**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

LAB REPORT

on

OPERATING SYSTEMS

(23CS4PCOPS)

***Submitted by***

Subramanya J (1BM23CS343)

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

COMPUTER SCIENCE AND ENGINEERING

****

B.M.S. COLLEGE OF ENGINEERING

**(Autonomous Institution under VTU)**

BENGALURU-560019

Feb-2025 to June-2025

B. M. S. College of Engineering,

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**

****

**CERTIFICATE**

This is to certify that the Lab work entitled “OPERATING SYSTEMS – 23CS4PCOPS” carried out by Subramanya J **(1BM23CS343),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Megha MJ **Dr. Kavitha Sooda**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Sl.**  **No.** | **Experiment Title** | **Page No.** |
| 1. | Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. (Any one)   1. FCFS 2. SJF | 5 |
| 2. | Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories –system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue. | 16 |
| 3. | Write a C program to simulate Real-Time CPU Scheduling algorithms   1. Rate- Monotonic | 19 |
| 4. | Write a C program to simulate:   1. Producer-Consumer problem using semaphores. 2. Dining-Philosopher’s problem | 21 |
| 5. | Write a C program to simulate:   1. Bankers’ algorithm for the purpose of deadlock avoidance. | 25 |
| 6. | Write a C program to simulate the following contiguous memory allocation techniques.   1. Worst-fit 2. Best-fit 3. First-fit | 27 |
| 7. | Write a C program to simulate page replacement algorithms.   1. FIFO 2. LRU 3. Optimal | 33 |

**Course Outcome**

|  |  |
| --- | --- |
| CO1 | Apply the different concepts and functionalities of Operating System |
| CO2 | Analyze various Operating system strategies and techniques |
| CO3 | Demonstrate the different functionalities of Operating System |
| CO4 | Conduct practical experiments to implement the functionalities of Operating system |

**Program 1**

Write -a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. (Any one)

1. FCFS
2. SJF

**Code :**

==> main\_fcfs.c <==

#include <stdio.h>

#include "process.h"

int main() {

int pnums, \*\*arr;

printf("Enter the number of processes : ");

scanf("%d", &pnums);

arr = create\_process\_table(pnums);

get\_pid\_data(arr, pnums);

sort\_index(arr, pnums, 1);

calculate\_params(arr, pnums);

display\_process\_table(arr, pnums);

free\_table(arr, pnums);

return 0;

}

==> main\_pr.c <==

#include "process.h"

#include <stdio.h>

#include <stdlib.h>

void calculate\_params\_priority(int \*\*arr, int pnums) {

sort\_index(arr, pnums, 1);

int time = 0;

int completed = 0;

int \*completed\_processes = calloc(pnums, sizeof(int));

while (completed < pnums) {

int highest\_priority = -1;

int next\_process = -1;

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

if (!completed\_processes[i] && arr[i][1] <= time) {

if (highest\_priority == -1 || arr[i][7] < highest\_priority) {

highest\_priority = arr[i][7];

next\_process = i;

}

}

}

if (next\_process == -1) {

time++;

continue;

}

completed\_processes[next\_process] = 1;

completed++;

time += arr[next\_process][2];

arr[next\_process][3] = time;

arr[next\_process][4] = arr[next\_process][3] - arr[next\_process][1]; // TAT = CT - AT

arr[next\_process][5] = arr[next\_process][4] - arr[next\_process][2]; // WT = TAT - BT

arr[next\_process][6] = arr[next\_process][5]; // RT = WT (for non-preemptive)

}

free(completed\_processes);

}

int main() {

int pnums, \*\*arr;

printf("Enter the number of processes : ");

scanf("%d", &pnums);

arr = create\_process\_table(pnums);

get\_pid\_data(arr, pnums);

sort\_index(arr, pnums, 1);

calculate\_params\_priority(arr, pnums);

display\_process\_table(arr, pnums);

free\_table(arr, pnums);

return 0;

}

==> main\_pr\_preempt.c <==

#include "process.h"

#include <stdio.h>

#include <stdlib.h>

void calculate\_params\_preemptive\_priority(int \*\*arr, int pnums) {

sort\_index(arr, pnums, 1);

int \*remaining\_time = malloc(pnums \* sizeof(int));

int \*is\_started = calloc(pnums, sizeof(int));

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

remaining\_time[i] = arr[i][2];

}

int time = 0;

int completed = 0;

while (completed < pnums) {

int highest\_priority = -1;

int next\_process = -1;

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

if (remaining\_time[i] > 0 && arr[i][1] <= time) {

if (highest\_priority == -1 || arr[i][7] < highest\_priority) {

highest\_priority = arr[i][7];

next\_process = i;

}

}

}

if (next\_process == -1) {

time++;

continue;

}

if (!is\_started[next\_process]) {

arr[next\_process][6] = time - arr[next\_process][1];

is\_started[next\_process] = 1;

}

remaining\_time[next\_process]--;

time++;

if (remaining\_time[next\_process] == 0) {

completed++;

arr[next\_process][3] = time; // CT = Current Time

arr[next\_process][4] = arr[next\_process][3] - arr[next\_process][1]; // TAT = CT - AT

arr[next\_process][5] = arr[next\_process][4] - arr[next\_process][2]; // WT = TAT - BT

