

**SUBJECT: CSE316**

**SECTION: EE029**

**COURSE TITLE: OPERATING SYSTEMS**

**REPORT**

**Submitted by REG NO Roll No Submitted to**

**Pinisetti V V Kalyan (11718833) A34 Prabhdeep Kaur**

**SCHOOL OF ELECTRONICS AND ELECTRICAL ENGINEERING**

**LOVELY PROFESSIONAL UNIVERSITY**

**PHAGWARA**

**2020.**

**QUESTION NO 16:-** A barrier is a tool for synchronizing the activity of a number of threads. When a thread reaches a barrier point, it cannot proceed until all other threads have reached this point as well. When the last thread reaches the barrier point, all threads are released and can resume concurrent execution. Assume that the barrier is initialized to N —the number of threads that must wait at the barrier point: init(N);

Each thread then performs some work until it reaches the barrier point:

/\* do some work for a while \*/ barrier point ();

/\* do some work for a while \*/

Using synchronization tools described in this chapter, construct a barrier that implements the following API:

• intinit(int n) —Initializes the barrier to the specified size.

• int barrier point(void) —Identifies the barrier point.

All threads are released from the barrier when the last thread reaches this point.

**Concepts**:-

**Threads**:-A Thread is a 'Light Weight Process'. A thread is a stream of instructions that can be scheduled as an independent unit. A thread exists within a process, and uses the process resources. Since threads are very small compared with processes, thread creation is relatively cheap in terms of CPU costs. As processes require their own resource bundle, and threads share resources, threads are likewise memory frugal. There can be multiple threads within a process. Multithreaded programs may have several threads running through different code paths "simultaneously".

**Pthreads:**

The Pthreads library is a POSIX C API thread library that has standardized functions for using threads across different platforms. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's. Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a header/include file and a library that you link with your program

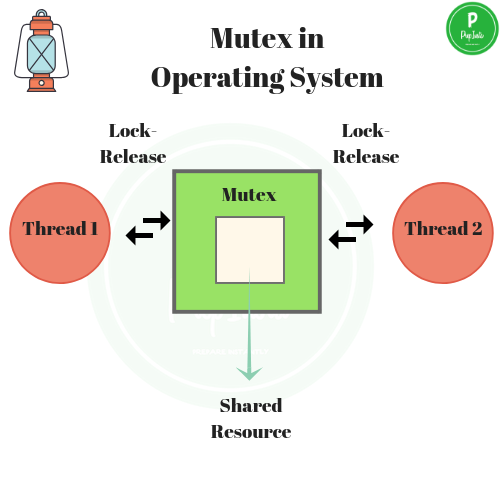
**Why Pthreads**

* The primary motivation for using Pthreads is to realize potential program performance gains.
* When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
* All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
* Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
* Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU intensive work.

**Mutex**:-

A mutex is a binary variable whose purpose is to provide locking mechanism. It is used to provide mutual exclusion to a section of code, means only one process can work on a particular code section at a time.

* Mutex is locking mechanism in OS
* 1. Mutex is used for thread.
* 2.mutex is work in userspace.
* 3.mutex is locking mechanism ownership method
* 4.thread to thread mutex is used.



**Basic Pthreads Library Calls**

* int pthread\_create
* void pthread\_exit
* pthread\_t pthread\_self()
* int pthread\_join
* **int pthread\_mutex\_lock** (pthread\_mutex\_t \*mutex);

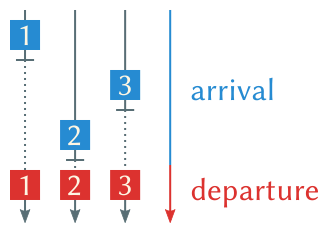
Locks a mutex.

The mutex object referenced by mutex is locked by calling pthread\_mutex\_lock. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

**Barriers**

A barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

A barrier is a method to implement synchronization. Synchronization ensures that concurrently executing threads or processes do not execute specific portions of the program at the same time. When a barrier is inserted at a specific point in a program for a group of threads [processes], any thread [process] must stop at this point and cannot proceed until all other threads [processes] reach this barrier.



**Algorithm:**

1. Initialize barrier\_size and thread\_count;

2. Create threads

3. Threads doing some work

4. Threads waiting at the barrier.

5. Barrier is released when last thread comes at the thread.

6. all threads complete thier task and exit.

7. exit.

**C language code** :-

#include<stdio.h>

#include<pthread.h>

#include<stdlib.h>

#include <unistd.h>

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

/\*

The mutex object referenced by mutex is locked by calling pthread\_mutex\_lock. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

\*/

/\*The function pthread\_mutex\_trylock is identifierentical to pthread\_mutex\_lock except that if the mutex object referenced by mutex is currently locked (by any thread, including the current thread), the call returns immediately.\*/

pthread\_cond\_t finish\_cond = PTHREAD\_COND\_INITIALIZER;

int barrier = 0; //intializing barrier is equal to zero.//

int thread\_count; //intializing thread\_count //

int barrier\_size; //intializing barrier\_size//

int counter=0; //intializing counter is equal to zero.//

int invoke\_barrier = 0; //intializing invoke\_barrier is equal to zero.//

/\*\*\*

\* params : number of threads a process is creating.

