**FCFS,SCAN,CSCAN:**#include <stdio.h>

#include <stdlib.h>

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

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

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

if (req[j] > req[j + 1]){

int temp = req[j];

req[j] = req[j + 1];

req[j + 1] = temp;

} } }}

void FCFS(int req[], int head, int n){

int ans = abs(head - req[0]);

printf("seek sequence:%d ",head);

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

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

ans += abs(req[i] - req[i + 1]);

}

printf("\n%d\n",ans);

}

void SCAN(int req[], int head, int trackMin, int trackMax, int n){

int temp[n + 1];

int ptr = 0;

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

if (req[i] >= head)

temp[ptr++] = req[i];

}

temp[ptr++] = (trackMax);

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

if (req[i] < head)

temp[ptr++] = (req[i]);

}

FCFS(temp, head, n + 1);

}

void C\_SCAN(int req[], int head, int trackMin, int trackMax, int n){

int temp[n + 2], ptr = 0;

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

if (req[i] >= head)

temp[ptr++] = req[i];

}

temp[ptr++] = (trackMax);

temp[ptr++] = (trackMin);

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

if (req[i] < head)

temp[ptr++] = (req[i]);

}

FCFS(temp, head, n + 2);

}

int main(){

int trackMin, trackMax,n, head;

printf("enter trackmin and trackmax:");

scanf("%d%d", &trackMin, &trackMax);

printf("enter number of tracks:");

scanf("%d", &n);

printf("enter current track position:");

scanf("%d",&head);

int req[n];

printf("enter sequence:");

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

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

printf("\nFCFS:\n");

FCFS(req, head, n);

sort(req,n);

printf("\nSCAN:\n");

SCAN(req, head, trackMin, trackMax, n);

printf("\nC\_SCAN:\n");

C\_SCAN(req, head, trackMin, trackMax, n);

return 0;}

**FIFO,LRU**#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int isPresent(int cache[], int size, int key) {

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

if (cache[i] == key)

return 1;

}

return 0;

}

void FIFO(int req[], int n, int pageSize) {

int cache[pageSize];

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

cache[i] = -1;

int hits = 0;

int ptr = 0;

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

if (isPresent(cache, pageSize, req[i])) {

hits++;

} else {

cache[ptr] = req[i];

ptr = (ptr + 1) % pageSize;

}

}

printf("%d", hits);

}

int getLRURank(int req[], int key, int pos) {

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

if (req[i] == key)

return i;

}

return -1;

}

void LRU(int req[], int n, int pageSize) {

int cache[pageSize];

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

cache[i] = -1;

int hits = 0;

int ptr = 0;

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

if (isPresent(cache, pageSize, req[i])) {

hits++;

} else {

if (ptr < pageSize) {

cache[ptr++] = req[i];

continue;

}

int minRank = INT\_MAX;

int minPos = 0;

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

int r = getLRURank(req, cache[j], i);

if (r < minRank) {

minRank = r;

minPos = j;

}

}

cache[minPos] = req[i];

}

}

printf("%d", hits);

}

int getOptimalRank(int req[], int key, int pos, int n) {

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

if (req[i] == key)

return i;

}

return n;

}

void Optimal(int req[], int n, int pageSize) {

int cache[pageSize];

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

cache[i] = -1;

int hits = 0;

int ptr = 0;

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

if (isPresent(cache, pageSize, req[i])) {

hits++;

} else {

if (ptr < pageSize) {

cache[ptr++] = req[i];

continue;

}

int maxRank = INT\_MIN;

int maxPos = 0;

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

int r = getOptimalRank(req, cache[j], i, n);

if (r > maxRank) {

maxRank = r;

maxPos = j;

}

}

cache[maxPos] = req[i];

}

}

printf("%d", hits);

}

int main() {

int n, pageSize;

printf("enter number of requests:");

scanf("%d", &n);

printf("enter page size:");

scanf("%d",&pageSize);

printf("enter page requests");

int req[n];

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

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

}

printf("FIFO:");

FIFO(req, n, pageSize);

printf("\nLRU:");

LRU(req, n, pageSize);

printf("\nOPTIMAL:");

Optimal(req, n, pageSize);

return 0;

}

**BEST,NEXT,FIRST,WORST FIT:**#include <stdio.h>

#include <limits.h>

struct Process

{

char id;

int size;

char alloc;

