Fcfs scheduling

#include<stdio.h>

int main() {

int n, i, waiting\_time = 0, turnaround\_time = 0;

float avg\_waiting\_time, avg\_turnaround\_time;

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

scanf("%d", &n);

int bt[n], at[n], wt[n], tat[n];

// get input for burst time and arrival time of each process

printf("Enter burst time and arrival time for each process:\n");

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

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

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

}

// calculate waiting time and turnaround time for each process

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

if(i == 0) {

wt[i] = 0;

} else {

wt[i] = wt[i-1] + bt[i-1] - at[i];

if(wt[i] < 0) {

wt[i] = 0;

}

}

tat[i] = wt[i] + bt[i];

waiting\_time += wt[i];

turnaround\_time += tat[i];

}

// calculate average waiting time and average turnaround time

avg\_waiting\_time = (float)waiting\_time / n;

avg\_turnaround\_time = (float)turnaround\_time / n;

// print the results

printf("Process\tBurst Time\tArrival Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i+1, bt[i], at[i], wt[i], tat[i]);

}

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

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

return 0;

}

Sjf scheduling

#include<stdio.h>

int main() {

int n, i, j, min\_index, total\_time = 0;

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

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

scanf("%d", &n);

int bt[n], at[n], wt[n], tat[n], ct[n];

printf("Enter burst time and arrival time for each process:\n");

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

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

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

}

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

min\_index = i;

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

if(bt[j] < bt[min\_index] && at[j] <= total\_time) {

min\_index = j;

}

}

if(min\_index != i) {

int temp = bt[i];

bt[i] = bt[min\_index];

bt[min\_index] = temp;

temp = at[i];

at[i] = at[min\_index];

at[min\_index] = temp;

}

wt[i] = total\_time - at[i];

tat[i] = wt[i] + bt[i];

ct[i] = total\_time + bt[i];

total\_time += bt[i];

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

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

}

printf("Process\tBurst Time\tArrival Time\tWaiting Time\tTurnaround Time\tCompletion Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i+1, bt[i], at[i], wt[i], tat[i], ct[i]);

}

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

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

return 0;

}

Best fit

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_PROCESSES 100

int main() {

int nb, np, b[MAX\_BLOCKS], p[MAX\_PROCESSES], i, j, allocation[MAX\_PROCESSES];

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

scanf("%d", &nb);

printf("Enter the size of each block:\n");

for(i=0; i<nb; i++) scanf("%d", &b[i]);

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

scanf("%d", &np);

printf("Enter the size of each process:\n");

for(i=0; i<np; i++) scanf("%d", &p[i]);

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

allocation[i] = -1;

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

if(b[j] >= p[i] && (allocation[i] == -1 || b[j] < b[allocation[i]])) {

allocation[i] = j;

}

}

if(allocation[i] != -1) b[allocation[i]] -= p[i];

}

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

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

printf("%d\t\t%d\t\t", i+1, p[i]);

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

else printf("Not Allocated\n");

}

return 0;

}

First fit

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_PROCESSES 100

int main() {

int nb, np, b[MAX\_BLOCKS], p[MAX\_PROCESSES], i, j, allocation[MAX\_PROCESSES];

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

scanf("%d", &nb);

printf("Enter the size of each block:\n");

for(i=0; i<nb; i++) scanf("%d", &b[i]);

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

scanf("%d", &np);

printf("Enter the size of each process:\n");

for(i=0; i<np; i++) scanf("%d", &p[i]);

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

allocation[i] = -1;

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

if(b[j] >= p[i]) {

allocation[i] = j;

b[j] -= p[i];

break;

}

}

}

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

for(i=0; i<np; i++) printf("%d\t\t%d\t\t%s\n", i+1, p[i], allocation[i] != -1 ? (char[]){allocation[i]+1+'0', '\0'} : "Not Allocated");

return 0;

}

Sstf disk

#include <stdio.h>

#include <stdlib.h>

int main() {

int n, i, head, total = 0;

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the requests in order: ");

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

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

}

printf("Enter the initial position of the head: ");

scanf("%d", &head);

total += abs(head - requests[0]);

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

total += abs(requests[i] - requests[i-1]);

}

printf("Total head movement = %d", total);

return 0;

}

fcfs disk

#include <stdio.h>

#include <stdlib.h>

int main() {

int n, i, j, head, pos, total = 0, min;

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the requests: ");

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

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

printf("Enter the initial position of the head: ");

scanf("%d", &head);

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

min = 1e9;

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

if(abs(requests[j] - head) < min) {

min = abs(requests[j] - head);

pos = j;

}

}

total += min;

head = requests[pos];

requests[pos] = 1e9;

}

printf("Total head movement = %d", total);

return 0;

}

Look disk

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 1000

void swap(int x, inty) {

int temp = \*x;

x =y;

\*y = temp;

}

void sort(int arr[], int n) {

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

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

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

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

}

}

}

}

int main() {

int requests[MAX\_REQUESTS];

int n, head, total\_distance = 0;

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

scanf("%d", &n);

printf("Enter the requests: ");

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

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

}

printf("Enter the starting head position: ");

scanf("%d", &head);

sort(requests, n);

int index = 0;

while (index < n && requests[index] < head) {

index++;

}

// Move towards higher requests

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

total\_distance += abs(head - requests[i]);

head = requests[i];

}

// Move towards lower requests

for (int i = index - 1; i >= 0; i--) {

total\_distance += abs(head - requests[i]);

head = requests[i];

}

printf("Total head movement: %d\n", total\_distance);

return 0;

}

Scan disk

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 1000

void swap(int x, inty) {

int temp = \*x;

x =y;

\*y = temp;

}

void sort(int arr[], int n) {

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

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

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

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

}

}

}

}

int main() {

int requests[MAX\_REQUESTS];

int n, head, total\_distance = 0;

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

scanf("%d", &n);

printf("Enter the requests: ");

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

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

}

printf("Enter the starting head position: ");

scanf("%d", &head);

sort(requests, n);

int index = 0;

while (index < n && requests[index] < head) {

index++;

}

// Move towards higher requests

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

total\_distance += abs(head - requests[i]);

head = requests[i];

}

// Move towards lower requests

total\_distance += abs(head - requests[index-1]);

head = requests[index-1];

for (int i = index - 2; i >= 0; i--) {

total\_distance += abs(head - requests[i]);

head = requests[i];

}

printf("Total head movement: %d\n", total\_distance);

return 0;

}

Producer

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 10 // Maximum size of the shared buffer

// Define shared variables and semaphores

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty, full;

pthread\_mutex\_t mutex;

// Define Producer thread function

void\* producer(void\* arg) {

int item;

while (1) {

item = rand() % 100;

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[in] = item;

printf("Producer produced item %d at position %d\n", item, in);

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

usleep(rand() % 100000);

}

}

// Define Consumer thread function

void\* consumer(void\* arg) {

int item;

while (1) {

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[out];

printf("Consumer consumed item %d from position %d\n", item, out);

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

usleep(rand() % 100000);

}

}

int main() {

// Initialize the semaphores and mutex

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

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

pthread\_mutex\_init(&mutex, NULL);

// Create Producer and Consumer threads

pthread\_t producer\_thread, consumer\_thread;

pthread\_create(&producer\_thread, NULL, producer, NULL);

pthread\_create(&consumer\_thread, NULL, consumer, NULL);

// Wait for the threads to finish (this will never happen, since the threads run indefinitely)

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

// Destroy the semaphores and mutex

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

RR

#include <stdio.h>

#define MAX\_PROCESS 10 // Maximum number of processes

#define TIME\_QUANTUM 2 // Time quantum

struct process {

int pid; // Process ID

int burst\_time; // Burst time

int remaining\_time; // Remaining time

int waiting\_time; // Waiting time

int turnaround\_time; // Turnaround time

};

void calculate\_time(struct process p[], int n);

void display(struct process p[], int n);

int main() {

struct process p[MAX\_PROCESS];

int n, i;

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

scanf("%d", &n);

printf("Enter burst time for each process:\n");

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

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

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

p[i].remaining\_time = p[i].burst\_time;

p[i].pid = i+1;

}

calculate\_time(p, n);

display(p, n);

return 0;

}

void calculate\_time(struct process p[], int n) {

int i, time = 0, completed = 0, flag = 0;

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

while (completed != n) {

flag = 0;

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

if (p[i].remaining\_time > 0) {

flag = 1;

if (p[i].remaining\_time > TIME\_QUANTUM) {

time += TIME\_QUANTUM;

p[i].remaining\_time -= TIME\_QUANTUM;