}

}

free(remaining\_time);

free(is\_started);

}

int main() {

int pnums, \*\*arr;

printf("Enter the number of processes : ");

scanf("%d", &pnums);

arr = create\_process\_table(pnums);

get\_pid\_data(arr, pnums);

sort\_index(arr, pnums, 1);

calculate\_params\_preemptive\_priority(arr, pnums);

display\_process\_table(arr, pnums);

free\_table(arr, pnums);

return 0;

}

==> main\_rr.c <==

#include "process.h"

#include <stdio.h>

#include <stdlib.h>

void calculate\_params\_rr(int \*\*arr, int pnums, int time\_quantum) {

int \*remaining\_time = malloc(pnums \* sizeof(int));

int \*is\_started = calloc(pnums, sizeof(int));

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

remaining\_time[i] = arr[i][2];

}

int time = 0;

int completed = 0;

while (completed < pnums) {

int done = 1;

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

if (remaining\_time[i] > 0 && arr[i][1] <= time) {

done = 0;

if (!is\_started[i]) {

arr[i][6] = time - arr[i][1]; // RT = Start Time - AT

is\_started[i] = 1;

}

if (remaining\_time[i] > time\_quantum) {

time += time\_quantum;

remaining\_time[i] -= time\_quantum;

} else {

time += remaining\_time[i];

arr[i][3] = time; // CT

arr[i][4] = arr[i][3] - arr[i][1]; // TAT

arr[i][5] = arr[i][4] - arr[i][2]; // WT

remaining\_time[i] = 0;

completed++;

}

}

}

if (done) {

time++;

}

}

free(remaining\_time);

free(is\_started);

}

int main() {

int pnums, time\_q, \*\*arr;

printf("Enter the number of processes : ");

scanf("%d", &pnums);

printf("Enter the time quantum : ");

scanf("%d", &time\_q);

arr = create\_process\_table(pnums);

get\_pid\_data(arr, pnums);

sort\_index(arr, pnums, 1);

calculate\_params\_rr(arr, pnums, time\_q);

display\_process\_table(arr, pnums);

free\_table(arr, pnums);

return 0;

}

==> main\_sjf.c <==

#include "process.h"

#include <stdio.h>

#include <stdlib.h>

void calculate\_params\_sjf(int \*\*arr, int pnums) {

sort\_index(arr, pnums, 1);

int time = 0;

int completed = 0;

int \*completed\_processes = calloc(pnums, sizeof(int));

while (completed < pnums) {

int shortest\_burst = -1;

int next\_process = -1;

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

if (!completed\_processes[i] && arr[i][1] <= time) {

if (shortest\_burst == -1 || arr[i][2] < shortest\_burst) {

shortest\_burst = arr[i][2];

next\_process = i;

}

}

}

if (next\_process == -1) {

time++;

continue;

}

completed\_processes[next\_process] = 1;

completed++;

time += arr[next\_process][2];

arr[next\_process][3] = time;

arr[next\_process][4] = arr[next\_process][3] - arr[next\_process][1]; // TAT = CT - AT

arr[next\_process][5] = arr[next\_process][4] - arr[next\_process][2]; // WT = TAT - BT

arr[next\_process][6] = arr[next\_process][5]; // RT = WT (for non-preemptive SJF)

}

free(completed\_processes);

}

int main() {

int pnums, \*\*arr;

printf("Enter the number of processes : ");

scanf("%d", &pnums);

arr = create\_process\_table(pnums);

get\_pid\_data(arr, pnums);

sort\_index(arr, pnums, 1);

calculate\_params\_sjf(arr, pnums);

display\_process\_table(arr, pnums);

free\_table(arr, pnums);

return 0;

}

==> main\_sjf\_preempt.c <==

#include "process.h"

#include <stdio.h>

#include <stdlib.h>

void calculate\_params\_srtf(int \*\*arr, int pnums) {

int \*remaining\_time = malloc(pnums \* sizeof(int));

int \*is\_started = calloc(pnums, sizeof(int));

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

remaining\_time[i] = arr[i][2];

}

int time = 0;

int completed = 0;

while (completed < pnums) {

int shortest\_remaining = -1;

int next\_process = -1;

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

if (remaining\_time[i] > 0 && arr[i][1] <= time) {

if (shortest\_remaining == -1 || remaining\_time[i] < shortest\_remaining) {

shortest\_remaining = remaining\_time[i];

next\_process = i;

}

}

}

if (next\_process == -1) {

time++;

continue;

}

if (!is\_started[next\_process]) {

arr[next\_process][6] = time - arr[next\_process][1];

is\_started[next\_process] = 1;

}

remaining\_time[next\_process]--;

time++;

if (remaining\_time[next\_process] == 0) {

completed++;

arr[next\_process][3] = time;

arr[next\_process][4] = arr[next\_process][3] - arr[next\_process][1];

arr[next\_process][5] = arr[next\_process][4] - arr[next\_process][2];

}

}

free(remaining\_time);

free(is\_started);

