**Ex. no: 2a) Date:**

**Shell Script**

**Aim:**

To write a Shellscript to to display basic calculator.

**Program:**

#!/bin/bash

# Read two numbers and the operator from the user

read -p "Enter first number: " num1

read -p "Enter an operator (+, -, \*, /): " op

read -p "Enter second number: " num2

# Perform calculation using Arithmetic Expansion

case $op in

+) result=$((num1 + num2));;

-) result=$((num1 - num2));;

\\*) result=$((num1 \* num2));;

/) if [ $num2 -ne 0 ]; then

result=$((num1 / num2)) # Integer division

else

echo "Error: Division by zero is not allowed."

exit 1

fi

;;

\*) echo "Invalid operator! Use +, -, \*, or /"

exit 1

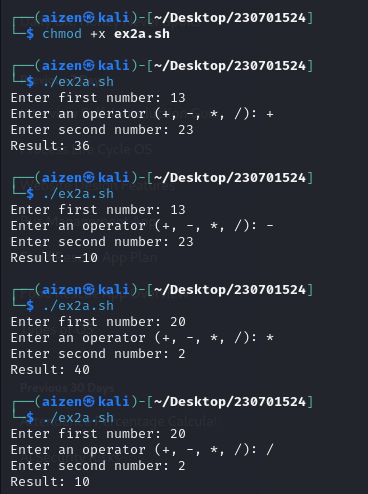
;;

esac

# Display the result

echo "Result: $result"

**output**



**Ex. no: 2b)**

**Aim:**

To write a Shellscript to test given year is leap or not using conditional statement

**Program:**

echo "Enter the year"

read y

b=`expr $y % 4`

if [ $b -eq 0 ]

then

echo "$y is a leap year"

else

echo "$y is not a leap year"

fi

**Output:**

**3a) Shell Script – Reverse of Digit**

**Aim:**

To write a Shell script to reverse a given digit using looping statement.

**Program:**

#!/bin/bash

read -p "Enter a number: " num

rev=0

while [ $num -gt 0 ]; do

digit=$((num % 10)) # Get last digit

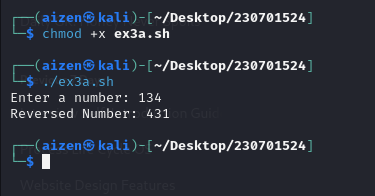
rev=$((rev \* 10 + digit)) # Append digit to reversed number

num=$((num / 10)) # Remove last digit

done

echo "Reversed Number: $rev"

**output:**



**3b) Shell Script – Fibbonacci Series**

**Aim:**

To write a Shell script to generate a Fibonacci series using for loop.

**Program:**

#!/bin/bash

# Read the number of terms

read -p "Enter the number of terms: " n

# First two terms

a=0

b=1

echo "Fibonacci Sequence:"

# Print first two terms

echo -n "$a $b "

# Loop to generate the remaining terms

for ((i=2; i<n; i++)); do

c=$((a + b)) # Next term

echo -n "$c " # Print next term

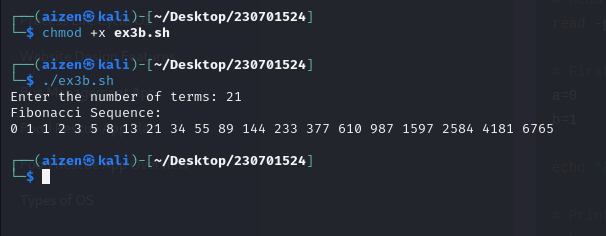
a=$b # Update previous terms

b=$c

done

echo “ “

**output**



**Ex4a)**

**Aim :**

To find out the average pay of all employees whose salary is more than 6000 and no. of days worked is more than 4.

