# DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

OPERATING SYSTEM LAB 22CSL44

|  |  |  |
| --- | --- | --- |
| **Prepared by** | **Verified by** | **Approved by** |
| **Course co-ordinator** | Rahul B | **HOD- CSE** |

# INSTITUTE VISION AND MISSION

## VISION

To emerge as an institute of eminence in the fields of engineering, technology and managementin serving the industry and the nation by empowering students with a high degree of technical,managerial and practical competence.

## MISSION

* To strengthen the theoretical, practical and ethical dimensions of the learning process byfostering a culture of research and innovation among faculty members and students
* To encourage long-term interaction between the academia and industry through theirinvolvement in the design of curriculum and its hands-on implementation
* To strengthen and mould students in professional, ethical, social and environmental dimensions by encouraging participation in co-curricular and extracurricular activities

# DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

## VISION

To emerge as a department of eminence in Computer Science and Engineering in serving the Information Technology Industry and the nation by empowering students with a high degree of technical and practical competence.

## MISSION

* To strengthen the theoretical and practical aspects of the learning process by strongly encouraging a culture of research, innovation and hands-on learning in Computer Science and Engineering
* To encourage long-term interaction between the department and the IT industry, through the involvement of the IT industry in the design of the curriculum and its hands-on implementation
* To widen the awareness of students in professional, ethical, social and environmental dimensions by encouraging their participation in co-curricular and extracurricular activities

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**PROGRAM OUTCOMES (POs)**

The student will be able to:

**PO1: Engineering Knowledge:** Apply knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex Computer Science and engineering problems.

**PO2: Problem Analysis:** Identify, formulate, review research literature and analyze complex engineering problems in Computer Science and Engineering reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.

**PO3: Design / Development of Solutions:** Design solutions for complex engineering problems and design system components or processes of Computer Science and Engineering that meet the specified needs with appropriate consideration for public health and safety, cultural, societaland environmental considerations.

**PO4: Conduct Investigations of Complex Problems:** Use research-based knowledge and research methods including design of experiments in Computer Science and Engineering, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

**PO5: Modern tool usage:** Create, select and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities related to Computer Science and Engineering with an understanding of the limitations.

**PO6: The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice in Computer Science and Engineering.

**PO7: Environment and sustainability:** Understand the impact of the professional engineering solutions of Computer Science and Engineering in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**PO8: Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

**PO9: Individual and Team Work:** Function effectively as an individual and as a member orleader to diverse teams, and in multidisciplinary settings.

**PO10: Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective report and design documentation, make effective presentations, and give and receiveclear instructions.

**PO11: Project Management and Finance:** Demonstrate knowledge and understanding of theengineering and management principles and apply these to one’s own work, as a member andleader in a team, to manage projects and in multidisciplinary environments.

**PO12: Life-Long Learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

## PROGRAM SPECIFIC OUTCOMES (PSOs)

The student will be able to:

**PSO1:** Ability to design, develop, implement computer programs and use knowledge in various domains to identify research gaps and hence to provide solution to new ideas and innovations.

**PSO2:** Work with and communicate effectively with professionals in various fields and pursuelifelong professional development in computing.

## PROGRAM EDUCATIONAL OBJECTIVES (PEOS)

The Graduate of the program will be able to:

**PE01:** Develop proficiency as computer scientists with an ability to solve a wide range of computational problems in industry, government, or other work environments.

**PE02:** Attain the ability to adapt quickly to new environments and technologies, assimilate new information, and work in multi-disciplinary areas with a strong focus on innovation and entrepreneurship.

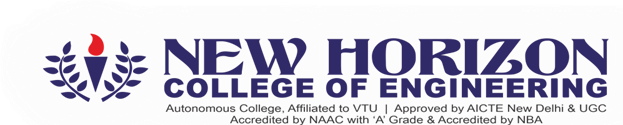
**PE03:** Possess the ability to think logically and the capacity to understand technical problemswith computational systems.

**PE04:** Possess the ability to collaborate as team members and team leaders to facilitate cutting-edge technical solutions for computing systems and thereby providing improved functionality.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Operating System Lab** | | | | | | | | | | | | | | | | | | |
| **Course Code** | | **22CSL44** | | | | | | | | **CIE Marks** | | | | | **50** | | | |
| **L:T:P:S** | | **0:0:1:0** | | | | | | | | **SEE Marks** | | | | | **50** | | | |
| **Hrs / Week** | | **2** | | | | | | | | **Total Marks** | | | | | **100** | | | |
| **Credits** | | **01** | | | | | | | | **Exam Hours** | | | | | **03** | | | |
| **Course outcomes:**  At the end of the course, the student will be able to: | | | | | | | | | | | | | | | | | | |
| **22CSL44.1** | | Perform UNIX System Calls and implement CPU Scheduling algorithms | | | | | | | | | | | | | | | | |
| **22CSL44.2** | | Devise solutions for process synchronization, deadlock avoidance, and prevention in a specified scenario | | | | | | | | | | | | | | | | |
| **22CSL44.3** | | Evaluate different methods of memory allocation and page replacement strategies. | | | | | | | | | | | | | | | | |
| **22CSL44.4** | | Implement disk scheduling algorithms based on a provided process description. | | | | | | | | | | | | | | | | |
| **Mapping of Course Outcomes to Program Outcomes and Program Specific Outcomes:** | | | | | | | | | | | | | | | | | | |
|  | **PO1** | | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | | **PO9** | **PO10** | **PO11** | | **PO12** | | **PSO1** | **PSO2** |
| **22CSL44.1** | 3 | | 3 | 3 | 3 | 3 | - | - | - | | - | - | - | | 3 | | 3 | - |
| **22CSL44.2** | 3 | | 3 | 3 | 3 | 3 | - | - | - | | - | - | - | | 3 | | 3 | - |
| **22CSL44.3** | 3 | | 3 | 3 | 3 | 3 | - | - | - | | - | - | - | | 3 | | 3 | - |
| **22CSL44.4** | 3 | | 3 | 3 | 3 | 3 | - | - | - | | - | - | - | | 3 | | 3 | - |
|  | | | | | | | | | | | | | | | | | | |
| **Exp. No. / Pgm. No.** | **List of Experiments / Programs** | | | | | | | | | | | | | **Hours** | | **Cos** | | |
| **Prerequisite Experiments / Programs / Demo** | | | | | | | | | | | | | | | | | | |
|  | Proficiency in programming languages like C or C++ is essential for operating system development since many OS components are typically written in these languages. | | | | | | | | | | | | | 2 | | NA | | |
| **PART-A** | | | | | | | | | | | | | | | | | | |
| 1 | Compose a program utilizing the following system calls:   * opendir, readdir, closedir, * fork, exec, getpid | | | | | | | | | | | | | 2 | | **22CSL44.1** | | |
| 2 | Develop a program to model non-preemptive SJF CPU scheduling algorithms. | | | | | | | | | | | | | 2 | | **22CSL44.1** | | |
| 3 | Create a program to simulate the round-robin scheduling algorithm | | | | | | | | | | | | | 2 | | **22CSL44.1** | | |
| 4 | Develop a program that demonstrates Shared Memory and Inter-  Process Communication. | | | | | | | | | | | | | 2 | | **22CSL44.2** | | |
| 5 | Create a program that simulates the Producer-Consumer problem using semaphores. | | | | | | | | | | | | | 2 | | **22CSL44.2** | | |
| 6 | Implement a program to depict the Dining Philosopher's problem concept. | | | | | | | | | | | | | 2 | | **22CSL44.2** | | |
| **PART-B** | | | | | | | | | | | | | | | | | | |
| 7 | Develop a program for simulating the Banker's Algorithm to prevent  deadlock avoidance. | | | | | | | | | | | | | 2 | | **22CSL44.2** | | |
| 8 | Create a program to simulate the Banker's Algorithm for deadlock prevention. | | | | | | | | | | | | | 2 | | **22CSL44.2** | | |
| 9 | Implement a program to emulate first-fit contiguous memory allocation. | | | | | | | | | | | | | 2 | | **22CSL44.3** | | |
| 10 | Develop a program for simulating paging table implementation and determining the actual physical address in memory. | | | | | | | | | | | | | 2 | | **22CSL44.3** | | |
| 11 | Create a program to execute the FIFO page replacement algorithm. | | | | | | | | | | | | | 2 | | **22CSL44.3** | | |
| 12 | Implement a program for simulating the SCAN disk scheduling algorithm. | | | | | | | | | | | | | 2 | | **22CSL44.4** | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PART-C**  **Beyond Syllabus Virtual Lab Content**  **(To be done during Lab but not to be included for CIE or SEE)**  https://naim30.github.io/OS-virtual-lab/ | | | | | |
| **CIE Assessment Pattern (50 Marks – Lab)** | | | | | |
|  | **RBT Levels** | | **Test (s)** | **Weekly Assessment** |  |
| **20** | **30** |
|  | **L1** | **Remember** | **-** | 5 |
|  | **L2** | **Understand** | 5 | 5 |
|  | **L3** | **Apply** | 5 | 10 |
|  | **L4** | **Analyze** | 5 | 10 |
|  | **L5** | **Evaluate** | 5 |  |
|  | **L6** | **Create** | - |  |
| **SEE Assessment Pattern (50 Marks – Lab)** | | | | | |
| **Suggested Learning Resources:**  1) Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons, Inc., 10th Edition, 2018, ISBN978-1-118-06333-0. | | | | | |