\* returns : none.

\*

\* Initialize barrier with total number of threads.

\*\* \*/

void barrier\_init(int n\_threads)

{

if ( thread\_count < barrier\_size ) { barrier = thread\_count; return; }

barrier = n\_threads;

}

/\*

\* params: none.

\* returns: -1 on failure, 0 on success.

\* decrement the count by 1.

\*/

int decrement()

{

if (barrier == 0) {

return 0;

}

if(pthread\_mutex\_lock(&lock) != 0)

{

perror("Failed to take lock.");

return -1;

}

barrier--;

if(pthread\_mutex\_unlock(&lock) != 0)

{

perror("Failed to unlock.");

return -1;

}

return 0;

}

/\*

\* params: none.

\* returns: int : 0 on sucess, -1 on failure.

\* wait for other threads to complete.

\*/

int wait\_barrier()

{

if(decrement() < 0)

{

return -1;

}

while (barrier)

{

if(pthread\_mutex\_lock(&lock) != 0)

{

perror("\n Error in locking mutex");

return -1;

}

if(pthread\_cond\_wait(&finish\_cond, &lock) != 0)

{

perror("\n Error in cond wait.");

return -1;

}

}

/\*

\* last thread will execute this.

\*/

if(0 == barrier)

{

if(pthread\_mutex\_unlock(&lock) != 0)

{

perror("\n Error in locking mutex");

return -1;

}

if(pthread\_cond\_signal(&finish\_cond) != 0)

{

perror("\n Error while signaling.");

return -1;

}

}

return 0;

}

void \* barrier\_point(void \*numthreads)

{

int r = rand() % 5;

printf("\nThread %d \nPerforming init task of length %d sec\n",++counter,r);

int sleep(r);

wait\_barrier();

if (barrier\_size!=0) {

if ((thread\_count - (invoke\_barrier++) ) % barrier\_size == 0) {

printf("\nBarrier is Released\n");

}

printf("\nI am task after barrier\n");

}

printf("Thread completed job.\n");

return NULL;

}

int main()

{

printf("Enter Barrier Size\n");

scanf("%d", &barrier\_size);

printf("Enter no. of thread\n");

scanf("%d", &thread\_count);

//Checking valid input

if (barrier\_size>=0 && thread\_count>=0) {

pthread\_t tid[thread\_count];

barrier\_init(barrier\_size);

for(int i =0; i < thread\_count; i++)

{

pthread\_create(&(tid[i]), NULL, &barrier\_point, &thread\_count);

}

for(int j = 0; j < thread\_count; j++)

{

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

}

}

//when user give wrong input then this section will execute.

else{

printf("You are entering wrong data.\n");

main();

}

return 0;

}

**Complexity**:

O (n) complexity. “n” is no of thread\_count.

**Constraints and Boundary conditions:**

• All the data entered must be integers.

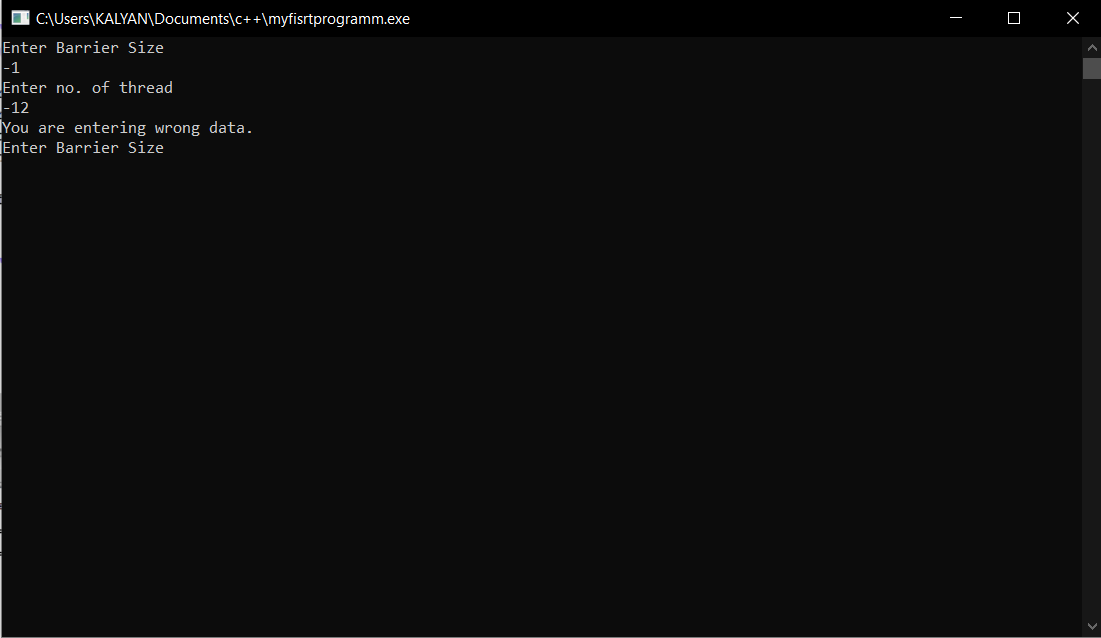
• Number of processes cannot be changed once the program starts running.

