }
    }

    return cache_file_ts;

    // if filename == cur_filename

    // return ts
}
```

```cpp
/*
    @name   --> update_cache
    @params --> string filename, int new_ts, int old_ts
    @return --> void
    @desc   --> updates the timestamp of the filename in the cache record
(replaces old_ts with new_ts)
*/

void update_cache(const char *filename, time_t new_ts, time_t old_ts)
{
    // udate the cache so its holds the latest ts of updation
    ifstream fin("./cache/cache_record.txt");

    string content = "";

    // read content of file in better way
    while (!fin.eof())
    {

        string fn, ts;

        fin >> fn >> ts;

        content += fn + ' ' + ts;
    }

    // find old
    string to_remove = string(filename) + ' ' + to_string(old_ts);

    cout << "............ Remove string ....................." << endl;
    cout << to_remove << endl;
    cout << "............................................." << endl;

    string to_insert = string(filename) + ' ' + to_string(new_ts);

    cout << "................ Insert string ................." << endl;
    cout << to_insert << endl;
    cout << "............................................." << endl;
    // remove string from content
    int pos = content.find(to_remove);

    if (pos != string::npos)
    {

        cout << "------ record found at string -----------" << endl;

        cout << pos << endl;

        cout << "-------------------------------------------" << endl;

        // to_remove ... not being removed
        content.erase(pos, to_remove.size());

        content += to_insert;
    }
    cout << "File content of cache record" << endl;
    cout << "..............................." << endl;
    cout << content << endl;
    cout << "..............................." << endl;

    fin.close();

    ofstream fout("./cache/cache_record.txt");
```

```cpp
    fout << content << endl;

    fout.close();
}
/*
    @name   --> get_file_from_server
    @params --> int client_socket, string filename
    @return --> string filecontent
    @desc   --> sends the filename to the server, recieves the filecontent.
Then it returns the filecontent
*/
string get_file_from_server(int client_socket, char filename[])
{
    printf("Filename sending to the server = %s\n", filename);

    //////////////////////////////////////////////////////
    /*
    @TODO --> add a header to the filename
            ==> header shows the type of request being sent to the server
    */
    int filename_len = string(filename).size();

    int datagram_len = filename_len + 1;
    char header = '0';
    // send the whole header

    // construct header
    string datagram = header + string(filename);

    const char *filename_datagram = datagram.c_str();

    printf("%s\n", filename_datagram);

    send(client_socket, filename_datagram, datagram_len, 0);

    cout << "-----------------------------------------" << endl;

    cout << "Filename datagram sent to the server" << string(filename_datagram)
<< endl;
    //////////////////////////////////////////////////////
    cout << "-------------------------------------" << endl;

    // // recieve length of file
    int file_size;
    recv(client_socket, &file_size, sizeof(file_size), 0);

    // // // revieve file from server
    cout << file_size << endl;

    int num_blocks;

    recv(client_socket, &num_blocks, sizeof(num_blocks), 0);

    cout << "-------------------------------------" << endl;

    printf("Num blocks to recieve = %d\n", num_blocks);
    cout << "-------------------------------------" << endl;

    // // // send acknowledgement
    bool is_reached = true;
    send(client_socket, &is_reached, sizeof(is_reached), 0);

    // // recieve blocks
```

```cpp
    string file_content = recieve_file(client_socket, file_size, num_blocks);
    // cache file after recieving

    return file_content;
}

/*
Driver Program
*/
int main()
{
    // ifstream fin("./cache/cache_record.txt", "r");
    char filename[] = "./folder1/file_to_recieve.txt";

    int client_socket = socket(AF_INET, SOCK_STREAM, 0);

    // server object
    sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(PORT);
    server_address.sin_addr.s_addr = INADDR_ANY;

    if (!is_in_cache(filename))
    {
        // connect to the client
        connect(client_socket, (sockaddr *)&server_address,
sizeof(server_address));

        // send name of file to server
        string file_content = get_file_from_server(client_socket, filename);

        cache_file(string(filename), file_content);

        fflush(stdout);

        close(client_socket);
    }
    else
    {
        connect(client_socket, (sockaddr *)&server_address,
sizeof(server_address));

        cout << "Require timestamp" << endl;

        time_t last_update_ts = recieve_timestamp_from_server(client_socket,
filename);

        time_t client_cache_ts = get_file_cache_ts(filename);

        if (difftime(last_update_ts, client_cache_ts) <= 0)
        {
            cout << "---------------------------------------" << endl;

            cout << "File up to date" << endl;

            cout << "---------------------------------------" << endl;
        }
        else
        {
            cout << "---------------------------------------" << endl;
```

