Barrier implementation using synchronization tools

Problem statement:
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 otherthreads have reached this point as well. When the last thread reachesthe barrier point, all threads are released and can resume concurrentexecution.

Assume that the barrier is initialized to N—the number of threads thatmust wait at the barrier point: init(N);

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

/* do some work for awhile */ barrier point();

/* do some work for awhile */

Using synchronization tools like locks, semaphores and monitors, construct a barrierthat 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.

Barrier:

In parallel computing, 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.
Many collective routines and directive-based parallel languages impose implicit barriers. For example, a parallel do loop in Fortran with OpenMP will not be allowed to continue on any thread until the last iteration is completed. This is in case the program relies on the result of the loop immediately after its completion. In message passing, any global communication (such as reduction or scatter) may imply a barrier.

Implementation:

The basic barrier has mainly two variables, one of which records the pass/stop state of the barrier, the other of which keeps the total number of threads that have entered in the barrier. The barrier state was initialized to be "stop" by the first threads coming into the barrier. Whenever a thread enters, based on the number of threads already in the barrier, only if it is the last one, the thread sets the barrier state to be "pass" so that all the threads can get out of the barrier. On the other hand, when the incoming thread is not the last one, it is trapped in the barrier and keeps testing if the barrier state has changed from "stop" to "pass", and it gets out only when the barrier state changes to "pass". The following C++ code demonstrates this procedure.

The potential problems are as follows:

1. When sequential barriers using the same pass/block state variable are implemented, a deadlock could happen in the first barrier whenever a thread reaches the second and there are still some threads have not got out of the first barrier.
2. Due to all the threads repeatedly accessing the global variable for pass/stop, the communication traffic is rather high, which decreases the scalability.

The following Sense-Reversal Centralized Barrier is designed to resolve the first problem. And the second problem can be resolved by regrouping the threads and using multi-level barrier, e.g. Combining Tree Barrier. Also hardware implementations may have the advantage of higher scalability.

Sense-Reversal Centralized Barrier:

A Sense-Reversal Centralized Barrier solves the potential deadlock problem arising when sequential barriers are used. Instead of using the same value to represent pass/stop, sequential barriers use opposite values for pass/stop state. For example, if barrier 1 uses 0 to stop the threads, barrier 2 will use 1 to stop threads and barrier 3 will use 0 to stop threads again and so on.[The following C++ code demonstrates this.

```cpp
struct barrier_type
{
    int counter; // initialize to 0
    int flag; // initialize to 0
    std::mutex lock;
};

int local_sense = 0; // private per processor

// barrier for p processors
void barrier(barrier_type* b, int p)
{
    local_sense = 1 - local_sense;
    b->lock.lock();
    b->counter++;
    int arrived = b->counter;
    if (arrived == p) // last arriver sets flag
    {
        b->lock.unlock();
        b->counter = 0;
        // memory fence to ensure that the change to counter
        // is seen before the change to flag
        b->flag = local_sense;
    }
    else
    {
        b->lock.unlock();
        while (b->flag != local_sense); // wait for flag
    }
}
```

Combining Tree Barrier

A Combining Tree Barrier is a hierarchical way of implementing barrier to resolve the scalability by avoiding the case that all threads spinning on a same location.

In k-Tree Barrier, all threads are equally divided into subgroups of k threads and a first-round synchronizations are done within these subgroups. Once all subgroups have done their synchronizations, the first thread in each subgroup enters the second level for further synchronization. In the second level, like in the first level, the threads form new subgroups of k threads and synchronize within groups, sending out one thread in each subgroup to next level and so on. Eventually, in the final level there is only one subgroup to be synchronized. After the final-level synchronization, the releasing signal is transmitted to upper levels and all threads get past the barrier.

Hardware Barrier Implementation

The hardware barrier uses hardware to implement the above basic barrier model.