}

int main() {

int pnums, \*\*arr;

printf("Enter the number of processes : ");

scanf("%d", &pnums);

arr = create\_process\_table(pnums);

get\_pid\_data(arr, pnums);

sort\_index(arr, pnums, 1);

calculate\_params\_srtf(arr, pnums);

display\_process\_table(arr, pnums);

free\_table(arr, pnums);

return 0;

}

==> process.c <==

#include <stdio.h>

#include <stdlib.h>

/\* When there are pnum processes,

\* Return a 2-D array of dimension

\* pnum x 8

\* \*/

int \*\*create\_process\_table(int pnum) {

int \*\*arr = calloc(pnum, sizeof(int \*));

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

arr[i] = calloc(8, sizeof(int));

}

return arr;

}

/\* The index to column mapping is :

\* 0 : Process ID

\* 1 : Arrival Time

\* 2 : Burst Time

\* 3 : Completion Time

\* 4 : Turnaround Time

\* 5 : Waiting Time

\* 6 : Response Time

\* 7 : Priority

\* \*/

void get\_pid\_data(int \*\*arr, int pnums) {

int pids = 0;

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

printf("\nEnter the data for process %d", ++pids);

arr[i][0] = pids;

printf("\nArrival Time : ");

scanf("%d", &arr[i][1]);

printf("\nBurst Time : ");

scanf("%d", &arr[i][2]);

printf("\nPriority :");

scanf("%d", &arr[i][7]);

}

}

void sort\_index(int \*\*arr, int pnums, int index) {

int \*ptr;

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

for(int j = i + 1; j < pnums; j++) {

if(arr[i][index] > arr[j][index]) {

ptr = arr[i];

arr[i] = arr[j];

arr[j] = ptr;

}

}

}

}

void display\_process\_table(int \*\*arr, int pnums) {

printf("\n|PID\t|AT\t|BT\t|CT\t|TAT\t|WT\t|RT\t|Priority\n");

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

for(int j = 0; j < 8; j++) {

printf("|%d\t", arr[i][j]);

} putchar('\n');

} putchar('\n');

}

void calculate\_params(int \*\*arr, int pnums) {

int time = 0;

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

time = arr[i][1] > time ? arr[i][1] : time;

/\* CT = Time + BT \*/

arr[i][3] = time + arr[i][2];

/\* TAT = CT - AT \*/

arr[i][4] = arr[i][3] - arr[i][1];

/\* WT = TAT - BT \*/

/\* FCFS, so RT = WT \*/

arr[i][6] = arr[i][5] = arr[i][4] - arr[i][2];

time = arr[i][3];

