//Basic Linux Commands

$ls - Lists the files and directories in the current directory.

$ ls

Documents Downloads Pictures Videos

$ pwd - Prints the working directory (current directory)

$ pwd

/home/user

$ cd Documents

$ pwd

/home/user/Documents

$ cp

$ cp file.txt /home/user/Backup/

$rm

$ rm unwanted\_file.txt

//Advanced

1. $ grep "error" log.txt

[ERROR] Disk space is low

$ find /home/user -name "\*.pdf"

/home/user/Documents/file1.pdf

/home/user/Downloads/file2.pdf

2.$ chmod 755 script.sh

3.$ tar -czvf archive.tar.gz Documents/

4.$ top

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

1234 user 20 0 1.2g 123m 10m S 2.3 1.5 00:02.34 chrome

// Shell script sum

echo "Enter a number:"

read number

# Initialize sum to 0

sum=0

# Loop through each digit

while [ $number -gt 0 ]; do

# Extract the last digit

digit=$((number % 10))

# Add the digit to the sum

sum=$((sum + digit))

# Remove the last digit

number=$((number / 10))

done

# Display the result

echo "The sum of the digits is: $sum"

//Shell descending order

echo "Enter the elements of the array separated by space:"

read -a array

# Get the length of the array

n=${#array[@]}

# Bubble sort to sort the array in descending order

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

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

if [ ${array[j]} -lt ${array[j+1]} ]; then

# Swap elements

temp=${array[j]}

array[j]=${array[j+1]}

array[j+1]=$temp

fi

done

done

# Display the sorted array

echo "Array elements in descending order:"

echo "${array[@]}"

//FCFS

#include <stdio.h>

#include <stdlib.h>

// Function to sort processes by Arrival Time

void sort\_by\_arrival\_time(int \*arrival\_time, int \*cpu\_burst, int n) {

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

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

if (arrival\_time[j] > arrival\_time[j + 1]) {

// Swap arrival time

int temp = arrival\_time[j];

arrival\_time[j] = arrival\_time[j + 1];

arrival\_time[j + 1] = temp;

// Swap CPU burst time

temp = cpu\_burst[j];

cpu\_burst[j] = cpu\_burst[j + 1];

cpu\_burst[j + 1] = temp;

}

}

}

}

// FCFS Scheduling Function

void fcfs(int \*cpu\_burst, int \*arrival\_time, int n, int \*waiting\_time, int \*turnaround\_time) {

int completion\_time[n];

int current\_time = 0;

// Calculate Completion Time, Turnaround Time, and Waiting Time

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

if (current\_time < arrival\_time[i]) {

current\_time = arrival\_time[i];

}

completion\_time[i] = current\_time + cpu\_burst[i];

current\_time += cpu\_burst[i];

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

waiting\_time[i] = turnaround\_time[i] - cpu\_burst[i];

}

}

// Function to print results

void print\_results(int \*waiting\_time, int \*turnaround\_time, int n, int \*arrival\_time, int \*cpu\_burst) {

double avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

printf("\nProcess\tAT\tBT\tWT\tTAT\n");

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

printf("P%d\t%d\t%d\t%d\t%d\n", i + 1, arrival\_time[i], cpu\_burst[i], waiting\_time[i], turnaround\_time[i]);

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

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

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

}

int main() {

int n;

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

scanf("%d", &n);

if (n <= 0) {

printf("Number of processes must be greater than zero.\n");

return 1;

}

int arrival\_time[n], cpu\_burst[n], waiting\_time[n], turnaround\_time[n];

// Input Arrival Time and CPU Burst Time

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

printf("Enter Arrival Time of process %d: ", i + 1);

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

printf("Enter CPU Burst Time of process %d: ", i + 1);

scanf("%d", &cpu\_burst[i]);

if (arrival\_time[i] < 0 || cpu\_burst[i] <= 0) {

printf("Arrival time must be non-negative and CPU burst time must be greater than zero.\n");

return 1;

}

}

// Sort processes by Arrival Time

sort\_by\_arrival\_time(arrival\_time, cpu\_burst, n);

