OS SOLUTION

**Bankers Algorithm**

#include <stdio.h>

int main() {

int p, r, i, j, count = 0;

int claim[10][10], alloc[10][10], req[10][10], rsrc[10], avail[10], comp[10] = {0};

// Prompt for the number of processes

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

scanf("%d", &p); // Read the number of processes

// Prompt for the number of resources

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

scanf("%d", &r); // Read the number of resources

// Input Claim matrix

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

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

printf("For Process %d:\n", i); // Output the process number

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

scanf("%d", &claim[i][j]); // Read claim for each resource for the current process

}

}

// Input Allocation matrix

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

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

printf("For Process %d:\n", i); // Output the process number

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

scanf("%d", &alloc[i][j]); // Read the allocated resources for the current process

}

}

// Input total resources

printf("Enter the total number of each resource:\n");

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

scanf("%d", &rsrc[j]); // Read the total number of each resource

}

// Calculate available resources

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

int total = 0;

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

total += alloc[i][j]; // Sum the allocated resources for the current resource type

}

avail[j] = rsrc[j] - total; // The available resources are total resources minus allocated resources

}

int done;

do {

done = 0; // Flag to check if any process has been completed in this cycle

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

if (comp[i] == 0) { // Check if the process has not completed

int can\_run = 1; // Flag to check if the process can run

// Calculate the required resources for each process

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

req[i][j] = claim[i][j] - alloc[i][j]; // Needed resources = claimed - allocated

if (req[i][j] > avail[j]) { // If the required resources are more than available, the process cannot run

can\_run = 0;

break; // Break out of the loop as the process cannot run

}

}

// If the process can run, allocate resources and mark it as completed

if (can\_run) {

printf("Process %d runs to completion.\n", i); // Process runs to completion

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

avail[j] += alloc[i][j]; // Release the allocated resources back to the available pool

}

comp[i] = 1; // Mark the process as completed

done = 1; // Set the done flag to indicate a process was completed

count++; // Increment the count of completed processes

}

}

}

} while (done); // Repeat the process as long as a process was completed in the current cycle

// After all cycles, check if all processes have been completed

if (count == p) // If the number of completed processes equals the total number of processes, the system is safe

printf("The system is in a safe state.\n");

else

printf("The system is in an unsafe state.\n"); // If some processes couldn't complete, the system is unsafe

return 0; // Return 0 to indicate the program has finished

}

**IPC FIFO**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h> // Required for strcmp

#include <sys/stat.h>

#include <sys/types.h>

#define FIFO\_NAME "myfifo" // FIFO name definition

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

char buffer[80]; // Buffer to store the data read from FIFO

int fd;

// Create the FIFO (if it doesn't exist)

mkfifo(FIFO\_NAME, 0666); // Create with read/write permissions for all

// Check if the argument is "writer" to send data to the FIFO

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

// Writer process

fd = open(FIFO\_NAME, O\_WRONLY); // Open FIFO for writing

if (fd == -1) {

perror("open"); // Handle error if open fails

exit(EXIT\_FAILURE);

}

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

ssize\_t bytes\_written = write(fd, message, strlen(message) + 1); // Write message (+1 for '\0')

if (bytes\_written == -1) {

perror("write"); // Handle error if write fails

exit(EXIT\_FAILURE);

}

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

close(fd); // Close FIFO after writing

}

// Check if the argument is "reader" to read from the FIFO

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

// Reader process

fd = open(FIFO\_NAME, O\_RDONLY); // Open FIFO for reading

if (fd == -1) {

perror("open"); // Handle error if open fails

exit(EXIT\_FAILURE);

}

ssize\_t bytes\_read = read(fd, buffer, sizeof(buffer)); // Read from FIFO into buffer

if (bytes\_read == -1) {

perror("read"); // Handle error if read fails

exit(EXIT\_FAILURE);

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

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

} else {

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

}

close(fd); // Close FIFO after reading

}

else {

// Error message if invalid argument

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

exit(EXIT\_FAILURE);

}

return 0;

}

**ROUND ROBIN**

#include <stdio.h> // Standard I/O functions

// Structure to store process details

struct rr {

int pno; // Process number

int btime; // Remaining burst time

int com; // Completion time of the last execution slice

int wt; // Waiting time before the current slice

int cal; // Total accumulated waiting time

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

};

int main() {

int n, i, wait = 0, temp = 20; // wait = current time, temp = time quantum (20ms)

float avg = 0, avg1 = 0; // avg = average waiting time, avg1 = avg turnaround time

struct rr r[10]; // Array of processes (max 10)

// Input number of processes

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

scanf("%d", &n);

// Input details for each process

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);

