* **Implement the solutions for following synchronization problems using semaphore:**

**i. Producer Consumer**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

sem\_t empty, full, mutex;

void \*producer(void \*arg) {

int item = 0;

while (1) {

sem\_wait(&empty);

sem\_wait(&mutex);

buffer[item] = item;

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

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

sem\_post(&mutex);

sem\_post(&full);

}

}

void \*consumer(void \*arg) {

int item;

while (1) {

sem\_wait(&full);

sem\_wait(&mutex);

item = buffer[item];

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

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

sem\_post(&mutex);

sem\_post(&empty);

}

}

int main() {

pthread\_t producerThread, consumerThread;

sem\_init(&mutex, 0, 1);

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

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

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

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

pthread\_join(producerThread, NULL);

pthread\_join(consumerThread, NULL);

sem\_destroy(&mutex);

sem\_destroy(&empty);

sem\_destroy(&full);

return 0;

}

**ii. Reader Writer**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

sem\_t mutex, wrt;

int readCount = 0;

void \*reader(void \*arg) {

while (1) {

sem\_wait(&mutex);

readCount++;

if (readCount == 1) {

sem\_wait(&wrt);

}

sem\_post(&mutex);

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

sem\_wait(&mutex);

readCount--;

if (readCount == 0) {

sem\_post(&wrt);

}

sem\_post(&mutex);

}

}

void \*writer(void \*arg) {

while (1) {

sem\_wait(&wrt);

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

sem\_post(&wrt);

}

}

int main() {

pthread\_t readerThread[5], writerThread[5];

sem\_init(&mutex, 0, 1);

sem\_init(&wrt, 0, 1);

for (int i = 0; i < 5; i++) {

pthread\_create(&readerThread[i], NULL, reader, NULL);

pthread\_create(&writerThread[i], NULL, writer, NULL);

}

for (int i = 0; i < 5; i++) {

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

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

}

sem\_destroy(&mutex);

sem\_destroy(&wrt);

return 0;

}

**iii. Dinning Philosopher**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define N 5

enum { THINKING, HUNGRY, EATING } state[N];

sem\_t mutex;

sem\_t s[N];

void test(int i) {

if (state[i] == HUNGRY && state[(i + 4) % N] != EATING && state[(i + 1) % N] != EATING) {

state[i] = EATING;

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

sem\_post(&s[i]);

}

}

void take\_forks(int i) {

sem\_wait(&mutex);

state[i] = HUNGRY;

printf("Philosopher %d is hungry\n", i);

test(i);

sem\_post(&mutex);

sem\_wait(&s[i]);

}

void put\_forks(int i) {

sem\_wait(&mutex);

state[i] = THINKING;

printf("Philosopher %d puts down forks and starts thinking\n", i);

test((i + 4) % N);

test((i + 1) % N);

sem\_post(&mutex);

}

void \*philosopher(void \*arg) {

int i = \*(int \*)arg;

while (1) {

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

sleep(1);

take\_forks(i);

sleep(1);

put\_forks(i);

}

}

int main() {

pthread\_t threads[N];

int threadArgs[N];

sem\_init(&mutex, 0, 1);

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

sem\_init(&s[i], 0, 0);

}

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

threadArgs[i] = i;

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

}

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

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

}

sem\_destroy(&mutex);

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

sem\_destroy(&s[i]);

}

return 0;

}

* **Write a program to implement following CPU scheduling algorithms:**
* **FCFS**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {

wt[0] = 0; // waiting time for first process is 0

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

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

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

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

}

}

void findAvgTime(int processes[], int n, int bt[]) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst Time Waiting Time Turn-Around Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

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

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

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

printf(" %d\n", tat[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

int processes[] = {1, 2, 3};

int n = sizeof(processes) / sizeof(processes[0]);

int burst\_time[] = {24, 3, 3};

findAvgTime(processes, n, burst\_time);

return 0;

}

* **Round Robin – time slice=4ms**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {

int rem\_bt[n];

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

rem\_bt[i] = bt[i];

int t = 0;

while (1) {

int done = 1;

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

if (rem\_bt[i] > 0) {

done = 0;

if (rem\_bt[i] > quantum) {

t += quantum;

rem\_bt[i] -= quantum;

} else {

t += rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done == 1)

break;

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

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

}

}

void findAvgTime(int processes[], int n, int bt[], int quantum) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt, quantum);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst Time Waiting Time Turn-Around Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

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

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

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

printf(" %d\n", tat[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

int processes[] = {1, 2, 3};

int n = sizeof(processes) / sizeof(processes[0]);

int burst\_time[] = {24, 3, 3};

int quantum = 4;

findAvgTime(processes, n, burst\_time, quantum);

return 0;

}

* **Write a program to implement following CPU scheduling algorithms:**
* **SJF**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {

int temp;

int pos;

int bt\_copy[n];

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

bt\_copy[i] = bt[i];

}

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

pos = i;

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

if (bt\_copy[j] < bt\_copy[pos]) {

pos = j;