```cpp
            cout << "File updated at server " << endl;

            cout << "--------------------------------------" << endl;
            // close(client_socket);
            // make sure that file is fetched in this case
            connect(client_socket, (sockaddr *)&server_address,
sizeof(server_address));

            string file_content = get_file_from_server(client_socket,
filename);

            // this func not working
            // make sure the timestamps are rig
            update_cache(filename, last_update_ts, client_cache_ts);

            close(client_socket);
            // update_cache(filename, )
        }
    }
    return 0;
}
```

OUTPUT:

```
--------------------------------------
file ./folder1/file_to_recieve.txt found in cache
--------------------------------------
Require timestamp
1./folder1/file_to_recieve.txt
sent timestamp_datagram
--------------------------------------
./folder1/file_to_recieve.txt
16046464989
--------------------------------------
--------------------------------------
File updated at server
--------------------------------------
Filename sending to the server = ./folder1/file_to_recieve.txt
0./folder1/file_to_recieve.txt
--------------------------------------
Filename datagram sent to the server0./folder1/file_to_recieve.txt
--------------------------------------
758477
--------------------------------------
Num blocks to recieve = 741
--------------------------------------
............ Remove string ....................
./folder1/file_to_recieve.txt 16046464989
.............................................
................ Insert string ..................
./folder1/file_to_recieve.txt 160464649898
.............................................
------ record found at string -----------
0
-----------------------------------------
File content of cache record
..............................
 ./folder1/file_to_recieve.txt 160464649898
..............................
```

Server module:

```cpp
#include <string>
#include <fstream>
#include <iostream>
#include <string.h>

#include <math.h>

#include <sys/socket.h>
#include <sys/types.h>

#include <netinet/in.h>
#include <unistd.h>

using namespace std;

#define PORT 5400

/*
    @name   --> filesize
    @params --> string filename
    @return --> size of the file having name filename
    @desc   --> gets the size of the file and returns the size of the file
*/
ifstream::pos_type filesize(const char *filename)
{
    std::ifstream in(filename, std::ifstream::ate | std::ifstream::binary);
    return in.tellg();
}

/*
    @name   --> get_length
    @params --> string filename
    @return --> length of filename
    @desc   --> uses filesize() func to get the size of the file and return to
the user
*/
int get_length(const char *filename)
{

    ifstream fin(filename);

    if (!fin.is_open())
    {
        return -1;
    }

    return filesize(filename);
}

/*
    @name   --> send_blocks
    @params --> int client_socket, string filename, int filesize
    @return --> void
    @desc   --> breaks the file into blocks and sends them to the server
*/
void send_blocks(int client_socket, string content, int filesize, int
num_blocks)
{
    int max_block_size = 1024;
    for (int i = 0; i < num_blocks; i++)
    {
```

```cpp
        // this code has bug
        // fix this code

        int start_ptr = i * max_block_size;
        int end_ptr = min((i + 1) * max_block_size, filesize);

        int cur_block_size = (end_ptr - start_ptr);

        // create a buffer to send
        char buffer[cur_block_size];

        // copy file content to buffer.
        for (int i = start_ptr; i < end_ptr; i++)
        {
            buffer[i - start_ptr] = content[i];
        }
        // send buffer
        send(client_socket, &buffer, sizeof(buffer), 0);

        // cout << "Block " << i << " sent to the client" << endl;
    }
}

/*
    @name   --> send_file
    @params --> int client_socket, string filename, int filesize
    @return --> void
    @desc   --> breaks the file into blocks and sends each block to the client
connected via client socket
*/
void send_file(int client_socket, const char *filename, int filesize)
{
    int max_block_size = 1024;

    int num_blocks = ceil(1.0 * filesize / max_block_size);

    bool is_recieved = false;

    // send num_blocks to client
    while (!is_recieved)
    {

        send(client_socket, &num_blocks, sizeof(num_blocks), 0);