};

struct Partition

{

char id;

int size;

};

void FirstFit(struct Process proc[], int numProc, struct Partition parts[], int numParts){

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

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

if (parts[j].size >= proc[i].size)

{

proc[i].alloc = parts[j].id;

parts[j].size = parts[j].size - proc[i].size;

break;

}

}

}

}

void BestFit(struct Process proc[], int numProc, struct Partition parts[], int numParts){

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

int minDif = INT\_MAX;

struct Partition \*best = NULL;

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

if (parts[j].size >= proc[i].size && parts[j].size - proc[i].size < minDif)

{

best = &parts[j];

minDif = parts[j].size - proc[i].size;

}

}

if (!best)

continue;

proc[i].alloc = best->id;

best->size = best->size - proc[i].size;

}

}

void WorstFit(struct Process proc[], int numProc, struct Partition parts[], int numParts){

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

int maxDif = INT\_MIN;

struct Partition \*worst = NULL;

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

if (parts[j].size >= proc[i].size && parts[j].size - proc[i].size > maxDif)

{

worst = &parts[j];

maxDif = parts[j].size - proc[i].size;

}

}

if (!worst)

continue;

proc[i].alloc = worst->id;

worst->size = worst->size - proc[i].size;

}

}

int main(){

int numParts;

printf("enter no of partitions:");

scanf("%d", &numParts);

struct Partition parts[numParts];

printf("enter partitions(id)(size):\n");

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

scanf(" %c %d", &parts[i].id, &parts[i].size);

int numProc;

printf("enter no of processes:");

scanf("%d", &numProc);

struct Process proc[numProc];

printf("enter processes(id)(size):\n");

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

{

scanf(" %c %d", &proc[i].id, &proc[i].size);

proc[i].alloc = 'X';

}

int choice;

printf("\n1.First\t2.Best\t3.Worst\nchoice:");

scanf("%d",&choice);

if(choice==1)

FirstFit(proc, numProc, parts, numParts);

else if(choice==2)

BestFit(proc, numProc, parts, numParts);

else if(choice==3)

WorstFit(proc, numProc, parts, numParts);

else

printf("wrong choice");

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

printf("%c %c\n", proc[i].id, proc[i].alloc);

return 0;

}

**FORK:**

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <unistd.h>

int wait\_func() {

int pid\_1 = fork();

if (pid\_1 == 0) {

printf("Process pid\_1 successfully created\n");

printf("Process pid\_1 id: %d\n", getpid());

exit(0);

}

waitpid(pid\_1, NULL, 0);

printf("pid\_1 process terminated.\n");

fflush(stdout); // Flush output buffer

return 0;

}

int main() {

int pid = fork();

if (pid == 0) {

printf("Child process created\n");

printf("Current child is: %d\n", getppid());

printf("Current parent is: %d\n", getpid());

exit(0);

}

wait(NULL);

printf("Child process is terminated.\n");

fflush(stdout); // Flush output buffer

wait\_func();

return 0;

}

**BANKERS:**

#include <stdio.h>

void ip(int arr[][10],int p, int r){

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

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

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

}

}

int main() {

int p, r, count = 0, i, j, alloc[10][10], max[10][10], need[10][10], safeseq[10], avail[10], finish[10], terminate = 0;

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

scanf("%d %d", &p, &r);

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

finish[i] = 0;

printf("Enter allocation of resources:\n");

ip(alloc,p,r);

printf("Enter the maximum:\n");

ip(max,p,r);

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

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

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

printf("\nNeed resources matrix:\n");

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

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

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

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

}

printf("\n");

}

while (count < p) {

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

if (finish[i] == 0) {

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

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

break;

}

if (j == r) {

safeseq[count] = i;

finish[i] = 1;

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

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

count++;

terminate = 0;

}

else

terminate++;

}

}

if (terminate == (p - 1)) {

printf("safe sequence does not exist.\n");

break;

}

}

if (terminate != (p - 1)) {

printf("\n\navail resource after completion:\n");

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

printf("%d\t", avail[j]);

printf("\nsafeseq sequence:\n");

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

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

}

return 0;}

**DINING PHIL:**

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<unistd.h>

#include<semaphore.h>

sem\_t room;

sem\_t chopstick[5];

void eat(int phil){

printf("\nphil%d is eating",phil);