}

}

temp = bt\_copy[i];

bt\_copy[i] = bt\_copy[pos];

bt\_copy[pos] = temp;

temp = processes[i];

processes[i] = processes[pos];

processes[pos] = temp;

}

wt[0] = 0; // waiting time for first process is 0

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

wt[i] = 0;

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

wt[i] += bt\_copy[j];

}

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

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

}

}

void findAvgTime(int processes[], int n, int bt[]) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst Time Waiting Time Turn-Around Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

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

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

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

printf(" %d\n", tat[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

int processes[] = {1, 2, 3};

int n = sizeof(processes) / sizeof(processes[0]);

int burst\_time[] = {24, 3, 3};

findAvgTime(processes, n, burst\_time);

return 0;

}

* **SRTF**

#include <stdio.h>

#include <limits.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {

int rt[n];

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

rt[i] = bt[i];

}

int complete = 0, t = 0, minm = INT\_MAX;

int shortest = 0, finish\_time;

int check = 0;

while (complete != n) {

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

if ((rt[j] < minm) && (rt[j] > 0)) {

minm = rt[j];

shortest = j;

check = 1;

}

}

if (check == 0) {

t++;

continue;

}

rt[shortest]--;

minm = rt[shortest];

if (minm == 0) {

minm = INT\_MAX;

}

if (rt[shortest] == 0) {

complete++;

check = 0;

finish\_time = t + 1;

wt[shortest] = finish\_time - bt[shortest];

if (wt[shortest] < 0) {

wt[shortest] = 0;

}

}

t++;

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

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

}

}

void findAvgTime(int processes[], int n, int bt[]) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst Time Waiting Time Turn-Around Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

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

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

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

printf(" %d\n", tat[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

int processes[] = {1, 2, 3};

int n = sizeof(processes) / sizeof(processes[0]);

int burst\_time[] = {24, 3, 3};

findAvgTime(processes, n, burst\_time);

return 0;

}

* **Write a program to implement following CPU scheduling algorithms:**
* **Priority –preemptive**

#include <stdio.h>

#include <limits.h>

struct Process {

int id;

int bt; // Burst time

int pri; // Priority

int rt; // Remaining time

int wt; // Waiting time

int tat; // Turn-around time

};

void findWaitingTime(struct Process proc[], int n) {

int completed = 0, t = 0, min\_pri = INT\_MAX;

int shortest = 0, finish\_time;

int check = 0;

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

proc[i].rt = proc[i].bt;

}

while (completed != n) {

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

if ((proc[j].pri < min\_pri) && (proc[j].rt > 0)) {

min\_pri = proc[j].pri;

shortest = j;

check = 1;

}

}

if (check == 0) {

t++;

continue;

}

proc[shortest].rt--;

min\_pri = proc[shortest].pri;

if (proc[shortest].rt == 0) {

completed++;

check = 0;

finish\_time = t + 1;

proc[shortest].wt = finish\_time - proc[shortest].bt;

if (proc[shortest].wt < 0) {

proc[shortest].wt = 0;

}

min\_pri = INT\_MAX;

}

t++;

}

}

void findTurnAroundTime(struct Process proc[], int n) {

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

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

}

}

void findAvgTime(struct Process proc[], int n) {

int total\_wt = 0, total\_tat = 0;

findWaitingTime(proc, n);

findTurnAroundTime(proc, n);

printf("Processes Burst Time Priority Waiting Time Turn-Around Time\n");

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

total\_wt += proc[i].wt;

total\_tat += proc[i].tat;

printf(" %d ", proc[i].id);

printf(" %d ", proc[i].bt);

printf(" %d ", proc[i].pri);

printf(" %d", proc[i].wt);

printf(" %d\n", proc[i].tat);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

struct Process proc[] = {{1, 10, 1}, {2, 1, 3}, {3, 2, 2}, {4, 1, 4}};

int n = sizeof(proc) / sizeof(proc[0]);

findAvgTime(proc, n);

return 0;

}

* **Priority –Nonpreemptive**

#include <stdio.h>

struct Process {

int id;

int bt; // Burst time

int pri; // Priority

int wt; // Waiting time

int tat; // Turn-around time

};

void findWaitingTime(struct Process proc[], int n) {

int temp;

int pos;

struct Process temp\_proc;

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

pos = i;

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

if (proc[j].pri < proc[pos].pri) {

pos = j;

}

}

temp\_proc = proc[i];

proc[i] = proc[pos];

proc[pos] = temp\_proc;

}

proc[0].wt = 0;

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

proc[i].wt = 0;

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

proc[i].wt += proc[j].bt;

}

}

}

void findTurnAroundTime(struct Process proc[], int n) {

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

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

}

}

void findAvgTime(struct Process proc[], int n) {

int total\_wt = 0, total\_tat = 0;

findWaitingTime(proc, n);

findTurnAroundTime(proc, n);

printf("Processes Burst Time Priority Waiting Time Turn-Around Time\n");