**Program:**

#!/usr/bin/awk -f

BEGIN {

print "Employee Name\tSalary\tDays Worked\tTotal Pay"

print "--------------------------------------------------"

totalPay = 0

count = 0

}

# Process each line

{

total = $2 \* $3

totalPay += total

count++

print $1, "\t\t", $2, "\t", $3, "\t\t", total

}

# END block to print total and average pay

END {

averagePay = (count > 0) ? totalPay / count : 0

print "--------------------------------------------------"

print "Total Pay:\t\t\t\t", totalPay

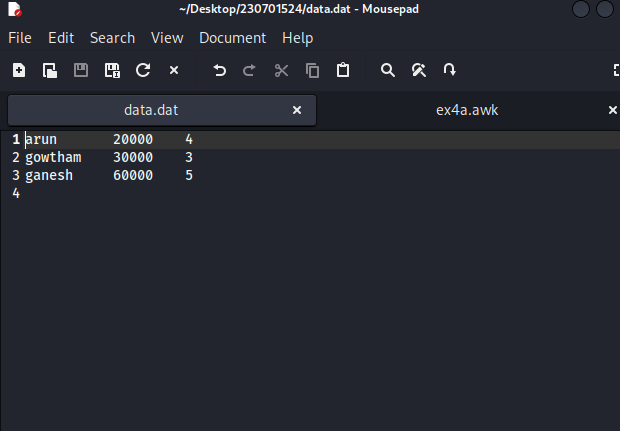
print "Average Pay:\t\t\t", averagePay

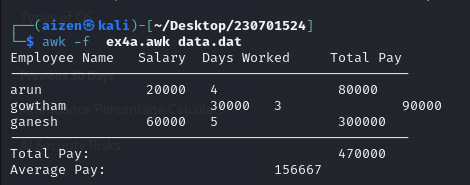
}

l

**Output:**

**data.dat**





**5) System calls**

**Aim:**

To experiment system calls using fork(), execlp() and pid() functions.

**Program:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

int pid;

pid = fork();

if (pid < 0) {

perror("Fork failed");

return 1;

}if (pid == 0) {

printf("THIS LINE EXECUTED TWICE\n");

printf("Child Process ID: %d\n", getpid());

printf("Parent Process ID: %d\n", getppid());

execlp("/bin/ls", "ls", "-l", (char \*)NULL);

perror("execlp failed");

return 1;

} else {

printf("THIS LINE EXECUTED TWICE\n");

printf("Parent Process ID: %d\n", getpid());

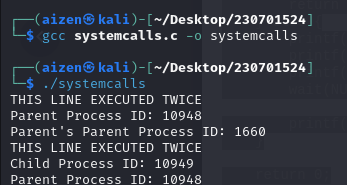
printf("Parent's Parent Process ID: %d\n", getppid());

wait(NULL);

printf("IT CAN BE EXECUTED TWICE\n");

}}

**Output**

****

**6a) first come first serve**

**Program:**

#include <stdio.h>

int main()

{

int pid[15];

int bt[15];

int n;

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

scanf("%d",&n);

printf("Enter process id of all the processes: ");

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

{

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

}

printf("Enter burst time of all the processes: ");

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

{

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

}

int i, wt[n];

wt[0]=0;

//for calculating waiting time of each process

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

{

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

}

printf("Process ID Burst Time Waiting Time TurnAround Time\n");

float twt=0.0;

float tat= 0.0;

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

{

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

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

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

//calculating and printing turnaround time of each process

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

printf("\n");

//for calculating total waiting time

twt += wt[i];

//for calculating total turnaround time

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

}

float att,awt;

//for calculating average waiting time

awt = twt/n;

//for calculating average turnaround time

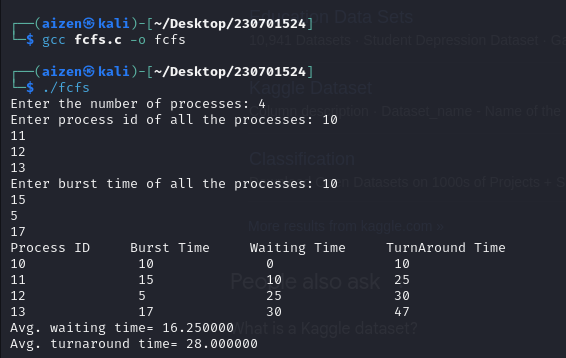
att = tat/n;

printf("Avg. waiting time= %f\n",awt);

printf("Avg. turnaround time= %f",att);

}

**output**

****

**6b)shortest job first**

**Program:**

#include<stdio.h>

int main() {

int time, burst\_time[10], at[10], sum\_burst\_time = 0, smallest, n, i;

int sumt = 0, sumw = 0;

printf("enter the no of processes : ");

scanf("%d", & n);

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

printf("the arrival time for process P%d : ", i + 1);

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

printf("the burst time for process P%d : ", i + 1);

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

sum\_burst\_time += burst\_time[i];

}

burst\_time[9] = 9999;

for (time = 0; time < sum\_burst\_time;) {

smallest = 9;

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

if (at[i] <= time && burst\_time[i] > 0 && burst\_time[i] < burst\_time[smallest])

smallest = i;

}

printf("P[%d]\t|\t%d\t|\t%d\n", smallest + 1, time + burst\_time[smallest] - at[smallest], time - at[smallest]);

sumt += time + burst\_time[smallest] - at[smallest];

sumw += time - at[smallest];

time += burst\_time[smallest];

burst\_time[smallest] = 0;

}

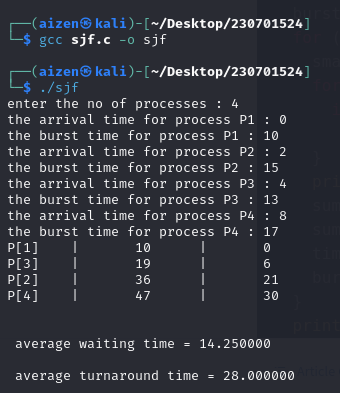
printf("\n\n average waiting time = %f", sumw \* 1.0 / n);

printf("\n\n average turnaround time = %f", sumt \* 1.0 / n);

return 0;

}

**Output**

****

**6c)priority scheduling**

**Program:**

#include <stdio.h>

//Function to swap two variables

void swap(int \*a,int \*b)

{

int temp=\*a;

\*a=\*b;

\*b=temp;

}

int main()

{

int n;

printf("Enter Number of Processes: ");

scanf("%d",&n);

// b is array for burst time, p for priority and index for process id

int b[n],p[n],index[n];

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

{

printf("Enter Burst Time and Priority Value for Process %d: ",i+1);

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

index[i]=i+1;

}

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

{

int a=p[i],m=i;

//Finding out highest priority element and placing it at its desired position

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

{

if(p[j] > a)

{

a=p[j];

m=j;

}

}

//Swapping processes

swap(&p[i], &p[m]);

swap(&b[i], &b[m]);

swap(&index[i],&index[m]);

}

// T stores the starting time of process

int t=0;

//Printing scheduled process

printf("Order of process Execution is\n");

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

{

printf("P%d is executed from %d to %d\n",index[i],t,t+b[i]);

t+=b[i];

}

printf("\n");

printf("Process Id Burst Time Wait Time TurnAround Time\n");

int wait\_time=0;

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

{

printf("P%d %d %d %d\n",index[i],b[i],wait\_time,wait\_time + b[i]);

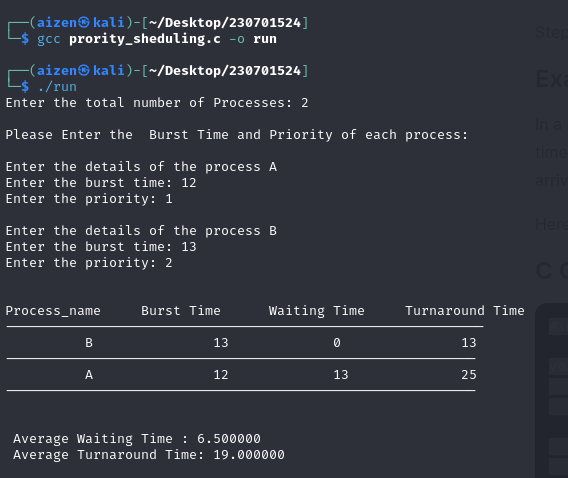
wait\_time += b[i];

}

return 0;