// Initialize process values

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

r[i].wt = 0; // Initial waiting time

r[i].com = 0; // No execution yet

r[i].cal = 0; // Total wait starts at 0

}

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

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

printf("\nPROCESS\tBURST TIME\tWAITING TIME\tCOMPLETION TIME");

// Start Round Robin execution

int all\_done;

do {

all\_done = 1; // Assume all done until we find an active process

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

if (r[i].st == 'a') { // If process is active

all\_done = 0;

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

// Process needs more than 1 quantum

r[i].wt = wait; // Waiting time before this slice

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

r[i].com = wait + temp; // Completion of this slice

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

wait += temp; // Increase global time

r[i].btime -= temp; // Reduce burst time

} else {

// Process can complete in this slice

r[i].wt = wait;

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

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

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

wait += r[i].btime;

r[i].btime = 0;

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

}

}

}

} while (!all\_done); // Continue until all processes are completed

// 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;

printf("\n\nTHE AVERAGE WAITING TIME IS: %.2f ms", avg);

printf("\nTHE AVERAGE TURNAROUND TIME IS: %.2f ms\n", avg1);

return 0;

}

**REVERSE STRING**

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

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

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

char buf; // To hold one character at a time

int size, fd; // 'size' for current position; 'fd' is file descriptor

// Open file passed as command-line argument in read-only mode

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

// Move file pointer to the end of the file, then back by one byte

size = lseek(fd, -1, SEEK\_END); // Get position of the last byte

// Loop to read and print characters from end to start

while (size-- >= 0) {

read(fd, &buf, 1); // Read current byte

write(STDOUT\_FILENO, &buf, 1); // Print it to standard output

lseek(fd, -2, SEEK\_CUR); // Move back 2 bytes to get previous char

}

close(fd); // Close the file

return 0;

}

**FCFS**

#include <stdio.h>

struct process {

int pno; // Process number

int atime; // Arrival time

int btime; // Burst time

int wtime; // Waiting time

int ttime; // Turnaround time

int ctime; // Completion time

};

void sortByArrival(struct process p[], int n) {

struct process temp;

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

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

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

temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

int main() {

struct process p[20];

int n;

float avg\_wt = 0, avg\_tt = 0;

printf("ENTER NUMBER OF PROCESSES: ");

scanf("%d", &n);

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

printf("\nPROCESS %d:\n", i + 1);

printf("Process Number: ");

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

printf("Arrival Time: ");

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

printf("Burst Time: ");

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

}

sortByArrival(p, n);

int time = 0;

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

if (time < p[i].atime)

time = p[i].atime; // CPU idle until the process arrives

p[i].wtime = time - p[i].atime;

p[i].ctime = time + p[i].btime;

p[i].ttime = p[i].ctime - p[i].atime;

time += p[i].btime;

avg\_wt += p[i].wtime;

avg\_tt += p[i].ttime;

}

// Output table

printf("\nPROCESS\tATIME\tBTIME\tCTIME\tWTIME\tTTIME\n");

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

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

p[i].pno, p[i].atime, p[i].btime,

p[i].ctime, p[i].wtime, p[i].ttime);

}

printf("\nAVERAGE WAITING TIME: %.2f", avg\_wt / n);

printf("\nAVERAGE TURNAROUND TIME: %.2f\n", avg\_tt / n);

// Gantt Chart

printf("\nGANTT CHART:\n|");

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

printf(" P%d |", p[i].pno);

}

printf("\n0");

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

printf(" %d", p[i].ctime);

}

return 0;

}

**FILE MANAGEMENT**

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

#include <fcntl.h> // For open() and O\_RDWR flag

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

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

int main() {

int fd[2]; // File descriptor array for reading and writing

char buf1[100] = "JUST A TEXT"; // Buffer to store user input (expanded size for safety)

char buf2[100]; // Buffer to read data back

// Open the same file twice with read-write access

fd[0] = open("SAMPLE", O\_RDWR);

fd[1] = open("SAMPLE", O\_RDWR);

// Write the initial string to the file using fd[0]

write(fd[0], buf1, strlen(buf1));

// Prompt user for input

printf("\nENTER YOUR TEXT NOW: ");

scanf("%s", buf1); // Read a word into buf1 (unsafe for spaces; consider fgets for multi-word input)

// Write user input to file

write(fd[0], buf1, strlen(buf1));

// Read content from file using fd[1] and write to STDOUT (fd = 1)

write(1, buf2, read(fd[1], buf2, sizeof(buf2)));