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

total\_wt += proc[i].wt;

total\_tat += proc[i].tat;

printf(" %d ", proc[i].id);

printf(" %d ", proc[i].bt);

printf(" %d ", proc[i].pri);

printf(" %d", proc[i].wt);

printf(" %d\n", proc[i].tat);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

struct Process proc[] = {{1, 10, 1}, {2, 1, 3}, {3, 2, 2}, {4, 1, 4}};

int n = sizeof(proc) / sizeof(proc[0]);

findAvgTime(proc, n);

return 0;

}

* **Write a program to implement following CPU scheduling algorithms:**
* **SJF**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {

int temp;

int pos;

int bt\_copy[n];

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

bt\_copy[i] = bt[i];

}

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

pos = i;

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

if (bt\_copy[j] < bt\_copy[pos]) {

pos = j;

}

}

temp = bt\_copy[i];

bt\_copy[i] = bt\_copy[pos];

bt\_copy[pos] = temp;

temp = processes[i];

processes[i] = processes[pos];

processes[pos] = temp;

}

wt[0] = 0; // waiting time for first process is 0

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

wt[i] = 0;

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

wt[i] += bt\_copy[j];

}

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

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

}

}

void findAvgTime(int processes[], int n, int bt[]) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst Time Waiting Time Turn-Around Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

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

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

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

printf(" %d\n", tat[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

int processes[] = {1, 2, 3};

int n = sizeof(processes) / sizeof(processes[0]);

int burst\_time[] = {24, 3, 3};

findAvgTime(processes, n, burst\_time);

return 0;

}

* **FCFS**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {

wt[0] = 0; // waiting time for first process is 0

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

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

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

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

}

}

void findAvgTime(int processes[], int n, int bt[]) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst Time Waiting Time Turn-Around Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

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

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

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

printf(" %d\n", tat[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

int processes[] = {1, 2, 3};

int n = sizeof(processes) / sizeof(processes[0]);

int burst\_time[] = {24, 3, 3};

findAvgTime(processes, n, burst\_time);

return 0;

}

* **SRTF**

#include <stdio.h>

#include <limits.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {

int rt[n];

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

rt[i] = bt[i];

}

int complete = 0, t = 0, minm = INT\_MAX;

int shortest = 0, finish\_time;

int check = 0;

while (complete != n) {

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

if ((rt[j] < minm) && (rt[j] > 0)) {

minm = rt[j];

shortest = j;

check = 1;

}

}

if (check == 0) {

t++;

continue;

}

rt[shortest]--;

minm = rt[shortest];

if (minm == 0) {

minm = INT\_MAX;

}

if (rt[shortest] == 0) {

complete++;

check = 0;

finish\_time = t + 1;

wt[shortest] = finish\_time - bt[shortest];

if (wt[shortest] < 0) {

wt[shortest] = 0;

}

}

t++;

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

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

}

}

void findAvgTime(int processes[], int n, int bt[]) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst Time Waiting Time Turn-Around Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

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

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

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

printf(" %d\n", tat[i]);

}

printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);

}

int main() {

int processes[] = {1, 2, 3};

int n = sizeof(processes) / sizeof(processes[0]);

int burst\_time[] = {6, 8, 7};

findAvgTime(processes, n, burst\_time);

return 0;

}

* **Write a program to implement bankers algorithm**

**Number of processes: 5 (P0 to P4)**

**Resources:**

**Resource A: Number of instances: 10**

**Resource B: Number of instances: 5**

**Resource C: Number of instances: 7**



* **Write a program for deadlock detection algorithm for following example.**



* **Following is the list of available blocks in the main memory and the process size for each process (P0 to P3).**

**blockSize[] = {100, 500, 200, 300, 600}**

**processSize[] = {212, 417, 112, 426}**

**Find which memory block is allocated to which process using following placement strategies:**

* **First Fit**

#include <stdio.h>

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

int allocation[n];

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

allocation[i] = -1;

}

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

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

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

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

printf("Process No. Process Size Block No.\n");

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

printf(" %d ", i);

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

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

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

} else {

printf(" Not Allocated\n");

}

}

}

int main() {

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

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

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

int n = sizeof(processSize) / sizeof(processSize[0]);

firstFit(blockSize, m, processSize, n);

return 0;

}

* **Worst fit**

#include <stdio.h>

void worstFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

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

int worstIdx = -1;

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

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

if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {

worstIdx = j;

}

}

}

if (worstIdx != -1) {

allocation[i] = worstIdx;

blockSize[worstIdx] -= processSize[i];

}

}

printf("Process No. Process Size Block No.\n");

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

printf(" %d ", i);

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

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

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

} else {

printf(" Not Allocated\n");

}

}

}

int main() {

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

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

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

int n = sizeof(processSize) / sizeof(processSize[0]);

worstFit(blockSize, m, processSize, n);

return 0;

}

* **Best fit**

#include <stdio.h>

void bestFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

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

int bestIdx = -1;

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

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

if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) {

bestIdx = j;

}

}

}

if (bestIdx != -1) {

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

printf("Process No. Process Size Block No.\n");

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

printf(" %d ", i);

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

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

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

} else {

printf(" Not Allocated\n");

}

}

}

int main() {

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

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

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

int n = sizeof(processSize) / sizeof(processSize[0]);

bestFit(blockSize, m, processSize, n);

return 0;