}

**Output**

****

**6d)round robin**

**Program:**

#include <stdio.h>

void main() {

int i, processes, sum = 0, cnt = 0, y, q, wt = 0, tat = 0, at[10], bt[10], temp[10];

float avg\_waitt, avg\_turnat;

// Input the number of processes

printf("Total number of processes in the system: ");

scanf("%d", &processes);

y = processes; // Assign number of processes to y

// Input arrival time and burst time for each process

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

printf("\nEnter the Arrival and Burst time of Process[%d]\n", i + 1);

printf("Arrival time: ");

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

printf("Burst time: ");

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

temp[i] = bt[i]; // Initialize remaining burst time

}

// Input the time quantum

printf("Enter the Time Quantum: ");

scanf("%d", &q);

// Display header for the process info

printf("\nProcess No \tBurst Time \tTAT \t\tWaiting Time\n");

// Scheduling loop

for(sum = 0, i = 0; y != 0;) {

if(temp[i] <= q && temp[i] > 0) {

sum = sum + temp[i];

temp[i] = 0;

cnt = 1;

} else if(temp[i] > 0) {

temp[i] = temp[i] - q;

sum = sum + q;

}

if(temp[i] == 0 && cnt == 1) {

y--; // Decrement remaining processes

printf("\nProcess No[%d] \t%d \t\t%d \t\t%d", i + 1, bt[i], sum - at[i], sum - at[i] - bt[i]);

wt = wt + sum - at[i] - bt[i]; // Calculate waiting time

tat = tat + sum - at[i]; // Calculate turnaround time

cnt = 0;

}

if(i == processes - 1) {

i = 0;

} else if(at[i + 1] <= sum) {

i++;

} else {

i = 0;

}

}

// Calculate average waiting time and turnaround time

avg\_waitt = wt \* 1.0 / processes;

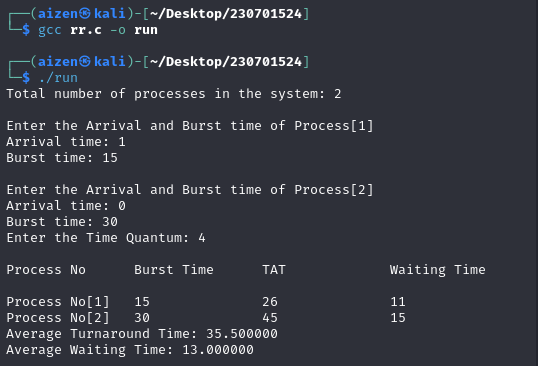
avg\_turnat = tat \* 1.0 / processes;

printf("\nAverage Turnaround Time: %f", avg\_turnat);

printf("\nAverage Waiting Time: %f", avg\_waitt);

}

**Output**

****

**7)Ipc using shared memory**

**Program:**

**Sender.c:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1024 // size of shared memory

int main() {

key\_t key = ftok("shmfile", 65); // generate unique key

int shmid = shmget(key, SHM\_SIZE, 0666|IPC\_CREAT); // create shared memory

if (shmid == -1) {

perror("shmget");

exit(1);

}

char \*str = (char\*) shmat(shmid, (void\*)0, 0); // attach to shared memory

if (str == (char\*)(-1)) {

perror("shmat");

exit(1);

}

printf("Enter a message: ");

fgets(str, SHM\_SIZE, stdin); // write user input to shared memory

printf("Message sent to shared memory.\n");

shmdt(str); // detach from shared memory

return 0;

}

**Reciver.c**

// receiver.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1024

int main() {

key\_t key = ftok("shmfile", 65); // same key as sender

int shmid = shmget(key, SHM\_SIZE, 0666|IPC\_CREAT); // access shared memory

if (shmid == -1) {

perror("shmget");

exit(1);

}

char \*str = (char\*) shmat(shmid, (void\*)0, 0); // attach to shared memory

if (str == (char\*)(-1)) {

perror("shmat");

exit(1);

}

printf("Received message: %s", str);

