**FCFS**

#include <stdio.h>

// Define a structure to store process number and burst time

struct fcfs {

int pno; // Process Number

int btime; // Burst Time

};

int main() {

int n, i, wait;

float avg, avg1, avg2, tc;

struct fcfs f[20];

// Get the number of processes

printf("ENTER THE NUMBER OF PROCESS: ");

scanf("%d", &n);

// Input each process's number and burst time

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

printf("\nENTER THE PROCESS NUMBER: ");

scanf("%d", &f[i].pno);

printf("ENTER THE BURST TIME: ");

scanf("%d", &f[i].btime);

}

// Print header for the process table

printf("THE FOLLOWING ARE THE PROCESS DETAILS\n");

printf("PROCESS NO\tBURST TIME\tWAITING TIME\tTURNAROUND TIME\n");

// Initialize accumulators

avg = 0; // Sum of waiting times

avg1 = 0; // Average waiting time

avg2 = 0; // Average turnaround time

wait = 0; // Waiting time for first process is 0

tc = 0; // Total turnaround time

// Loop to calculate and display each process's details

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

printf("%d\t\t%d\t\t%d\t\t%d\n", f[i].pno, f[i].btime, wait, wait + f[i].btime);

// Add turnaround time to total

tc = tc + (wait + f[i].btime);

// Accumulate waiting time

avg = avg + wait;

// Update waiting time for next process

wait = wait + f[i].btime;

}

// Compute averages

avg1 = avg / n;

avg2 = tc / n;

// Display average waiting and turnaround times

printf("AVERAGE WAITING TIME FOR THE PROCESS IS: %f\n", avg1);

printf("AVERAGE TURNAROUND TIME FOR THE PROCESS IS: %f\n", avg2);

return 0;

}

**SJF**

#include <stdio.h>

// Define a structure to store process number and burst time

struct s {

int pno; // Process Number

int btime; // Burst Time

} s[20];

int main() {

int i, j, n, wait, ptemp, ptemp1;

float avg, avg1, avg2, tc;

// Initialize variables

ptemp = 0;

ptemp1 = 0;

wait = 0;

avg = 0;

avg1 = 0;

avg2 = 0;

tc = 0;

// Get number of processes

printf("ENTER THE NUMBER OF PROCESS: ");

scanf("%d", &n);

// Input process number and burst time

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

printf("\nENTER THE PROCESS NUMBER: ");

scanf("%d", &s[i].pno);

printf("ENTER THE BURST TIME: ");

scanf("%d", &s[i].btime);

}

// Sort processes by burst time using simple bubble sort (SJF logic)

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

for(j = i + 1; j < n; j++) {

if(s[i].btime > s[j].btime) {

// Swap burst times

ptemp1 = s[i].btime;

s[i].btime = s[j].btime;

s[j].btime = ptemp1;

// Swap process numbers accordingly

ptemp = s[i].pno;

s[i].pno = s[j].pno;

s[j].pno = ptemp;

}

}

}

// Print table header

printf("\nTHE FOLLOWING ARE THE PROCESS DETAILS:\n");

printf("\nPROCESS NO\tBURST TIME\tWAITING TIME\tTURN AROUND TIME\n");

// Calculate waiting time and turnaround time

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

printf("%d\t\t%d\t\t%d\t\t%d\n", s[i].pno, s[i].btime, wait, wait + s[i].btime);

tc += (wait + s[i].btime); // Accumulate turnaround time

avg += wait; // Accumulate waiting time

wait += s[i].btime; // Update waiting time for next process

}

// Calculate average waiting and turnaround times

avg1 = avg / n;

avg2 = tc / n;

// Display average results

printf("\nTHE AVERAGE WAITING TIME OF PROCESS IS: %f", avg1);

printf("\nTHE AVERAGE TURN AROUND TIME OF PROCESS IS: %f\n", avg2);

return 0;

}

**Priority**

#include <stdio.h>

// Structure to store process details

struct pr {

int prino; // Priority Number (lower number = higher priority)

int pno; // Process Number

int btime; // Burst Time

};

int main() {

int n, i, j;

int wait = 0, ptemp, prtemp, btemp;

float avg = 0, avg1 = 0, avg2 = 0, tc = 0;

struct pr p[15];

printf("PRIORITY SCHEDULING\n");

