**Practical-8**

**Aim:** **Thread management using pthread library. Write a simple program to understand it.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void \*print\_message\_function(void \*ptr);

main()

{

pthread\_t thread1, thread2;

char \*message1 = "Thread 1";

char \*message2 = "Thread 2";

int iret1, iret2;

iret1 = pthread\_create(&thread1, NULL, print\_message\_function, (void \*)message1);

iret2 = pthread\_create(&thread2, NULL, print\_message\_function, (void \*)message2);

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

printf("Thread 1 returns: %d\n", iret1);

printf("Thread 2 returns: %d\n", iret2);

exit(0);

}

void \*print\_message\_function(void \*ptr)

{

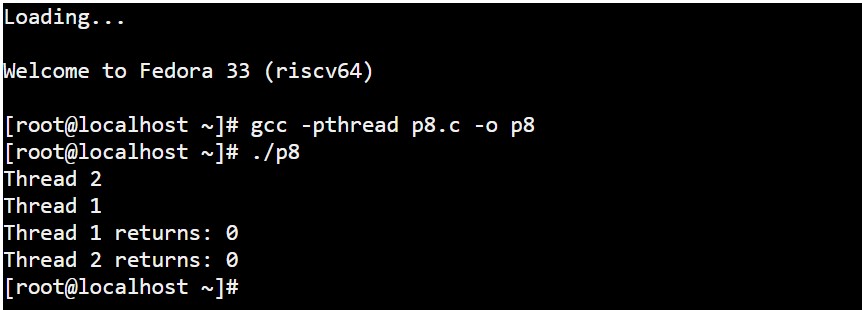
char \*message;

message = (char \*)ptr;

printf("%s \n", message);

}

**Output:**



**Practical-9**

**Aim:** **Write a C program in LINUX to implement inter process communication (IPC) Using Semaphore**

**Code:**

#include<stdio.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/sem.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#define SHM\_KEY 0x12345

#define SEM\_KEY 0x54321

#define MAX\_TRIES 20

struct shmseg {

int cntr;

int write\_complete;

int read\_complete;

};

void shared\_memory\_cntr\_increment(int, struct shmseg\*, int);

void remove\_semaphore();

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

int shmid;

struct shmseg \*shmp;

char \*bufptr;

int total\_count;

int sleep\_time;

pid\_t pid;

if (argc != 2)

total\_count = 10000;

else {

total\_count = atoi(argv[1]);

if (total\_count < 10000)

total\_count = 10000;

}

printf("Total Count is %d\n", total\_count);

shmid = shmget(SHM\_KEY, sizeof(struct shmseg), 0644|IPC\_CREAT);

if (shmid == -1) {

perror("Shared memory");

return 1;

}

// Attach to the segment to get a pointer to it.

shmp = shmat(shmid, NULL, 0);

if (shmp == (void \*) -1) {

perror("Shared memory attach: ");

return 1;

}

shmp->cntr = 0;

pid = fork();

if (pid > 0) {

shared\_memory\_cntr\_increment(pid, shmp, total\_count);

} else if (pid == 0) {

shared\_memory\_cntr\_increment(pid, shmp, total\_count);

return 0;

} else {

perror("Fork Failure\n");

return 1;

}

while (shmp->read\_complete != 1)

sleep(1);

if (shmdt(shmp) == -1) {

perror("shmdt");

return 1;

}

if (shmctl(shmid, IPC\_RMID, 0) == -1) {

perror("shmctl");

return 1;

}

printf("Writing Process: Complete\n");

remove\_semaphore();

return 0;

}

void shared\_memory\_cntr\_increment(int pid, struct shmseg \*shmp, int total\_count) {

int cntr;

int numtimes;

int sleep\_time;

int semid;

struct sembuf sem\_buf;

struct semid\_ds buf;

int tries;

int retval;

semid = semget(SEM\_KEY, 1, IPC\_CREAT | IPC\_EXCL | 0666);