// Close both file descriptors

close(fd[0]);

close(fd[1]);

printf("\n"); // Print a newline

return 0; // Exit main

}

**SJF**

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

// Define structure for process information

struct s {

int pno; // Process number

int btime; // Burst time

} s[20]; // Array of process structures

int main() {

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

float avg, avg1, avg2, tc;

// Prompt for number of processes

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

scanf("%d", &n);

// Initialize all totals to 0

ptemp = 0;

ptemp1 = 0;

wait = 0;

avg = 0;

avg1 = 0;

avg2 = 0;

tc = 0;

// 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 (SJF - Non-Preemptive)

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

ptemp = s[i].pno;

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

s[j].pno = ptemp;

}

}

}

// Display table header

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

printf("PROCESS NO\tBURST TIME\tWAITING TIME\tTURNAROUND 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; // Total turnaround time

avg += wait; // Total waiting time

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

}

// Calculate averages

avg1 = avg / n;

avg2 = tc / n;

// Print averages

printf("\nTHE AVERAGE WAITING TIME OF PROCESS IS: %.2f", avg1);

printf("\nTHE AVERAGE TURN AROUND TIME OF PROCESS IS: %.2f\n", avg2);

return 0;

}

**PRIORITY**

#include <stdio.h>

// Define a structure to hold process details

struct pr {

int prino; // Priority number

int pno; // Process number

int btime; // Burst time

};

int main() {

// Variable declarations

int btemp = 0, ptemp = 0, prtemp = 0; // Temporary variables for swapping

int wait = 0, n, i, j;

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

struct pr p[15]; // Array to hold up to 15 processes

printf("\nPRIORITY SCHEDULING\n");

// Get number of processes from user

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

scanf("%d", &n);

// Input process details

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

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

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

printf("ENTER THE PRIORITY NUMBER: ");

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

printf("ENTER THE BURST TIME: ");

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

}

// Sort processes based on priority (lower number = higher priority)

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

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

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

// Swap all fields if current process has lower priority

prtemp = p[i].prino;

btemp = p[i].btime;

ptemp = p[i].pno;

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

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

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

p[j].prino = prtemp;

p[j].btime = btemp;

p[j].pno = ptemp;

}

}

}

// Display sorted process details

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

printf("PROCESS NO\tPRIORITY NO\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 waiting time for next process

}

// Calculate averages

avg1 = avg / n;

avg2 = tc / n;

// Display results

printf("\nAVERAGE WAITING TIME FOR THE PROCESS IS: %.2f\n", avg1);

printf("AVERAGE TURN AROUND TIME FOR THE PROCESS IS: %.2f\n", avg2);

return 0;

}

**Directory Management**

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

#include <dirent.h> // For directory handling functions

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

DIR \*dir; // Pointer to directory structure

struct dirent \*directory; // Pointer to structure for reading entries

// Open the directory provided as a command line argument

dir = opendir(argy[1]);

// Check if directory was opened successfully

if (dir == NULL) {

printf("Unable to open directory.\n");

return 1;

}

// Read and print entries in the directory one by one

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

// Print inode number and file/directory name

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

}

// Close the directory

closedir(dir);

return 0;

}

**PIPES**

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

#include <stdlib.h> // For exit() and EXIT\_FAILURE

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

int main() {

int pipefd[2]; // Array to hold the two ends of the pipe: [0] for reading, [1] for writing

char buffer[80]; // Buffer to store data read from the pipe

// Create a pipe

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

perror("pipe"); // Print error if pipe creation fails

exit(EXIT\_FAILURE); // Exit with failure

}

// Create a child process

pid\_t pid = fork(); // fork() creates a new process

if (pid < 0) {

perror("fork"); // Print error if fork fails

exit(EXIT\_FAILURE); // Exit with failure

}

// Parent process (writer)

if (pid > 0) {

close(pipefd[0]); // Close the reading end in the parent process

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

// Write the message to the pipe

ssize\_t bytes\_written = write(pipefd[1], message, sizeof(message));

if (bytes\_written == -1) {

perror("write"); // Print error if write fails

exit(EXIT\_FAILURE);

}

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

}

// Child process (reader)

else {

close(pipefd[1]); // Close the writing end in the child process

// Read the message from the pipe into buffer

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

if (bytes\_read == -1) {

perror("read"); // Print error if read fails

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); // Display received message

}

}

// Close any remaining open pipe file descriptors

close(pipefd[0]);

close(pipefd[1]);

return 0;

}

**Process Management**

#include <stdio.h> // For printf

#include <unistd.h> // For fork(), execlp()

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

#include <sys/types.h> // For pid\_t

#include <stdlib.h> // For exit()

int main()

{

int pid;

pid = fork(); // Create a new process

if (pid < 0) {

// If fork fails

printf("\nFORK FAILED\n");

exit(-1);