// Input number of processes

printf("ENTER THE NUMBER OF PROCESSES: ");

scanf("%d", &n);

// Input details for each process

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

printf("\nENTER PROCESS NUMBER: ");

scanf("%d", &p[i].pno);

printf("ENTER PRIORITY NUMBER (lower = higher priority): ");

scanf("%d", &p[i].prino);

printf("ENTER BURST TIME: ");

scanf("%d", &p[i].btime);

}

// Sort processes based on priority using simple bubble sort

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

for(j = i + 1; j < n; j++) {

if(p[i].prino > p[j].prino) {

// Swap priority

prtemp = p[i].prino;

p[i].prino = p[j].prino;

p[j].prino = prtemp;

// Swap burst time

btemp = p[i].btime;

p[i].btime = p[j].btime;

p[j].btime = btemp;

// Swap process number

ptemp = p[i].pno;

p[i].pno = p[j].pno;

p[j].pno = ptemp;

}

}

}

// Output process table

printf("\nPROCESS NO\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME\n");

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

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

p[i].pno, p[i].prino, p[i].btime, wait, wait + p[i].btime);

tc += wait + p[i].btime; // Total turnaround time

avg += wait; // Total waiting time

wait += p[i].btime; // Update wait for next process

}

// Calculate averages

avg1 = avg / n;

avg2 = tc / n;

// Display averages

printf("\nAVERAGE WAITING TIME: %f", avg1);

printf("\nAVERAGE TURNAROUND TIME: %f\n", avg2);

return 0;

}

**Robin**

#include <stdio.h>

// Define structure for Round Robin

struct rr {

int pno; // Process number

int btime; // Burst time

int com; // Completion time

int wt; // Waiting time

int cal; // Cumulative waiting time

char st; // Status: 'a' = active, 'p' = processed

};

int main() {

int n, i, j, wait = 0, temp = 4; // Time quantum is 4 ms

float avg = 0, avg1 = 0;

struct rr r[20];

// Input number of processes

printf("\nENTER THE NUMBER OF PROCESSES: ");

scanf("%d", &n);

// Input burst times and initialize other fields

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

printf("\nENTER THE PROCESS NUMBER: ");

scanf("%d", &r[i].pno);

printf("ENTER THE BURST TIME: ");

scanf("%d", &r[i].btime);

r[i].st = 'a'; // 'a' means active

r[i].wt = 0;

r[i].com = 0;

r[i].cal = 0;

}

printf("\nTIME QUANTUM = %d\n", temp);

printf("PROCESS\tBURST\tWAITING\tCOMPLETION\n");

// Round Robin Scheduling

int done;

do {

done = 1; // Assume all done

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

if(r[i].st == 'a') {

done = 0; // Still some process left

if(r[i].btime > temp) {

r[i].wt = wait;

r[i].cal += (wait - r[i].com);

r[i].com = wait + temp;

printf("%d\t%d\t%d\t%d\n", r[i].pno, temp, wait, wait + temp);

wait += temp;

r[i].btime -= temp;

} else {

r[i].wt = wait;

r[i].cal += (wait - r[i].com);

r[i].com = wait + r[i].btime;

printf("%d\t%d\t%d\t%d\n", r[i].pno, r[i].btime, wait, wait + r[i].btime);

wait += r[i].btime;

r[i].btime = 0;

r[i].st = 'p'; // Mark process as done

}

}

}

} while(!done);

// Calculate averages

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

avg += r[i].cal; // Total waiting time

avg1 += r[i].com; // Total turnaround time

}

avg /= n;

avg1 /= n;

// Output averages

printf("\nAVERAGE WAITING TIME = %.2f ms", avg);

printf("\nAVERAGE TURNAROUND TIME = %.2f ms\n", avg1);

return 0;

}

**UNIX**

#include <fcntl.h> // For open(), O\_RDONLY, etc.

#include <stdio.h> // For printf(), etc.