}

* **Next fit**

#include <stdio.h>

void nextFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

int lastIndex = 0; // Initialize the index of the last block allocated

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

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

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

allocation[i] = j;

blockSize[j] -= processSize[i];

lastIndex = j; // Update the index of the last block allocated

break;

}

}

}

printf("Process No. Process Size Block No.\n");

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

printf(" %d ", i);

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

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

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

} else {

printf(" Not Allocated\n");

}

}

}

int main() {

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

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

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

int n = sizeof(processSize) / sizeof(processSize[0]);

nextFit(blockSize, m, processSize, n);

return 0;

}

* **Consider the page reference string as 7,1,0,2,0,3,0,4,2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3-page frames. Find the total number of page faults using following page replacement algorithms:**
* **First In First Out**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_FRAMES 3

int pageFaultsFIFO(int pages[], int n) {

int frames[MAX\_FRAMES];

bool frameFilled[MAX\_FRAMES];

int pageFaults = 0;

int nextIndex = 0; // Next index to replace in the frames array

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

frames[i] = -1; // Initialize all frames to -1 (empty)

frameFilled[i] = false; // Initialize all frame filled flags to false

}

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

bool pageExists = false;

// Check if page is already in frames

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

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

pageExists = true;

break;

}

}

if (!pageExists) {

// If frame is not filled, insert page into frame

if (!frameFilled[nextIndex]) {

frames[nextIndex] = pages[i];

frameFilled[nextIndex] = true;

nextIndex = (nextIndex + 1) % MAX\_FRAMES; // Update next index to replace

} else { // If frame is filled, replace oldest page with current page

frames[nextIndex] = pages[i];

nextIndex = (nextIndex + 1) % MAX\_FRAMES; // Update next index to replace

}

pageFaults++;

}

}

return pageFaults;

}

int main() {

int pages[] = {7, 1, 0, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int n = sizeof(pages) / sizeof(pages[0]);

int pageFaults = pageFaultsFIFO(pages, n);

printf("Total number of page faults using FIFO: %d\n", pageFaults);

return 0;

}

* **Optimal**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_FRAMES 3

int pageFaultsOptimal(int pages[], int n) {

int frames[MAX\_FRAMES];

bool frameFilled[MAX\_FRAMES];

int pageFaults = 0;

int nextIndex = 0; // Next index to replace in the frames array

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

frames[i] = -1; // Initialize all frames to -1 (empty)

frameFilled[i] = false; // Initialize all frame filled flags to false

}

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

bool pageExists = false;

// Check if page is already in frames

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

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

pageExists = true;

break;

}

}

if (!pageExists) {

// If frame is not filled, insert page into frame

if (!frameFilled[nextIndex]) {

frames[nextIndex] = pages[i];

frameFilled[nextIndex] = true;

nextIndex = (nextIndex + 1) % MAX\_FRAMES; // Update next index to replace

} else { // If frame is filled, find the page that will not be used furthest in the future and replace it

int replaceIndex = -1;

int farthestIndex = -1;

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

int k;

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

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

if (k > farthestIndex) {

farthestIndex = k;

replaceIndex = j;

}

break;

}

}

if (k == n) { // Page will not be used in future

replaceIndex = j;

break;

}

}

frames[replaceIndex] = pages[i];

}

pageFaults++;

}

}

return pageFaults;

}

int main() {

int pages[] = {7, 1, 0, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int n = sizeof(pages) / sizeof(pages[0]);

int pageFaults = pageFaultsOptimal(pages, n);

printf("Total number of page faults using Optimal: %d\n", pageFaults);

return 0;

}

* **Consider the page reference string as 7,1,0,2,0,3,0,4,2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3-page frames. Find the total number of page faults using following page replacement algorithms:**
* **Least recently used**
* **Second Chance (Clock)**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 3

// Function to find the index of the least recently used frame

int findLRU(int lru[]) {

int min = lru[0];

int index = 0;

for (int i = 1; i < MAX\_FRAMES; i++) {

if (lru[i] < min) {

min = lru[i];

index = i;

}

}

return index;

}

// Function to perform Least Recently Used (LRU) page replacement algorithm

int pageFaultsLRU(int pages[], int n) {

int frames[MAX\_FRAMES];

int pageFaults = 0;

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

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

frames[i] = -1; // Initialize all frames to -1 (empty)

}

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

bool pageExists = false;

// Check if page is already in frames

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

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

pageExists = true;

// Update LRU value for the current frame

lru[j] = i + 1; // Timestamp is the index of the page reference + 1

break;

}

}

if (!pageExists) {

pageFaults++;

// Find the frame with the lowest LRU value

int replaceIndex = findLRU(lru);

// Replace the page in the frame with the lowest LRU value

frames[replaceIndex] = pages[i];

// Update LRU value for the current frame

lru[replaceIndex] = i + 1; // Timestamp is the index of the page reference + 1

}

}

return pageFaults;

}

// Function to perform Second Chance (Clock) page replacement algorithm

int pageFaultsSecondChance(int pages[], int n) {

int frames[MAX\_FRAMES];

bool referenceBit[MAX\_FRAMES];

int pageFaults = 0;

int hand = 0; // Hand points to the next frame to check for replacement

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

frames[i] = -1; // Initialize all frames to -1 (empty)

referenceBit[i] = false; // Initialize all reference bits to false

}

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

bool pageExists = false;

// Check if page is already in frames

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

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

pageExists = true;

referenceBit[j] = true; // Set the reference bit to true

break;

}

}

if (!pageExists) {

pageFaults++;

while (true) {

if (!referenceBit[hand]) {

// Replace the frame pointed by the hand

frames[hand] = pages[i];

referenceBit[hand] = true; // Set the reference bit to true

hand = (hand + 1) % MAX\_FRAMES; // Move the hand to the next frame

break;

} else {

referenceBit[hand] = false; // Reset the reference bit to false

hand = (hand + 1) % MAX\_FRAMES; // Move the hand to the next frame

}

}

}

}

return pageFaults;

}

int main() {

int pages[] = {7, 1, 0, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int n = sizeof(pages) / sizeof(pages[0]);

int pageFaultsLRUVal = pageFaultsLRU(pages, n);

printf("Total number of page faults using LRU: %d\n", pageFaultsLRUVal);

int pageFaultsSecondChanceVal = pageFaultsSecondChance(pages, n);

printf("Total number of page faults using Second Chance: %d\n", pageFaultsSecondChanceVal);

return 0;

}

* **Consider the page reference string as 7,1,0,2,0,3,0,4,2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3-page frames. Find the total number of page faults using following page replacement algorithms:**
* **Least recently used**
* **Optimal**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX\_FRAMES 100

int lruPageFaults(int pages[], int numPages, int numFrames) {

int pageMap[MAX\_FRAMES];

int lruList[MAX\_FRAMES];

int pageFaults = 0;

int i, j;

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

pageMap[i] = -1;

lruList[i] = 0;

}

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

int page = pages[i];

int found = 0;

int leastRecentPage = 0; // Declare leastRecentPage here

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

if (pageMap[j] == page) {

found = 1;

break;

}

}

if (!found) {

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

if (lruList[j] < lruList[leastRecentPage]) {

leastRecentPage = j;

}

}

pageMap[leastRecentPage] = page;

pageFaults++;

}

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

if (j != leastRecentPage) {

lruList[j]++;

} else {

lruList[j] = 0;

}

}

}

return pageFaults;

}

int optimalPageFaults(int pages[], int numPages, int numFrames) {

int nextPageMap[MAX\_FRAMES];

int pageMap[MAX\_FRAMES];

int pageFaults = 0;

int i, j;

for (i = 0; i < MAX\_FRAMES; ++i) {

nextPageMap[i] = INT\_MAX;

}

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

nextPageMap[pages[i]] = i;

int found = 0;

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

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

found = 1;

break;

}

}

if (!found) {

if (numFrames > pageFaults) {

pageMap[pageFaults] = pages[i];

pageFaults++;

} else {

int pageToRemove = -1;

int farthestNextPage = INT\_MIN;

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

if (nextPageMap[pageMap[j]] > farthestNextPage) {

farthestNextPage = nextPageMap[pageMap[j]];

pageToRemove = j;

}

}

pageMap[pageToRemove] = pages[i];

pageFaults++;

}

}

}

return pageFaults;

}

int main() {

int pageReferenceString[] = {7, 1, 0, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int numFrames = 3;

int numPages = sizeof(pageReferenceString) / sizeof(pageReferenceString[0]);

int lruFaults = lruPageFaults(pageReferenceString, numPages, numFrames);

int optimalFaults = optimalPageFaults(pageReferenceString, numPages, numFrames);

printf("Total page faults using LRU: %d\n", lruFaults);

printf("Total page faults using Optimal: %d\n", optimalFaults);

return 0;

}

* **Consider the page reference string as 7,1,0,2,0,3,0,4,2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3-page frames. Find the total number of page faults using following page replacement algorithms:**

**1) Least recently used**

**2)FIFO**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 3

// Function to find the index of the least recently used frame

int findLRU(int lru[]) {

int min = lru[0];

int index = 0;

for (int i = 1; i < MAX\_FRAMES; i++) {

if (lru[i] < min) {

min = lru[i];

index = i;

}

}

return index;

}

// Function to perform Least Recently Used (LRU) page replacement algorithm

int pageFaultsLRU(int pages[], int n) {

int frames[MAX\_FRAMES];

int pageFaults = 0;

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

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

frames[i] = -1; // Initialize all frames to -1 (empty)

}

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

bool pageExists = false;

// Check if page is already in frames

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

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

pageExists = true;

// Update LRU value for the current frame

lru[j] = i + 1; // Timestamp is the index of the page reference + 1

break;

}

}

if (!pageExists) {

pageFaults++;

// Find the frame with the lowest LRU value

int replaceIndex = findLRU(lru);

// Replace the page in the frame with the lowest LRU value

frames[replaceIndex] = pages[i];

// Update LRU value for the current frame

lru[replaceIndex] = i + 1; // Timestamp is the index of the page reference + 1

}

}

return pageFaults;