The simplest hardware implementation uses dedicated wires to transmit signal to implement barrier. This dedicated wire performs OR/AND operation to act as the pass/block flags and thread counter. For small systems, such a model works and communication speed is not a major concern. In large multiprocessor systems this hardware design can make barrier implementation have high latency. The network connection among processors is one implementation to lower the latency, which is analogous to Combining Tree Barrier.

Code:

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include <unistd.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  finished  = PTHREAD_COND_INITIALIZER;
int barrier = 0, count, size, counter=0, invoke=0;
void init(int new_threads)
{
   if ( count < size ) { barrier = count; return; }
   barrier = new_threads;
}

int decrease()
{
   if (barrier == 0) {

      return 0;
   }

   if(pthread_mutex_lock(&lock) != 0)
   {
      perror("Lock Failed.");
      return -1;
   }

   barrier--;

   if(pthread_mutex_unlock(&lock) != 0)
   {
      perror("Unlock failed.");
      return -1;
   }

   return 0;
}
int wait()
{
   if(decrease() < 0)
   {
      return -1;
   }

   while (barrier)
   {
      if(pthread_mutex_lock(&lock) != 0)
      {
         perror("\n Error locking mutex");
         return -1;
```

```
        }

        if(pthread_cond_wait(&finished, &lock) != 0)
        {
            perror("\n Error condition wait.");
            return -1;
        }
    }
    if(0 == barrier)
    {
        if(pthread_mutex_unlock(&lock) != 0)
        {
            perror("\n Error locking mutex");
            return -1;
        }
        if(pthread_cond_signal(&finished) != 0)
        {
            perror("\n Error signaling.");
            return -1;
        }
    }

    return 0;
}

void * barrier_point(void *num)
{

     int r = rand() % 5;

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

     wait();
     if (size!=0) {
       if ((count - (invoke++) ) % size == 0) {
         printf("\nBarrier Released\n");
       }
       printf("\ntask after barrier\n");

     }


     return  NULL;
}
int main()
{

    printf("Enter Barrier Size:\t");
    scanf("%d", &size);
    printf("Enter no. of thread:\t");
    scanf("%d", &count);



    if (size>=0 && count>=0) {
        pthread_t tid[count];

        init(size);

        for(int i =0; i < count; i++)
        {
            pthread_create(&(tid[i]), NULL, &barrier_point, &count);
        }


        for(int j = 0; j < count; j++)
        {
            pthread_join(tid[j], NULL);
        }
    }

    else{
      printf("wrong data...\n");
      main();
    }

    return 0;
}
```

References

1. ^ Jump up to:[a] [b] [c] *Solihin, Yan (2015-01-01). Fundamentals of Parallel Multicore Architecture (1st ed.). Chapman & Hall/CRC. ISBN 978-1482211184.*
2. ^ Jump up to:[a] [b] *"Implementing Barriers". Carnegie Mellon University.*
3. ^ Jump up to:[a] [b] *Culler, David (1998). Parallel Computer Architecture, A Hardware/Software Approach. ISBN 978-1558603431.*
4. ^ Jump up to:[a] [b] *Nanjegowda, Ramachandra; Hernandez, Oscar; Chapman, Barbara; Jin, Haoqiang H. (2009-06-03). Müller, Matthias S.; Supinski, Bronis R. de; Chapman, Barbara M. (eds.). Evolving OpenMP in an Age of Extreme Parallelism. Lecture Notes in Computer Science. Springer Berlin Heidelberg. pp. 42–52. doi:10.1007/978-3-642-02303-3_4. ISBN 9783642022845.*
5. ^ *Nikolopoulos, Dimitrios S.; Papatheodorou, Theodore S. (1999-01-01). A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin2000. Proceedings of the 13th International Conference on Supercomputing. ICS '99. New York, NY, USA: ACM. pp. 319–328. doi:10.1145/305138.305209. ISBN 978-1581131642.*
6. ^ N.R. Adiga, et al. An Overview of the BlueGene/L Supercomputer. *Proceedings of the Conference on High Performance Networking and Computing,* 2002.