#include <unistd.h> // For fork(), write(), read(), close()

#include <sys/wait.h> // For wait()

int main()

{

int fp, exitstatus;

char chr = 'A';

int pid;

// Create a child process

pid = fork();

if(pid == 0) // Child process

{

fp = open("testfile.txt", O\_WRONLY | O\_CREAT, 0644); // Open file in write mode

if(fp < 0) {

perror("File open error in child");

return 1;

}

printf("\nIN CHILD: CHR is %c\n", chr);

chr = 'B'; // Modify character

write(fp, &chr, 1); // Write one byte

printf("CHILD PROCESS ID: %d\n", getpid());

printf("CHILD: CHR AFTER CHANGE IS %c\n", chr);

printf("CHILD TERMINATED\n");

close(fp);

}

else // Parent process

{

wait(&exitstatus); // Wait for child to finish

fp = open("testfile.txt", O\_RDONLY); // Open file in read mode

if(fp < 0) {

perror("File open error in parent");

return 1;

}

read(fp, &chr, 1); // Read one byte

printf("\nPARENT PROCESS ID: %d\n", getpid());

printf("CHR AFTER PARENT READ IS: %c\n", chr);

close(fp);

}

return 0;

}

**Lseek**

#include <fcntl.h> // For open()

#include <unistd.h> // For read(), write(), lseek()

#include <stdio.h> // For perror()

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

{

char buf;

int size, fd;

if(argc != 2) {

write(STDERR\_FILENO, "Usage: ./reverse <filename>\n", 29);

return 1;

}

fd = open(argv[1], O\_RDONLY);

if(fd < 0) {

perror("Error opening file");

return 1;

}

// Move file pointer to the end and get the size

size = lseek(fd, -1, SEEK\_END);

if(size < 0) {

perror("Error seeking file");

close(fd);

return 1;

}

while(size-- >= 0) {

read(fd, &buf, 1); // Read 1 character

write(STDOUT\_FILENO, &buf, 1); // Print to stdout

lseek(fd, -2, SEEK\_CUR); // Move back 2 positions

}

close(fd);

return 0;

}

**DIR**

#include <stdio.h>

#include <dirent.h>

#include <stdlib.h>

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

{

DIR \*dir;

struct dirent \*directory;

if (argc != 2) {

printf("Usage: %s <directory path>\n", argv[0]);

return 1;

}

dir = opendir(argv[1]);

if (dir == NULL) {

perror("Unable to open directory");

return 1;

}

while ((directory = readdir(dir)) != NULL) {

printf("%lu %s\n", (unsigned long)directory->d\_ino, directory->d\_name);

}

closedir(dir);

return 0;

}

**Banker**

#include <stdio.h>

int main() {

int p, r, i, j;

int alloc[10][10], max[10][10], need[10][10], avail[10];

int finish[10] = {0}, safeSeq[10];

int count = 0;

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

scanf("%d", &p);

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

scanf("%d", &r);

printf("Enter allocation matrix:\n");

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

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

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

}

}

printf("Enter maximum claim matrix:\n");

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

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

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

}

}

printf("Enter available resources:\n");

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

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

}

// Calculate need matrix

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

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

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

}

}

// Banker's Algorithm

while (count < p) {

int found = 0;

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

if (!finish[i]) {

int canAllocate = 1;

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

if (need[i][j] > avail[j]) {

canAllocate = 0;

break;

}

}

if (canAllocate) {

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

avail[j] += alloc[i][j];

}

safeSeq[count++] = i;

finish[i] = 1;

found = 1;

}

}

}

if (!found) {

break;

}

}

if (count == p) {

printf("\nThe system is in a SAFE STATE.\nSafe Sequence: ");

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

printf("P%d ", safeSeq[i]);

}

printf("\n");

} else {

printf("\nThe system is in an UNSAFE STATE!\n");

}

return 0;

}

**SEMO**

#include <stdio.h>

#include <stdlib.h>

#define SIZE 20

typedef int semaphore;

semaphore mutex = 1;

semaphore full = 0;

semaphore empty;

int buffer[SIZE];

int in = 0;

void producer() {

int item;

if (empty != 0 && mutex == 1) {

mutex = 0;

printf("Enter the item to produce: ");

scanf("%d", &item);

buffer[in++] = item;

full++;

empty--;

printf("Item produced: %d\n", item);

printf("Buffer: ");

