

## CHAPTER 4. BARRIERS.

Next to spinlocks, barriers are the most important synchronization mechanisms for fine grained parallel programs, programs in which processes communicate a great deal with each other.

Barriers are often used with computations that go on in stages. The function of a barrier is to ensure that a given stage of computation has been finished before starting on the next. For instance, suppose that a particular computation is done in stages, and that the input to one stage is the output of the previous stage. Suppose also that a particular process *races* ahead to finish ahead of others. It then goes on to the next stage, without waiting for others to finish their assigned part. Then, all resulting computations would be in error. A barrier serves to ensure that a specified number of processes (usually all) have finished their work before they can continue.

### 4.1 The need for a barrier - an example

The following program attempts to find the mean (average) and absolute deviation of a given array  $A$  real numbers. There are two stages to the calculation. The mean is computed first, and then the absolute deviation. The formulas for the mean and absolute deviation are:

$$m = \sum_{i=0}^{n-1} \frac{A(i)}{n} \qquad a = \sum_{i=0}^{n-1} \frac{|m - A(i)|}{n}$$

The program may not work correctly! There are two stages, the first to find the mean and the second, which requires the mean, to find the absolute deviation. A program outline is as follows:

```

#include <iostream.h>
#include <math.h>
#include "utility.h"

const int size = 20;
int procs = 50;
double *A;    // an array of SIZE elements
double *mean // mean of the array
double *ad    // absolute deviation of the array
void main()
{
    int i;
    int id;
    int gate;
    int shareid[4];

    // Populate the array

    // Define A, mean, ad to be shared variables

    // Define gate to be semaphore id

    id=process_fork(procs);

// Stage1 : Compute mean
    tempsum=0;
    for (i=id; i<=size; i++)
        tempsum = tempsum + A[i] ;

    spin_lock(gate);
        *mean += tempsum/size; //update in critical section
    spin_unlock(gate);

// Stage 2: Compute ad
    tempsum = 0;
    for (i=id; i<=size; i++)
        tempsum =(float)fabs(A[i] - (*mean));;

    spin_lock(gate);
        *ad += tempsum/size; //update in critical section
    spin_unlock(gate);

    process_join(procs, id);

    //output results
    cout << "Mean = " << *mean << "\n Absol dev = " << *ad << endl;

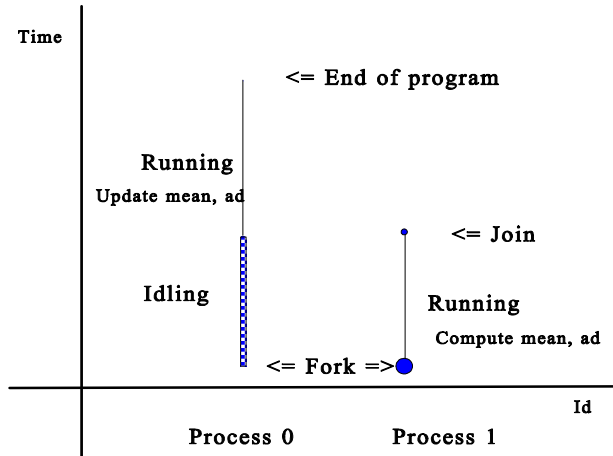
    // Release Semaphores and shared memory

}

```

The program given above has a bug! It may not work correctly always. To keep matters simple, consider the following scenario with *procs* equal to two.

1. Process 0 idles at fork.
2. Process 1 executes its code, and dies.
3. Process 0 wakes up, executes and finishes the program.



Here, the results of the computation assigned to Process 0 are not available to Process 1, and it would compute the mean incorrectly. It then goes on to find the absolute deviation incorrectly.