|  |  |  |
| --- | --- | --- |
| **RBT Levels** | | **Exam Marks Distribution (50)** |
| **L1** | **Remember** | **-** |
| **L2** | **Understand** | **-** |
| **L3** | **Apply** | **20** |
| **L4** | **Analyze** | **20** |
| **L5** | **Evaluate** | **10** |
| **L6** | **Create** | **-** |



# Department of Computer Science & Engineering.

**OPERATING SYSTEM LAB [22CSL44]**

# LAB RUBRICS

### Internal Assessment Marks: 50 Divided into two components:

* + Continuous Assessment: 30Marks
  + Internal Test : 20 Marks

### Continuous Assessment:

1. Will be carried out in every lab (for 12 labs- 12 experiments)
2. Each lab will be evaluated for 30 marks

### Break up of 30 marks (in every lab):

Will be carried out in every lab (for 12 labs – 12 experiments)

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Descriptors** | **Scores** |
| Program Write- up (10) | Complete program with proper variable naming, commenting | 10 |
| Complete program with not-so-proper variable naming, poor commenting | 6-8 |
| Incomplete code | <5 |
| Not Written | 0 |
| Execution & Results (10) | Passes all specified test cases efficiently | 10 |
| Fails in some test cases | 6-8 |
| Incomplete execution | <5 |

|  |  |  |
| --- | --- | --- |
| Viva Voce (5) | Viva questions answered correctly | 5 |
| Viva questions answered satisfactorily | 3 |
| Do not answer any question | 0 |
| Record completion and submission (5) | Completed record and observation on time | 5 |
| Fails to submit the record in time / incomplete submission | 0 |

### Internal Test -1+2 : 10+10 Marks (Total of 2 CIE)

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **EXPLANATION** | **CIE1(MARKS)** | **CIE2(MARKS)** |
| 01 | Write up | 3 | 3 |
| 02 | Execution and results | 5 | 5 |
| 03 | Viva Voce | 2 | 2 |
|  | TOTAL | **10** | **10** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Descriptors** | **Scores** |
| Conduction of experiment/  Writing the program (3) | Student can completely write and type the program and able to explain to working of program | 3 |
| Student can explain 75% their program | 2 |
| Student can explain 50% of their program | 1 |
| Student can’t explain program | 0 |
| Execution of program /output  (5) | Execution of program without error | 5 |
| Partial Execution of program without error | 3-4 |
| No Execution of program | 0 |
| Viva Voce  (2) | Answers correctly | 2 |
| Answers satisfactorily | 1 |

|  |  |  |
| --- | --- | --- |
|  | Do not answer any question | 0 |

**SEE Assessment Marks: 50 Marks**

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Descriptors** | **Scores** |
| Conduction of experiment/  Writing the program (10) | Student can completely write and type the program and able to explain to working of program | 10 |
| Student can explain 75% their program | 6-8 |
| Student can explain 50% of their program | 3-5 |
| Student can’t explain program | 0-2 |
| Execution of program /output  (30) | Execution of program without error | 30 |
| Partial Execution of program without error | 15-20 |
| No Execution of program | 0 |
| Viva Voce (10) | Answers correctly | 10 |
| Answers satisfactorily | 6-8 |
| Do not answer any question | 0 |

**Lab Course Faculty Course Coordinator**

**CONTENT**

|  |  |  |
| --- | --- | --- |
| **Exp. No** | **List of Experiments** | **Page No** |
|  | **PART A** |  |
| 1 | Compose a program utilizing the following system calls:   * opendir, readdir, closedir, * fork, exec, getpid | 1 |
| 2 | Develop a program to model non-preemptive SJF CPU scheduling algorithms. | 5 |
| 3 | Create a program to simulate the round-robin scheduling algorithm | 9 |
| 4 | Develop a program that demonstrates Shared Memory and Inter-Process Communication. | 12 |
| 5 | Create a program that simulates the Producer-Consumer problem using semaphores. | 15 |
| 6 | Implement a program to depict the Dining Philosopher's problem concept. | 18 |
|  | **PART-B** |  |
| 7 | Develop a program for simulating the Banker's Algorithm to prevent Deadlock avoidance. | 21 |
| 8 | Create a program to simulate the Banker's Algorithm for deadlock prevention. | 24 |
| 9 | Implement a program to emulate first-fit contiguous memory allocation. | 27 |
| 10 | Develop a program for simulating paging table implementation and determining the actual physical address in memory. | 30 |
| 11 | Create a program to execute the FIFO page replacement algorithm. | 32 |
| 12 | Implement a program for simulating the SCAN disk scheduling algorithm. | 35 |

### Compose a program utilizing the following system calls:

* 1. **opendir, readdir, closedir,**

### fork, exec, getpid Theory:

The **opendir** subroutine opens the directory designated by the DirectoryName parameter and associates a directory stream with it.