}

else if (pid == 0) {

// Child process

// execlp executes 'ls -l' (long listing)

// Correction: "is" should be "ls"

execlp("/bin/ls", "ls", "-l", NULL);

// If execlp fails

perror("execlp failed");

exit(1);

}

else {

// Parent process waits for the child to complete

wait(NULL);

printf("\nCHILD COMPLETE\n");

exit(0);

}

}

**SEMAPHORE**

#include <stdio.h>

#include <stdlib.h>

typedef int semaphore;

semaphore mutex = 1;

semaphore full = 0;

semaphore empty = 0;

int main() {

int i, opt, size;

int buffer[20] = {0};

printf("\nEnter buffer capacity: ");

scanf("%d", &size);

empty = size; // Set available slots

do {

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

printf("\nEnter your option: ");

scanf("%d", &opt);

switch (opt) {

case 1:

if (empty != 0) {

mutex--;

empty--;

int item;

printf("\nEnter the item: ");

scanf("%d", &item);

buffer[full] = item;

full++;

mutex++;

printf("\nItems in buffer: ");

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

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

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

} else {

printf("\nBuffer Overflow\n");

}

break;

case 2:

if (full != 0) {

mutex--;

full--;

int consumed = buffer[0];

// Shift items left

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

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

empty++;

mutex++;

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

printf("\nItems in buffer: ");

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

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

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

} else {

printf("\nBuffer Underflow\n");

}

break;

case 3:

exit(0);

break;

default:

printf("\nEnter a valid option\n");

break;

}

} while (opt != 3);

return 0;

}

**SHARED MEMORY**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <fcntl.h> // For O\_CREAT and O\_RDWR

#include <errno.h> // For errno

#include <unistd.h> // For close()

#define SHM\_SIZE 1024

#define SHM\_KEY\_FILE "shared\_memory.txt" // File for ftok()

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

int shmid;

// Try to create shared memory segment

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

if (errno == EEXIST) {

// If it already exists, try to get it

shmid = shmget(key, size, 0);

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 for ftok to work

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

if (fd < 0) {

perror("open key file");

exit(EXIT\_FAILURE);

}

close(fd); // Close it immediately, only needed for ftok

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

if (key == -1) {

perror("ftok");

exit(EXIT\_FAILURE);

}

int mode = 0; // 0 = reader, 1 = writer

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

mode = 1;

}

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

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

if (mode == 1) {

printf("Writer process attached to shared memory.\n");

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

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

} else {

printf("Reader process attached to shared memory.\n");

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

}

detach\_shared\_memory(data);

return 0;

}

**SEGMENTATION**

#include <stdio.h>

#include <stdlib.h>

struct list {

int seg;

int base;

int limit;

struct list \*next;

};

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

struct list \*q = \*p;

struct list \*new\_node = malloc(sizeof(struct list));

if (new\_node == NULL) {

printf("Memory allocation failed\n");

return;

}

new\_node->base = base;

new\_node->limit = limit;

new\_node->seg = seg;

new\_node->next = NULL;

// If the list is empty, insert at the beginning

if (\*p == NULL) {

\*p = new\_node;

} else {

// Traverse to the end of the list

while (q->next != NULL) {

q = q->next;

}

q->next = new\_node;

}

}

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

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

q = q->next;

}

if (q == NULL) {

printf("Segment not found\n");

return -1; // Error: segment not found

}

return q->limit;

}

int search(struct list \*q, int seg) {

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

q = q->next;

}

if (q == NULL) {

printf("Segment not found\n");

return -1; // Error: segment not found

}

return q->base;

}

int main() {

struct list \*p = NULL;

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

printf("Enter segment table\n");

printf("Enter -1 as segment value for termination\n");

do {

printf("Enter segment number: ");

scanf("%d", &seg);

if (seg != -1) {

printf("Enter base value: ");

scanf("%d", &base);

printf("Enter value for limit: ");

scanf("%d", &limit);

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

}

} while (seg != -1);

printf("Enter offset: ");

scanf("%d", &offset);

printf("Enter segmentation number: ");

scanf("%d", &seg);

c = find(p, seg);

s = search(p, seg);

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

printf("Error: invalid segment\n");

} else if (offset < c) {

physical = s + offset;

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

} else {

printf("Error: offset exceeds segment limit\n");

}

return 0;

}

**PAGING**

#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 (in bytes) -- ");

scanf("%d", &ms);

printf("\nEnter the page size (in bytes) -- ");

scanf("%d", &ps);

nop = ms / ps;

printf("\nThe number of pages available in memory are -- %d", nop);

printf("\nEnter the number of processes -- ");

scanf("%d", &np);

rempages = nop;

