Student Name :- Gaurav Dhar Dubey

Registration Number :- 11803296

Roll Number :- RK18TGA16

Email:-gauravdhardubey88@gmail.com

Github Link :-

Problem NO.3 :

Code:-

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#define MIN\_INPUT 2

#define MAX\_INPUT 1000

void \*getUserInput(){

int \*number\_from\_user=(int \*)malloc(sizeof(int));

puts("Enter a number to print all prime numbers in range [0 - number]");

while(1){

printf("Input : ");

scanf("%d",number\_from\_user);

if(\*number\_from\_user<MIN\_INPUT || \*number\_from\_user>MAX\_INPUT){

printf("Please Enter a positive number from %d to %d\n",MIN\_INPUT,MAX\_INPUT);

rewind(stdin);

}else break;

}

puts("Input thread received user input.");

pthread\_exit(number\_from\_user);

}

int checkPrime(int number){

if(number==1) return 0;

int i;

for(i=2; i<=(int)(number/2); i++)

if(!(number%i)) return 0;

return 1;

}

void \*printPrimeUntil(void \*argument){

int number = \*((int \*)argument),i,count=0;

printf("Thread two received : %d\n",number);

for(i=2; i<=number; i++){

if(checkPrime(i)){

printf("%5d",i);

if(!(++count%5))

printf("\n");

}

}

pthread\_exit(NULL);

}

int main(){

pthread\_t input\_thread, output\_thread;

void \*ret\_number;

if(pthread\_create(&input\_thread,NULL,getUserInput,NULL))

return -1\*printf("Error in thread creation!!!\n");

pthread\_join(input\_thread,&ret\_number);

int \*number = (int \*)ret\_number;

if(pthread\_create(&output\_thread,NULL,&printPrimeUntil,(void \*)number))

return -1\*printf("Error in thread creation!!!\n");

return pthread\_join(output\_thread,NULL);

}

Description:-

Threads :-

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.

Advantages of Thread

* Threads minimize the context switching time.
* Use of threads provides concurrency within a process.
* Efficient communication.
* It is more economical to create and context switch threads.
* Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways −

* **User Level Threads** − User managed threads.
* **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

### **Advantages**

* Thread switching does not require Kernel mode privileges.
* User level thread can run on any operating system.
* Scheduling can be application specific in the user level thread.
* User level threads are fast to create and manage.

### **Disadvantages**

* In a typical operating system, most system calls are blocking.
* Multithreaded application cannot take advantage of multiprocessing.

## Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

### **Advantages**

* Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
* If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
* Kernel routines themselves can be multithreaded.

### **Disadvantages**

* Kernel threads are generally slower to create and manage than the user threads.
* Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

## Multithreading Models

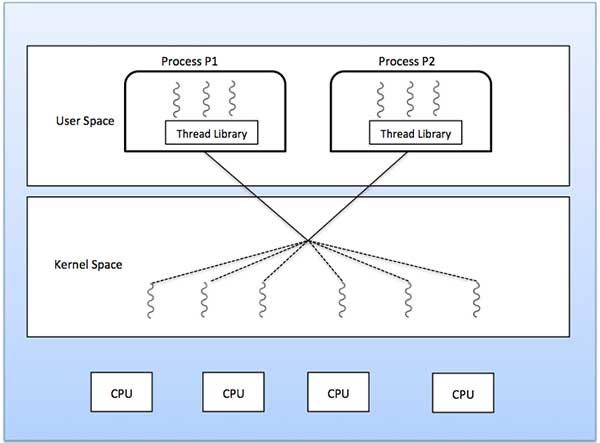
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

* Many to many relationship.
* Many to one relationship.
* One to one relationship.

## Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

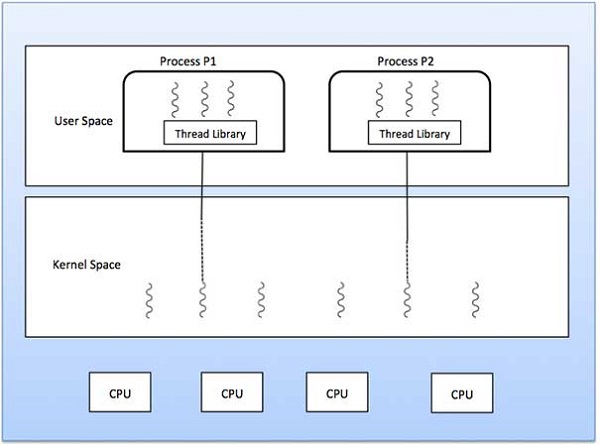
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



## Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



## One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

Test cases:-

Sample1:-

Enter a number to print all prime numbers in range [0 - number]

Input : 45

Input thread received user input.

Thread two received : 45

2 3 5 7 11

13 17 19 23 29

31 37 41 43

Sample 2:-

Enter a number to print all prime numbers in range [0 - number]

Input : 456

Input thread received user input.

Thread two received : 456

2 3 5 7 11

13 17 19 23 29

31 37 41 43 47

53 59 61 67 71

73 79 83 89 97

101 103 107 109 113

127 131 137 139 149

151 157 163 167 173

179 181 191 193 197

199 211 223 227 229

233 239 241 251 257

263 269 271 277 281

283 293 307 311 313

317 331 337 347 349

353 359 367 373 379

383 389 397 401 409

419 421 431 433 439

443 449

Problem no 24 :-

Code:-

#include <stdio.h>

int main()

{

int n, m, i, j, k;

n = 5;

m = 3;

int alloc[5][4] = { { 0,0,1,2 },

{ 1,0,0,0},

{ 1,3,5,4 },

{ 0,6,3,2 },

{ 0,0,1,4 } };

int max[5][4] = { { 0,0,1,2 },

{ 1,7,5,0 },

{ 2,3,5,6 },

{ 0,6,5,2 },

{ 0,6,5,6 } };

int avail[4] = { 1,5,2,0 };

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;

}

}

}

}

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);

}

Desciption:-

# **Banker’s Algorithm in Operating System**

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

**Why Banker’s algorithm is named so?**  
Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Following **Data structures** are used to implement the Banker’s Algorithm:

Let **‘n’**be the number of processes in the system and **‘m’**be the number of resources types.

**Available :**

* It is a 1-d array of size **‘m’** indicating the number of available resources of each type.
* Available[ j ] = k means there are **‘k’** instances of resource type **Rj**

**Max :**

* It is a 2-d array of size ‘**n\*m’**that defines the maximum demand of each process in a system.
* Max[ i, j ] = k means process **Pi** may request at most **‘k’** instances of resource type **Rj.**

**Allocation :**

* It is a 2-d array of size**‘n\*m’**that defines the number of resources of each type currently allocated to each process.
* Allocation[ i, j ] = k means process **Pi** is currently allocated **‘k’** instances of resource type **Rj**

**Need :**

* It is a 2-d array of size **‘n\*m’** that indicates the remaining resource need of each process.
* Need [ i,   j ] = k means process **Pi** currently need **‘k’** instances of resource type **Rj**

for its execution.

* Need [ i,   j ] = Max [ i,   j ] – Allocation [ i,   j ]

Allocationi specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.

Banker’s algorithm consists of Safety algorithm and Resource request algorithm

**Safety-request Algorithm**

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

*1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.  
Initialize: Work = Available  
Finish[i] = false; for i=1, 2, 3, 4….n*

*2) Find an i such that both  
a) Finish[i] = false  
b) Need <= Work  
if no such i exists goto step (4)*

*3) Work = Work + Allocation[i]  
Finish[i] = true  
goto step (2)*

*4) if Finish [i] = true for all i  
then the system is in a safe state*

**Resource-Request Algorithm**

Let Request be the request array for process Pi. Request[j] = k means process Pi wants k instances of resource type R. When a request for resources is made by process Pi, the following actions are taken:

*1) If Request <= Need  
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.*

*2) If Request <= Available  
Goto step (3); otherwise, Pi must wait, since the resources are not available.*

*3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as  
follows:  
Available = Available – Request  
Allocation = Allocation + Request  
Need = Need– Request*

Test cases:-

Following is the SAFE Sequence

P0 -> P2 -> P3 -> P4 -> P1