// Perform FCFS Scheduling

fcfs(cpu\_burst, arrival\_time, n, waiting\_time, turnaround\_time);

// Print Results

print\_results(waiting\_time, turnaround\_time, n, arrival\_time, cpu\_burst);

return 0;

}

//SJF

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to sort processes by Arrival Time

void sort\_by\_arrival\_time(int \*arrival\_time, int \*cpu\_burst, int \*process\_id, int n) {

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

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

if (arrival\_time[j] > arrival\_time[j + 1]) {

// Swap arrival time

int temp = arrival\_time[j];

arrival\_time[j] = arrival\_time[j + 1];

arrival\_time[j + 1] = temp;

// Swap CPU burst time

temp = cpu\_burst[j];

cpu\_burst[j] = cpu\_burst[j + 1];

cpu\_burst[j + 1] = temp;

// Swap process ID

temp = process\_id[j];

process\_id[j] = process\_id[j + 1];

process\_id[j + 1] = temp;

}

}

}

}

// SJF Non-Preemptive Scheduling

void sjf\_non\_preemptive(int \*arrival\_time, int \*cpu\_burst, int n, int \*waiting\_time, int \*turnaround\_time) {

int completed[n], completion\_time[n], current\_time = 0, completed\_count = 0;

memset(completed, 0, sizeof(completed));

while (completed\_count < n) {

int shortest\_index = -1;

int shortest\_burst = \_\_INT\_MAX\_\_;

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

if (!completed[i] && arrival\_time[i] <= current\_time && cpu\_burst[i] < shortest\_burst) {

shortest\_burst = cpu\_burst[i];

shortest\_index = i;

}

}

if (shortest\_index == -1) {

current\_time++;

continue;

}

completion\_time[shortest\_index] = current\_time + cpu\_burst[shortest\_index];

current\_time = completion\_time[shortest\_index];

turnaround\_time[shortest\_index] = completion\_time[shortest\_index] - arrival\_time[shortest\_index];

waiting\_time[shortest\_index] = turnaround\_time[shortest\_index] - cpu\_burst[shortest\_index];

completed[shortest\_index] = 1;

completed\_count++;

}

}

// SJF Preemptive Scheduling

void sjf\_preemptive(int \*arrival\_time, int \*cpu\_burst, int n, int \*waiting\_time, int \*turnaround\_time) {

int remaining\_time[n], completion\_time[n];

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

remaining\_time[i] = cpu\_burst[i];

}

int current\_time = 0, completed\_count = 0, shortest\_index = -1, min\_remaining = \_\_INT\_MAX\_\_;

while (completed\_count < n) {

shortest\_index = -1;

min\_remaining = \_\_INT\_MAX\_\_;

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

if (arrival\_time[i] <= current\_time && remaining\_time[i] > 0 && remaining\_time[i] < min\_remaining) {

min\_remaining = remaining\_time[i];

shortest\_index = i;

}

}

if (shortest\_index == -1) {

current\_time++;

continue;

}

remaining\_time[shortest\_index]--;

if (remaining\_time[shortest\_index] == 0) {

completion\_time[shortest\_index] = current\_time + 1;

turnaround\_time[shortest\_index] = completion\_time[shortest\_index] - arrival\_time[shortest\_index];

waiting\_time[shortest\_index] = turnaround\_time[shortest\_index] - cpu\_burst[shortest\_index];

completed\_count++;

}

current\_time++;

}

}

// Function to print results

void print\_results(int \*waiting\_time, int \*turnaround\_time, int n, int \*arrival\_time, int \*cpu\_burst) {

double avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

printf("\nProcess\tAT\tBT\tWT\tTAT\n");

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

printf("P%d\t%d\t%d\t%d\t%d\n", i + 1, arrival\_time[i], cpu\_burst[i], waiting\_time[i], turnaround\_time[i]);

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

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

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

}

int main() {

int n;

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

scanf("%d", &n);

if (n <= 0) {

printf("Number of processes must be greater than zero.\n");

return 1;

}

int arrival\_time[n], cpu\_burst[n], waiting\_time[n], turnaround\_time[n], process\_id[n];

// Input Arrival Time and CPU Burst Time

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

process\_id[i] = i + 1;

printf("Enter Arrival Time of process %d: ", i + 1);

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

printf("Enter CPU Burst Time of process %d: ", i + 1);

scanf("%d", &cpu\_burst[i]);

if (arrival\_time[i] < 0 || cpu\_burst[i] <= 0) {

printf("Arrival time must be non-negative and CPU burst time must be greater than zero.\n");

return 1;

}

}

// Sort processes by Arrival Time

sort\_by\_arrival\_time(arrival\_time, cpu\_burst, process\_id, n);

while (1) {

int choice;

printf("\nSelect scheduling mode:\n");

printf("1. SJF Non-Preemptive\n");

printf("2. SJF Preemptive\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

if (choice == 3) {

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

break;

}

switch (choice) {

case 1:

sjf\_non\_preemptive(arrival\_time, cpu\_burst, n, waiting\_time, turnaround\_time);

break;

case 2:

sjf\_preemptive(arrival\_time, cpu\_burst, n, waiting\_time, turnaround\_time);

break;

default:

printf("Invalid choice! Try again.\n");

continue;

}

// Print Results

print\_results(waiting\_time, turnaround\_time, n, arrival\_time, cpu\_burst);

}

return 0;

}

// Priority Scheduling

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to sort processes by Arrival Time

void sort\_by\_arrival\_time(int \*arrival\_time, int \*cpu\_burst, int \*priority, int \*process\_id, int n) {

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

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

if (arrival\_time[j] > arrival\_time[j + 1]) {

// Swap arrival time

int temp = arrival\_time[j];

arrival\_time[j] = arrival\_time[j + 1];

arrival\_time[j + 1] = temp;

// Swap CPU burst time

temp = cpu\_burst[j];

cpu\_burst[j] = cpu\_burst[j + 1];

cpu\_burst[j + 1] = temp;

// Swap priority

temp = priority[j];

priority[j] = priority[j + 1];

priority[j + 1] = temp;

// Swap process ID

temp = process\_id[j];

process\_id[j] = process\_id[j + 1];

process\_id[j + 1] = temp;

}

}

}

}

// Priority Non-Preemptive Scheduling

void priority\_non\_preemptive(int \*arrival\_time, int \*cpu\_burst, int \*priority, int n, int \*waiting\_time, int \*turnaround\_time) {

int completed[n], completion\_time[n], current\_time = 0, completed\_count = 0;

memset(completed, 0, sizeof(completed));

while (completed\_count < n) {

int highest\_priority = \_\_INT\_MAX\_\_;

int index = -1;

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

if (!completed[i] && arrival\_time[i] <= current\_time && priority[i] < highest\_priority) {

highest\_priority = priority[i];

index = i;

}

}

if (index == -1) {

current\_time++;

continue;

}

completion\_time[index] = current\_time + cpu\_burst[index];

current\_time = completion\_time[index];

turnaround\_time[index] = completion\_time[index] - arrival\_time[index];

waiting\_time[index] = turnaround\_time[index] - cpu\_burst[index];

completed[index] = 1;

completed\_count++;

}

}

// Priority Preemptive Scheduling

void priority\_preemptive(int \*arrival\_time, int \*cpu\_burst, int \*priority, int n, int \*waiting\_time, int \*turnaround\_time) {

int remaining\_time[n], completion\_time[n], current\_time = 0, completed\_count = 0;

int prev\_index = -1;

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

remaining\_time[i] = cpu\_burst[i];

}

memset(waiting\_time, 0, sizeof(int) \* n);

while (completed\_count < n) {

int highest\_priority = \_\_INT\_MAX\_\_;

int index = -1;

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

if (arrival\_time[i] <= current\_time && remaining\_time[i] > 0 && priority[i] < highest\_priority) {

highest\_priority = priority[i];

index = i;

}

}

if (index == -1) {

current\_time++;

continue;

}

remaining\_time[index]--;

current\_time++;

if (remaining\_time[index] == 0) {

completion\_time[index] = current\_time;

turnaround\_time[index] = completion\_time[index] - arrival\_time[index];

waiting\_time[index] = turnaround\_time[index] - cpu\_burst[index];

completed\_count++;