}

// Function to perform First In First Out (FIFO) page replacement algorithm

int pageFaultsFIFO(int pages[], int n) {

int frames[MAX\_FRAMES];

int pageFaults = 0;

int nextIndex = 0; // Next index to replace in the frames array

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

frames[i] = -1; // Initialize all frames to -1 (empty)

}

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

bool pageExists = false;

// Check if page is already in frames

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

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

pageExists = true;

break;

}

}

if (!pageExists) {

// If frame is not filled, insert page into frame

if (nextIndex < MAX\_FRAMES) {

frames[nextIndex] = pages[i];

nextIndex++;

} else { // If frame is filled, replace oldest page with current page

frames[0] = pages[i];

}

pageFaults++;

}

}

return pageFaults;

}

int main() {

int pages[] = {7, 1, 0, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int n = sizeof(pages) / sizeof(pages[0]);

int pageFaultsLRUVal = pageFaultsLRU(pages, n);

printf("Total number of page faults using LRU: %d\n", pageFaultsLRUVal);

int pageFaultsFIFOVal = pageFaultsFIFO(pages, n);

printf("Total number of page faults using FIFO: %d\n", pageFaultsFIFOVal);

return 0;

}

* **Consider the order of request for cylinder access of the disk is - (98, 183, 37, 122, 14, 124, 65, 67) and current position of Read/Write head is: 53. Find the total head movements to access the data on the cylinder using following disk scheduling algorithms:**
* **FCFS**
* **Shortest Seek Time First (SSTF)**

#include <stdio.h>

#include <stdlib.h>

// Function to calculate total head movements using FCFS algorithm

int totalHeadMovementsFCFS(int requests[], int n, int current) {

int totalMovements = 0;

// Iterate through requests and calculate head movements

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

totalMovements += abs(current - requests[i]);

current = requests[i];

}

return totalMovements;

}

// Function to calculate total head movements using SSTF algorithm

int totalHeadMovementsSSTF(int requests[], int n, int current) {

int totalMovements = 0;

int visited[n];

// Mark all requests as not visited

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

visited[i] = 0;

}

// Iterate through requests

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

int minDistance = \_\_INT\_MAX\_\_;

int nextRequest;

// Find the nearest unvisited request

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

if (!visited[j] && abs(current - requests[j]) < minDistance) {

minDistance = abs(current - requests[j]);

nextRequest = j;

}

}

// Update current position and total movements

totalMovements += minDistance;

current = requests[nextRequest];

visited[nextRequest] = 1;

}

return totalMovements;

}

int main() {

int requests[] = {98, 183, 37, 122, 14, 124, 65, 67};

int n = sizeof(requests) / sizeof(requests[0]);

int current = 53;

// Calculate total head movements using FCFS

int totalMovementsFCFS = totalHeadMovementsFCFS(requests, n, current);

printf("Total head movements using FCFS: %d\n", totalMovementsFCFS);

// Calculate total head movements using SSTF

int totalMovementsSSTF = totalHeadMovementsSSTF(requests, n, current);

printf("Total head movements using SSTF: %d\n", totalMovementsSSTF);

return 0;

}

* **Consider the order of request for cylinder access of the disk is - (98, 183, 37, 122, 14, 124, 65, 67) and current position of Read/Write head is: 53. Find the total head movements to access the data on the cylinder using following disk scheduling algorithms:**
* **SCAN**
* **C-SCAN**

#include <stdio.h>

#include <stdlib.h>

// Function to calculate total head movements using SCAN algorithm

int totalHeadMovementsSCAN(int requests[], int n, int current, int direction, int cylinderSize) {

int totalMovements = 0;

int visited[n];

// Mark all requests as not visited

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

visited[i] = 0;

}

// Move the head in the specified direction

while (current >= 0 && current <= cylinderSize) {

// Check if the current request exists

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

if (requests[i] == current) {

visited[i] = 1;

totalMovements++;

}

}

// Move the head in the specified direction

current += direction;

}

return totalMovements;

}

// Function to calculate total head movements using C-SCAN algorithm

int totalHeadMovementsCSCAN(int requests[], int n, int current, int direction, int cylinderSize) {

int totalMovements = 0;

int visited[n];

// Mark all requests as not visited

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

visited[i] = 0;

}

// Move the head in the specified direction

while (current >= 0 && current <= cylinderSize) {

// Check if the current request exists

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

if (requests[i] == current) {

visited[i] = 1;

totalMovements++;

}

}

// Move the head in the specified direction

current += direction;

}

// Wrap around to the beginning of the disk

current = 0;

while (current >= 0 && current <= cylinderSize) {

// Check if the current request exists

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

if (requests[i] == current && !visited[i]) {

visited[i] = 1;

totalMovements++;

}

}

// Move the head in the specified direction

current += direction;

}

return totalMovements;

}

int main() {

int requests[] = {98, 183, 37, 122, 14, 124, 65, 67};

int n = sizeof(requests) / sizeof(requests[0]);

int current = 53;

int cylinderSize = 199; // Assuming cylinder size is 200 (0 to 199)