} else {

time += p[i].remaining\_time;

p[i].waiting\_time = time - p[i].burst\_time;

p[i].remaining\_time = 0;

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

completed++;

}

}

}

if (flag == 0) // All processes are completed

break;

}

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

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

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

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

printf("Average waiting time: %.2f\n", avg\_waiting\_time);

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

}

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

int i;

printf("PID\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

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

}

}

Worst fit

#include <stdio.h>

#define MAX\_BLOCKS 10 // Maximum number of memory blocks

#define MAX\_PROCESSES 10 // Maximum number of processes

void worst\_fit(int blocks[], int m, int processes[], int n);

int main() {

int blocks[MAX\_BLOCKS], processes[MAX\_PROCESSES];

int m, n, i;

printf("Enter the number of memory blocks: ");

scanf("%d", &m);

printf("Enter the size of each memory block:\n");

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

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

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

}

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

scanf("%d", &n);

printf("Enter the size of each process:\n");

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

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

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

}

worst\_fit(blocks, m, processes, n);

return 0;

}

void worst\_fit(int blocks[], int m, int processes[], int n) {

int allocation[MAX\_PROCESSES] = {-1}; // To store block allocation for each process

int i, j;

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

int max\_block\_index = -1; // Index of the memory block with maximum size

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

if (blocks[j] >= processes[i]) {

if (max\_block\_index == -1 || blocks[j] > blocks[max\_block\_index]) {

max\_block\_index = j;

}

}

}

if (max\_block\_index != -1) {

allocation[i] = max\_block\_index;

blocks[max\_block\_index] -= processes[i];

}

}

printf("\nProcess\tProcess Size\tBlock Index\n");

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

printf("%d\t%d\t\t", i + 1, processes[i]);

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

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

} else {

printf("Not Allocated\n");

}

}

}

Next fit

#include <stdio.h>

#define MAX\_BLOCKS 10 // Maximum number of memory blocks

#define MAX\_PROCESSES 10 // Maximum number of processes

void next\_fit(int blocks[], int m, int processes[], int n);

int main() {

int blocks[MAX\_BLOCKS], processes[MAX\_PROCESSES];

int m, n, i;

printf("Enter the number of memory blocks: ");

scanf("%d", &m);

printf("Enter the size of each memory block:\n");

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

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

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

}

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

scanf("%d", &n);

printf("Enter the size of each process:\n");

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

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

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

}

next\_fit(blocks, m, processes, n);

return 0;

}

void next\_fit(int blocks[], int m, int processes[], int n) {

int allocation[MAX\_PROCESSES] = {-1}; // To store block allocation for each process

int i, j, last\_allocated = 0;

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

for (j = last\_allocated; j < m; j++) {

if (blocks[j] >= processes[i]) {

allocation[i] = j;

blocks[j] -= processes[i];

last\_allocated = j;

break;

}

}

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

for (j = 0; j < last\_allocated; j++) {

if (blocks[j] >= processes[i]) {

allocation[i] = j;

blocks[j] -= processes[i];

last\_allocated = j;

break;

}

}

}

}

printf("\nProcess\tProcess Size\tBlock Index\n");

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

printf("%d\t%d\t\t", i + 1, processes[i]);

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

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

} else {

printf("Not Allocated\n");

}

}

}

Fcfs page

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10 // Maximum number of page frames

#define MAX\_PAGES 50 // Maximum number of pages in the reference string

void fcfs(int frames[], int n, int pages[], int m);

int main() {

int frames[MAX\_FRAMES], pages[MAX\_PAGES];

int n, m, i;

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

scanf("%d", &n);

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

scanf("%d", &m);

printf("Enter the reference string:\n");

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

printf("Page %d: ", i + 1);

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

}

fcfs(frames, n, pages, m);

return 0;

}

void fcfs(int frames[], int n, int pages[], int m) {

int i, j, k = 0, faults = 0;

bool is\_present;

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

is\_present = false;

// Check if page is already present in any frame

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

if (frames[j] == pages[i]) {

is\_present = true;

break;

}

}

// If page is not present, replace the oldest page in the frame

if (!is\_present) {

frames[k] = pages[i];

k = (k + 1) % n;

faults++;