}

}

void free\_table(int \*\*arr, int pnums) {

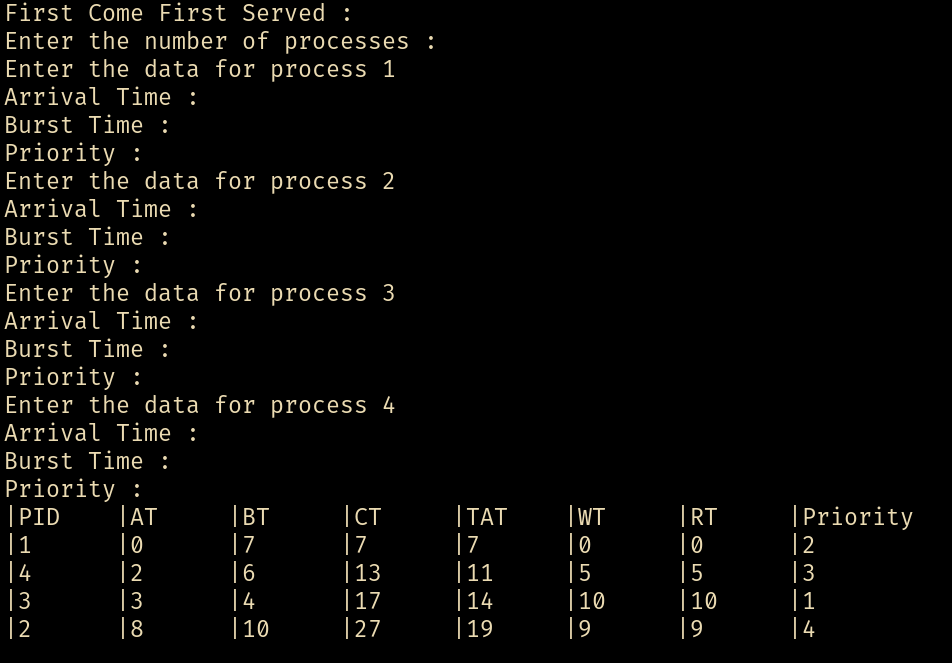
for(int i = 0; i < pnums; i++) {

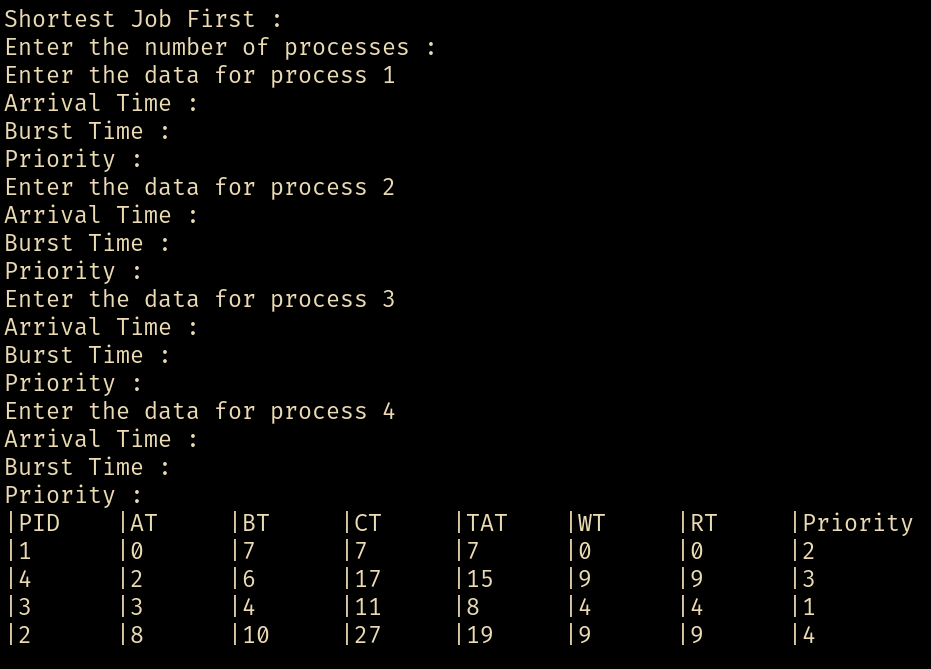
free(arr[i]);

} free(arr);

}

**Result :**

****

****

**Program 2**

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories –system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

**Code :**

#include <stdio.h>

#include <stdlib.h>

#include "queue.h"

struct Process {

int PID, AT, BT;

int CT, TAT, WT;

int PRI, RT;

};

struct Process \*get\_processes(int \*n) {

printf("Enter number of processes: ");

scanf("%d", n);

struct Process \*parr = calloc(\*n, sizeof(struct Process));

for (int i = 0; i < \*n; i++) {

parr[i].PID = i + 1;

printf("Enter Arrival Time, Burst Time, and Priority (0 for System, 1 for User) for Process %d: ", i + 1);

scanf("%d %d %d", &parr[i].AT, &parr[i].BT, &parr[i].PRI);

parr[i].RT = parr[i].BT;

}

return parr;

}

int ar\_sort(const void \*x, const void \*y) {

return ((struct Process \*)x)->AT - ((struct Process \*)y)->AT;

}

void classify\_processes(struct Process \*parr, int n, struct Queue \*\*sys, struct Queue \*\*usr) {

\*sys = create\_queue(n);

\*usr = create\_queue(n);

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

if(parr[i].PRI == 0) {

enqueue\_queue(\*sys, &parr[i]);

} else {

enqueue\_queue(\*usr, &parr[i]);

}

}

}

void fill\_data(struct Process \*p, int \*time) {

p->WT = \*time - p->AT;

\*time += p->BT;

p->CT = \*time;

p->TAT = p->CT - p->AT;

p->RT = 0;

}

void scheduler\_fcfs(struct Queue \*sys, struct Queue \*usr, int n) {

int time = 0;

struct Process \*current = NULL;

while(!isempty\_queue(sys) || !isempty\_queue(usr)) {

while(!isempty\_queue(sys)) {

struct Process \*sys\_proc = (struct Process \*)frontof\_queue(sys);

if(sys\_proc->AT <= time) {

current = dequeue\_queue(sys);

fill\_data(current, &time);

break;

} else {

break;

}

}

if(current == NULL && !isempty\_queue(usr)) {

struct Process \*usr\_proc = (struct Process \*)frontof\_queue(usr);

if(usr\_proc->AT <= time) {

current = dequeue\_queue(usr);

fill\_data(current, &time);

} else {

time++;

}

}

current = NULL;

}

}

void display\_process\_table(struct Process \*p\_list, int n) {

float avg\_TAT = 0, avg\_WT = 0;

printf("\nPID\tAT\tBT\tPRI\tCT\tTAT\tWT\n");

printf("------------------------------------------------\n");

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

struct Process p = p\_list[i];

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

p.PID, p.AT, p.BT, p.PRI, p.CT, p.TAT, p.WT);

avg\_TAT += p.TAT;

avg\_WT += p.WT;

}

avg\_TAT /= n;

avg\_WT /= n;

printf("\nAverage Turn Around Time: %.2f\n", avg\_TAT);

printf("Average Waiting Time: %.2f\n", avg\_WT);

}

int main() {

int n;

struct Process \*p\_list = get\_processes(&n);

struct Queue \*sys, \*usr;

qsort(p\_list, n, sizeof(struct Process), ar\_sort);

classify\_processes(p\_list, n, &sys, &usr);

scheduler\_fcfs(sys, usr, n);

display\_process\_table(p\_list, n);

free\_queue(sys);