}

}

}

// Function to print results

void print\_results(int \*waiting\_time, int \*turnaround\_time, int n, int \*arrival\_time, int \*cpu\_burst, int \*priority, int \*process\_id) {

double avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

printf("\nProcess\tAT\tBT\tPri\tWT\tTAT\n");

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

printf("P%d\t%d\t%d\t%d\t%d\t%d\n", process\_id[i], arrival\_time[i], cpu\_burst[i], priority[i], waiting\_time[i], turnaround\_time[i]);

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

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

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

}

int main() {

int n;

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

scanf("%d", &n);

if (n <= 0) {

printf("Number of processes must be greater than zero.\n");

return 1;

}

int arrival\_time[n], cpu\_burst[n], priority[n], waiting\_time[n], turnaround\_time[n], process\_id[n];

// Input Arrival Time, CPU Burst Time, and Priority

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

process\_id[i] = i + 1;

printf("Enter Arrival Time of process %d: ", i + 1);

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

printf("Enter CPU Burst Time of process %d: ", i + 1);

scanf("%d", &cpu\_burst[i]);

printf("Enter Priority of process %d (lower number means higher priority): ", i + 1);

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

if (arrival\_time[i] < 0 || cpu\_burst[i] <= 0 || priority[i] < 0) {

printf("Arrival time, CPU burst time, and priority must be non-negative, and burst time must be greater than zero.\n");

return 1;

}

}

// Sort processes by Arrival Time

sort\_by\_arrival\_time(arrival\_time, cpu\_burst, priority, process\_id, n);

while (1) {

int choice;

printf("\nSelect scheduling mode:\n");

printf("1. Priority Non-Preemptive\n");

printf("2. Priority Preemptive\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

if (choice == 3) {

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

break;

}

switch (choice) {

case 1:

priority\_non\_preemptive(arrival\_time, cpu\_burst, priority, n, waiting\_time, turnaround\_time);

break;

case 2:

priority\_preemptive(arrival\_time, cpu\_burst, priority, n, waiting\_time, turnaround\_time);

break;

default:

printf("Invalid choice! Try again.\n");

continue;

}

// Print Results

print\_results(waiting\_time, turnaround\_time, n, arrival\_time, cpu\_burst, priority, process\_id);

}

return 0;

}

//Round Robin Scheduling

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to sort processes by Arrival Time

void sort\_by\_arrival\_time(int \*arrival\_time, int \*cpu\_burst, int \*process\_id, int n) {

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

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

if (arrival\_time[j] > arrival\_time[j + 1]) {

// Swap arrival time

int temp = arrival\_time[j];

arrival\_time[j] = arrival\_time[j + 1];

arrival\_time[j + 1] = temp;

// Swap CPU burst time

temp = cpu\_burst[j];

cpu\_burst[j] = cpu\_burst[j + 1];

cpu\_burst[j + 1] = temp;

// Swap process ID

temp = process\_id[j];

process\_id[j] = process\_id[j + 1];

process\_id[j + 1] = temp;

}

}

}

}

// Round Robin Scheduling

void round\_robin(int \*arrival\_time, int \*cpu\_burst, int n, int \*waiting\_time, int \*turnaround\_time, int time\_quantum) {

int remaining\_time[n];

int current\_time = 0;

int completed\_count = 0;

int queue[n \* 2], front = 0, rear = 0; // Circular queue for ready processes

int visited[n]; // Tracks whether a process has been added to the queue

memset(remaining\_time, 0, sizeof(int) \* n);

memset(visited, 0, sizeof(int) \* n);

// Initialize remaining time

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

remaining\_time[i] = cpu\_burst[i];

waiting\_time[i] = 0; // Initialize waiting time

}

// Enqueue the first process

queue[rear++] = 0;

visited[0] = 1;

while (completed\_count < n) {

if (front == rear) {

// No process is ready; move time forward

current\_time++;

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

if (arrival\_time[i] <= current\_time && remaining\_time[i] > 0 && !visited[i]) {

queue[rear++] = i;

visited[i] = 1;

break;