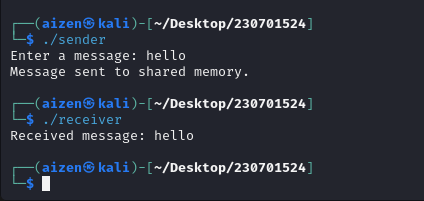
shmdt(str); // detach from shared memory

shmctl(shmid, IPC\_RMID, NULL); // destroy the shared memory

return 0;

}

**Output**

****

**8)producer consumer using semaphore**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#define SIZE 3

int buffer[SIZE];

int in = 0, out = 0, count = 0, item = 0;

sem\_t empty, full, mutex;

void produce() {

if (sem\_trywait(&empty) == 0) {

sem\_wait(&mutex);

item++;

buffer[in] = item;

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

in = (in + 1) % SIZE;

count++;

sem\_post(&mutex);

sem\_post(&full);

} else {

printf("Buffer is full!!\n");

}

}

void consume() {

if (sem\_trywait(&full) == 0) {

sem\_wait(&mutex);

int consumed = buffer[out];

printf("Consumer consumes item %d\n", consumed);

out = (out + 1) % SIZE;

count--;

sem\_post(&mutex);

sem\_post(&empty);

} else {

printf("Buffer is empty!!\n");

}

}

int main() {

int choice;

sem\_init(&empty, 0, SIZE);

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

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

// Show menu only once

printf("1. Producer\n");

printf("2. Consumer\n");

printf("3. Exit\n");

while (1) {

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

produce();

break;

case 2:

consume();

break;

case 3:

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

exit(0);

default:

printf("Invalid choice!\n");

}

}

// Cleanup (technically unreachable)

sem\_destroy(&empty);

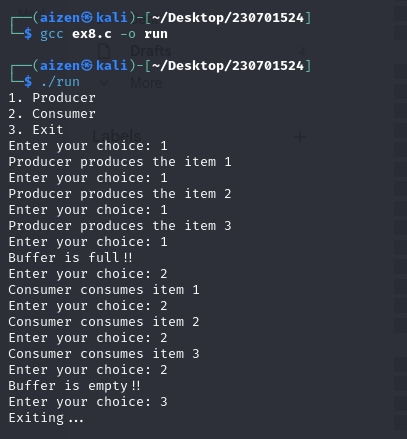
sem\_destroy(&full);

sem\_destroy(&mutex);

return 0;

}

**Output**

****

**9)deadloack avoidance**

Program:

#include <stdio.h>

#include <stdbool.h>

#define P 5 // Number of processes

#define R 3 // Number of resources

int main() {

int allocation[P][R] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int max[P][R] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

int available[R] = {3, 3, 2};

int need[P][R];

bool finish[P] = {false};

int safeSequence[P];

int work[R];

// Calculate need matrix

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

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

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

// Step 1: Initialize work = available

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

work[i] = available[i];

int count = 0;

while (count < P) {

bool found = false;

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

if (!finish[i]) {

int j;

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

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

break;

}

if (j == R) {

// This process can be safely executed

for (int k = 0; k < R; k++)

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

safeSequence[count++] = i;

finish[i] = true;

found = true;

}

}

}

if (!found) {

printf("System is not in a safe state (No safe sequence exists).\n");

return 1;

}

}

// If here, then all processes can finish

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

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

printf("P%d", safeSequence[i]);

if (i != P - 1)

printf(" -> ");

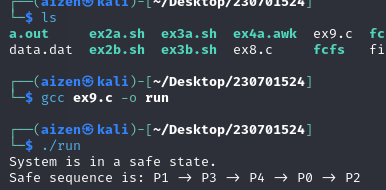
}

printf("\n");

return 0;

}

**Output**

****

**Ex10a)best fit**

**Program:**

#include <stdio.h>

#define MAX 25

int main() {

int blockSize[MAX], processSize[MAX];

int blockCount, processCount;

int allocation[MAX];

// Input number of blocks and processes

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

scanf("%d", &blockCount);

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

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

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

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

}

printf("\nEnter number of processes: ");

scanf("%d", &processCount);

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

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

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

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

allocation[i] = -1; // Initially all processes are unallocated

}

// Best Fit Allocation

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

int bestIdx = -1;

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

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

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

bestIdx = j;