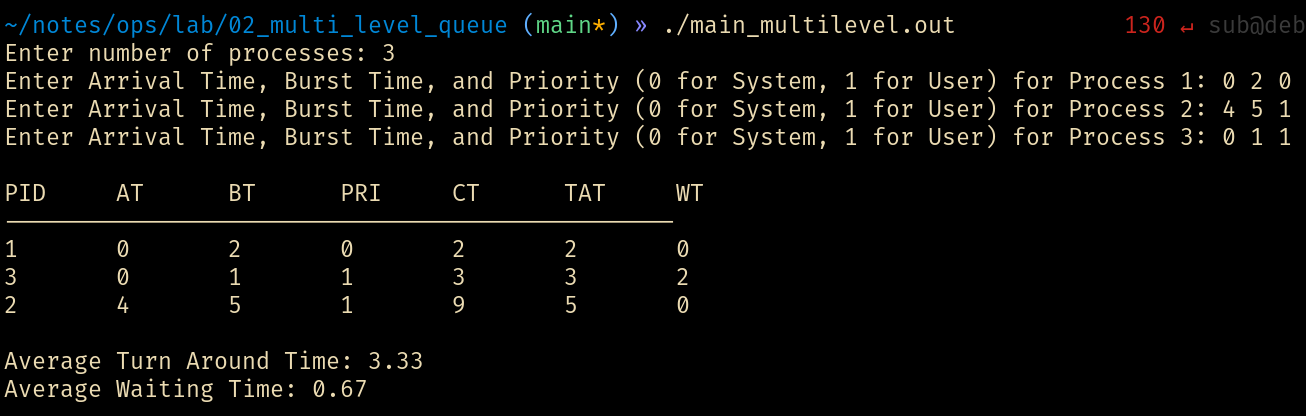
free\_queue(usr);

free(p\_list);

return 0;

}

**Result :**



**Program 3**

Write a C program to simulate Real-Time CPU Scheduling algorithms

a) Rate- Monotonic

**Code :**

#include <stdio.h>

#include <stdlib.h>

#include "rtos.h"

#define PCAST (struct Process \*)

int

proccmp(const void \*x, const void \*y) {

return (PCAST x)->ival - (PCAST y)->ival;

}

void

monotonic\_scheduler(struct Process \*arr, int n) {

int max\_time = arr[n-1].ival \* 2;

int time = 0;

struct Process \*curr = NULL, \*prev = NULL;

while(time < max\_time) {

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

if(arr[i].next\_release == time) {

arr[i].rtr = true;

arr[i].next\_release += arr[i].ival;

arr[i].rem\_bt = arr[i].bt;

}

}

curr = NULL;

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

if(arr[i].rtr == true && arr[i].rem\_bt > 0) {

curr = &arr[i];

break;

}

}

if(prev != curr) {

if (prev != NULL && prev->rem\_bt > 0)

printf("PID %d till %d\n", prev->PID, time);

if (curr != NULL)

printf("PID %d starts at %d\n", curr->PID, time);

prev = curr;

}

if(!curr) {time++; continue;}

curr->rem\_bt--;

if(curr->rem\_bt == 0) { curr->rtr = false; }

time++;

}

return;

}

int

main() {

int n;

struct Process \*arr = get\_processes(&n);

qsort(arr, n, sizeof(struct Process), proccmp);

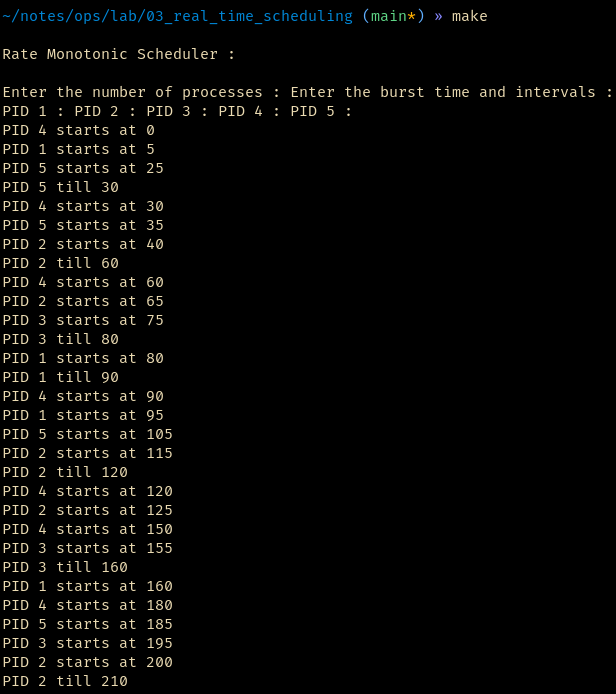
putchar('\n');

monotonic\_scheduler(arr, n);

return 0;

}

**Result :**



**Program**

Write a C program to simulate:

1. Producer-Consumer problem using semaphores.
2. Dining-Philosopher’s problem

**Code :**

**==> main\_dp.c <==**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <unistd.h>**

**#define N 5**

**sem\_t chop[N];**

**void \*philosopher(void \*num) {**

**int id = \*(int \*)num;**

**while (1) {**

**printf("Philosopher %d is thinking.\n", id);**

**sleep(1);**

**sem\_wait(&chop[id]);**

**sem\_wait(&chop[(id + 1) % N]);**

**printf("Philosopher %d is eating.\n", id);**

**sleep(1);**

**sem\_post(&chop[id]);**

**sem\_post(&chop[(id + 1) % N]);**

**printf("Philosopher %d finished eating.\n", id);**

**sleep(1);**

**}**

**}**

**int main() {**

**pthread\_t phil[N];**

**int i, ids[N];**

**for (i = 0; i < N; i++) sem\_init(&chop[i], 0, 1);**

**for (i = 0; i < N; i++) {**

**ids[i] = i;**

**pthread\_create(&phil[i], NULL, philosopher, &ids[i]);**

**}**

**for (i = 0; i < N; i++) pthread\_join(phil[i], NULL);**

**for (i = 0; i < N; i++) sem\_destroy(&chop[i]);**

**return 0;**

**}**

**==> main\_pc.c <==**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <unistd.h>**

**#define BUFFER\_SIZE 5**

**int buffer[BUFFER\_SIZE];**

**int in = 0, out = 0;**

**sem\_t empty, full, mutex;**

**void \*producer(void \*arg) {**

**int item;**

**while (1) {**

**item = rand() % 100;**

**sem\_wait(&empty);**

**sem\_wait(&mutex);**

**buffer[in] = item;**

**printf("Produced: %d at %d\n", item, in);**

**in = (in + 1) % BUFFER\_SIZE;**

**sem\_post(&mutex);**

**sem\_post(&full);**

**sleep(1);**

**}**

**}**

**void \*consumer(void \*arg) {**

**int item;**

**while (1) {**

**sem\_wait(&full);**

**sem\_wait(&mutex);**

**item = buffer[out];**

**printf("Consumed: %d from %d\n", item, out);**

**out = (out + 1) % BUFFER\_SIZE;**

**sem\_post(&mutex);**

**sem\_post(&empty);**

**sleep(1);**

**}**

**}**

**int main() {**

**pthread\_t prod, cons;**

**sem\_init(&empty, 0, BUFFER\_SIZE);**

**sem\_init(&full, 0, 0);**

**sem\_init(&mutex, 0, 1);**

**pthread\_create(&prod, NULL, producer, NULL);**

**pthread\_create(&cons, NULL, consumer, NULL);**

**pthread\_join(prod, NULL);**

**pthread\_join(cons, NULL);**

**sem\_destroy(&empty);**

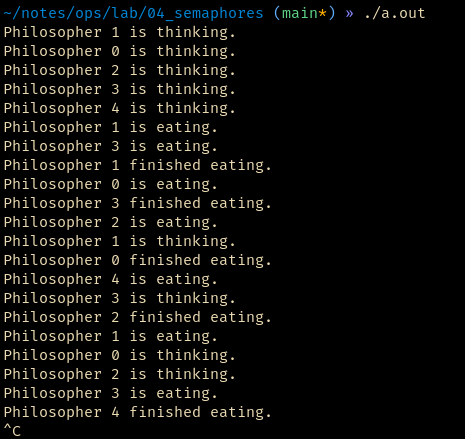
**sem\_destroy(&full);**

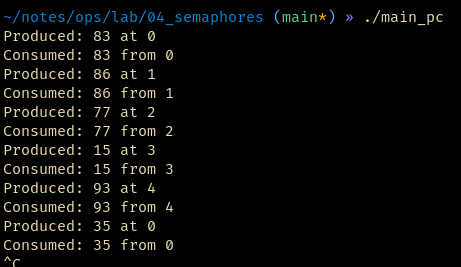
**sem\_destroy(&mutex);**

**return 0;**

**}**

**Result :**





**Program**

Write a C program to simulate:

a) Bankers’ algorithm for the purpose of deadlock avoidance.

#include <stdio.h>

int main() {

int n, m, i, j, k;

printf("Enter the number of processes and resources : ");

scanf("%d %d", &n, &m);

int allo[n][m], max[n][m], need[n][m], avail[m];

printf("Enter the allo matrix:\n");

for (i = 0; i < n; i++)

for (j = 0; j < m; j++)

scanf("%d", &allo[i][j]);

printf("Enter the maximum demand matrix:\n");

for (i = 0; i < n; i++)

for (j = 0; j < m; j++) {

scanf("%d", &max[i][j]);

need[i][j] = max[i][j] - allo[i][j];

}

printf("Enter the avail resources:\n");

for (i = 0; i < m; i++)

scanf("%d", &avail[i]);

int finished[n], safeSequence[n], count = 0;

for (i = 0; i < n; i++) finished[i] = 0;

while (count < n) {

int found = 0;

for (i = 0; i < n; i++) {

if (!finished[i]) {

for (j = 0; j < m; j++)

if (need[i][j] > avail[j]) break;

if (j == m) {

for (k = 0; k < m; k++)

avail[k] += allo[i][k];

safeSequence[count++] = i;

finished[i] = 1;

found = 1;

}

}

}

if (!found) {

printf("System is in an unsafe state.\n");

return 0;

}

}

printf("System is in a safe state.\nSafe sequence: ");

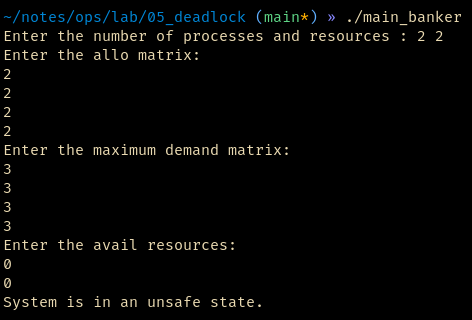
for (i = 0; i < n; i++) printf("%d ", safeSequence[i]);

printf("\n");

return 0;

}

**Result :**



**Program :**

Write a C program to simulate the following contiguous memory allocation techniques.