}

}

continue;

}

int current\_process = queue[front++];

if (front >= n \* 2) front = 0; // Circular queue

// Move current\_time if the process hasn't arrived yet

if (arrival\_time[current\_process] > current\_time) {

current\_time = arrival\_time[current\_process];

}

// Execute the process for either the time quantum or remaining time

int time\_spent = (remaining\_time[current\_process] > time\_quantum) ? time\_quantum : remaining\_time[current\_process];

current\_time += time\_spent;

remaining\_time[current\_process] -= time\_spent;

// Check for newly arrived processes and add them to the queue

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

if (arrival\_time[i] <= current\_time && remaining\_time[i] > 0 && !visited[i]) {

queue[rear++] = i;

visited[i] = 1;

if (rear >= n \* 2) rear = 0; // Circular queue

}

}

// If the current process is not finished, re-enqueue it

if (remaining\_time[current\_process] > 0) {

queue[rear++] = current\_process;

if (rear >= n \* 2) rear = 0; // Circular queue

} else {

// Process is finished

completed\_count++;

turnaround\_time[current\_process] = current\_time - arrival\_time[current\_process];

waiting\_time[current\_process] = turnaround\_time[current\_process] - cpu\_burst[current\_process];

}

}

}

// Function to print results

void print\_results(int \*waiting\_time, int \*turnaround\_time, int n, int \*arrival\_time, int \*cpu\_burst, int \*process\_id) {

double avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

printf("\nProcess\tAT\tBT\tWT\tTAT\n");

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

printf("P%d\t%d\t%d\t%d\t%d\n", process\_id[i], arrival\_time[i], cpu\_burst[i], waiting\_time[i], turnaround\_time[i]);

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

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

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

}

int main() {

int n, time\_quantum;

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

scanf("%d", &n);

if (n <= 0) {

printf("Number of processes must be greater than zero.\n");

return 1;

}

int arrival\_time[n], cpu\_burst[n], waiting\_time[n], turnaround\_time[n], process\_id[n];

// Input Arrival Time and CPU Burst Time

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

process\_id[i] = i + 1;

printf("Enter Arrival Time of process %d: ", i + 1);

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

printf("Enter CPU Burst Time of process %d: ", i + 1);

scanf("%d", &cpu\_burst[i]);

if (arrival\_time[i] < 0 || cpu\_burst[i] <= 0) {

printf("Arrival time must be non-negative, and burst time must be greater than zero.\n");

return 1;

}

}

// Sort processes by Arrival Time

sort\_by\_arrival\_time(arrival\_time, cpu\_burst, process\_id, n);

printf("\nEnter Time Quantum for Round Robin Scheduling: ");

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

if (time\_quantum <= 0) {

printf("Time Quantum must be greater than zero.\n");

return 1;

}

round\_robin(arrival\_time, cpu\_burst, n, waiting\_time, turnaround\_time, time\_quantum);

// Print Results

print\_results(waiting\_time, turnaround\_time, n, arrival\_time, cpu\_burst, process\_id);

return 0;

}

// Page Faults using FIFO

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Function to calculate page faults using FIFO

int fifoPageFaults(char referenceString[], int n, int capacity) {

int pageQueue[capacity];

int front = 0, rear = 0, size = 0;

bool pageSet[256] = {false}; // To track pages in memory

int pageFaults = 0;

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

char page = referenceString[i];

if (!pageSet[page]) { // Page not in memory

pageFaults++;

if (size == capacity) { // Memory is full

char oldestPage = pageQueue[front];

front = (front + 1) % capacity;

size--;

pageSet[oldestPage] = false;

}

pageQueue[rear] = page;

rear = (rear + 1) % capacity;

size++;

pageSet[page] = true;

}

}

return pageFaults;

}

int main() {

int capacity, n;

printf("Enter the number of page frames: ");

scanf("%d", &capacity);

printf("Enter the size of the reference string: ");

scanf("%d", &n);

char referenceString[n];

printf("Enter the elements of the reference string (separated by space): ");

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

scanf(" %c", &referenceString[i]);

}

int pageFaults = fifoPageFaults(referenceString, n, capacity);

printf("\nFIFO Page Faults: %d\n", pageFaults);

return 0;

}

// Page Faults using Optimal Replacement

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

// Function to calculate page faults using Optimal page replacement

int optimalPageFaults(char referenceString[], int n, int capacity) {

char memory[capacity];

int pageFaults = 0, size = 0;

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

char page = referenceString[i];

bool found = false;

// Check if the page is already in memory

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

if (memory[j] == page) {

found = true;

break;

}

}

if (!found) { // Page fault occurs

pageFaults++;

// If memory is not full, add the new page

if (size < capacity) {

memory[size++] = page;

} else {

// Memory is full, replace the optimal page

int farthestIndex = -1;

int replaceIndex = -1;

// Find the page that will be used the farthest in the future

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

int nextUse = INT\_MAX;

// Look for the next occurrence of the page in the reference string

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

if (memory[j] == referenceString[k]) {

nextUse = k;

break;

}

}

// If the page will not be used again, it's the best candidate for replacement

if (nextUse > farthestIndex) {

farthestIndex = nextUse;

replaceIndex = j;

}

}

// Replace the page at replaceIndex with the new page

memory[replaceIndex] = page;

}

}

}

return pageFaults;

}

int main() {

int capacity, n;

printf("Enter the number of page frames: ");

scanf("%d", &capacity);

printf("Enter the size of the reference string: ");

scanf("%d", &n);

char referenceString[n];

printf("Enter the elements of the reference string (separated by space): ");

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

scanf(" %c", &referenceString[i]);

}

// Calculate page faults using Optimal page replacement

int pageFaults = optimalPageFaults(referenceString, n, capacity);

printf("\nOptimal Page Faults: %d\n", pageFaults);

return 0;

}

// Page Fault LRU

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

// Function to calculate page faults using LRU

int lruPageFaults(char referenceString[], int n, int capacity) {

int pageList[capacity]; // To store pages in memory

int time[capacity]; // To store the last used time for each page

int pageFaults = 0;

int currentTime = 0; // To keep track of the time step

// Initialize pageList with -1 (indicating no page is loaded initially)

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

pageList[i] = -1; // No page loaded initially

time[i] = -1; // No time recorded initially

}

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

char page = referenceString[i];

bool found = false;

// Check if the page is already in memory

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

if (pageList[j] == page) {

// If found, update the last used time

time[j] = currentTime;

found = true;

break;

}

}

// If the page is not found in memory, we have a page fault

if (!found) {

pageFaults++;

// Try to place the page in an empty slot if available

bool inserted = false;

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

if (pageList[j] == -1) {

pageList[j] = page;

time[j] = currentTime;

inserted = true;

break;

}

}

// If no empty slot is found, we need to replace the least recently used page

if (!inserted) {

int lruIndex = 0;

int minTime = INT\_MAX;

// Find the least recently used page (the one with the smallest last used time)

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

if (time[j] < minTime) {

minTime = time[j];

lruIndex = j;

}

}

// Replace the least recently used page

pageList[lruIndex] = page;

time[lruIndex] = currentTime;

}

}

currentTime++;

}

return pageFaults;

}

int main() {

int capacity, n;

printf("Enter the number of page frames: ");

scanf("%d", &capacity);

printf("Enter the size of the reference string: ");

scanf("%d", &n);

char referenceString[n];

printf("Enter the elements of the reference string (separated by space): ");

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

scanf(" %c", &referenceString[i]);

}

int pageFaults = lruPageFaults(referenceString, n, capacity);

printf("\nLRU Page Faults: %d\n", pageFaults);

return 0;

}

//Banker’s Algorithm

#include <stdio.h>

int main() {

int n, m;

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

scanf("%d", &n);

printf("Enter the number of resource types: ");

scanf("%d", &m);

int max[n][m], allocation[n][m], need[n][m], available[m];

int finish[n], safeSeq[n];

printf("Enter the maximum resource instances for each resource type:\n");

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

printf("Resource type %d: ", i);

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

}

printf("Enter the maximum resource allocation for each process:\n");

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

printf("Process %d: ", i);

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

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

}

finish[i] = 0; // Initialize all processes as unfinished

}

printf("Enter the current resource allocation for each process:\n");

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

printf("Process %d: ", i);

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

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

available[j] -= allocation[i][j]; // Update available resources

}

}

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

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

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

}

}

int count = 0;

while (count < n) {

int found = 0;

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

if (finish[i] == 0) {

int j;

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

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

break;

}

}

if (j == m) {

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

available[k] += allocation[i][k];

}

safeSeq[count++] = i;

finish[i] = 1;

found = 1;

}

}

}

if (!found) {

printf("System is NOT in a safe state.\n");

return 1;

}

}

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

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

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

}

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

return 0;

}

//System Calls

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

int file\_descriptor;

char data\_to\_write[] = "Hello, System Calls!";

char buffer[100];

ssize\_t bytes\_read;

// 1. open() system call

file\_descriptor = open("example.txt", O\_WRONLY | O\_CREAT, 0644);

if (file\_descriptor == -1) {

perror("open");

exit(1);

}

// 2. write() system call

ssize\_t bytes\_written = write(file\_descriptor, data\_to\_write, sizeof(data\_to\_write) - 1);

if (bytes\_written == -1) {

perror("write");

exit(1);

}

printf("Wrote %zd bytes to the file.\n", bytes\_written);

// 3. lseek() system call

off\_t offset = lseek(file\_descriptor, 0, SEEK\_SET);

if (offset == (off\_t)-1) {

perror("lseek");

exit(1);

}

printf("Seeked to the beginning of the file.\n");

// Close the file after writing

if (close(file\_descriptor) == -1) {

perror("close");

exit(1);

}

// Reopen the file for reading

file\_descriptor = open("example.txt", O\_RDONLY);

if (file\_descriptor == -1) {

perror("open");

exit(1);

}

// 4. read() system call

bytes\_read = read(file\_descriptor, buffer, sizeof(buffer));

if (bytes\_read == -1) {

perror("read");

exit(1);

}

buffer[bytes\_read] = '\0';

printf("Read from file: %s\n", buffer);

// 5. close() system call

if (close(file\_descriptor) == -1) {

perror("close");

exit(1);

}

return 0;

}

// Pthreads

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define MAX 10 // Maximum size of the matrix

// Structure to pass arguments to the threads

typedef struct {

int row;

int col;

int (\*A)[MAX]; // Matrix A

int (\*B)[MAX]; // Matrix B

int (\*C)[MAX]; // Result matrix C

} ThreadData;

// Function to perform matrix multiplication for a single element

void\* multiply(void\* arg) {

ThreadData\* data = (ThreadData\*)arg;

int sum = 0;

// Calculate the element at C[row][col]

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

sum += data->A[data->row][i] \* data->B[i][data->col];

}

// Store the result in the C matrix

data->C[data->row][data->col] = sum;

return NULL;

}

// Function to print a matrix

void printMatrix(int matrix[MAX][MAX]) {

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

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

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

int main() {

int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];

pthread\_t threads[MAX][MAX]; // Array of threads

ThreadData threadData[MAX][MAX]; // Array of thread data

// Initialize matrices A and B

printf("Enter elements for matrix A (%dx%d):\n", MAX, MAX);

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

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

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

}

}

printf("Enter elements for matrix B (%dx%d):\n", MAX, MAX);

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

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

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

}

}

// Create threads for each element of the result matrix

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

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

threadData[i][j].row = i;

threadData[i][j].col = j;

threadData[i][j].A = A;

threadData[i][j].B = B;

threadData[i][j].C = C;

// Create thread for element C[i][j]

if (pthread\_create(&threads[i][j], NULL, multiply, (void\*)&threadData[i][j]) != 0) {

perror("Failed to create thread");

exit(1);

}

}

}

// Wait for all threads to complete

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

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

pthread\_join(threads[i][j], NULL);

}

}

// Print the result matrix

printf("\nResultant Matrix C (%dx%d):\n", MAX, MAX);

printMatrix(C);

return 0;

}