The **readdir** subroutine returns a pointer to the next directory entry. The readdir subroutine returns entries for . (dot) and .. (dot dot), if present, but never returns an invalid entry (with d\_ino set to 0). When it reaches the end of the directory, or when it detects an invalid seekdir operation, the readdir subroutine returns the null value. The returned pointer designates data that may be overwritten by another call to the readdir subroutine on the same directory stream. A call to the readdir subroutine on a different directory stream does not overwrite this data. The readdir subroutine marks the st\_atime field of the directory for update each time the directory is actually read.

The **closedir** subroutine closes a directory stream and frees the structure associated with the DirectoryPointer parameter. If the closedir subroutine is called for a directory that is already closed, the behavior is undefined. To prevent this, always initialize the DirectoryPointer parameter to null after closure.

The **fork()** is one of the syscalls that is very special and useful in Linux/Unix systems. It is used by processes to create the processes that are copies of themselves. With the help of such system calls, the child process can be created by the parent process. Until the child process is executed completely, the parent process is suspended.

Some of the important points on fork() are as follows.

* + - The parent will get the child process ID with non-zero value.
    - Zero Value is returned to the child.
    - If there will be any system or hardware errors while creating the child, -1 is returned to the fork().
    - With the unique process ID obtained by the child process, it does not match the ID of any existing process group.

The **exec()** is such a system call that runs by replacing the current process image with the new process image. However, the original process remains as a new process but the new process replaces the head data, stack data,etc. It runs the program from the entry point by loading the program into the current process space.

**getpid() :** returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

### Program: 1a

#include<stdio.h> #include<dirent.h> #include<stdlib.h> struct dirent \*dptr;

int main(int argc, char \*argv[])

{

char buff[100]; DIR \*dirp;

printf("\n\n ENTER DIRECTORY NAME"); scanf("%s", buff); if((dirp=opendir(buff))==NULL)

{printf("The given directory does not exist"); exit(1);

}

while(dptr=readdir(dirp))

{

printf("%s\n",dptr→d\_name);

}

closedir(dirp);

}

### Output:

$ mkdir d1

$ mkdir d2

$cd d1

$mkdir d11

$cc 1a.c

$./a.out

$ENTER DIRECTORY NAMEd1

d11

.

..

**Program: 1b** #include<stdio.h> #include<unistd.h> #include<stdlib.h> int main()

{

int pid,pid1,pid2; pid=fork(); if(pid==-1)

{

printf("ERROR IN PROCESS CREATION \n");

exit(1);

}

if(pid!=0)

{

}

else

{

pid1=getpid();

printf("\n the parent process ID is %d\n", pid1);

pid2=getpid();

printf("\n the child process ID is %d\n", pid2);

}

}

### OUTPUT:

$ cc 1b.c

$ ./a.out

the parent process ID is 43483 the child process ID is 43484

### Develop a program to model non-preemptive SJF CPU scheduling algorithms.

**Theory:**

The shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN, also known as Shortest Job Next (SJN), can be preemptive or non-preemptive.

Characteristics of SJF Scheduling:

* Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
* It is a Greedy Algorithm.
* It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
* It is practically infeasible as Operating System may not know burst times and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.
* SJF can be used in specialized environments where accurate estimates of running time are available.

### Program:

#include <stdio.h> struct process {

int pid; // Process ID int burst; // Burst time int arrival; // Arrival time

};

void swap(struct process\* a, struct process\* b) { struct process temp = \*a;

\*a = \*b;

\*b = temp;

}

void sort\_processes(struct process p[], int n) { for (int i = 0; i < n-1; i++) {

for (int j = i+1; j < n; j++) { if (p[i].burst > p[j].burst) {

swap(&p[i], &p[j]);

}

}

}

}

void sjf\_scheduling(struct process p[], int n) { int waiting\_time[n], turnaround\_time[n];

float total\_waiting\_time = 0, total\_turnaround\_time = 0;

// Sort processes by their burst time sort\_processes(p, n);

// Calculate waiting and turnaround time for each process waiting\_time[0] = 0;

turnaround\_time[0] = p[0].burst; for (int i = 1; i < n; i++) {

waiting\_time[i] = turnaround\_time[i-1]; turnaround\_time[i] = waiting\_time[i] + p[i].burst;

}

// Print process details and their times

printf("\nProcess ID\tBurst Time\tWaiting Time\tTurnaround Time\n"); for (int i = 0; i < n; i++) {

printf("%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].burst, waiting\_time[i], turnaround\_time[i]);

total\_waiting\_time += waiting\_time[i];

total\_turnaround\_time += turnaround\_time[i];

}

// Print average waiting and turnaround time

printf("\nAverage Waiting Time = %f\n", total\_waiting\_time/n); printf("Average Turnaround Time = %f\n", total\_turnaround\_time/n);

}

int main() { int n;

printf("Enter the number of processes: "); scanf("%d", &n);

struct process p[n];

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

printf("Enter burst time for process %d: ", i+1); scanf("%d", &p[i].burst);

printf("Enter arrival time for process %d: ", i+1); scanf("%d", &p[i].arrival);

p[i].pid = i+1;

}

sjf\_scheduling(p, n);

return 0;

}

### OUTPUT:

$ cc 2b.c

$ ./a.out

Enter the number of processes: 3 Enter burst time for process 1: 20 Enter arrival time for process 1: 0 Enter burst time for process 2: 7 Enter arrival time for process 2: 0 Enter burst time for process 3: 5 Enter arrival time for process 3: 3

|  |  |  |  |
| --- | --- | --- | --- |
| Process ID | Burst Time | Waiting Time | Turnaround Time |
| 3 | 5 | 0 | 5 |
| 2 | 7 | 5 | 12 |
| 1 | 20 | 12 | 32 |

Average Waiting Time = 5.666667 Average Turnaround Time = 16.333334

### Create a program to simulate the round-robin scheduling algorithm

**THEORY:**

A round-robin is a CPU scheduling algorithm that shares equal portions of resources in circular orders to each process and handles all processes without prioritization. In the round-robin, each process gets a fixed time interval of the slice to utilize the resources or execute its task called time quantum or time slice. Some of the round-robin processes are pre-empted if it executed in a given time slot, while the rest of the processes go back to the ready queue and wait to run in a circular order with the scheduled time slot until they complete their task. It removes the starvation for each process to achieve CPU scheduling by proper partitioning of the CPU.

### PROGRAM:

#include <stdio.h> struct process {

int pid; // Process ID int burst; // Burst time

int remaining; // Remaining burst time

};

void round\_robin\_scheduling(struct process p[], int n, int quantum) {

int time = 0, total\_waiting\_time = 0, total\_turnaround\_time = 0, count = n;

while (count > 0) {

for (int i = 0; i < n; i++) { if (p[i].remaining > 0) {

if (p[i].remaining <= quantum) { time += p[i].remaining;

total\_waiting\_time += time - p[i].burst; total\_turnaround\_time += time; p[i].remaining = 0;

count--;

} else {

time += quantum; p[i].remaining -= quantum;

}

}

}

}

// Print process details and their times

printf("\nProcess ID\tBurst Time\tWaiting Time\tTurnaround Time\n"); for (int i = 0; i < n; i++) {

printf("%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].burst, time - p[i].burst, time - p[i].burst - p[i].remaining);

}

// Print average waiting and turnaround time

printf("\nAverage Waiting Time = %f\n", (float)total\_waiting\_time/n); printf("Average Turnaround Time = %f\n", (float)total\_turnaround\_time/n);

}

int main() {

int n, quantum;

printf("Enter the number of processes: "); scanf("%d", &n);

struct process p[n];

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

printf("Enter burst time for process %d: ", i+1); scanf("%d", &p[i].burst);

p[i].remaining = p[i].burst; p[i].pid = i+1;

}

printf("Enter time quantum: "); scanf("%d", &quantum);

round\_robin\_scheduling(p, n, quantum);

return 0;

}

### OUTPUT:

$ cc 3a.c

$ ./a.out

Enter the number of processes: 3 Enter burst time for process 1: 20 Enter burst time for process 2: 3 Enter burst time for process 3: 5 Enter time quantum: 2

|  |  |  |  |
| --- | --- | --- | --- |
| Process ID | Burst Time | Waiting Time | Turnaround Time |
| 1 | 20 | 8 | 8 |
| 2 | 3 | 25 | 25 |
| 3 | 5 | 23 | 23 |

Average Waiting Time = 7.666667 Average Turnaround Time = 17.000000

### Develop a program that demonstrates Shared Memory and InterProcess Communication

**Theory:**

In this program, we first generate a key for the shared memory segment using the ftok function. We then create the shared memory segment using shmget, and attach it to our process using shmat.

We then write some data to the shared memory segment by copying it into the shm pointer. In the child process, we read the data from the shared memory segment by copying it into another pointer s and printing it out.

After the child process completes, we detach the shared memory segment from our process using shmdt, and then remove it from the system using shmctl.

Note that this is just a simple example, and in practice you will need to do error checking and handle other cases as well.

### Program:

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h> #include<sys/wait.h> #include <sys/ipc.h> #include <sys/shm.h> #include <string.h>

#define SHM\_SIZE 1024 // size of shared memory segment int main(int argc, char \*argv[]) {

int shmid; key\_t key; char \*shm, \*s;

// generate key for shared memory segment key = ftok(".", 'S');

if (key == -1) {

perror("ftok"); exit(1);

}

// create shared memory segment

shmid = shmget(key, SHM\_SIZE, IPC\_CREAT | 0666); if (shmid == -1) {

perror("shmget"); exit(1);

}

// attach shared memory segment to process shm = shmat(shmid, NULL, 0);

if (shm == (char \*) -1) {

perror("shmat"); exit(1);

}

// write data to shared memory segment s = shm;

strcpy(s, "Hello, world!");

// read data from shared memory segment in child process if (fork() == 0) {

s = shm;

printf("Child process read: %s\n", s); exit(0);

}

// wait for child process to complete wait(NULL);

// detach shared memory segment from process if (shmdt(shm) == -1) {

perror("shmdt"); exit(1);

}

// remove shared memory segment from system if (shmctl(shmid, IPC\_RMID, 0) == -1) {

perror("shmctl"); exit(1);

}

return 0;

}

### OUTPUT:

$cc 4.c

$ ./a.out

Child process read: Hello, world!

### Create a program that simulates the Producer-Consumer problem using semaphores.

**Theory:**

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

### Program:

#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <semaphore.h>

#define BUFFER\_SIZE 5

sem\_t mutex, full, empty; int buffer[BUFFER\_SIZE]; int in = 0, out = 0;

void \*producer(void \*arg) { int item;

for (int i = 0; i < 10; i++) {

item = rand() % 100; // generate random item to produce sem\_wait(&empty);

sem\_wait(&mutex); buffer[in] = item;

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

printf("Producer produced item %d\n", item); sem\_post(&mutex);

sem\_post(&full);

}

pthread\_exit(NULL);

}

void \*consumer(void \*arg) { int item;

for (int i = 0; i < 10; i++) { sem\_wait(&full); sem\_wait(&mutex); item = buffer[out];

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

printf("Consumer consumed item %d\n", item); sem\_post(&mutex);

sem\_post(&empty);

}

pthread\_exit(NULL);

}

int main() {

pthread\_t prod, cons; sem\_init(&mutex, 0, 1);

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

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

pthread\_create(&prod, NULL, producer, NULL); pthread\_create(&cons, NULL, consumer, NULL);

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

sem\_destroy(&mutex); sem\_destroy(&full); sem\_destroy(&empty);

return 0;

}

### OUTPUT:

$ cc 5.c

$ ./a.out

Producer produced item 83 Producer produced item 86 Producer produced item 77 Producer produced item 15 Producer produced item 93 Consumer consumed item 83 Consumer consumed item 86 Consumer consumed item 77 Consumer consumed item 15 Consumer consumed item 93 Producer produced item 35 Producer produced item 86 Producer produced item 92 Producer produced item 49 Producer produced item 21 Consumer consumed item 35 Consumer consumed item 86 Consumer consumed item 92 Consumer consumed item 49 Consumer consumed item 21

### Implement a program to depict the Dining Philosopher's problem concept.

**Theory:**

The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbours). A philosopher may pick up only one chopstick at a time. Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbour. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock.

### Program:

#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include<unistd.h>

#define N 5 // number of philosophers pthread\_mutex\_t forks[N];

void \*philosopher(void \*arg) { int id = \*(int\*)arg;

int left = id;

int right = (id + 1) % N;

while (1) {

printf("Philosopher %d is thinking\n", id); sleep(rand() % 5); // simulate thinking

printf("Philosopher %d is hungry\n", id); pthread\_mutex\_lock(&forks[left]); printf("Philosopher %d picked up left fork\n", id); pthread\_mutex\_lock(&forks[right]); printf("Philosopher %d picked up right fork\n", id);

printf("Philosopher %d is eating\n", id); sleep(rand() % 5); // simulate eating

printf("Philosopher %d is done eating\n", id); pthread\_mutex\_unlock(&forks[left]); printf("Philosopher %d put down left fork\n", id); pthread\_mutex\_unlock(&forks[right]); printf("Philosopher %d put down right fork\n", id);

}

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[N]; int ids[N]; srand(time(NULL));

for (int i = 0; i < N; i++) { pthread\_mutex\_init(&forks[i], NULL); ids[i] = i;

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

}

for (int i = 0; i < N; i++) { pthread\_join(threads[i], NULL); pthread\_mutex\_destroy(&forks[i]);

}

return 0;

}

### Output:

$cc 6.c

$ ./a.out

Philosopher 0 is thinking Philosopher 3 is thinking Philosopher 1 is thinking Philosopher 4 is thinking Philosopher 2 is thinking Philosopher 1 is hungry Philosopher 1 picked up left fork Philosopher 1 picked up right fork Philosopher 1 is eating Philosopher 2 is hungry Philosopher 3 is hungry Philosopher 3 picked up left fork Philosopher 3 picked up right fork Philosopher 3 is eating Philosopher 1 is done eating Philosopher 1 put down left fork Philosopher 1 put down right fork Philosopher 1 is thinking Philosopher 2 picked up left fork Philosopher 1 is hungry Philosopher 1 picked up left fork Philosopher 4 is hungry Philosopher 0 is hungry

1. **Develop a program for simulating the Banker's Algorithm to prevent deadlock avoidance**

### Theory:

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

**Program:** // Banker's Algorithm #include <stdio.h>

int main()

{

// P0, P1, P2, P3, P4 are the Process names here

int n, m, i, j, k;

n = 5; // Number of processes m = 3; // Number of resources

int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

{ 2, 0, 0 }, // P1

{ 3, 0, 2 }, // P2

{ 2, 1, 1 }, // P3

{ 0, 0, 2 } }; // P4

int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

{ 3, 2, 2 }, // P1

{ 9, 0, 2 }, // P2

{ 2, 2, 2 }, // P3

{ 4, 3, 3 } }; // P4

int avail[3] = { 3, 3, 2 }; // Available Resources int f[n], ans[n], ind = 0;

for (k = 0; k < n; k++) { f[k] = 0;

}

int need[n][m];

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

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

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

}

int y = 0;

for (k = 0; k < 5; k++) {

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

if (f[i] == 0) {

int flag = 0;

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

if (need[i][j] > avail[j]){ flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

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

avail[y] += alloc[i][y]; f[i] = 1;

}

}

}

}

int flag = 1;

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

{

if(f[i]==0)

{

flag=0;

printf("The following system is not safe"); break;

}

}

if(flag==1)

{

printf("Following is the SAFE Sequence\n"); for (i = 0; i < n - 1; i++)

printf(" P%d ->", ans[i]);

printf(" P%d", ans[n - 1]);

}

return (0);

}

### OUTPUT:

Following is the SAFE Sequence P1 -> P3 -> P4 -> P0 -> P2

### Create a program to simulate the Banker's Algorithm for deadlock prevention.

**Theory:**

A deadlock in the operating system is a situation of indefinite blocking of one or more processes that compete for resources.

Deadlock involves resources needed by two or more processes at the same time that cannot be shared. We can understand this from the above example, two cars require the road at the same time but it cannot be shared as it is one way.

* The processes must anticipate the maximum number of resources required as they en- ter the system, which may be done in a reasonable amount of time.
* In contrast to interactive systems, this method maintains a fixed number of processes.
* A specific number of resources must be available to distribute in order for this tech- nique to work. The algorithm would not function if a gadget broke and went unex-pectedly offline.
* The method will incur overhead costs since it must be invoked for each process when there are several processes and resources.

#include<stdio.h> int main()

{

// P0 , P1 , P2 , P3 , P4 are the Process names here int n , m , i , j , k;

n = 5; // Number of processes m = 3; // Number of resources

int alloc[ 5 ] [ 3 ] = { { 0 , 1 , 0 }, // P0 // Allocation Matrix

{ 2 , 0 , 0 } , // P1

{ 3 , 0 , 2 } , // P2

{ 2 , 1 , 1 } , // P3

{ 0 , 0 , 2 } } ; // P4

int max[ 5 ] [ 3 ] = { { 7 , 5 , 3 } , // P0 // MAX Matrix

{ 3 , 2 , 2 } , // P1

{ 9 , 0 , 2 } , // P2

{ 2 , 2 , 2 } , // P3

{ 4 , 3 , 3 } } ; // P4

int avail[3] = { 3 , 3 , 2 } ; // Available Resources int f[n] , ans[n] , ind = 0 ;

for (k = 0; k < n; k++) { f[k] = 0;

}

int need[n][m];

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

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

}

int y = 0;

for (k = 0; k < 5; k++){ for (i = 0; i < n; i++){

if (f[i] == 0){ int flag = 0;

for (j = 0; j < m; j++) { if(need[i][j] > avail[j]){

flag = 1; break;

}

}

if ( flag == 0 ) { ans[ind++] = i;

for (y = 0; y < m; y++) avail[y] += alloc[i][y] ;

f[i] = 1;

}

}

}

}

int flag = 1;

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

{

if(f[i] == 0)

{

flag = 0;

printf(" The following system is not safe "); break;

}

}

if (flag == 1)

{

printf(" Following is the SAFE Sequence \ n "); for (i = 0; i < n - 1; i++)

printf(" P%d -> " , ans[i]);

printf(" P%d ", ans[n - 1]);

}

return(0);

}

### Output:

Following is the SAFE Sequence P1 -> P3 -> P4 -> P0 -> P2

........................................................

Process execute din 1.33 seconds Press any key to continue.

### Implement a program to emulate first-fit contiguous memory allocation

**Theory:**

First Fit: In the first fit, the partition is allocated which is the first sufficient block from the top of Main Memory. It scans memory from the beginning and chooses the first available block that is large enough. Thus it allocates the first hole that is large enough.

In the first fit, the partition is allocated which is first sufficient from the top of Main Memory. Example :

Input : blockSize[] = {100, 500, 200, 300, 600};

processSize[] = {212, 417, 112, 426}; Output:

Process No. Process Size Block no.

|  |  |  |
| --- | --- | --- |
| 1 | 212 | 2 |
| 2 | 417 | 5 |
| 3 | 112 | 2 |
| 4 | 426 | Not Allocated |

Its advantage is that it is the fastest search as it searches only the first block i.e. enough to assign a process.

It may have problems of not allowing processes to take space even if it was possible to allocate. Consider the above example, process number 4 (of size 426) does not get memory. However it was possible to allocate memory if we had allocated using best fit allocation [block number 4 (of size 300) to process 1, block number 2 to process 2, block number 3 to process 3 and block number 5 to process 4].

### Program:// C implementation of First - Fit algorithm

#include<stdio.h>

// Function to allocate memory to

// blocks as per First fit algorithm

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

{

int i, j;

// Stores block id of the

// block allocated to a process int allocation[n];

// Initially no block is assigned to any process for(i = 0; i < n; i++)

{

allocation[i] = -1;

}

// pick each process and find suitable blocks

// according to its size ad assign to it

for (i = 0; i < n; i++) //here, n -> number of processes

{

for (j = 0; j < m; j++) //here, m -> number of blocks

{

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

{

// allocating block j to the ith process allocation[i] = j;

// Reduce available memory in this block. blockSize[j] -= processSize[i];

break; //go to the next process in the queue

}

}

}

printf("\nProcess No.\tProcess Size\tBlock no.\n"); for (int i = 0; i < n; i++)

{

printf(" %i\t\t\t", i+1); printf("%i\t\t\t\t", processSize[i]); if (allocation[i] != -1)

else

printf("%i", allocation[i] + 1);

printf("Not Allocated");

printf("\n");

}

}

// Driver code int main()

{

int m; //number of blocks in the memory

int n; //number of processes in the input queue int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

m = sizeof(blockSize) / sizeof(blockSize[0]);

n = sizeof(processSize) / sizeof(processSize[0]); firstFit(blockSize, m, processSize, n);

return 0 ;