1. Worst-fit
2. Best-fit
3. First-fit

**Code:**

**==> main\_best.c <==**

**#include <stdio.h>**

**#include "main\_common.c"**

**void bestFit(int blockSize[], int m, int processSize[], int n) {**

**int allocation[n];**

**for (int i = 0; i < n; i++) {**

**allocation[i] = -1;**

**}**

**for (int i = 0; i < n; i++) {**

**int bestIdx = -1;**

**for (int j = 0; j < m; j++) {**

**if (blockSize[j] >= processSize[i]) {**

**if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) {**

**bestIdx = j;**

**}**

**}**

**}**

**if (bestIdx != -1) {**

**allocation[i] = bestIdx;**

**blockSize[bestIdx] -= processSize[i];**

**}**

**}**

**printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");**

**for (int i = 0; i < n; i++) {**

**if (allocation[i] != -1) {**

**printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, processSize[i], allocation[i] + 1, blockSize[allocation[i]]);**

**} else {**

**printf("%d\t\t%d\t\tNot Allocated\n", i + 1, processSize[i]);**

**}**

**}**

**}**

**int main() {**

**COMMON\_MAIN()**

**bestFit(blockSize, m, processSize, n);**

**return 0;**

**}**

**==> main\_common.c <==**

**#define COMMON\_MAIN() \**

**int m, n; \**

**printf("Enter the number of blocks: "); \**

**scanf("%d", &m); \**

**printf("Enter the number of processes: "); \**

**scanf("%d", &n); \**

**int blockSize[m], processSize[n]; \**

**printf("\nEnter the sizes of the blocks:\n"); \**

**for (int i = 0; i < m; i++) { \**

**printf("Block %d size: ", i + 1); \**

**scanf("%d", &blockSize[i]); \**

**} \**

**printf("\nEnter the sizes of the processes:\n"); \**

**for (int i = 0; i < n; i++) { \**

**printf("Process %d size: ", i + 1); \**

**scanf("%d", &processSize[i]); \**

**}**

**==> main\_first.c <==**

**#include <stdio.h>**

**#include "main\_common.c"**

**void firstFit(int blockSize[], int m, int processSize[], int n) {**

**int allocation[n];**

**for (int i = 0; i < n; i++) {**

**allocation[i] = -1;**

**}**

**for (int i = 0; i < n; i++) {**

**for (int j = 0; j < m; j++) {**

**if (blockSize[j] >= processSize[i]) {**

**allocation[i] = j;**

**blockSize[j] -= processSize[i];**

**break;**

**}**

**}**

**}**

**printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");**

**for (int i = 0; i < n; i++) {**

**if (allocation[i] != -1) {**

**printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, processSize[i], allocation[i] + 1, blockSize[allocation[i]]);**

**} else {**

**printf("%d\t\t%d\t\tNot Allocated\n", i + 1, processSize[i]);**

**}**

**}**

**}**

**int main() {**

**COMMON\_MAIN()**

**firstFit(blockSize, m, processSize, n);**

**return 0;**

**}**

**==> main\_worst.c <==**

**#include <stdio.h>**

**#include "main\_common.c"**

**void worstFit(int blockSize[], int m, int processSize[], int n) {**

**int allocation[n];**

**for (int i = 0; i < n; i++) {**

**allocation[i] = -1;**

**}**

**for (int i = 0; i < n; i++) {**

**int worstIdx = -1;**

**for (int j = 0; j < m; j++) {**

**if (blockSize[j] >= processSize[i]) {**

**if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {**

**worstIdx = j;**

**}**

**}**

**}**

**if (worstIdx != -1) {**

**allocation[i] = worstIdx;**

**blockSize[worstIdx] -= processSize[i];**

**}**

**}**

**printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");**

**for (int i = 0; i < n; i++) {**

**if (allocation[i] != -1) {**

**printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, processSize[i], allocation[i] + 1, blockSize[allocation[i]]);**