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

printf("%d ", buffer[i]);

printf("\nBuffer size: %d\n", in);

mutex = 1;

} else {

printf("Buffer is full!\n");

}

}

void consumer() {

if (full != 0 && mutex == 1) {

mutex = 0;

int item = buffer[0];

for (int i = 0; i < in - 1; i++)

buffer[i] = buffer[i + 1];

in--;

full--;

empty++;

printf("Item consumed: %d\n", item);

printf("Buffer: ");

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

printf("%d ", buffer[i]);

printf("\nBuffer size: %d\n", in);

mutex = 1;

} else {

printf("Buffer is empty!\n");

}

}

int main() {

int opt;

printf("Enter buffer capacity (max 20): ");

scanf("%d", &empty);

if (empty > SIZE || empty <= 0) {

printf("Invalid size. Exiting.\n");

return 1;

}

do {

printf("\nMenu:\n1. Producer\n2. Consumer\n3. Quit\nEnter option: ");

scanf("%d", &opt);

switch (opt) {

case 1: producer(); break;

case 2: consumer(); break;

case 3: exit(0);

default: printf("Invalid option. Try again.\n");

}

} while (opt != 3);

return 0;

}

**PIPES**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h> // For strlen()

int main() {

int pipefd[2];

char buffer[80];

// Create a pipe

if (pipe(pipefd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

// Create a child process

pid\_t pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

// Parent process (writer)

if (pid > 0) {

close(pipefd[0]); // Close reading end

const char\* message = "Hello from the parent process!\n";

ssize\_t bytes\_written = write(pipefd[1], message, strlen(message) + 1); // +1 to include '\0'

if (bytes\_written == -1) {

perror("write");

exit(EXIT\_FAILURE);

}

printf("Parent: Sent message - %s\n", message);

close(pipefd[1]); // Close writing end

}

// Child process (reader)

else {

close(pipefd[1]); // Close writing end

ssize\_t bytes\_read = read(pipefd[0], buffer, sizeof(buffer));

if (bytes\_read == -1) {

perror("read");

exit(EXIT\_FAILURE);

} else if (bytes\_read == 0) {

printf("Child: End of pipe reached (no data).\n");

} else {

printf("Child: Received message - %s\n", buffer);

}

close(pipefd[0]); // Close reading end

}

return 0;

}

**FIFO**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/types.h>

#define FIFO\_NAME "myfifo"

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

char buffer[80];

int fd;

// Create the FIFO (named pipe) if it does not exist

mkfifo(FIFO\_NAME, 0666); // rw-rw-rw-

// Check command line argument

if (argc == 2 && strcmp(argv[1], "writer") == 0) {

// Writer process

fd = open(FIFO\_NAME, O\_WRONLY);

if (fd == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

const char\* message = "Hello from the writer process!\n";

ssize\_t bytes\_written = write(fd, message, strlen(message) + 1);

if (bytes\_written == -1) {

perror("write");

exit(EXIT\_FAILURE);

}

printf("Writer: Sent message - %s\n", message);

close(fd);

}

else if (argc == 2 && strcmp(argv[1], "reader") == 0) {

// Reader process

fd = open(FIFO\_NAME, O\_RDONLY);

if (fd == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

ssize\_t bytes\_read = read(fd, buffer, sizeof(buffer));

if (bytes\_read == -1) {

perror("read");

exit(EXIT\_FAILURE);

} else if (bytes\_read == 0) {

printf("Reader: End of pipe reached (no data).\n");

} else {

printf("Reader: Received message - %s\n", buffer);

}

close(fd);

}

else {

fprintf(stderr, "Usage: %s [writer|reader]\n", argv[0]);

exit(EXIT\_FAILURE);

}

return 0;

}

**MSG QUEUE**

#ifndef COMMON\_H

#define COMMON\_H

struct my\_msg {

long msg\_type;

char some\_text[100]; // MAX\_TEXT should match this

};

#endif

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include "common.h"