        // recv ack
        recv(client_socket, &is_recieved, sizeof(is_recieved), 0);
    }
    cout << "-------------------------------------" << endl;

    printf("Acknowledgement recieved... sending file..\n");
    cout << "-------------------------------------" << endl;

    ifstream ifs(filename);
    string content((std::istreambuf_iterator<char>(ifs)),
                   (std::istreambuf_iterator<char>()));

    // send file to client in blocks
    send_blocks(client_socket, content, filesize, num_blocks);
    cout << "-------------------------------------" << endl;

    cout << "file send to client ..... " << endl;
    cout << "Closing connection ...... " << endl;
    cout << "-------------------------------------" << endl;
```

```cpp
}
/*
    @name   --> get_file_ts
    @params --> char *filename
    @return --> last update timestamp of filename
    @desc   --> looks up for the last update timestamp of the filename and
returns it
*/
time_t get_file_ts(const char *filename)
{
    string filename_str = string(filename);

    // open directory.txt
    ifstream fin_dir("./directory.txt");

    time_t final_update_ts;
    while (!fin_dir.eof())
    {

        string cur_filename;
        time_t cur_ts;

        fin_dir >> cur_filename >> cur_ts;

        cout << "----------------------------------------" << endl;

        cout << "getting the last update timestamp of the file " << endl;
        cout << cur_ts << endl;

        cout << "----------------------------------------" << endl;

        if (cur_filename == filename_str)
        {
            final_update_ts = cur_ts;
            break;
        }
    }

    cout << "----------------------------------------" << endl;

    cout << "Printing the final ts of the filename" << endl;
    cout << final_update_ts << endl;

    cout << "----------------------------------------" << endl;
    return final_update_ts;
}

/*
    @name   --> serve_client
    @params --> int client_socket
    @return --> void
    @desc   --> recieves the client socket and serves the requests of the
client
*/
void serve_client(int client_socket)
{

    char datagram[1024];

    recv(client_socket, &datagram, sizeof(datagram), 0);

    // extract filename and header from datagram
    char header = datagram[0];
```

```cpp
    string payload = string(datagram).substr(1, string(datagram).size());

    const char *filename = payload.c_str();

    // if operation is 0, then send file
    if (header == '0')
    {
        int file_length = get_length(filename);

        cout << "--------------------------------------" << endl;

        cout << "GOT FILE LENGTH = " << file_length << endl;
        cout << "--------------------------------------" << endl;

        send(client_socket, &file_length, sizeof(file_length), 0);

        cout << file_length << endl;

        send_file(client_socket, filename, file_length);
    }
    // else send the timestamp details of the file
    else if (header == '1')
    {
        cout << "--------------------------------------" << endl;

        cout << "Recieved request for timestamp" << endl;
        // get timestamp of filename
        time_t file_write_ts = get_file_ts(filename);

        // send timestamp of filename
        send(client_socket, &file_write_ts, sizeof(file_write_ts), 0);
    }

    // // close connection
    // close(client_socket);
}

/*
    Driver program
*/
int main()
{
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in server_address;

    server_address.sin_port = htons(PORT);
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;

    // bind
    bind(server_socket, (sockaddr *)&server_address, sizeof(server_address));

    // listern
    listen(server_socket, 5);

    // recieve connection

    int client_socket = accept(server_socket, NULL, NULL);

    while (true)
    {
```

```
        serve_client(client_socket);
    }

    fflush(stdout);

    close(server_socket);

    // socket struct
}
```

Output:


```
rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/course_work/lab_practica
l/advanced_os/practical3$ cd server_module/
rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/course_work/lab_practical/ad
ver_module$ g++ -o server server.cpp
rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/course_work/lab_practical/ad
ver_module$ ./server
----------------------------------------
Recieved request for timestamp
----------------------------------------
getting the last update timestamp of the file
160464649898
----------------------------------------
----------------------------------------
Printing the final ts of the filename
160464649898
----------------------------------------
----------------------------------------
GOT FILE LENGTH = 758477
----------------------------------------
758477
----------------------------------------
Acknowledgement recieved... sending file..
----------------------------------------
----------------------------------------
file send to client .....
Closing connection ......
----------------------------------------
----------------------------------------
GOT FILE LENGTH = 758477
----------------------------------------
758477
rishabh@rishabh HP Pavilion Laptop 15 cc1xx /Documents/course work/la
```