**} else {**

**printf("%d\t\t%d\t\tNot Allocated\n", i + 1, processSize[i]);**

**}**

**}**

**}**

**int main() {**

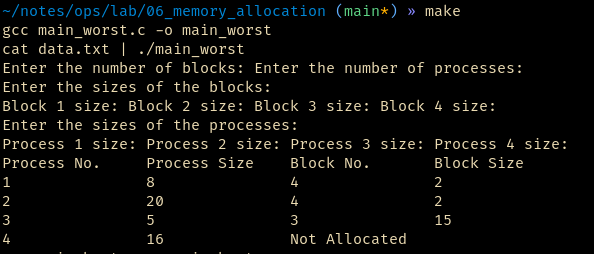
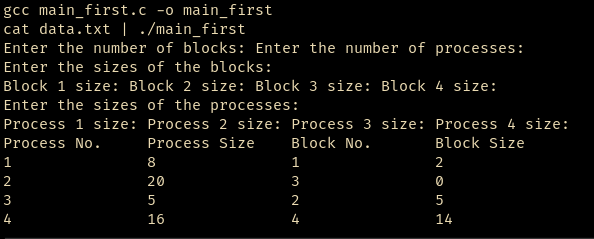
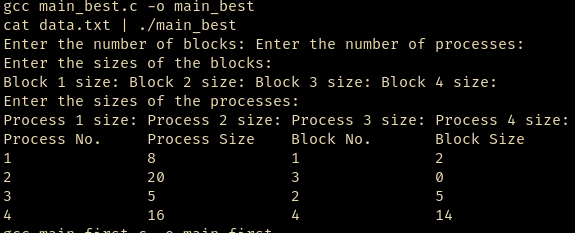
**COMMON\_MAIN()**

**worstFit(blockSize, m, processSize, n);**

**return 0;**

**}**

**Result :**



**Program**

Write a C program to simulate page replacement algorithms.

1. FIFO
2. LRU
3. Optimal

Code :

==> main\_fifo.c <==

#include "page\_replacement.h"

int main() {

int frames, pages, pageFaults = 0, current = 0;

int referenceString[100], frame[100]; // Max size to prevent runtime issues

GET\_INPUT(frames, pages, referenceString);

// Initialize frames

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

frame[i] = -1;

}

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

int found = 0;

for (int j = 0; j < frames; j++) {

if (frame[j] == referenceString[i]) {

found = 1;

break;

}

}

if (!found) {

frame[current] = referenceString[i];

current = (current + 1) % frames;

pageFaults++;

}

DISPLAY\_FRAMES(referenceString[i], frames, frame);

}

PRINT\_PAGE\_FAULTS(pageFaults);

return 0;

}

==> main\_lru.c <==

#include "page\_replacement.h"

int main() {

int frames, pages, pageFaults = 0;

int referenceString[100], frame[100]; // Max size to prevent runtime issues

GET\_INPUT(frames, pages, referenceString);

// Initialize frames

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

frame[i] = -1;

}

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

int found = 0;

for (int j = 0; j < frames; j++) {

if (frame[j] == referenceString[i]) {

found = 1;

break;

}

}

if (!found) {

int max = -1, index = -1;

for (int j = 0; j < frames; j++) {

if (frame[j] == -1) { index = j; break; }

int next = 0;

for (int k = i + 1; k < pages; k++) {

if (frame[j] == referenceString[k]) { next = k; break; }

}

if (next == 0) { index = j; break; }

if (next > max) { max = next; index = j; }

}

frame[index] = referenceString[i];

pageFaults++;

}

DISPLAY\_FRAMES(referenceString[i], frames, frame);

}

PRINT\_PAGE\_FAULTS(pageFaults);

return 0;

}

==> main\_optimal.c <==

#include "page\_replacement.h"

int main() {

int frames, pages, pageFaults = 0;

int referenceString[100], frame[100];

GET\_INPUT(frames, pages, referenceString);

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

frame[i] = -1;

}

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

int found = 0;

for (int j = 0; j < frames; j++) {

if (frame[j] == referenceString[i]) {

found = 1;

break;

}

}

if (!found) {

int max = -1, index = -1;

for (int j = 0; j < frames; j++) {

if (frame[j] == -1) { index = j; break; }

int next = 0;

for (int k = i + 1; k < pages; k++) {

if (frame[j] == referenceString[k]) { next = k; break; }

}

if (next == 0) { index = j; break; }

if (next > max) { max = next; index = j; }

}

frame[index] = referenceString[i];

pageFaults++;

}

DISPLAY\_FRAMES(referenceString[i], frames, frame);

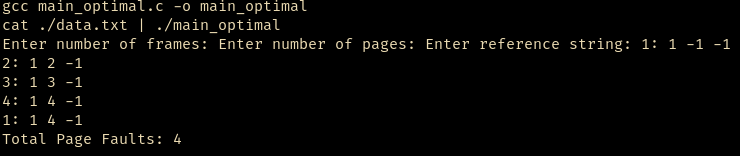
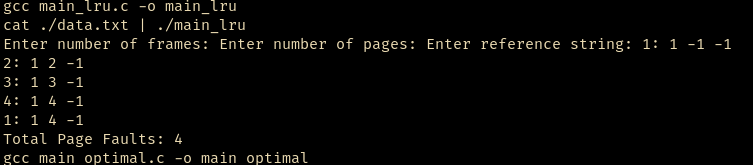
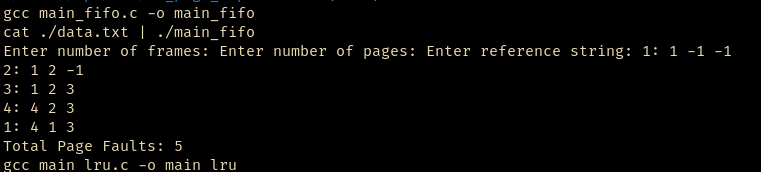
}

PRINT\_PAGE\_FAULTS(pageFaults);

return 0;

}

**Result :**



**Program :**

Write a C program to simulate the following file allocation strategies.

1. Sequential
2. Indexed
3. Linked

Code :