}

}

}

// If a suitable block was found

if (bestIdx != -1) {

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

// Display Allocation Results

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

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

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

if (allocation[i] != -1)

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

else

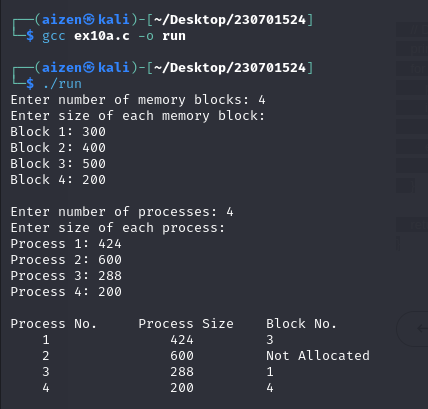
printf("Not Allocated\n");

}

return 0;

}

**Output**

****

**Ex10b)first fit**

**Program**

#include <stdio.h>

#define MAX 25 // Maximum number of blocks or files

void main() {

int frag[MAX], b[MAX], f[MAX];

int i, j, nb, nf, temp, highest = 0;

int bf[MAX], ff[MAX]; // bf for block flag, ff for file flag

// Get the number of blocks and files

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

scanf("%d", &nb);

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

scanf("%d", &nf);

// Get the size of the blocks

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

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

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

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

}

// Get the size of the files

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

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

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

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

}

// Initialize all blocks and files as not allocated

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

bf[i] = 0; // block is not allocated

}

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

ff[i] = 0; // file is not allocated

}

// First Fit algorithm

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

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

// If block is not allocated and file fits in the block

if (bf[j] == 0 && b[j] >= f[i]) {

// Allocate the block to the file

bf[j] = 1;

ff[i] = 1;

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

break;

}

}

}

// Displaying the allocation results

printf("\nBlock Allocation using First Fit:\n");

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

if (ff[i] == 1) {

printf("File %d allocated to Block %d with %d units of fragment\n", i + 1, i + 1, frag[i]);

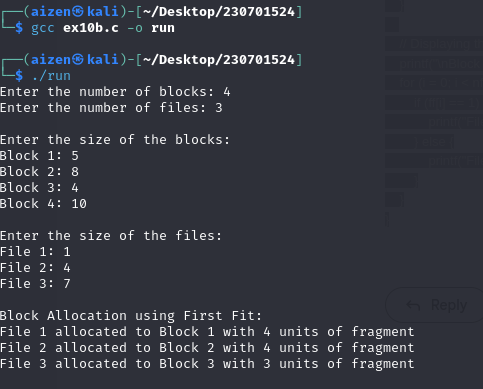
} else {

printf("File %d cannot be allocated\n", i + 1);

}

}

}



**11a)fifo page replacement**

**Program**

#include <stdio.h>

int main() {

int pages[50], frames[10], n, f, i, j, k, pageFaults = 0, index = 0, flag;

// Step 1: Get the number of pages

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

scanf("%d", &n);

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

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

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

}

// Step 2: Get the number of frames (memory size)

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

scanf("%d", &f);

// Step 3: Initialize all frames to -1 (empty)

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

frames[i] = -1;

}

// Step 4-6: FIFO logic with page fault tracking

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

flag = 0;

// Check if page already exists in frame

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

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

flag = 1; // Page hit, no page fault

break;

} }

// If page is not found (page fault)

if (flag == 0) {

frames[index] = pages[i]; // Replace the oldest page

index = (index + 1) % f; // Move to next frame position (circular)

pageFaults++;

}

// Display current state of frames

printf("Frames: ");

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

if (frames[k] != -1)

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

else

printf("- ");

}

printf("\n");

}

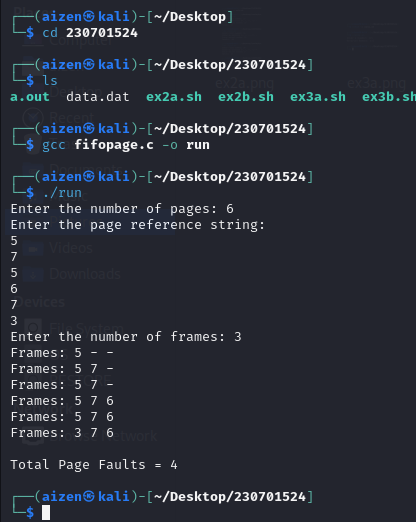
// Step 8: Display number of page faults