// Input for each process's page table

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

printf("\nEnter the number of pages required for p[%d]-- ", i);

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

if (s[i] > rempages) {

printf("\nMemory is Full\n");

break;

}

rempages -= s[i];

printf("\nEnter page table for p[%d] --- ", i);

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

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

}

}

// Logical address translation

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

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

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

// Check if the process number, page number, or offset is invalid

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

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

} else {

pa = fno[x][y] \* ps + offset; // Calculate Physical Address

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

}

return 0;

}

**Page Replacement**

#include<stdio.h>

int n, pg[30], fr[10];

void fifo();

void optimal();

void lru();

void main() {

int i, ch;

// User input for total number of pages

printf("\nEnter total number of pages: ");

scanf("%d", &n);

printf("\nEnter sequence of pages: ");

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

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

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(); break;

case 2: optimal(); break;

case 3: lru(); break;

case 4: break;

default: printf("Invalid choice! Please try again.\n");

}

} while (ch != 4);

}

void fifo() {

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

printf("\nEnter size of page frame: ");

scanf("%d", &psize);

// Initialize frame with -1

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

fr[i] = -1;

}

f = 0;

r = 0;

s = 0;

while (s < n) {

flag = 0;

num = pg[s];

// Check if page is already in the frame

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

if (num == fr[i]) {

s++;

flag = 1;

break;

}

}

// If page is not found in the frame

if (flag == 0) {

if (r < psize) {

fr[r] = pg[s];

r++;

} else {

fr[f] = pg[s];

f = (f + 1) % psize; // Circular replacement

}

s++;

count++;

}

// Print current page frames

printf("\n");

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

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

}

}

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

}

void optimal() {

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

printf("\nEnter frame size: ");

scanf("%d", &f);

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

count[i] = 0;

fr[i] = -1;

}

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

flag = 0;

temp = pg[i];

// Check if the page is in the frame

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

if (temp == fr[j]) {

flag = 1;

break;

}

}

// If page is not in the frame, handle it

if (flag == 0) {

fault++;

if (k < f) {

fr[k] = temp;

k++;

} else {

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

current = fr[cnt];

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

if (current != pg[c]) {

count[cnt]++;

} else {

break;

}

}

}

max = 0;

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

if (count[m] > max) {

max = count[m];

p = m;

}

}

fr[p] = temp;

}

}

// Print current frame

printf("\n");

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

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

}

}

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

}

void lru() {

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

printf("\nEnter frame size: ");

scanf("%d", &f);

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

count[i] = 0;

fr[i] = -1;

}

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

flag = 0;

temp = pg[i];

// Check if the page is already in the frame

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

if (temp == fr[j]) {

flag = 1;

break;

}

}

if (flag == 0) {

fault++;

if (k < f) {

fr[k] = temp;

k++;

} else {

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

current = fr[cnt];

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

if (current != pg[c]) {

count[cnt]++;

} else {

break;

}

}

}

max = 0;

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

if (count[m] > max) {

max = count[m];

p = m;

}

}

fr[p] = temp;

}

}

// Print current frame

printf("\n");

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

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

}

}

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

}

**Message queue**

#ifndef COMMON\_H

#define COMMON\_H

struct my\_msg {

long msg\_type; // Message type

char some\_text[100]; // Adjust size as needed

};

#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" // Include the header file

#define MAX\_TEXT 100

int main() {

int msgid;

struct my\_msg some\_data;

int running = 1;

key\_t key = ftok("/tmp/myqueue", 65); // Replace with desired key generation method

// Get message queue ID (create if it doesn't exist)

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

if (msgid == -1) {

perror("msgget");

exit(1);

}

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'; // Remove trailing newline

// Check for end message

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

running = 0;

} else {

some\_data.msg\_type = 1; // Set message type (e.g., 1 for request)

// Send message to queue

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

perror("msgsnd");

exit(1);

}

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" // Include the header file

#define MAX\_TEXT 100

int main() {

int msgid;

struct my\_msg some\_data;

long int msg\_to\_rec = 0; // Any message type

key\_t key = ftok("/tmp/myqueue", 65); // Replace with desired key generation method

// Get message queue ID (create if it doesn't exist)

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

if (msgid == -1) {

perror("msgget");

exit(1);

}

while (1) {

// Receive message from queue

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

perror("msgrcv");

exit(1);

}

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

// Check for end message

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

break;

}

}

// Cleanup and remove the message queue

if (msgctl(msgid, IPC\_RMID, NULL) == -1) {

perror("msgctl");

exit(1);

}

return 0;

}