**ASSIGNMENT 5**

**Aim:**

Thread management using pthread library. Implement matrix multiplication

using multithreading. Application should have pthread\_create, pthread\_join,

pthread\_exit. In the program, every thread must return the value and must be

collected in pthread\_join in the main function. Final sum of row column

multiplication must be done by main thread (main function).

**Theory:**

Thread is a single sequence stream within a process. Threads have same

properties as of the

process so they are called as light weight processes. Threads are executed one

after another

but gives the illusion as if they are executing in parallel. Each thread has

different states. Each

thread has

1. Program Counter

2. Register Set

3. Stack Space

Threads are not independent of each other as they share the code, data, OS

resources etc.

**Types of Threads:**

**1. User Level thread (ULT)**

Is implemented in the user level library, they are not created using the system

calls. Thread switching does not need to call OS and to cause interrupt to

Kernel. Kernel doesn’t know about the user level thread and manages them as

if they were single-threaded processes.

**2. Kernel Level Thread (KLT)**

Kernel knows and manages the threads. Instead of thread table in each

process, the kernel itself has thread table (a master one) that keeps track of all

the threads in the system. In addition kernel also maintains the traditional

process table to keep track.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

#define SIZE 10

struct matrix

{

int r, c;

};

int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];

void\* runner(void\* param);

int add[SIZE];

int main()

{

double time\_spent = 0.0;

clock\_t begin = clock();

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

{

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

{

a[i][j] = rand()%20;

b[i][j] = rand()%20;

}

}

printf("Matrix A :\n");

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

{

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

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

printf("\n");

}

printf("\n");

printf("Matrix B :\n");

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

{

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

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

printf("\n");

}

printf("\n");

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

{

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

{

pthread\_t thread;

struct matrix\* newC = (struct matrix\*)malloc(sizeof(struct

matrix));

newC->r = i;

newC->c = j;

pthread\_create(&thread, NULL, runner, (void\*)newC);

pthread\_join(thread, NULL);

// adding the multiplications made by thread stored in array

add

for(int k = 0; k < 4; k++)

{

c[i][j] += add[k];

}

free(newC);

}

}

printf("Matrix C :\n");

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

{

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

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

printf("\n");

}

clock\_t end = clock();

time\_spent += (double) (end - begin) / CLOCKS\_PER\_SEC;

printf("\nTime difference is %f seconds\n", time\_spent);

return 0;

}

void\* runner(void\* param)

{

struct matrix\* temp = (struct matrix\*)param;

for(int c = 0; c < SIZE; c++)

{

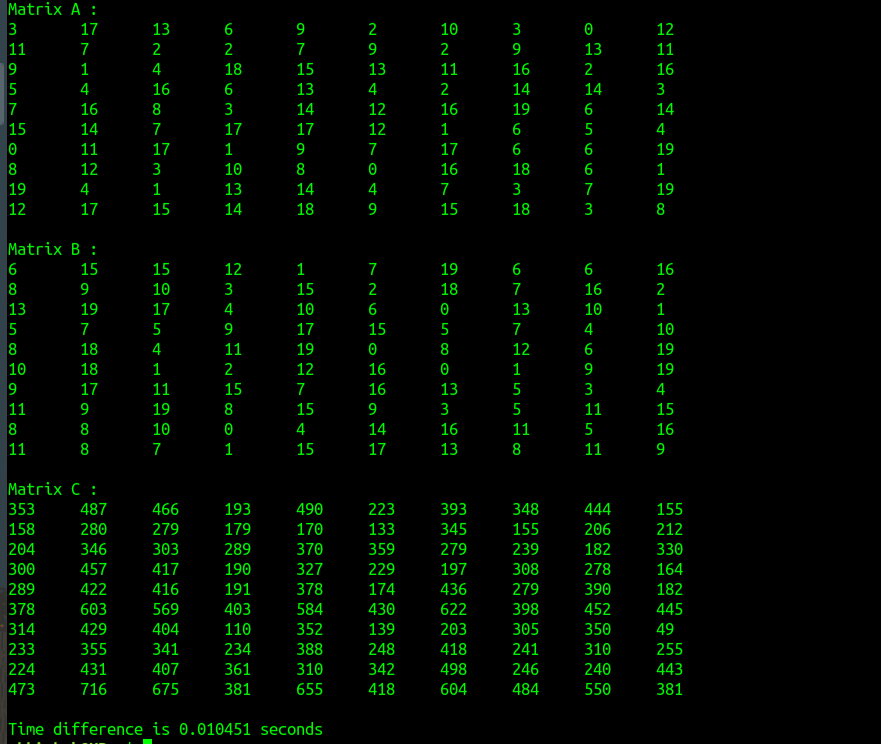
add[c] = a[temp->r][c] \* b[c][temp->c];

}

pthread\_exit(0);

}

**Output:**



**Conclusion:**

We are able to perform basic operations of POSIX.