To see how this can happen, consider the following example:

Size = 5, Array A: 2, 4, 6, 8, 10

At fork, only Process 1 is working. It first finds the mean with its data:

$$\text{Tempsum} = 4 + 8 = 12.$$

$$\text{Mean} = 0 + 12/5 = 2.4 \quad \text{7 Stored and shared!}$$

Next Process 1 finds the absolute deviation:

$$\text{Tempsum} = |(4 - 2.4)| + |(8 - 2.4)| = 1.6 + 5.6 = 7.2$$

$$\text{ad} = 7.2/5 = 1.44 \quad \text{7 Stored and shared}$$

At this point Process 1 dies and Process 0 begins execution:

$$\text{Tempsum} = 2 + 6 + 10 = 18$$

$$\text{Mean} = 2.4 + (18/5) = 2.4 + 3.6 = 6.0 \quad \text{Mean is computed correctly!}$$

Next, Process 1 updates the value of Ad.

$$\text{Tempsum} = |(2 - 6)| + |(6 - 6)| + |(10 - 6)| = 4 + 0 + 4 = 8.$$

$$\text{ad} = 1.44 + 8/5 = 1.44 + 1.6 = 3.04 \quad ; \text{ Incorrectly computed!}$$

Notice that the value of the mean is computed correctly but not that of the absolute deviation.

The correct value of the latter is:

$$(4 + 2 + 0 + 2 + 4)/5 = 12/5 = 2.4$$

## 4.2 The barrier construct

What went wrong in the previous example? When Process 1 was computing the mean, Process 0 had not yet computed its contribution to the mean, so the absolute deviation computed by Process 1 is incorrect. Although the mean is updated correctly later, Process 0 updates the *incorrectly* updated value of *ad*.

The solution to the problem is not to let any process *race ahead* of another and finish its work. The absolute deviation cannot be computed until all processes have finished computing their contribution to the mean. What we need is a *barrier gate*.

A barrier gate is a synchronization mechanism. Its sole function is to keep track of process that have finished their work. It is enforced by a spinlock and hence is *shared*.

When a process calls at a barrier gate, the number of processes that have called at the barrier is incremented by one. Once this equals the required number, the gate opens and *all* the processes are free to go on. Otherwise, they spin at the gate waiting for other processes to catch up.

## Implementation of a barrier

We will set up a barrier as a structure with functions to initialize the barrier, open the barrier and close the barrier. It would be ideally implemented as a class in C++.

Consider the code given in *sbarrier.h* (a header file.)

```
// This Barrier can be used only once and cannot be reused.
// Header file sbarrier.h for class Barrier. (SingleBarrier use)
// Includes all functions associated with the Barrier
// To use a Barrier, first share a pointer variable of size
// (Barrier*) and use Barrierinit function on that pointer
// passing the numtoblock.

// For example, if 3 processes are to be blocked,
//   Barrier *B;
//   B = (Barrier*)shareint(sizeof(Barrier*), shareid);
//   B->Barrierinit(3);

#ifndef BARRIER_H
#define BARRIER_H

#include <iostream.h>
#include "utility.h"

class Barrier {
public:
    Barrier(); //default constructor
    ~Barrier() {} //destructor
    void Barrierinit(int numtoblock); // initialize barrier
    void WaitAtBarrier(); //Blocks processes
    void SemRelease(); //Releases Semaphore

private:
    int toblock; // # of processes barrier will block at the gate
    int gateopen; // status of the gate
    int blocked; // # of processes currently spinning at the gate
    int gate; // Spinloc (semaphore) id that enforces the gate
};

Barrier::Barrier()
{
    toblock = blocked = gateopen = gate = 0;
}

void Barrier::Barrierinit(int numtoblock)
{
    toblock = numtoblock; //Number of processes to block
    gateopen = 0; //status of gate
    blocked = 0; //No process have called yet
}
```

```

        spin_lock_init(&gate, 0);           //Initialize spinlock controlling
barrier
    }

void Barrier::WaitAtBarrier()
{
    spin_lock(&gate);                       // Critical Section !!
    blocked++;                               //Increment number of processes at
gate
    if (blocked == toblock)                 // All processes at the gate
    {
        gateopen = 1;                       // Open the gate
    }
    spin_unlock(&gate);

    while (!gateopen);                       // Wait till gate opens
}

void Barrier::SemRelease()                  // a public function to release
semaphore
{
    clean_up_sem(&gate);
}

#endif

```

Note the code in the three functions : *Barrierinit* (for initializing the barrier), *WaitAtBarrier* (which keeps track of the processes that have called at the gate and prevents racing), *SemRelease* (for releasing the semaphores). A complete program to compute mean and absolute deviation (as described earlier on Page 2) is as follows:

```

#include <iostream.h>
#include <math.h>
#include "utility.h"
#include "sbarrier.h"

const int size = 20;
int nproc = 5;

Barrier* b;           // define a barrier b

void main()
{
    int i;