#define MAX\_TEXT 100

int main() {

int msgid;

struct my\_msg some\_data;

int running = 1;

key\_t key = ftok("/tmp/myqueue", 65);

msgid = msgget(key, 0666 | IPC\_CREAT);

if (msgid == -1) {

perror("msgget");

exit(EXIT\_FAILURE);

}

while (running) {

printf("Enter message (or 'end' to quit): ");

fgets(some\_data.some\_text, MAX\_TEXT, stdin);

some\_data.some\_text[strcspn(some\_data.some\_text, "\n")] = '\0';

some\_data.msg\_type = 1;

if (strncmp(some\_data.some\_text, "end", 3) == 0) {

running = 0;

}

if (msgsnd(msgid, &some\_data, strlen(some\_data.some\_text) + 1, 0) == -1) {

perror("msgsnd");

exit(EXIT\_FAILURE);

}

printf("Message sent!\n");

}

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include "common.h"

#define MAX\_TEXT 100

int main() {

int msgid;

struct my\_msg some\_data;

long int msg\_to\_rec = 0;

key\_t key = ftok("/tmp/myqueue", 65);

msgid = msgget(key, 0666 | IPC\_CREAT);

if (msgid == -1) {

perror("msgget");

exit(EXIT\_FAILURE);

}

while (1) {

if (msgrcv(msgid, &some\_data, MAX\_TEXT, msg\_to\_rec, 0) == -1) {

perror("msgrcv");

exit(EXIT\_FAILURE);

}

printf("Received message: %s\n", some\_data.some\_text);

if (strncmp(some\_data.some\_text, "end", 3) == 0) {

break;

}

}

return 0;

}

**SHARED MEM**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <fcntl.h>

#include <errno.h>

#include <unistd.h>

#define SHM\_SIZE 1024

#define SHM\_KEY\_FILE "shared\_memory.txt"

int create\_shared\_memory(key\_t key, size\_t size) {

int shmid;

// Try creating shared memory

if ((shmid = shmget(key, size, IPC\_CREAT | IPC\_EXCL | 0666)) < 0) {

if (errno == EEXIST) {

// Already exists, try attaching

shmid = shmget(key, size, 0666);

if (shmid < 0) {

perror("shmget (existing)");

exit(EXIT\_FAILURE);

}

} else {

perror("shmget");

exit(EXIT\_FAILURE);

}

}

return shmid;

}

void\* attach\_shared\_memory(int shmid) {

void \*data;

data = shmat(shmid, NULL, 0);

if (data == (void \*)-1) {

perror("shmat");

exit(EXIT\_FAILURE);

}

return data;

}

void detach\_shared\_memory(void \*data) {

if (shmdt(data) < 0) {

perror("shmdt");

exit(EXIT\_FAILURE);

}

}

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

// Ensure the key file exists

int fd = open(SHM\_KEY\_FILE, O\_CREAT | O\_RDWR, 0666);

if (fd == -1) {

perror("open key file");

exit(EXIT\_FAILURE);

}

close(fd);

key\_t key = ftok(SHM\_KEY\_FILE, 'X');

if (key == -1) {

perror("ftok");

exit(EXIT\_FAILURE);

}

int shmid = create\_shared\_memory(key, SHM\_SIZE);

char \*data = (char \*)attach\_shared\_memory(shmid);

if (argc > 1 && strcmp(argv[1], "-w") == 0) {

// Writer

printf("Writer process attached.\n");

strcpy(data, "Hello from the writer process!");

printf("Writer wrote: %s\n", data);

} else if (argc == 1 || strcmp(argv[1], "-r") == 0) {

// Reader

printf("Reader process attached.\n");

printf("Reader read: %s\n", data);

} else {

fprintf(stderr, "Usage: %s [-w] for writer or [-r] for reader (default)\n", argv[0]);

exit(EXIT\_FAILURE);

}

detach\_shared\_memory(data);

return 0;

}

**PAGING**

#include <stdio.h> // For standard input/output functions

int main() {

int ms, ps, nop, np, rempages, i, j; // Variables for memory size, page size, etc.