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

return 0;

}

**Output**

****

**11b)LRU**

**Program**

#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 frames[10], pages[50], time[10];

int i, j, pos, max, faults = 0, counter = 0, n, flag1, flag2;

// Step 3: Get number of pages

printf("Enter number of pages: ");

scanf("%d", &n);

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

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

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

}

// Step 2: Get frame size

printf("Enter number of frames: ");

scanf("%d", &max);

// Initialize frames and time arrays

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

frames[i] = -1;

time[i] = 0;

}

// LRU Page Replacement Logic

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

flag1 = flag2 = 0;

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

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

counter++;

time[j] = counter;

flag1 = flag2 = 1;

break;

}

}

// Check if there is an empty frame

if (flag1 == 0) {

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

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

counter++;

faults++;

frames[j] = pages[i];

time[j] = counter;

flag2 = 1;

break;

}

}

}

// Replace least recently used page

if (flag2 == 0) {

pos = findLRU(time, max);

counter++;

faults++;

frames[pos] = pages[i];

time[pos] = counter;

}

// Display current frame status

printf("Frames: ");

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

if (frames[j] != -1)

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

else

printf("- ");

}

printf("\n");

}

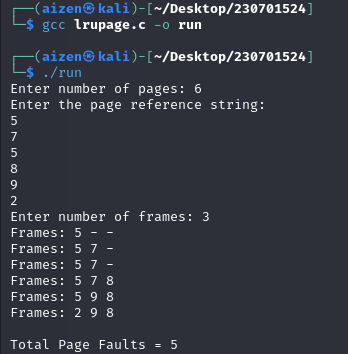
// Step 8: Display total page faults

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

return 0;

}

**Output**

****

**11c) optaimal**

**Program**

#include <stdio.h>

int predict(int pages[], int frames[], int n, int index, int f) {

int res = -1, farthest = index;

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

int j;

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

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

if (j > farthest) {

farthest = j;

res = i;

}

break;

}

}

// If the page is never used again

if (j == n)

return i;

}

return (res == -1) ? 0 : res;

}

int main() {

int pages[50], frames[10], n, f;

int pageFaults = 0, i, j, flag, frameIndex = 0;

// Step 3: Get number of pages

printf("Enter number of pages: ");

scanf("%d", &n);

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

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

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

}

// Step 2: Get frame size

printf("Enter number of frames: ");

scanf("%d", &f);

// Initialize all frames to -1 (empty)

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

frames[i] = -1;

}

// Step 6: Optimal replacement logic

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

flag = 0;

// Check if page is already in frame

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

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

flag = 1; // Page hit

break;

}

}

// If page not found

if (!flag) {

// If there's an empty frame

if (frameIndex < f) {

frames[frameIndex++] = pages[i];

} else {

int pos = predict(pages, frames, n, i + 1, f);

frames[pos] = pages[i];

}

pageFaults++;

}

// Step 8: Display current frames

printf("Frames: ");

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

if (frames[j] != -1)

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

else

printf("- ");

}

printf("\n");

}

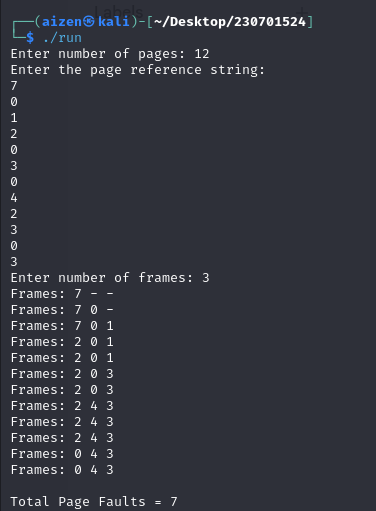
// Step 8: Display total page faults

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

return 0;

}

**Output**

****

**12)fille organization Techique**