}

### Output :

Process No. Process Size Block no.

|  |  |  |
| --- | --- | --- |
| 1 | 212 | 2 |
| 2 | 417 | 5 |
| 3 | 112 | 2 |
| 4 | 426 | Not Allocated |

### Develop a program for simulating paging table implementation and determining the actual physical address in memory

**Theory:**

In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be non-contiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

### Program:

#include<stdio.h> int main()

{

int ms, ps, nop, np, rempages, i, j, x, y, pa, offset; int s[10], fno[10][20];

printf("\nEnter the memory size -- "); scanf("%d",&ms);

printf("\nEnter the page size -- "); scanf("%d",&ps);

nop = ms/ps;

printf("\nThe no. of pages available in memory are -- %d ",nop); printf("\nEnter number of processes -- ");

scanf("%d",&np); rempages = nop; for(i=1;i<=np;i++)

{

printf("\nEnter no. of pages required for p[%d]-- ",i); scanf("%d",&s[i]);

if(s[i] >rempages)

{

printf("\nMemory is Full");

break;

}

rempages = rempages - s[i]; printf("\nEnter pagetable for p[%d] --- ",i); for(j=0;j<s[i];j++)

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

}

printf("\nEnter Logical Address to find Physical Address "); printf("\nEnter process no. and pagenumber and offset -- "); scanf("%d %d %d",&x,&y, &offset);

if(x>np || y>=s[i] || offset>=ps)

printf("\nInvalid Process or Page Number or offset"); else

{

pa=fno[x][y]\*ps+offset;

printf("\nThe Physical Address is -- %d",pa);

}

}

### OUTPUT:

Enter the memory size -- 1000 Enter the page size -- 100

The no. of pages available in memory are -- 10 Enter number of processes -- 3

Enter no. of pages required for p[1]-- 4 Enter pagetable for p[1] --- 8 6 9 5 Enter no. of pages required for p[2]-- 5 Enter pagetable for p[2] --- 1 4 5 7 3 Enter no. of pages required for p[3]-- 5 Memory is Full

Enter Logical Address to find Physical Address

Enter process no. and pagenumber and offset -- 2 3 60 The Physical Address is – 760

### Create a program to execute the FIFO page replacement algorithm.

**Theory:**

A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced.

### Program:

#include<stdio.h> #include<conio.h>

int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1; void main()

{

clrscr();

printf("\n \t\t\t FIFO PAGE REPLACEMENT ALGORITHM");

printf("\n Enter no.of frames. ");

scanf("%d",&nof);

printf("Enter number of Pages.\n"); scanf("%d",&nor);

printf("\n Enter the Page No. ");

for(i=0;i<nor;i++) scanf("%d",&ref[i]); printf("\nThe given Pages are:"); for(i=0;i<nor;i++) printf("%4d",ref[i]); for(i=1;i<=nof;i++)

frm[i]=-1;

printf("\n"); for(i=0;i<nor;i++)

{

flag=0;

printf("\n\t page no %d->\t",ref[i]);

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

{

if(frm[j]==ref[i])

{

flag=1; break;

}}

if(flag==0)

{

pf++; victim++;

victim=victim%nof; frm[victim]=ref[i]; for(j=0;j<nof;j++) printf("%4d",frm[j]);

} }

printf("\n\n\t\t No.of pages faults...%d",pf); getch();

}

### OUTPUT

FIFO PAGE REPLACEMENT ALGORITHM

Enter no.of frames. 4

Enter number of Pages. 6

Enter the Page No. 2 1 4 5 6 3

2 1 4 5 6 3

The given Pages are: 2 1 4 5 6 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| page no 2-> | 2 | -1 | -1 | -1 |
| page no 1-> | 2 | 1 | -1 | -1 |
| page no 4-> | 2 | 1 | 4 | -1 |
| page no 5-> | 2 | 1 | 4 | 5 |
| page no 6-> | 6 | 1 | 4 | 5 |
| page no 3-> | 6 | 3 | 4 | 5 |

No.of pages faults...6

### Implement a program for simulating the SCAN disk scheduling algorithm.

**Theory:**

In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

### Program:

#include<stdio.h> main()

{

int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0; clrscr();

printf("Enter the no of tracks to be traversed: "); scanf("%d'",&n);

printf("Enter the position of head: "); scanf("%d",&h);

t[0]=0;t[1]=h;

printf("Enter the tracks: "); for(i=2;i<n+2;i++) scanf("%d",&t[i]); for(i=0;i<n+2;i++)

{

for(j=0;j<(n+2)-i-1;j++)

{ if(t[j]>t[j+1])

{

temp=t[j]; t[j]=t[j+1]; t[j+1]=temp;

} } }

for(i=0;i<n+2;i++) if(t[i]==h)

j=i;k=i;

p=0;

while(t[j]!=0)

{

atr[p]=t[j]; j--;

p++;

}

atr[p]=t[j]; for(p=k+1;p<n+2;p++,k++) atr[p]=t[k+1]; for(j=0;j<n+1;j++)

{

if(atr[j]>atr[j+1])

d[j]=atr[j]-atr[j+1]; else

d[j]=atr[j+1]-atr[j]; sum+=d[j];

}

printf("\nAverage header movements:%f",(float)sum/n); getch();

}

### OUTPUT:

Enter the no of tracks to be traveresed: 5

Enter the position of head:

50

Enter the tracks:

40

50

60

70

80

Average header movements:10.000000

### Additional Program:

1. **Implement a program to simulate the following CPU scheduling algorithms and draw a Gantt Chart:**

**FCFS**

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm.

First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue. In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.

### PROGRAM

#include<stdio.h> struct Process {

int pid;

int arrival\_time; int burst\_time;

};

void fcfs(struct Process \*processes, int num\_processes) {

// Sort the processes by arrival time int i, j;

struct Process temp; for(i=0;i<num\_processes;i++) {

for(j=i+1;j<num\_processes;j++) { if(processes[i].arrival\_time > processes[j].arrival\_time) {

temp = processes[i]; processes[i] = processes[j]; processes[j] = temp;

}

}

}

int current\_time = 0;

printf("Gantt Chart:\n"); printf(" \n");

for(i=0;i<num\_processes;i++) {

// Wait for the process to arrive if(current\_time < processes[i].arrival\_time) {

printf("| IDLE |\t");

current\_time = processes[i].arrival\_time;

}

// Execute the process

printf("| P%d |\t", processes[i].pid); current\_time += processes[i].burst\_time;

}

printf("\n");

// Calculate average waiting time and turnaround time float avg\_waiting\_time = 0.0, avg\_turnaround\_time = 0.0; int completion\_time[num\_processes];

int waiting\_time[num\_processes];

int turnaround\_time[num\_processes];

completion\_time[0] = processes[0].burst\_time + processes[0].arrival\_time; waiting\_time[0] = 0;

turnaround\_time[0] = completion\_time[0] - processes[0].arrival\_time;

for(i=1;i<num\_processes;i++) {

// Calculate completion time

completion\_time[i] = completion\_time[i-1] + processes[i].burst\_time;

