Operating Systems 2
SPRING SEMESTER 2018
Programming Assignment 4
Implementing Barriers using Semaphores

INSTRUCTOR: DR. SATHYA PERI
Mayank Hooda -cs16btech11022@iith.ac.in

---

0. Task :

Our task it to implement barrier using semaphore and compare the running time of threads with the pthread implemented barrier.

1. Design of the Program :

We are using *pthread_barrier_t* for implementing a barrier with *n* threads with *k* iterations each.

Critical Section :

Ad segment of code which we want only a single thread to access at a given time is called a critical section.

Entry Section :

A part of code which is available to all the threads where the threads contend to enter into the Critical Section. **Bounded wait** guarantees that no thread waits forever in the entry section and gets a fair chance to enter Critical section.

Exit Section :

A segment of code wherein a thread exits from the critical section and opens the lock for other threads to enter the CS.

Waiting Time :

The time that a thread spends in the entry section waiting to get into the critical section in called waiting time.

Semaphores :

A semaphore is a class of objects which is used to control access to a common resource by multiple processes in a concurrent system. It is similar to a lock but it allows ***n*** number of processes to enter the critical section , which is called as the semaphore value. Semaphore has two functions which are **atomic** in nature which means that only one thread can execute in a semaphore function at a time. The functions are :

Wait(*sem_t value***)**:
The wait method blocks a given process if *value<=0* . At every call , the wait method decrements *value* by 1 and henceforth allows **n** number of processes to pass through depending on *value*

Post(*sem_t value***)**:
The postl method signals the **wait** method to release a blocked process, if blocked and increments the *value* by 1 and hence if *value* again crosses 0 then more number of processes will be able to go through.

Pthread_barrier_t:

Pthreads allows us to create and manage threads in a program.When multiple threads are working together, it might be required that the threads wait for each other at a certain event or point in the program before proceeding ahead. A barrier is a point where the thread is going to wait for other threads and will proceed further only when predefined number of threads reach the same barrier in their respective programs.

To use a barrier we need to use variables of the type *pthread_barrier_t*. A barrier can be initialized using the function *pthread_barrier_init*, whose syntax is

```
int pthread_barrier_init ( pthread_barrier_t *restrict barrier , const pthread_barrierattr_t *restrict attr , unsigned count);
```

Arguments:

barrier: The variable used for the barrier
attr: Attributes for the barrier, which can be set to NULL to use default attributes
count: Number of threads which must wait call pthread_barrier_wait on this barrier before the threads can proceed further.

Once a barrier in initialized, a thread can be made to wait on the barrier for other threads using the function pthread_barrier_wait whose syntax is

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

2. Implementation of the Program :

New Implementation

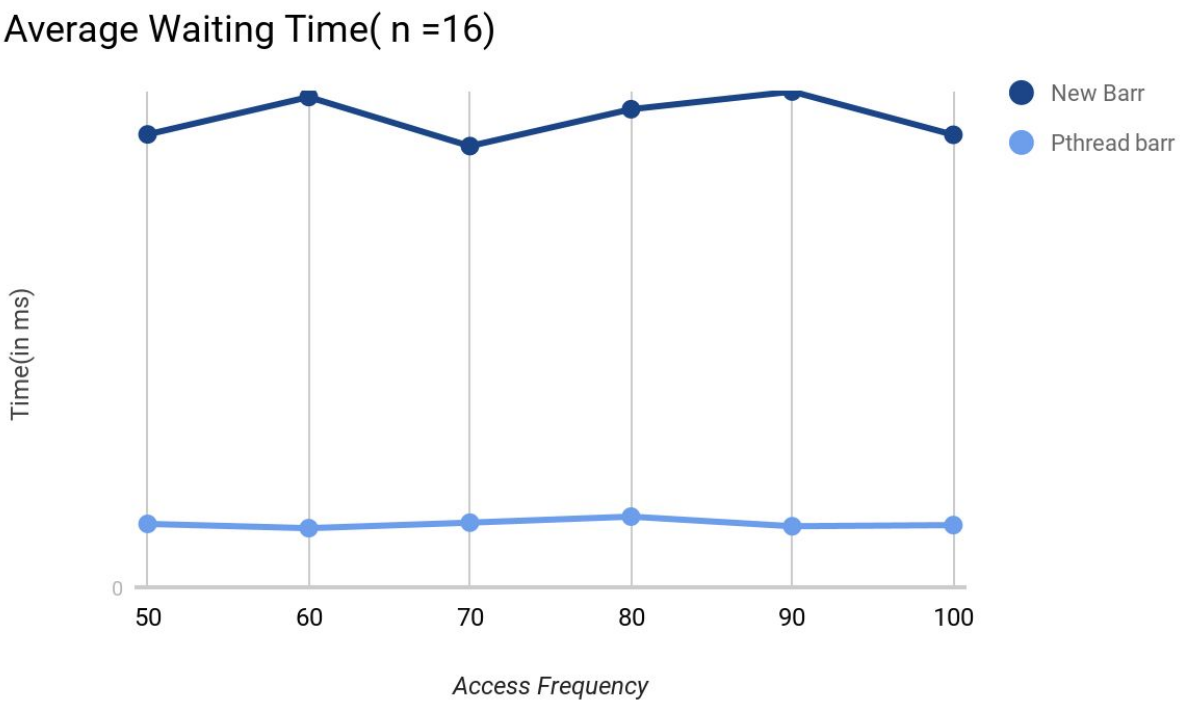
- We used 3 semaphores mutec , barrier1 and barrier2 to implement our own version of barrier.
- Mutec is used for providing atomicity to writing changes to count variable. Count variable stores the number of threads that are blocked by the barrier at any given instant , local for every thread.
- barrier1 is the barrier at which all threads except last one are stuck. Now the last thread frees all the threads to move but in doing so the status of the barrier is changed as the number of waits is less than posts and hence the value of barrier1 goes from 0 to 1 due to which if left like this , the barrier will not be reusable.
- To mitigate this problem we use barrier2 , another semaphore initialised to 1. The last thread arriving at barrier1 waits on barrier2 and changes it value to 0. Now all the threads wait on barrier2 which are eventually freed when count on decrementing goes to 0.This ensures the status of the two barriers is same as we started.