//printf("errno is %d and semid is %d\n", errno, semid);

if (semid >= 0) {

printf("First Process\n");

sem\_buf.sem\_op = 1;

sem\_buf.sem\_flg = 0;

sem\_buf.sem\_num = 0;

retval = semop(semid, &sem\_buf, 1);

if (retval == -1) {

perror("Semaphore Operation: ");

return;

}

} else if (errno == EEXIST) { // Already other process got it

int ready = 0;

printf("Second Process\n");

semid = semget(SEM\_KEY, 1, 0);

if (semid < 0) {

perror("Semaphore GET: ");

return;

}

sem\_buf.sem\_num = 0;

sem\_buf.sem\_op = 0;

sem\_buf.sem\_flg = SEM\_UNDO;

retval = semop(semid, &sem\_buf, 1);

if (retval == -1) {

perror("Semaphore Locked: ");

return;

}

}

sem\_buf.sem\_num = 0;

sem\_buf.sem\_op = -1; /\* Allocating the resources \*/

sem\_buf.sem\_flg = SEM\_UNDO;

retval = semop(semid, &sem\_buf, 1);

if (retval == -1) {

perror("Semaphore Locked: ");

return;

}

cntr = shmp->cntr;

shmp->write\_complete = 0;

if (pid == 0)

printf("SHM\_WRITE: CHILD: Now writing\n");

else if (pid > 0)

printf("SHM\_WRITE: PARENT: Now writing\n");

for (numtimes = 0; numtimes < total\_count; numtimes++) {

cntr += 1;

shmp->cntr = cntr;

/\* Sleeping for a second for every thousand \*/

sleep\_time = cntr % 1000;

if (sleep\_time == 0)

sleep(1);

}

shmp->write\_complete = 1;

sem\_buf.sem\_op = 1; /\* Releasing the resource \*/

retval = semop(semid, &sem\_buf, 1);

if (retval == -1) {

perror("Semaphore Locked\n");

return;

}

if (pid == 0)

printf("SHM\_WRITE: CHILD: Writing Done\n");

else if (pid > 0)

printf("SHM\_WRITE: PARENT: Writing Done\n");

return;

}

void remove\_semaphore() {

int semid;

int retval;

semid = semget(SEM\_KEY, 1, 0);

if (semid < 0) {

perror("Remove Semaphore: Semaphore GET: ");

return;

}

retval = semctl(semid, 0, IPC\_RMID);

if (retval == -1) {

perror("Remove Semaphore: Semaphore CTL: ");

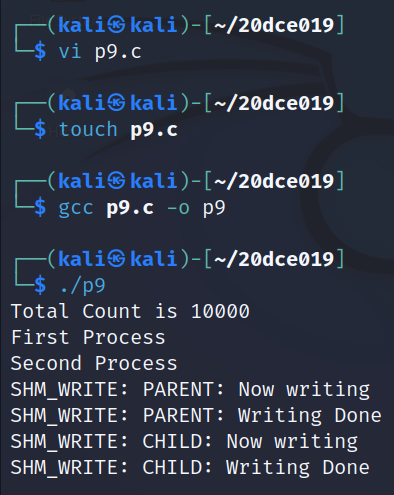
return;

}

return;

}

**Output:**

****

**Practical-10**

**Aim:** Simulate Following Page Replacement Algorithms.

1. First In First Out Algorithm
2. Least Recently Used Algorithm
3. Optimal Algorithm

**Code for FIFO Algorithm:**

#include <stdio.h>

int main()

{

int referenceString[10], pageFaults = 0, m, n, s, pages, frames;

printf("\nEnter the number of Pages:\t");

scanf("%d", &pages);

printf("\nEnter reference string values:\n");

for( m = 0; m < pages; m++)

{

printf("Value No. [%d]:\t", m + 1);

scanf("%d", &referenceString[m]);

}

printf("\n What are the total number of frames:\t");

{

scanf("%d", &frames);

}

int temp[frames];

for(m = 0; m < frames; m++)

{

temp[m] = -1;

}

for(m = 0; m < pages; m++)