// Calculate waiting time

waiting\_time[i] = completion\_time[i-1] - processes[i].arrival\_time;

// Calculate turnaround time

turnaround\_time[i] = completion\_time[i] - processes[i].arrival\_time;

// Add to total waiting and turnaround time avg\_waiting\_time += waiting\_time[i]; avg\_turnaround\_time += turnaround\_time[i];

}

// Calculate average waiting and turnaround time avg\_waiting\_time /= num\_processes; avg\_turnaround\_time /= num\_processes;

printf("Process\t Arrival Time\t Burst Time\t Completion Time\t Waiting Time\t Turnaround Time\n");

for(i=0;i<num\_processes;i++) {

printf("%d\t\t %d\t\t %d\t\t %d\t\t\t %d\t\t %d\n", processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time, completion\_time[i], waiting\_time[i], turnaround\_time[i]);

}

printf("Average Waiting Time: %f\n", avg\_waiting\_time); printf("Average Turnaround Time: %f\n", avg\_turnaround\_time);

}

int main() {

int num\_processes, i;

printf("Enter the number of processes: "); scanf("%d", &num\_processes);

struct Process processes[num\_processes]; for(i=0;i<num\_processes;i++) {

printf("Enter arrival time and burst time for process %d: ", i+1); scanf("%d %d", &processes[i].arrival\_time, &processes[i].burst\_time); processes[i].pid = i+1;

}

fcfs(processes, num\_processes);

return 0;

}

### OUTPUT

$ cc 2a.c

$ ./a.out

Enter the number of processes: 3

Enter arrival time and burst time for process 1: 0 20 Enter arrival time and burst time for process 2: 0 7 Enter arrival time and burst time for process 3: 3 5 Gantt Chart:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| | P1 |  Process | | P2 | | | P3 |  Arrival Time | Burst Time | Completion Time | Waiting Time | |
| Turnaround Time | | | | | | |
| 1 | 0 | | 20 | 20 | 0 | 20 |
| 2 | 0 | | 7 | 27 | 20 | 27 |
| 3 | 3 | | 5 | 32 | 24 | 29 |

Average Waiting Time: 14.666667 Average Turnaround Time: 18.666666

### Write a C program to implement memory management using segmentation. Theory:

Segmentation is a memory management technique in which the memory is divided into the variable size parts. Each part is known as a segment which can be allocated to a process. The details about each segment are stored in a table called a segment table.

**Program:** #include <stdio.h> #include <math.h> int sost;

void gstinfo(); void ptladdr(); struct segtab

{

int sno; int baddr; int limit;

int val[10];

}st[10];

void gstinfo()

{

int i,j;

printf("\n\tEnter the size of the segment table: "); scanf("%d",&sost);

for(i=1;i<=sost;i++)

{

printf("\n\tEnter the information about segment: %d",i); st[i].sno = i;

printf("\n\tEnter the base Address: "); scanf("%d",&st[i].baddr); printf("\n\tEnter the Limit: "); scanf("%d",&st[i].limit); for(j=0;j<st[i].limit;j++)

{

printf("Enter the %d address Value: ",(st[i].baddr + j)); scanf("%d",&st[i].val[j]);

}

}

}

void ptladdr()

{

int i,swd,d=0,n,s,disp,paddr;

//clrscr();

printf("\n\n\t\t\t SEGMENT TABLE \n\n");

printf("\n\t SEG.NO\tBASE ADDRESS\t LIMIT \n\n"); for(i=1;i<=sost;i++)

printf("\t\t%d \t\t%d\t\t%d\n\n",st[i].sno,st[i].baddr,st[i].limit);

printf("\n\nEnter the logical Address: "); scanf("%d",&swd);

n=swd;

while (n != 0)

{

n=n/10; d++;

}

s = swd/pow(10,d-1);

disp = swd%(int)pow(10,d-1); if(s<=sost)

{

if(disp < st[s].limit)

{

paddr = st[s].baddr + disp; printf("\n\t\tLogical Address is: %d",swd);

printf("\n\t\tMapped Physical address is: %d",paddr); printf("\n\tThe value is: %d",( st[s].val[disp] ) );

}

else

printf("\n\t\tLimit of segment %d is high\n\n",s);

}

else

printf("\n\t\tInvalid Segment Address \n");

}

void main()

{

char ch;

//clrscr(); gstinfo(); do

{

ptladdr();

printf("\n\t Do U want to Continue(Y/N)");

//flushall(); scanf("%c",&ch);

}while (ch == 'Y' || ch == 'y' );

//getch();

}

### OUTPUT:

Enter the size of the segment table: 3 Enter the information about segment: 1

Enter the base Address: 4 Enter the Limit: 5

Enter the 4 address Value: 11

Enter the 5 address Value: 12

Enter the 6 address Value: 13

Enter the 7 address Value: 14

Enter the 8 address Value: 15

Enter the information about segment: 2 Enter the base Address: 5

Enter the Limit: 4

Enter the 5 address Value: 21

Enter the 6 address Value: 31

Enter the 7 address Value: 41

Enter the 8 address Value: 51

Enter the information about segment: 3 Enter the base Address: 3

Enter the Limit: 4

Enter the 3 address Value: 31

Enter the 4 address Value: 41

Enter the 5 address Value: 41

Enter the 6 address Value: 51 SEGMENT TABLE

SEG.NO BASE ADDRESS LIMIT

1 4 5

|  |  |  |
| --- | --- | --- |
| 2 | 5 | 4 |
| 3 | 3 | 4 |

Enter the logical Address: 3 Logical Address is: 3

Mapped Physical address is: 3 The value is: 31

### Write a C program using to simulate Optimal page replacement algorithms Theory:

Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. The basic idea is to replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.

### PROGRAM

#include<stdio.h>

int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1; int recent[10],optcal[50],count=0;

int optvictim(); void main()

{

printf("\n OPTIMAL PAGE REPLACEMENT ALGORITHN"); printf("\n. ");

printf("\nEnter the no.of frames:"); scanf("%d",&nof);

printf("Enter the no.of reference string:"); scanf("%d",&nor);

printf("Enter the reference string:"); for(i=0;i<nor;i++) scanf("%d",&ref[i]);

printf("\n OPTIMAL PAGE REPLACEMENT ALGORITHM"); printf("\n. ");

printf("\nThe given string"); printf("\n. \n");

for(i=0;i<nor;i++) printf("%4d",ref[i]); for(i=0;i<nof;i++)

{

frm[i]=-1; optcal[i]=0;

}

for(i=0;i<10;i++) recent[i]=0; printf("\n"); for(i=0;i<nor;i++)

{

flag=0;

printf("\n\tref no %d ->\t",ref[i]); for(j=0;j<nof;j++)

{

if(frm[j]==ref[i])

{

flag=1; break;

}

}

if(flag==0)

{

count++; if(count<=nof) victim++;

else victim=optvictim(i); pf++;

frm[victim]=ref[i]; for(j=0;j<nof;j++) printf("%4d",frm[j]);

}

}

printf("\n Number of page faults: %d",pf);

}

int optvictim(int index)

{

int i,j,temp,notfound; for(i=0;i<nof;i++)

{

notfound=1; for(j=index;j<nor;j++) if(frm[i]==ref[j])

{

notfound=0; optcal[i]=j; break;

}

if(notfound==1) return i;

}

temp=optcal[0]; for(i=1;i<nof;i++) if(temp<optcal[i]) temp=optcal[i]; for(i=0;i<nof;i++) if(frm[temp]==frm[i]) return i;

return 0;

}

### OUTPUT:

OPTIMAL PAGE REPLACEMENT ALGORITHN

......................... ........