• Number of resources cannot be changed once the program starts running.

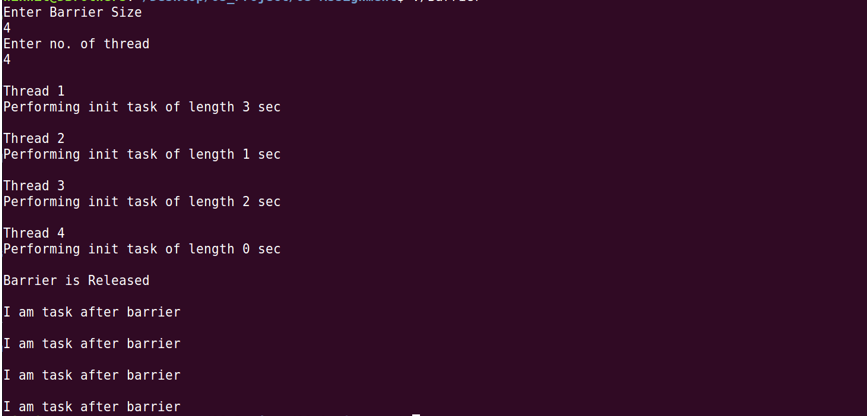
• No request for 2 processes is allowed simultaneously.

**Test Cases:-**

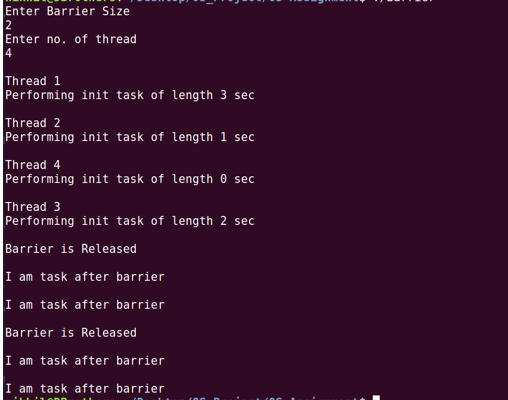
**Case1**:-When input is invalid



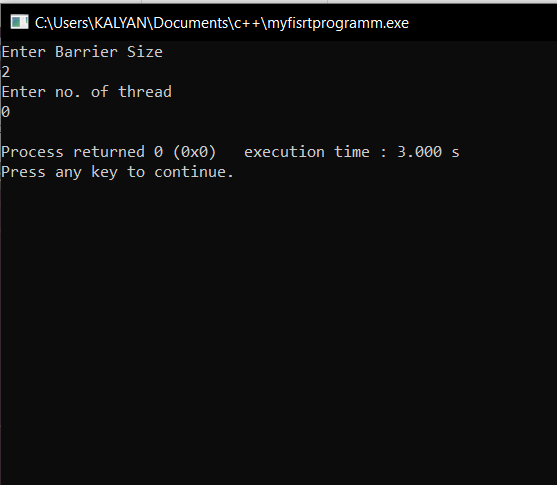
**Case2**:-when no of thread equal to size of barrier.



**Case3**:- when no of thread is greater than Barrier Size.



**Case4**:- when no of thread is equal to zero.



**Case** 5:- when barrier size is equal to zero.



**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.

**Disadvantages of threads include:**

* Global variables are shared between threads.
* Many library functions are not thread safe.
* If one thread crashes the whole application crashes.
* Memory crash in one thread kills other threads sharing the same memory, unlike processes.

**Limitations to mutex:-**

mutex have property that only owner can release it. So in a threading environment if some thread sets mutex on some shared address and it goes for long time I/O sleep or not scheduled for a long time (may be cause of lower priority) then other threads needy of that locked address will have to wait (may be infinitely causing deadlock). So generally in some operating systems ,afaik threads reset the mutex lock if any before going to sleep.

**Github Link:-**

**References**:-

* <https://www.careercup.com/question?id=9847074>
* <https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html>
* <http://www.cse.iitd.ernet.in/~dheerajb/Pthreads/Document/Pthreads_Information.html>