{

s = 0;

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

{

if(referenceString[m] == temp[n])

{

s++;

pageFaults--;

}

}

pageFaults++;

if((pageFaults <= frames) && (s == 0))

{

temp[m] = referenceString[m];

}

else if(s == 0)

{

temp[(pageFaults - 1) % frames] = referenceString[m];

}

printf("\n");

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

{

printf("%d\t", temp[n]);

}

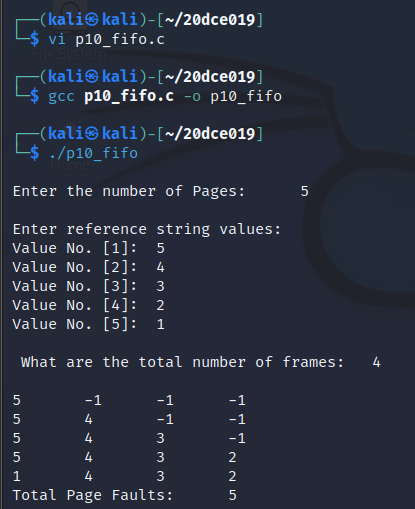
}

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

return 0;

}

**Output:**



**Code for LRU Algorithm:**

#include<stdio.h>

int findLRU(int time[], int n){

int i, minimum = time[0], pos = 0;

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

if(time[i] < minimum){

minimum = time[i];

pos = i;

}

}

return pos;

}

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j, pos, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

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

counter++;

time[j] = counter;

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

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

counter++;

faults++;

frames[j] = pages[i];

time[j] = counter;

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

pos = findLRU(time, no\_of\_frames);

counter++;

faults++;

frames[pos] = pages[i];

time[pos] = counter;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

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

}

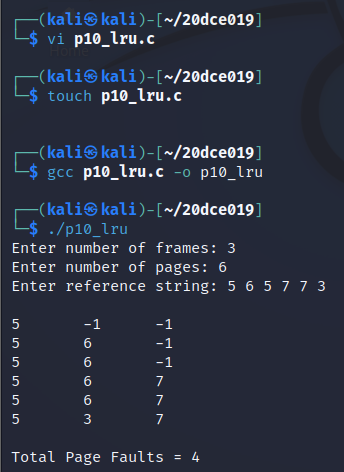
}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}

**Output:**

****

**Code for Optimal Algorithm:**

#include<stdio.h>

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter page reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

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

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

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

faults++;

frames[j] = pages[i];

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

flag3 =0;

for(j = 0; j < no\_of\_frames; ++j){

temp[j] = -1;

for(k = i + 1; k < no\_of\_pages; ++k){

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

temp[j] = k;

break;

}

}

}

for(j = 0; j < no\_of\_frames; ++j){

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

pos = j;

flag3 = 1;

break;

}

}

if(flag3 ==0){

max = temp[0];

pos = 0;

for(j = 1; j < no\_of\_frames; ++j){

if(temp[j] > max){

max = temp[j];

pos = j;

}

}

}

frames[pos] = pages[i];

faults++;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

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

}

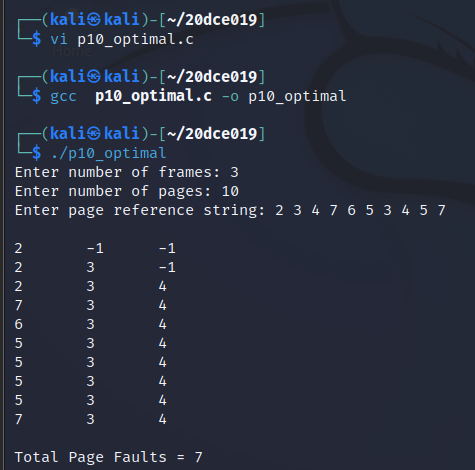
}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}

**Output:**

****

**Practical-11**

**Aim:** **Thread synchronization using counting semaphores and mutual exclusion using mutex.**

**Code:**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

pthread\_t tid[2];

int counter;

pthread\_mutex\_t lock;

void \*trythis(void \*arg)