Enter the no.of frames: 3

Enter the no.of reference string: 6 Enter the reference string

6 5 2 4 3 1

OPTIMAL PAGE REPLACEMENT ALGORITHM

................................

The given string

....................

6 5 2 4 3 1

|  |  |  |  |
| --- | --- | --- | --- |
| ref no 6 -> | 6 | -1 | -1 |
| ref no 5 -> | 6 | 5 | -1 |
| ref no 2 -> | 6 | 5 | 2 |
| ref no 4 -> | 4 | 5 | 2 |
| ref no 3 -> | 3 | 5 | 2 |
| ref no 1 -> | 1 | 5 | 2 |

Number of page faults: 6

### Write a C program to simulate FCFS disk scheduling algorithms

**Theory:**

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service.

**Program:** #include<stdio.h> int main()

{

int queue[20],n,head,i,j,k,seek=0,max,diff; float avg;

printf("Enter the max range of disk\n"); scanf("%d",&max);

printf("Enter the size of queue request\n"); scanf("%d",&n);

printf("Enter the queue of disk positions to be read\n"); for(i=1;i<=n;i++)

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

printf("Enter the initial head position\n"); scanf("%d",&head);

queue[0]=head; for(j=0;j<=n-1;j++)

{

diff=abs(queue[j+1]-queue[j]); seek+=diff;

printf("Disk head moves from %d to %d with %d\n",queue[j],queue[j+1],diff);

}

printf("Total seek time is %d\n",seek); avg=seek/(float)n;

printf("Average seek time is %f\n",avg); return 0;

}

### OUTPUT:

Enter the max range of disk 200

Enter the size of queue request 8

Enter the queue of disk positions to be read 90 120 35 122 38 128 65 68

Enter the initial head position 50

Disk head moves from 50 to 90 with 40

|  |  |
| --- | --- |
| Disk head moves from 90 to 120 with | 30 |
| Disk head moves from 120 to 35 with | 85 |
| Disk head moves from 35 to 122 with | 87 |
| Disk head moves from 122 to 38 with | 84 |
| Disk head moves from 38 to 128 with | 90 |
| Disk head moves from 128 to 65 with | 63 |
| Disk head moves from 65 to 68 with 3 |  |
| Total seek time is 482 |  |
| Average seek time is 60.250000 |  |

### VIVA QUESTIONS

1. What is a boot-strap program?
2. What are various scheduling queues?
3. What is context switching?
4. What is critical section problem?
5. What is a safe state and a safe sequence?
6. Explain the following terms and their working with diagram
   1. Time sharing b) Distributed system c) Real-time
7. Explain Dual Mode operation?
8. What are multithreading models?
9. Distinguish between process and program.
10. Define a process. Draw and explain process state diagram.
11. State dining philosopher’s problem and give a solution using semaphores. Write structure of philosopher.
12. What is starvation and aging?
13. What are the necessary and sufficient conditions for deadlock to occur
14. Explain semaphore in detail. Also explain the use of counting semaphore with respect to client server environment.
15. Explain readers and writer problem in detail and show how this problem can be solved using semaphores.
16. Define the following terms: i) Thrashing ii) Belady’s anomaly
17. Explain Indexed File Allocation Method

20. What is Throughput, Turnaround time, waiting time and Response time?

Round robin

#include<stdio.h>

int main()

{

//Input no of processed

int n;

printf("Enter Total Number of Processes:");

scanf("%d", &n);

int wait\_time = 0, ta\_time = 0, arr\_time[n], burst\_time[n], temp\_burst\_time[n];

int x = n;

//Input details of processes

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

{

printf("Enter Details of Process %d \n", i + 1);

printf("Arrival Time: ");

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

printf("Burst Time: ");

scanf("%d", &burst\_time[i]);

temp\_burst\_time[i] = burst\_time[i];

}

//Input time slot

int time\_slot;

printf("Enter Time Slot:");

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

//Total indicates total time

//counter indicates which process is executed

int total = 0, counter = 0,i;

printf("Process ID Burst Time Turnaround Time Waiting Time\n");

for(total=0, i = 0; x!=0; )

{

// define the conditions

if(temp\_burst\_time[i] <= time\_slot && temp\_burst\_time[i] > 0)

{

total = total + temp\_burst\_time[i];

temp\_burst\_time[i] = 0;

counter=1;

}

else if(temp\_burst\_time[i] > 0)

{

temp\_burst\_time[i] = temp\_burst\_time[i] - time\_slot;

total += time\_slot;

}

if(temp\_burst\_time[i]==0 && counter==1)

{

x--; //decrement the process no.

printf("\nProcess No %d \t\t %d\t\t\t\t %d\t\t\t %d", i+1, burst\_time[i],

total-arr\_time[i], total-arr\_time[i]-burst\_time[i]);

wait\_time = wait\_time+total-arr\_time[i]-burst\_time[i];

ta\_time += total -arr\_time[i];

counter =0;

}

if(i==n-1)

{

i=0;

}

else if(arr\_time[i+1]<=total)

{

i++;

}

else

{

i=0;

}

}

float average\_wait\_time = wait\_time \* 1.0 / n;

float average\_turnaround\_time = ta\_time \* 1.0 / n;

printf("\nAverage Waiting Time:%f", average\_wait\_time);

printf("\nAvg Turnaround Time:%f", average\_turnaround\_time);

return 0;

}

#include<stdio.h>

int main() {

int time, burst\_time[10], at[10], sum\_burst\_time = 0, smallest, n, i;

int sumt = 0, sumw = 0;

printf("enter the no of processes : ");

scanf("%d", & n);

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

printf("the arrival time for process P%d : ", i + 1);

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

printf("the burst time for process P%d : ", i + 1);

scanf("%d", & burst\_time[i]);

sum\_burst\_time += burst\_time[i];

}

burst\_time[9] = 9999;

for (time = 0; time < sum\_burst\_time;) {

smallest = 9;

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

if (at[i] <= time && burst\_time[i] > 0 && burst\_time[i] < burst\_time[smallest])

smallest = i;

}

printf("P[%d]\t|\t%d\t|\t%d\n", smallest + 1, time + burst\_time[smallest] - at[smallest], time - at[smallest]);

sumt += time + burst\_time[smallest] - at[smallest];

sumw += time - at[smallest];

time += burst\_time[smallest];

burst\_time[smallest] = 0;

}

printf("\n\n average waiting time = %f", sumw \* 1.0 / n);

printf("\n\n average turnaround time = %f", sumt \* 1.0 / n);

return 0;

}