// Calculate total head movements using SCAN

int totalMovementsSCAN = totalHeadMovementsSCAN(requests, n, current, 1, cylinderSize);

printf("Total head movements using SCAN: %d\n", totalMovementsSCAN);

// Calculate total head movements using C-SCAN

int totalMovementsCSCAN = totalHeadMovementsCSCAN(requests, n, current, 1, cylinderSize);

printf("Total head movements using C-SCAN: %d\n", totalMovementsCSCAN);

return 0;

}

* **Consider the order of request for cylinder access of the disk is - (98, 183, 37, 122, 14, 124, 65, 67) and current position of Read/Write head is: 53. Find the total head movements to access the data on the cylinder using following disk scheduling algorithms:**
* **SSTF**
* **SCAN**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Function to calculate total head movements using SSTF algorithm

int totalHeadMovementsSSTF(int requests[], int n, int current) {

int totalMovements = 0;

int visited[n];

// Mark all requests as not visited

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

visited[i] = 0;

}

// Iterate through requests

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

int minDistance = INT\_MAX;

int nextRequest = -1;

// Find the nearest unvisited request

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

if (!visited[j] && abs(current - requests[j]) < minDistance) {

minDistance = abs(current - requests[j]);

nextRequest = j;

}

}

// Update current position and total movements

totalMovements += minDistance;

current = requests[nextRequest];

visited[nextRequest] = 1;

}

return totalMovements;

}

// Function to calculate total head movements using SCAN algorithm

int totalHeadMovementsSCAN(int requests[], int n, int current, int direction, int cylinderSize) {

int totalMovements = 0;

int visited[n];

// Mark all requests as not visited

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

visited[i] = 0;

}

// Move the head in the specified direction

while (current >= 0 && current <= cylinderSize) {

// Check if the current request exists

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

if (requests[i] == current) {

visited[i] = 1;

totalMovements++;

}

}

// Move the head in the specified direction

current += direction;

}

return totalMovements;

}

int main() {

int requests[] = {98, 183, 37, 122, 14, 124, 65, 67};

int n = sizeof(requests) / sizeof(requests[0]);

int current = 53;

int cylinderSize = 199; // Assuming cylinder size is 200 (0 to 199)

// Calculate total head movements using SSTF

int totalMovementsSSTF = totalHeadMovementsSSTF(requests, n, current);

printf("Total head movements using SSTF: %d\n", totalMovementsSSTF);

// Calculate total head movements using SCAN

int totalMovementsSCAN = totalHeadMovementsSCAN(requests, n, current, 1, cylinderSize);

printf("Total head movements using SCAN: %d\n", totalMovementsSCAN);

return 0;

}

* **Consider the order of request for cylinder access of the disk is - (98, 183, 37, 122, 14, 124, 65, 67) and current position of Read/Write head is: 53. Find the total head movements to access the data on the cylinder using following disk scheduling algorithms:**
* **FCFS**
* **SCAN**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Function to calculate total head movements using SSTF algorithm

int totalHeadMovementsSSTF(int requests[], int n, int current) {

int totalMovements = 0;

int visited[n];

// Mark all requests as not visited

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

visited[i] = 0;

}

// Iterate through requests

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

int minDistance = INT\_MAX;

int nextRequest = -1;

// Find the nearest unvisited request

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

if (!visited[j] && abs(current - requests[j]) < minDistance) {

minDistance = abs(current - requests[j]);

nextRequest = j;

}

}

// Update current position and total movements

totalMovements += minDistance;

current = requests[nextRequest];

visited[nextRequest] = 1;

}

return totalMovements;

}

// Function to calculate total head movements using SCAN algorithm

int totalHeadMovementsSCAN(int requests[], int n, int current, int direction, int cylinderSize) {

int totalMovements = 0;

int visited[n];

// Mark all requests as not visited

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

visited[i] = 0;

}

// Move the head in the specified direction

while (current >= 0 && current <= cylinderSize) {

// Check if the current request exists

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

if (requests[i] == current) {

visited[i] = 1;

totalMovements++;

}

}

// Move the head in the specified direction

current += direction;

}

return totalMovements;

}

int main() {

int requests[] = {98, 183, 37, 122, 14, 124, 65, 67};

int n = sizeof(requests) / sizeof(requests[0]);

int current = 53;

int cylinderSize = 199; // Assuming cylinder size is 200 (0 to 199)

// Calculate total head movements using SSTF

int totalMovementsSSTF = totalHeadMovementsSSTF(requests, n, current);

printf("Total head movements using SSTF: %d\n", totalMovementsSSTF);

// Calculate total head movements using SCAN

int totalMovementsSCAN = totalHeadMovementsSCAN(requests, n, current, 1, cylinderSize);

printf("Total head movements using SCAN: %d\n", totalMovementsSCAN);

return 0;