int x, y, pa, offset; // Variables for address translation

int s[10], fno[10][20]; // s[i] = number of pages for process i; fno[i][j] = frame number for j-th page of i-th process

// Input total memory size

printf("\nEnter the memory size: ");

scanf("%d", &ms);

// Input page size

printf("Enter the page size: ");

scanf("%d", &ps);

// Calculate the number of pages available in memory

nop = ms / ps;

printf("The number of pages available in memory: %d\n", nop);

// Input number of processes

printf("Enter number of processes: ");

scanf("%d", &np);

rempages = nop; // Initialize remaining pages with total pages available

// Loop through each process to allocate pages and store frame numbers

for (i = 1; i <= np; i++) {

printf("\nEnter number of pages required for process P[%d]: ", i);

scanf("%d", &s[i]); // Store number of pages required for process i

// Check if memory has enough pages left

if (s[i] > rempages) {

printf("Memory is Full\n");

break; // Stop further allocation if not enough pages

}

rempages -= s[i]; // Deduct allocated pages from remaining

// Input the frame number mapping for each page of process i

printf("Enter page table for process P[%d] (frame numbers):\n", i);

for (j = 0; j < s[i]; j++) {

printf("Page %d: ", j);

scanf("%d", &fno[i][j]); // Store the frame number for page j

}

}

// Prompt for logical address to convert to physical address

printf("\nEnter Logical Address to find Physical Address");

printf("\nEnter process number, page number, and offset: ");

scanf("%d %d %d", &x, &y, &offset);

// Validate the process number, page number, and offset

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

printf("Invalid Process or Page Number or Offset\n");

} else {

// Calculate physical address using: frame number \* page size + offset

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

printf("The Physical Address is: %d\n", pa);

}

return 0;

}

**SEGMENT**

#include <stdio.h>

#include <stdlib.h>

struct list {

int seg;

int base;

int limit;

struct list \*next;

};

struct list \*p = NULL;

void insert(struct list \*\*q, int base, int limit, int seg) {

struct list \*newNode = (struct list \*)malloc(sizeof(struct list));

newNode->seg = seg;

newNode->base = base;

newNode->limit = limit;

newNode->next = NULL;

if (\*q == NULL) {

\*q = newNode;

} else {

struct list \*temp = \*q;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

int find\_limit(struct list \*q, int seg) {

while (q != NULL && q->seg != seg) {

q = q->next;

}

if (q == NULL) return -1;

return q->limit;

}

int find\_base(struct list \*q, int seg) {

while (q != NULL && q->seg != seg) {

q = q->next;

}

if (q == NULL) return -1;

return q->base;

}

int main() {

int seg, offset, limit, base, c, s, physical;

printf("Enter segment table\n");

printf("Enter -1 as segment number to stop\n");

while (1) {

printf("Enter segment number: ");

scanf("%d", &seg);

if (seg == -1) break;

printf("Enter base value: ");

scanf("%d", &base);

printf("Enter limit value: ");

scanf("%d", &limit);

insert(&p, base, limit, seg);

}

printf("Enter offset: ");

scanf("%d", &offset);

printf("Enter segment number: ");

scanf("%d", &seg);

c = find\_limit(p, seg);

s = find\_base(p, seg);

if (c == -1 || s == -1) {

printf("Segment not found\n");

} else if (offset < c) {

physical = s + offset;

printf("Address in physical memory: %d\n", physical);

} else {

printf("Error: Offset exceeds limit\n");

}

return 0;

}

**PAGE REPLACEMENT**

#include <stdio.h>

// Global declarations

int n, pg[30], fr[10]; // 'n' is number of pages, 'pg' stores the page sequence, 'fr' represents frame slots

// Function declarations

void fifo();

void optimal();

void lru();

int main() {

int i, ch;

// Input total number of pages

printf("\n Enter total number of pages: ");

scanf("%d", &n);

// Input page reference sequence

printf("\n Enter sequence: ");

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

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

// Menu for selecting algorithm

do {

printf("\nMENU\n");

printf("1) FIFO\n");

printf("2) OPTIMAL\n");

printf("3) LRU\n");

printf("4) EXIT\n");

printf("ENTER YOUR CHOICE: ");

scanf("%d", &ch);

switch (ch) {

case 1:

fifo(); // Call FIFO

break;

case 2:

optimal(); // Call OPTIMAL

break;

case 3:

lru(); // Call LRU

break;

case 4:

break;

default:

printf("Invalid choice\n");