```

```

    int id;
    int shareid[4];
    int gate;
    double tempsum;

    int *A;    // an array of SIZE elements
    double *mean ; // mean of the array
    double *ad    ;// absolute deviation of the array

// Define b, arr, mean, ad as shared
    b = (Barrier*)shareint(sizeof(Barrier), shareid[0]);
    A = shareint(size*sizeof(double), shareid[1]);
    mean = (double*)sharereal(sizeof(double), shareid[2]);
    ad = (double*)sharereal(sizeof(double), shareid[3]);
    *mean=0;
    *ad=0;
// Define gate to be semaphore id
    spin_lock_init(gate, 0);

// Populate the array
    for (i=0; i<size; i++)
        A[i]= i ;

// Initialize the barrier, b
    b -> Barrierinit(5);    //Initialize barrier to
                            // block 5 processes.

    id=process_fork(5);

//STEP I: computation of mean using 5 processes

    tempsum=0;
    for (i=id; i<size; i=i+5)
        tempsum = tempsum + A[i] ;

    spin_lock(gate);
        *mean += tempsum/size; //update in critical section
    spin_unlock(gate);

    cout<<id<<" arriving at the barrier " << endl;

    b->WaitAtBarrier();    // Block the processes till
                            // all of them reach this point

    cout<<id<<" leaving the barrier " << endl;

//STEP II : computation of absolute deviation

    tempsum = 0;
    for (i=id; i<size; i=i+5)
        tempsum = tempsum + (float)fabs(A[i] - (*mean));

    spin_lock(gate);

```

```

        *ad += tempsum/size;    //update in critical section
    spin_unlock(gate);

    process_join(5, id);

    for (i=0; i<4; i++)        // Release shared memory
        clean_up_shared(shareid[i]);

    b->SemRelease();          // Release Semaphore used in barrier
    clean_up_sem(gate);       // Release Semaphore

    //output results
    cout << "Mean = " << *mean << "\n Absolute deviation = " << *ad << endl;
}

```

In this program, the barrier is enforced between STEP I (computation of mean) and STEP II (computation of absolute deviation). The barrier gate prevents processes from racing ahead after completing STEP I till all processes have arrived at the gate. All the processes then go on to complete STEP II. A sample output for the above program is as follows:

```

1 arriving at the barrier
2 arriving at the barrier
4 arriving at the barrier
3 arriving at the barrier
0 arriving at the barrier
0 leaving the barrier
2 leaving the barrier
1 leaving the barrier
3 leaving the barrier
4 leaving the barrier
Mean = 9.5
Absolute deviation = 5

```

You should observe the fact that the processes do not call on the barrier in order! Also, they do not seem to leave the barrier in the order in which they entered, we cannot be sure

of this. The variation in the output could very well be the result of interleaving. You should carefully note that with a barrier, a process begins to compute the ad only after all other processes have finished contributing to the mean.

## 4.2 Multiple use of the same barrier

Can the same barrier be used in more than one place in a program. The answer is "Yes", provided some care is taken. For example, procedure *WaitAtBarrier* as given above will not work correctly in this case. The problem is that our procedure does not distinguish between *trap* mode and *release* mode of the barrier. Suppose that *WaitAtBarrier* is called at two places in the program, first at point A and then at point B:

```
WaitAtBarrier (bargate);    {Point A}
    {Work to be done}
WaitAtBarrier (bargate)    {Point B}
```