{

pthread\_mutex\_lock(&lock);

unsigned long i = 0;

counter += 1;

printf("\n Job %d has started\n", counter);

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

;

printf("\n Job %d has finished\n", counter);

pthread\_mutex\_unlock(&lock);

return NULL;

}

int main(void)

{

int i = 0;

int error;

if (pthread\_mutex\_init(&lock, NULL) != 0)

{

printf("\n mutex init has failed\n");

return 1;

}

while (i < 2)

{

error = pthread\_create(&(tid[i]), NULL,

&trythis, NULL);

if (error != 0)

printf("\nThread can't be created :[%s]", strerror(error));

i++;

}

pthread\_join(tid[0], NULL);

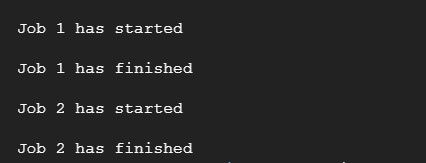
pthread\_join(tid[1], NULL);

pthread\_mutex\_destroy(&lock);

return 0;

}

**Output:**



**Practical-12**

**Aim:** **Write a C program in LINUX to implement Bankers algorithm for Deadlock Avoidance.**

**Code:**

#include <stdio.h>

int main()

{

int n, m, i, j, k;

n = 5; // Number of processes

m = 3; // Number of resources

int alloc[5][3] = { { 0, 1, 0 },

{ 2, 0, 0 },

{ 3, 0, 2 },

{ 2, 1, 1 },

{ 0, 0, 2 } };

int max[5][3] = { { 7, 5, 3 },

{ 3, 2, 2 },

{ 9, 0, 2 },

{ 2, 2, 2 },

{ 4, 3, 3 } };

int avail[3] = { 3, 3, 2 }; // Available Resources

int f[n], ans[n], ind = 0;

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

f[k] = 0;

}

int need[n][m];

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

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

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

}

int y = 0;

for (k = 0; k < 5; k++) {

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

if (f[i] == 0) {

int flag = 0;

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

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

flag = 1;

break;

}

if (flag == 0) {

ans[ind++] = i;

for (y = 0; y < m; y++)

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

f[i] = 1;

}

}

}

}

int flag = 1;

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

{

if(f[i]==0)

{

flag=0;

printf("The following system is not safe");

break;

}

if(flag==1)

{

printf("Following is the SAFE Sequence\n");

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

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

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

}

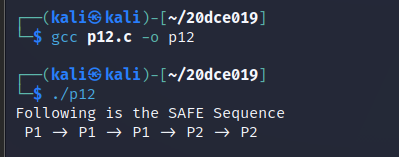
return (0);

}

}

}

**Output:**

****

**Practical-13**

**Aim:** **Write a C program in LINUX to perform Memory allocation algorithms and Calculate Internal and External Fragmentation. (First Fit, Best Fit, Worst Fit).**

**Code for First Fit:**

#include <stdio.h>

void implimentFirstFit(int blockSize[], int blocks, int processSize[], int processes)

{

int allocate[processes];

int occupied[blocks];

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

{

allocate[i] = -1;

}

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

occupied[i] = 0;

}

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

{

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

{

if (!occupied[j] && blockSize[j] >= processSize[i])

{

allocate[i] = j;

occupied[j] = 1;

break;

}

}

}

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

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

{

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

if (allocate[i] != -1)

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

else

printf("Not Allocated\n");

}

int total\_mem = 0;

int allocated\_mem = 0;

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

total\_mem += blockSize[i];

}

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

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

allocated\_mem += processSize[i];

}

}

printf("\n Total mem = %d and allocated memory = %d hence internal fragmentation = %d \n", total\_mem, allocated\_mem, (total\_mem-allocated\_mem));

}

void main()

{

int blockSize[] = {30, 5, 10};

int processSize[] = {10, 6, 9};