}

} while (ch != 4); // Continue until EXIT is selected

return 0;

}

// FIFO Algorithm

void fifo() {

int i, f = 0, r = 0, s = 0, count = 0, flag, num, psize;

// Frame size input

printf("\n Enter size of page frame: ");

scanf("%d", &psize);

// Initialize all frames to -1

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

fr[i] = -1;

}

// Traverse page sequence

while (s < n) {

flag = 0;

num = pg[s];

// Check if page already exists in frame

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

if (num == fr[i]) {

s++; // Move to next page

flag = 1; // Page hit

break;

}

}

// If page not found (page fault)

if (flag == 0) {

if (r < psize) {

fr[r] = pg[s]; // Fill the frame initially

r++;

s++;

count++; // Increase page fault count

} else {

if (f < psize) {

fr[f] = pg[s]; // Replace oldest page (FIFO)

s++;

f++;

count++;

} else {

f = 0; // Reset pointer to start

}

}

}

// Print current frame state

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

printf("%d \t", fr[i]);

}

}

// Output total page faults

printf("\nPage fault = %d", count);

}

// OPTIMAL Algorithm

void optimal() {

int count[20], i, j, k = 0, fault = 0, f, flag, temp, current, c, dist, max, m, cnt, p, x;

// Input frame size

printf("\n Enter Frame size: ");

scanf("%d", &f);

// Initialize frames and count array

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

count[i] = 0;

fr[i] = -1;

}

// Traverse each page in sequence

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

flag = 0;

temp = pg[i];

// Check if page is already in frame

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

if (temp == fr[j]) {

flag = 1; // Page hit

break;

}

}

// If not in frame

if (flag == 0 && k < f) {

fault++;

fr[k] = temp; // Fill available slot

k++;

} else if (flag == 0 && k == f) {

fault++;

// Calculate distance to next use

for (cnt = 0; cnt < f; cnt++) {

count[cnt] = 0;

current = fr[cnt];

for (c = i + 1; c < n; c++) {

if (current != pg[c]) {

count[cnt]++;

} else {

break; // Stop when found next occurrence

}

}

if (c == n) count[cnt] = n; // Not found in future

}

// Find farthest page

max = count[0];

p = 0;

for (m = 1; m < f; m++) {

if (count[m] > max) {

max = count[m];

p = m;

}

}

// Replace with farthest page

fr[p] = temp;

}

// Print frame content

for (x = 0; x < f; x++) {

printf("%d\t", fr[x]);

}

}

// Print total faults

printf("\nTotal page faults = %d", fault);

}

// LRU Algorithm

void lru() {

int count[10], i, j, k = 0, fault = 0, flag, temp, current, c, max, m, cnt, p, x, f;

// Input frame size

printf("\n Enter frame size: ");

scanf("%d", &f);

// Initialize frames and count array

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

count[i] = 0;

fr[i] = -1;

}

// Traverse pages

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

flag = 0;

temp = pg[i];

// Check for page hit

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

if (temp == fr[j]) {

flag = 1;

break;

}

}

// If page fault and space available

if (flag == 0 && k < f) {

fault++;

fr[k] = temp;

k++;

} else if (flag == 0 && k == f) {

fault++;

// Count backward to see how recently each page was used

for (cnt = 0; cnt < f; cnt++) {

current = fr[cnt];

count[cnt] = 0;

for (c = i - 1; c >= 0; c--) {

if (current != pg[c]) {

count[cnt]++;

} else {

break;

}

}

}

// Find least recently used page

max = count[0];

p = 0;

for (m = 1; m < f; m++) {

if (count[m] > max) {

max = count[m];

p = m;

}

}

// Replace LRU page

fr[p] = temp;

}

// Print current frame status

for (x = 0; x < f; x++) {

printf("%d\t", fr[x]);

}

}

// Output total faults

printf("\nTotal page faults = %d", fault);

}