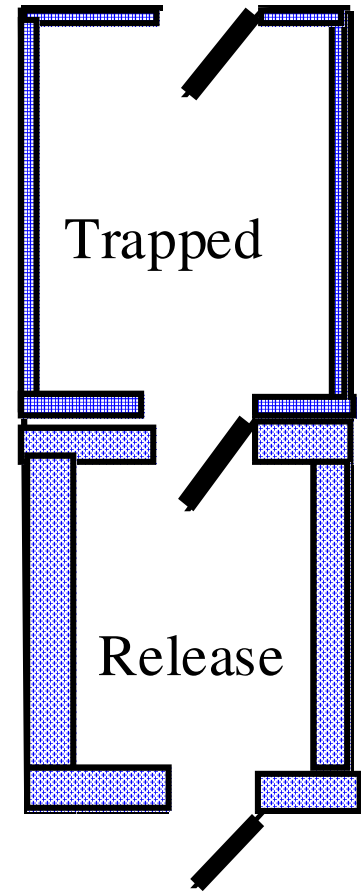
Suppose that *toblock* = 5, and that 5 processes have called at the barrier (*trap* mode) at point A. The barrier gate opens (*release* mode) allowing all of them to go on. Now suppose that all are idling except one process, say Process 0. Then Process 0 races to finish the work following the first call, and calls on the same barrier again, now at point B. But the gate is open and it will be released along with the rest of the processes. Process 0 then proceeds ahead past point B without waiting for the other processes, essentially defeating the purpose of the barrier at point B.

One way out of this problem is to use different barriers at different places. However, this involves lots of shared memory since all barriers are shared. If you have enough memory

available, this is the simplest way. Nevertheless, there are many situations where the same barrier is needed and used. Then, we can modify the

*WaitAtBarrier* procedure as follows.

Visualize a barrier as in *two* modes: *trap* phase and a *release* phase. Our current definition of the barrier illustrates only the trapping phase. To include the release phase, we first modify the definition of a barrier, to include another field *released*, which keeps track of the number released. Our aim is to release all processes when the required number have called, *and not allow any others to call once the release phase has commenced*. We shall do this by making all processes that call at the barrier to identify if they are "old" or are "new." When the barrier is in the release mode, old processes are released, new ones simply spin. If the barrier is in the trapping mode, new processes update the number at the gate and become "old" processes, waiting for their turn to be released.



Such an implementation is described in *mbarrier.h* (a header file, that implements a barrier that can be called multiple number of times)

```
// This Barrier works even for multiple calls.  
  
// Header file barrierm.h for class Barrier  
// Includes all functions associated with the Barrier
```

```

#ifndef BARRIER_H
#define BARRIER_H

#include <iostream.h>
#include "utility.h"

class Barrier {

public:
    Barrier();                //default constructor
    ~Barrier() {}            //destructor
    void Barrierinit(int numtoblock); // Initialize barrier
    void WaitAtBarrier();     //Blocks processes
    void SemRelease();        //Releases Semaphores

private:
    int toblock;             //number of processes to be blocked
    int blocked;            // number blocked
    int released;           //number released
    int gate;               //ID of spinlock to enforce barrier

};

Barrier::Barrier()
{
    toblock = blocked = gate = 0;
}

void Barrier::Barrierinit(int numtoblock)
{
    toblock = numtoblock;    //number of processes that will call
    released = 0;           // none released yet
    blocked = 0;            // no process trapped yet
    spin_lock_init(gate, 0); //initialize spinlock controlling barrier
}

void Barrier::WaitAtBarrier()
{
    int done;
    int newproc;

    done=0;
    newproc=1;             // a new process arrives at the barrier
                          // and is blocked till done =1;
                          // Note done =1 only when all processes arrive
                          // at the barrier and is in release mode;

    while (done==0)
    {
        spin_lock(gate);
        if (newproc==1 && released>0)
        {
            spin_unlock(gate);           //New process and in release
                                          // mode. This new process is at
                                          // a new subsequent barrier, while
                                          // others are still being released
        }
    }
}

```

```

// at the previous barrier.
//Simply keep spinning

else
//one or both of the following are true
// released=0 | newproc=0
{
if (newproc==1) //New process and is being trapped.
{ blocked++; // released =0 at this point.
newproc=0; // This process is no longer new
} // and is waiting to be released

if ((blocked<toblock) && (released==0))
{ //No process released
//Still in Trapping state

spin_unlock(gate) ; //Continue to spin
}
else
if (released==0)
{ released = toblock-1; // Ready to release
// First process released
blocked =0; // Reset barrier for blocking
// at the next point
done=1; // Process will release by
// exiting the loop
spin_unlock(gate);
}
else
{ released = released -1; // Release this process
spin_unlock(gate);
done = 1 ; // by exiting the loop
}
}
}

void Barrier::SemRelease()
{
clean_up_sem(gate);
}

#endif

```

You should observe that in this implementation of the barrier, a "new caller" at the barrier checks the status of the barrier. If in trapping mode, the process updates the variable blocked, becomes "old" and waits for the barrier to go into release mode. An "old caller" spins in the trapping mode, or is released when in release mode. Also observe that the barrier goes into the release mode only if there are requisite number of old callers. Once in this mode, it does not allow any

one else to be released until all the "called" processes have left.

### 4.3 An Application

*Expression Splitting* refers to the method of apportioning the sum of several terms among several processes, using one or more barriers. When  $p$ , the number of processes used is small, and we use loop splitting to do a loop, the load may not be balanced. This is particularly true if the work to be done inside the loop is large.

Suppose we have the following loop,

```
for (i=0;i<n;i++)          for(i=id; i<n; i=i+p)
    x[i] = E[i];           x[i]=E[i];
```

*Serial Version*

*Parallel Version*

where  $E[i]$  is an expression with several terms. In  $n=10$ , and  $p=4$ , under loop splitting, Processes 0 and 1 would do three iterations of the loop. Processes 3 and 4 would do only two iterations. If  $E[i]$  is fairly involved, Processes 0 and 1, which decide the speedup, would adversely affect the performance of the program. Here, we can distribute the work more equally by splitting  $E[i]$ .

Suppose  $E[i] = U(i) + V(i)$ , and that  $U$  and  $V$  roughly are of the same complexity. The work can be more evenly distributed by doing parallel computation of  $U$  and  $V$  as follows for  $n=10$  and  $p=4$ :

id=0	id=1	id=2	id=3
U(0)	V(0)	U(1)	V(1)

U(2)	V(2)	U(3)	V(3)
U(4)	V(4)	U(5)	V(5)
..	..	..	..

i.e., pairs of processes execute on the same value of  $i$ . However, we need a barrier

implementation to prevent that particular pair of processes from racing ahead. A pseudocode

to evaluate  $x[i]$  with  $p$  processes is as follows:

```
#include <iostream.h>
#include "utility.h"
#include "mbarrier.h"

const int p = 4;
Barrier* b[p/2]; // declare a barrier.

void main()
{
    int i;
    int id;
    int shareid[4];
    int *sum ;    sum is a shared array;

    b[0] = (Barrier*)shareint(sizeof(Barrier), shareid[0]);
    b[1] = (Barrier*)shareint(sizeof(Barrier), shareid[1]);

    // define sum as shared

    for (int j=0; j< p/2; j++)
        b[j]->Barrierinit(2); //Initialize p/2 barriers to
                               // block 2 processes each.

    id=process_fork(p);

    for(i=id/2;i<n;i=i+p/2)
    { if (id/2 == 0)
        sum[id]=u[i]
      else
        sum[id]= v[i];

      b[id/2]->WaitAtBarrier(); // Block the pair of processes
                               //b[0] blocks 0 and 1, and only these two, and so on
                               // This ensures that u[i] & v[i] are computed before continuing

      if (id/2==0)
        x[i] = sum[id] + sum[id+1];
        //Each even numbered process accumulates its own and its
```

```

        // successor's sum. i.e., process computes x[0]= u[0]+v[0]
        // as x[0] = sum[0]+sum[1] and so on.

    b[id/2]->WaitAtBarrier();
        // This is needed to prevent odd id's from racing ahead while
        // even id's are still computing x[i] in the above step.

    }
    process_join (id,p);
    // output results
}

```

## Remarks

There is a considerable overhead with two barrier calls in each loop. This could slow execution. This can be avoided by using a single barrier, but the tradeoff is that it requires more memory. In this version, one uses two additional shared arrays, *sum1*, *sum2* each with *n* indices:

```

Barrier* b; // we need one single barrier.
.
b->Barrierinit(p) ;// that blocks all processes.
.
id=process_fork(p);

    for(i=id/2;i<n;i=i+p/2)
    { if (id/2 == 0)
        sum1[i]=u[i]
      else
        sum2[i]=v[i];
    }
    b->WaitAtBarrier(); // wait for all processes to finish

    //Accumulate all partial sums in the two arrays sum1 and sum2
    // by another loop splitting technique

    for(i=id;i<n;i=i+p)
        x[i]=sum1[i]+sum2[i]

    process_join (id,p);
    // output results
.

```

In the next chapter, we will see additional examples of barriers.