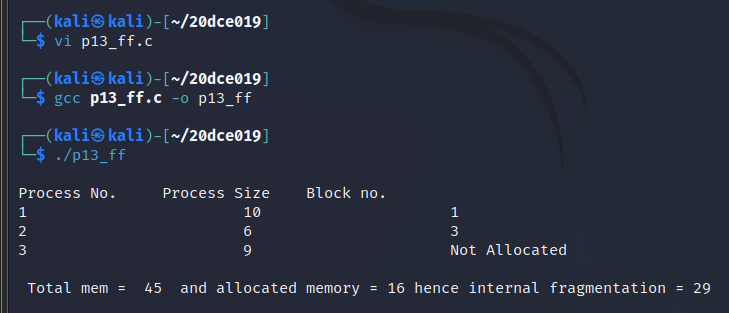
int m = sizeof(blockSize)/sizeof(blockSize[0]);

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

implimentFirstFit(blockSize, m, processSize, n);

}

**Output:**

****

**Code for Best Fit:**

#include <stdio.h>

void implimentBestFit(int blockSize[], int blocks, int processSize[], int proccesses)

{

int allocation[proccesses];

int occupied[blocks];

int total\_mem = 0;

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

total\_mem += blockSize[i];

}

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

allocation[i] = -1;

}

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

occupied[i] = 0;

}

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

{

int indexPlaced = -1;

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

{

if (blockSize[j] >= processSize[i] && !occupied[j])

{

if (indexPlaced == -1)

indexPlaced = j;

else if (blockSize[indexPlaced] < blockSize[j])

indexPlaced = j;

}

}

if (indexPlaced != -1)

{

allocation[i] = indexPlaced;

occupied[indexPlaced] = 1;

blockSize[indexPlaced] -= processSize[i];

}

}

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

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

{

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

if (allocation[i] != -1)

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

else

printf("Not Allocated\n");

}

int allocated\_mem = 0;

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

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

allocated\_mem += processSize[i];

}

}

printf("\n Total mem = %d and allocated memory = %d hence internal fragmentation = %d \n", total\_mem, allocated\_mem, (total\_mem-allocated\_mem));

}

int main()

{

int blockSize[] = {5, 4, 3, 6, 7};

int processSize[] = {1, 3, 5, 3};

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

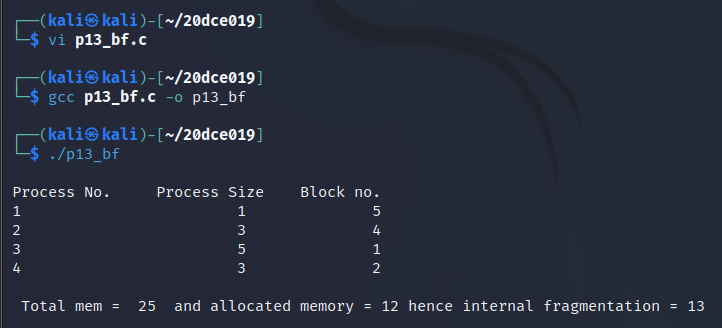
int proccesses = sizeof(processSize)/sizeof(processSize[0]);

implimentBestFit(blockSize, blocks, processSize, proccesses);

return 0 ;

}

**Output:**



**Code for Worst Fit:**

#include<stdio.h>

#define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp;

static int bf[max],ff[max];

printf("\n\tMemory Management Scheme - First Fit");

printf("\nEnter the number of blocks:");

scanf("%d",&nb);

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

scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n");

for(i=1;i<=nb;i++)

{

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

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

}

printf("Enter the size of the files :-\n");

for(i=1;i<=nf;i++)

{

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

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

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i];

if(temp>=0)

{

ff[i]=j;

break;

}

}

}

frag[i]=temp;

bf[ff[i]]=1;

}

printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragement");

for(i=1;i<=nf;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

int fragmentation = 0;

for (i=1; i<=nf ; i++)

fragmentation += frag[i];

printf("\n Total fragmentation that occured is %d \n", fragmentation);

}

**Output:**