}

void \*phil(void \* num){

int p=\*(int \*)num;

sem\_wait(&room);

printf("\nphil%d has enterd room",p);

sem\_wait(&chopstick[p]);

sem\_wait(&chopstick[(p+1)%5]);

eat(p);

sleep(2);

printf("\nphil%d has finished eating",p);

sem\_post(&chopstick[(p+1)%5]);

sem\_post(&chopstick[p]);

sem\_post(&room);

}

int main(){

int i,a[5];

sem\_init(&room,0,4);

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

sem\_init(&chopstick[i],0,1);

}

pthread\_t tid[5];

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

a[i]=i;

pthread\_create(&tid[i],NULL,phil,(void \*)&a[i]);

}

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

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

}

}

**PRODUCER CONSUMER:**#include<stdio.h>

#include<stdlib.h>

int full = 0, empty = 10, x = 0, mutex = 1;

void Producer() {

--mutex;

++full;

--empty;

x++;

printf("Producer produces the item %d", x);

++mutex;

}

void Consumer() {

--mutex;

--full;

++empty;

printf("Consumer consumes the item %d", x);

x--;

++mutex;

}

int main() {

int i,choice;

printf("\n1.Producer \n2.Consumer \n3.Exit");

#pragma omp critical

for (i = 1; i > 0; i++) {

printf("\nEnter the choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

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

Producer();

else

printf("Buffer is full");

break;

case 2:

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

Consumer();

else

printf("Buffer is empty");

break;

case 3:

printf("Exiting......");

exit(0);

default:

printf("Invalid choice!!!!");

break;

}

}

}

**NON PREMPTIVE: FCFS**#include <stdio.h>

struct Process {

int pid;

int at;

int bt;

};

int sort(const void\* a, const void\* b) {

return ((struct Process\*)a)->at - ((struct Process\*)b)->at;

}

int main() {

int n;

printf("Enter Number of processes : ");

scanf("%d", &n);

struct Process p[n];

printf("Enter process id, arrival time, burst time : \n");

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

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

qsort(p, n, sizeof(struct Process), sort);

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

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

int count = p[0].at;

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

if (p[i].at <= count) {

count += p[i].bt;

ct[i] = count;

tat[i] = ct[i] - p[i].at;

total\_tat += tat[i];

wt[i] = tat[i] - p[i].bt;

total\_wt += wt[i];

} else {

count++;

i--;

}

}

printf("\n%s\t%s\t%s\t%s\t%s\t%s\n","PID","AT","BT", "CT", "TAT", "WT");

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

printf(" %d\t%d\t%d\t%d\t%d\t%d\n",p[i].pid,p[i].at,p[i].bt,ct[i],tat[i],wt[i]);

printf("\nAvg WT : %.2f\n", (total\_wt / n));

printf("Avg TAT : %.2f\n", (total\_tat / n));

return 0;

}

**PREEMPTIVE : SRTF**

#include <stdio.h>

#include <limits.h>

struct Process {

int pid;

int at;

int bt;

};

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

int remt[n];

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

remt[i] = p[i].bt;

int completed = 0, currt = 0, minremt = INT\_MAX,shortestp, finish\_time, check = 0;

while (completed != n) {

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

if ((p[j].at <= currt) && (remt[j] < minremt) && remt[j] > 0) {

minremt = remt[j];

shortestp = j;

check = 1;

}

}

if (check == 0) {

currt++;

continue;

}

remt[shortestp]--;

minremt = remt[shortestp];

if (minremt == 0)

minremt = INT\_MAX;

if (remt[shortestp] == 0) {

completed++;

check = 0;

finish\_time = currt + 1;

wt[shortestp] = finish\_time - p[shortestp].bt - p[shortestp].at;

if (wt[shortestp] < 0)

wt[shortestp] = 0;

}

currt++;

}

}

void SRTF(struct Process p[], int n) {

int wt[n], tat[n];

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

findWaitingTime(p, n, wt);

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

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

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

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

total\_wt += wt[i];

total\_tat += tat[i];

printf(" %d\t%d\t%d\t%d\t%d\n", p[i].pid,p[i].at, p[i].bt, wt[i], tat[i]);

}

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

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

}

int main() {

int n;

printf("Enter no. of processes : ");

scanf("%d", &n);

struct Process p[n];

printf("Enter process id, arrival time and burst times : \n");

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

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

SRTF(p, n);

return 0;}