Pthread Implementation

- We use the pthread library functions *pthread_barrier_wait* for halting all the threads , it takes parameter *n* i.e. number of threads.
- Pthread_barrier_init is used to initialise the *pthread_barrier_t* .

Important Libraries Used :

- **<pthread.h> :** Used for creation and work assignment of the threads.
- **<semaphore.h> :** Used for implementing the semaphore , provides three crucial functions *sem_init* , *sem_wait* , *sem_post*.
- **<random> :** Used for the random number generator for seeding the sleep interval.
- **<unistd.h> :** Used for calculating the current system time.



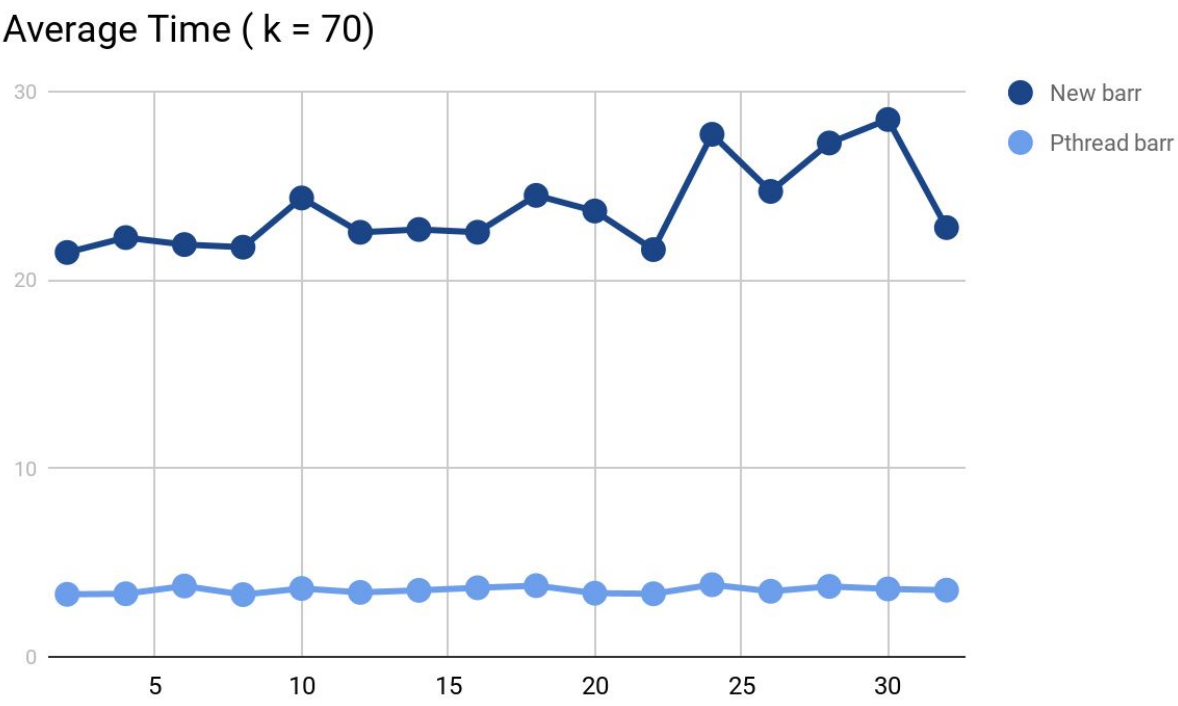
Graph Plotted for following parameters :

n = 16

50<=k <=100

preLam = 10

postLam = 20



Graph Plotted for following parameters :

k = 70

2<= n <=32

preLam = 10

postLam = 20

As we can see in the graph , our version of barrier is less efficient than the pthread version and it takes a little less time to finish execution.