}

printf("\nReference String: ");

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

printf("%d ", pages[j]);

}

printf("\nCurrent Page Frames: ");

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

printf("%d ", frames[j]);

}

printf("\nPage Faults: %d\n", faults);

}

}

Lru

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10 // Maximum number of page frames

#define MAX\_PAGES 50 // Maximum number of pages in the reference string

void lru(int frames[], int n, int pages[], int m);

int main() {

int frames[MAX\_FRAMES], pages[MAX\_PAGES];

int n, m, i;

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

scanf("%d", &n);

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

scanf("%d", &m);

printf("Enter the reference string:\n");

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

printf("Page %d: ", i + 1);

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

}

lru(frames, n, pages, m);

return 0;

}

void lru(int frames[], int n, int pages[], int m) {

int i, j, k, faults = 0, min, max;

int counter[MAX\_FRAMES] = {0};

bool is\_present;

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

is\_present = false;

// Check if page is already present in any frame

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

if (frames[j] == pages[i]) {

is\_present = true;

counter[j] = i;

break;

}

}

// If page is not present, replace the least recently used page in the frame

if (!is\_present) {

min = counter[0];

k = 0;

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

if (counter[j] < min) {

min = counter[j];

k = j;

}

}

frames[k] = pages[i];

counter[k] = i;

faults++;

}

printf("\nReference String: ");

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

printf("%d ", pages[j]);

}

printf("\nCurrent Page Frames: ");

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

printf("%d ", frames[j]);

}

printf("\nPage Faults: %d\n", faults);

}

}

Opr

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10 // Maximum number of page frames

#define MAX\_PAGES 50 // Maximum number of pages in the reference string

void opr(int frames[], int n, int pages[], int m);

int main() {

int frames[MAX\_FRAMES], pages[MAX\_PAGES];

int n, m, i;

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

scanf("%d", &n);

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

scanf("%d", &m);

printf("Enter the reference string:\n");

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

printf("Page %d: ", i + 1);

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

}

opr(frames, n, pages, m);

return 0;

}

void opr(int frames[], int n, int pages[], int m) {

int i, j, k, faults = 0, max, index;

bool is\_present;

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

is\_present = false;

// Check if page is already present in any frame

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

if (frames[j] == pages[i]) {

is\_present = true;

break;

}

}

// If page is not present, replace the page that will not be used for the longest period of time

if (!is\_present) {

faults++;

// Find the page that will not be used for the longest period of time

max = -1;

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

for (k = i + 1; k < m; k++) {

if (frames[j] == pages[k]) {

if (k > max) {

max = k;

index = j;

}

break;

}

else if (k == m - 1 && max == -1) {

index = j;

}

}

}

frames[index] = pages[i];

}

printf("\nReference String: ");

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

printf("%d ", pages[j]);

}

printf("\nCurrent Page Frames: ");

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

printf("%d ", frames[j]);

}

printf("\nPage Faults: %d\n", faults);

}

}

Dining philosopher

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <semaphore.h>

#define NUM\_PHILOSOPHERS 5

pthread\_mutex\_t forks[NUM\_PHILOSOPHERS];

sem\_t waiter;

void \*philosopher(void \*arg);

int main() {

int i;

pthread\_t threads[NUM\_PHILOSOPHERS];

// Initialize mutexes and semaphore

sem\_init(&waiter, 0, NUM\_PHILOSOPHERS - 1);

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_mutex\_init(&forks[i], NULL);

}

// Create threads for philosophers

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_create(&threads[i], NULL, philosopher, (void \*) (long) i);

}

// Join threads

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

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

}

// Destroy mutexes and semaphore

sem\_destroy(&waiter);

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_mutex\_destroy(&forks[i]);

}

return 0;

}

void \*philosopher(void \*arg) {

int id = (int) (long) arg;

int left = id;

int right = (id + 1) % NUM\_PHILOSOPHERS;

while (1) {

// Thinking

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

// Waiting for forks

sem\_wait(&waiter);

pthread\_mutex\_lock(&forks[left]);

pthread\_mutex\_lock(&forks[right]);

// Eating

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

sleep(1);

// Releasing forks

pthread\_mutex\_unlock(&forks[right]);

pthread\_mutex\_unlock(&forks[left]);

sem\_post(&waiter);

}

}