}

* **Implement the following algorithms:**
* **FCFS- CPU scheduling algo**
* **Optimal –page replacement algo –( Consider the page reference string as 7,1,0,2,0,3,0,4,2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3-page frames. Find the total number of page faults)**
* **Implement the following algorithms:**
* **SJF- CPU scheduling algo**
* **FIFO –page replacement algo –( Consider the page reference string as 7,1,0,2,0,3,0,4,2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3-page frames. Find the total number of page faults)**
* **Implement the following algorithms:**
* **Round Robin- CPU scheduling algo(time slice =4ms)**
* **FIFO –page replacement algo –( Consider the page reference string as 7,1,0,2,0,3,0,4,2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3-page frames. Find the total number of page faults)**
* **Implement the following algorithms:**
* **FCFS- Disk scheduling algo (Consider the order of request for cylinder access of the disk is - (98, 183, 37, 122, 14, 124, 65, 67) and current position of Read/Write head is: 53. Find the total head movements to access the data on the cylinder)**
* **Best Fit –memory placement algo**

**(process (P0 to P3).**

**blockSize[] = {100, 500, 200, 300, 600}**

**processSize[] = {212, 417, 112, 426} Find which memory block is allocated to which process**

* **Implement the following algorithms:**
* **SSTF- Disk scheduling algo (Consider the order of request for cylinder access of the disk is - (98, 183, 37, 122, 14, 124, 65, 67) and current position of Read/Write head is: 53. Find the total head movements to access the data on the cylinder)**
* **First Fit –memory placement algo**

**(process (P0 to P3).**

**blockSize[] = {100, 500, 200, 300, 600}**

**processSize[] = {212, 417, 112, 426} Find which memory block is allocated to which process**

* **Implement the following algorithms:**
* **SCAN- Disk scheduling algo (Consider the order of request for cylinder access of the disk is - (98, 183, 37, 122, 14, 124, 65, 67) and current position of Read/Write head is: 53. Find the total head movements to access the data on the cylinder)**
* **Worst Fit –memory placement algo**

**(process (P0 to P3).**

**blockSize[] = {100, 500, 200, 300, 600}**

**processSize[] = {212, 417, 112, 426} Find which memory block is allocated to which process**

* **Write a program to implement bankers algorithm**

**Number of processes: 5 (P0 to P4)**

**Resources:**

**Resource A: Number of instances: 10**

**Resource B: Number of instances: 5**

**Resource C: Number of instances: 7**



* **Write a program for deadlock detection algorithm for following example.**



* **Write a program to implement bankers algorithm**

**Number of processes: 5 (P0 to P4)**

**Resources:**

**Resource A: Number of instances: 10**

**Resource B: Number of instances: 5**

**Resource C: Number of instances: 7**



* **Write a program for deadlock detection algorithm for following example.**



* **Consider the page reference string as 7,1,0,2,0,3,0,4,2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3-page frames. Find the total number of page faults using following page replacement algorithms:**
* **First In First Out**
* **Optimal**

#include <stdio.h>

#include <stdbool.h>

// Function to find if a page exists in a frame

bool isPageInFrame(int page, int frame[], int numFrames) {

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

if (frame[i] == page) {

return true;

}

}

return false;

}

// Function to find the index of the oldest page in the frame

int findOldestPageIndex(int frame[], int numFrames, int refStr[], int refStrSize, int currentIndex) {

int oldestIndex = 0;

int maxIndex = -1;

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

int j;

for (j = currentIndex - 1; j >= 0; j--) {

if (frame[i] == refStr[j]) {

if (j > maxIndex) {

maxIndex = j;

oldestIndex = i;

}

break;

}

}

if (j == -1) {

oldestIndex = i;

break;

}

}

return oldestIndex;

}

// Function to simulate FIFO page replacement algorithm

int pageFaultsFIFO(int refStr[], int refStrSize, int numFrames) {

int frame[numFrames];

int pageFaults = 0;

int frameIndex = 0;

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

frame[i] = -1; // Initialize frame to -1 (empty)

}

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

if (!isPageInFrame(refStr[i], frame, numFrames)) {

frame[frameIndex] = refStr[i];

frameIndex = (frameIndex + 1) % numFrames;

pageFaults++;

}

}

return pageFaults;

}

// Function to simulate Optimal page replacement algorithm

int pageFaultsOptimal(int refStr[], int refStrSize, int numFrames) {

int frame[numFrames];

int pageFaults = 0;

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

frame[i] = -1; // Initialize frame to -1 (empty)

}

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

if (!isPageInFrame(refStr[i], frame, numFrames)) {

int oldestPageIndex = findOldestPageIndex(frame, numFrames, refStr, refStrSize, i);

frame[oldestPageIndex] = refStr[i];

pageFaults++;

}

}

return pageFaults;

}

int main() {

int refStr[] = {7, 1, 0, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int refStrSize = sizeof(refStr) / sizeof(refStr[0]);

int numFrames = 3;

int pageFaultsFIFOVal = pageFaultsFIFO(refStr, refStrSize, numFrames);

printf("Total number of page faults using FIFO: %d\n", pageFaultsFIFOVal);

int pageFaultsOptimalVal = pageFaultsOptimal(refStr, refStrSize, numFrames);

printf("Total number of page faults using Optimal: %d\n", pageFaultsOptimalVal);

return 0;

}