**CSE 316 : OPERATING SYSTEM ASSIGNMENT**

**SESSION 2019-2020**

**Student Name: Uttam Veeturi**

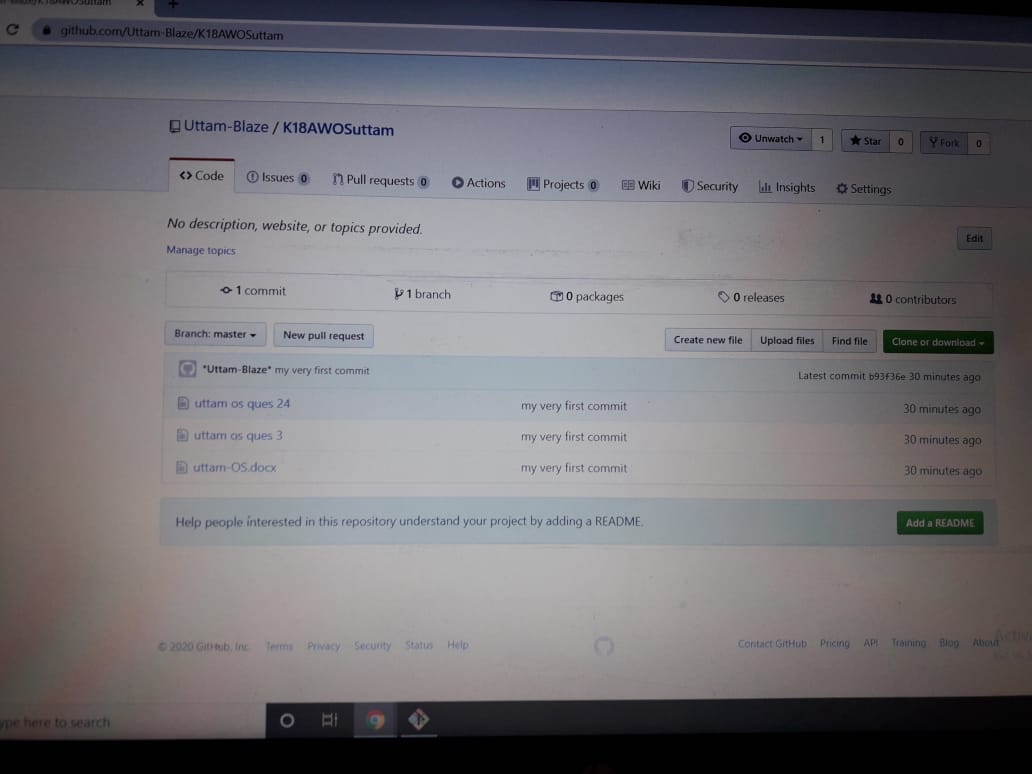
**Student ID: 11802235**

**Email Address:** [**uttamveeturi50@gmail.com**](mailto:uttamveeturi50@gmail.com)

**Section :K18AW Roll: 55**

**GitHub Link: https://uttam-blaze.github.io/K18AWOSuttam/**

(sir the link isn’t loading the correct page, however, repository has been uploaded :



)

**Ques no 3**

**DESCRIPTION**:

Thread: also known as LWP (Light Weight Process). A thread is a sequential flow of tasks of process. A thread is a basic unit of CPU utilization, it consists of –

* Program counter
* Register set
* Stack space

A thread shares with its peer threads its :

* Code section
* Data section
* Operating system resources.

While process switching happens, switching time makes the CPU to sit idle, thus more time is consumed in completing any task.

To eradicate this problem, we use a concept known as multithreaded processing.

In a multi-threaded task, while one thread is blocked and waiting, a second thread in the same task can run.

* In a web browser, one thread displays images, other texts, other fetches data from network.
* Word processor, graphics, response to keystrokes, spell checker.

POSIX thread or pthread: it is a standard for threads, implementation of pthread is available with gcc compiler. To compile a multithreaded program using gcc, we need to link it with pthread library.

Advantage:

* Threads are used to improve task completion through parallelism/
* Thread creation is faster than process.
* Context switcjing between threads is much faster.
* Threads can be terminated easily.
* Communication between threads is faster.

**ALGORITHM:**

A prime number is prime when it is only divisible by one and itself. Take for example the number 3, 3 can only be divided by one and three and have a remainder of zero. If we try to divide by two then will get a remainder of one. Therefore 3 is a prime number. Lets take another example the number 4, 4 is not a prime number because when you divide it by 2 you get a remainder of zero. Our definition states that a number can only have a remainder of zero when divided by one and itself, the number 4 clearly breaks this rule when divided by 2. 5 on the other hand can’t be divided by anything but one and itself so it’s prime.

A simple sudo code algorithm that depicts this behavior:

Lets assume if we want to know all prime numbers that come before a certain digit(Say upper limit), then :

* FOR every number X that comes before our upper limit
* FOR every number Y that comes before X
* IF X MOD Y doesn’t equal zero
* THEN this number must be prime
* ELSE X must be divisible by some Y so it is NOT a prime number
* [REMEMBER] Mod or Modulus (%) division is remainder division. EX 1: 3 %(MOD) 1 = 0; Because when you divide 3 by 1, 1 goes into 3, 3 times. This gives us a remainder of zero. EX 2: 5 % 3 = 2; Because 3 goes into 5 once, so 2 is left over as the remainder.
* Modulus division is really going to help us with this prime number problem**.**

**COMPLEXITY:**

Time complexity is O(sqrt(n)), since the loop iterates itself (sqrt(n)+1-3)/2 times, which is in O(sqrt(n)).

In best case, when number is divided by 2,3,4,5,etc. , loop will get terminated quite soon, so then complexity will be O(1).

**CODE:**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

void \*runner(void \*uttam);

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

if(argc < 2) {

fprintf(stderr, "USAGE: ./prime.out <Integer value>\n");

exit(1);

}

if(atoi(argv[1]) < 2) {

fprintf(stderr, "USAGE: %d must be >= 2\n", atoi(argv[1]));

exit(1);

}

pthread\_t tid;

pthread\_attr\_t attr;

printf("Prime Numbers: ");

pthread\_attr\_init(&attr);

pthread\_create(&tid,&attr,runner,argv[1]);

pthread\_join(tid,NULL);

printf("\nComplete\n");

}

void \*runner(void \*uttam) {

int i,j,upper = atoi(uttam);

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

int trap = 0;

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

int result = i % j;

if(result == 0) {

trap = 1;

break;

}

}

if(trap == 0) {

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

}

}

pthread\_exit(0);

}

**OUTPUT:**

~$ ./prime.out 100

Prime Numbers: [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]

**TEST CASES**:

* Let the prime number be n.
* Divide the number n by, remainder = 0, pass.
* Divide the number n by n, remainder = 0, pass.
* Divide by the number 2, remainder!=0, pass.
* Divide the number n by up to n-1 and if remainder nit equal to zero, it is a prime number.

**GITHUB LINK**:

**Ques no. 24:**

Consider following and Generate a solution in C to find whether the system is in safe state or not?

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Available | | | | Processes | Allocation | | | | Max | | | |
| A | B | C | D | A | B | C | D | A | B | C | D |
| 1 | 5 | 2 | 0 | P0 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 |
|  | | | | P1 | 1 | 0 | 0 | 0 | 1 | 7 | 5 | 0 |
| P2 | 1 | 3 | 5 | 4 | 2 | 3 | 5 | 6 |
| P3 | 0 | 6 | 3 | 2 | 0 | 6 | 5 | 2 |
| P4 | 0 | 0 | 1 | 4 | 0 | 6 | 5 | 6 |

**DESCRIPTION:**

Above given is a table with resources A,B,C,D and processes P0, P1, P2, P3, P4.

To check if the above given system is in safe state or not, we need to use banker’s algorithm for deadlock avoidance.

To deadlock avoidance techniques are used. For a deadlock avoidance,the Operating System will require prior database regarding :

* the availability of resources
* preoccupancy of the resources
* the need by the processes(resources-wise)
* the allocation of resources to th respective processes

Allocation to the future demand could be done accordingly and system will remain in safe state.

SAFE STATE : a state with no deadlock

Although the unsafe state may/may not have a deadlock state, depend on the given scenario. Unsafe state can transition to safe state if deadlock condition is resolved by some additional factor.

For storing data, we have four database/data structures :

(here m= no. of resources and n= no. of process)

1. **Available:** it is a one dimensional array of size “m” which stores no. of instances available for the resources provided in the system.
2. **Allocation:** stores the data about which process has got how many instances for each distinct resource. Two dimensional array matrix of size [n \* m]
3. **Max:** stores the data about if a process can demand maximum how many no. of instances for a resource, e.g. p0 can demand maximum 0 no. of instances for resource A. two dimensional array matrix of size [n\*m]
4. **Need:** stores the data about how many instances more will be required by a process for a resource. Also a 2-d array of size[n\*m].

**{ need = max-allocation }**

**ALGORITHM:**

Step 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

Step 2: Find an i such that both

a) Finish[i] = false

b) Need[i]<= Work

if no such i exists go to step (4)

Step3: Work = Work + Allocation[i]

Finish[i] = true

Go to step (2)

Step4: if Finish [i] = true for all i

then the system is in a safe state.

**COMPLEXITY:**

The time complexity for above given algorithm will be O(m\*n\*n), where m is no. of resources and n is no. of active process in the system.

**CODE:**

#include <stdio.h>

int main()

{

// P0, P1, P2, P3, P4 are the Process names here

int n, m, i, j, k;

n = 5; // Number of processes

m = 4; // Number of resources

int alloc[5][4] = { { 0, 0, 1, 2 }, // P0 // Allocation Matrix

{ 1, 0, 0, 0 }, // P1

{ 1, 3, 5, 4 }, // P2

{ 0, 6, 3, 2 }, // P3

{ 0, 0, 1, 4 } }; // P4

int max[5][4] = { { 0, 0, 1, 2 }, // P0 // MAX Matrix

{ 1, 7, 5, 0 }, // P1

{ 2, 3, 5, 6 }, // P2

{ 0, 6, 5, 2 }, // P3

{ 0, 6, 5, 6 } }; // P4

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);

}

**OUTPUT:**

Following is the SAFE Sequence

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

**BOUNDARY CONDITIONS:**

1. If finish[i]= true, system may return work= work +allocation
2. If finish[i]- true for all, system has reached safe state.

**TEST CASES:**

1. If allocation>max, system may fail, because need=max-allocation can’t be negative.\
2. In the end, work array has total as allocation of a resource (in all process ) + available

e.g. in A, we have 1(P1)+1(P2)+1(available)= 3 (work array)

3. in the end, work array for [A B C D]=[3, 14, 12, 12].

**GITHUB LINK**: