CHAPTER 3: PARALLEL PROGRAMMING TECHNIQUES

This chapter considers some simple parallel programming techniques. In the last chapter, you learned about process contention and the errors it can cause. We will look into that topic in greater detail here. Synchronization mechanisms for interprocess communication, which avoid such errors, are considered.

3.1 Loop splitting

We have used loop splitting in several examples in the previous chapter. Loop splitting is an easy and efficient way to apportion work among several processes. It involves distributing various iterations of a loop among the processes. Simply stated, a loop such as the following,

for (i = 0; i < n; i++)

{Job (i)};

is distributed among the p processes evenly as follows:

Process Id	Values of i for which Job (i) is done
0	$0, \quad p, \quad 2.p, \quad \ldots \quad j.p$
1	$1, \quad p+1, \quad 2.p+1, \ldots \quad j.p+1$
..	....
$p\text{-}1$	$p\text{-}1, \quad 2.p\text{-}1, \quad 3.p\text{-}1 \quad \ldots \quad (j+1)p\text{-}1$

If $(j+1)p = n$, then all the processes do an equal number of iterations, namely $(j + 1)$. If not, some processes will do only j iterations.

Consider the following example in which the maximum of an array of real numbers is found by loop splitting. Each process finds the maximum of a subset of the given array and stores it in another array. The maximum of the latter is found in the usual way serially.

```
#include <iostream.h>
const int size = 1000, procs = 50, shareidsize = 5;
float realarray[size];
float procarray[procs];
float max;
int i, maxi;
int shareid[shareidsize];  //shareid's assigned by O.S

float *lp = NULL;
float *prp = NULL;



extern "C"{                                //link c functions from util.o
int process_fork(int nproc);
void process_join(int nproc,int id);
int *sharelist (int size, int& id);
int *sharerealarray (int size, int& id);
void clean_up_shared (int id);

}

// header for function splitmax
void splitmax(float *prp, float* lp);

main()
{
   lp = (float*)sharelist(size*sizeof(float), shareid[0]);
   prp = (float*)sharerealarray(procs*sizeof(float), shareid[1]);


   for (i = 0; i < size; i++){      //populate the array
            lp[i] = (float)(i * 2.2);
        }

     lp[48]=9999.99;

      splitmax (prp, lp);

   max = prp[0];  // initialize max
   maxi = 0;      // initialize maxindex


      for (i = 0; i < procs; i++){  //get largest maximum from all processes
            if (max < prp[i])
                  max = prp[i];
                  maxi = i;
       }

   clean_up_shared(shareid[0]);    // Release memory resources
   clean_up_shared(shareid[1]);

      cout << "Max of this array = " << max << endl<< maxi<<endl;
      return 0;
}

void splitmax(float *prp, float* lp){
      int id, j;
      float tempmax;
```

```
        id = process_fork(procs);
        tempmax = lp[0];
        for (j = id; j <= size; j += procs){
              if (tempmax < lp[j])
                    tempmax = lp[j];
        prp[id] = tempmax;
        }

        process_join(procs, id);
}
```

You should particularly note the *synchronization* portion of the program above. This is the part where the results of the various processes are collected and the result found. Note also that there is no contention. Each process stores the max of its portion of the array in a separate place. If we attempted to parallelize the synchronization part also, perhaps by the parent updating her own max by constantly inspecting those of her children, we will be led to process contention, leading further to possible errors. Parallelizing the synchronization portion by carefully avoiding process contention is often called *mutual exclusion.* This will be considered later in this chapter.

3.2 Loop splitting and speedup

In the last chapter we considered the speedup of loop splitting achieved by creating processes. Recall that speedup is

speedup = (time for serial execution)/(time for parallel execution).

If n is the number of iterations in a loop, p is the number of processes created, and t is the time to execute the loop once, then the serial time is nt. Suppose it costs τ units to create a single process. If $i = n/p$ is an integer, then the parallel cost of executing the loop is $p\tau + it$. Note that i represents the number of loop iteration *each* process has to do. In this case there is load balancing and the

speedup = $nt/(p\tau + it)$.

If we exclude the overhead cost $p\tau$, we see that the maximum speed up possible is $n/i = p$. That is the speedup is never more than the number of processes.

What happens if n/p is not an integer, that is p does not divide n ? In this case we would not have load balancing, that is some processes would do more iterations through the loop than the others. What is the speed up then? To fix ideas, suppose $n = 100$, $p = 8$. In the above example, the loop splitting is accomplished as follows:

```
for (i = id; i < n; i=i+p)
    {Code to be executed here};
```

With this, Process 0 iterates through the loop for thirteen values of i:

0, 8, 16, ... 96

Process 1 also iterates through thirteen values:

1, 9, 17, ... 97

This is true for Processes 0, 1, 2, and 3 However, beginning with Process 4, all others do only twelve iterations only and hence their loads are not balanced. The speedup is determined by the process that does the most work (often referred to *rate determining step*). Since several processes do 13 iterations, the speedup therefore, is $100/13 \cong 7.6$. In general, each process would have to do at least n div p iterations (in the above example, 100 div 8 = 12), and if n mod $p > 0$, that is p does not divide n (in the example, 100 mod 8 = 4 > 0), then at least one process would have to do an additional iteration. In this case, the rate determining step (n div p + 1). This is precisely the value of the ceiling function, $\lceil n/p \rceil$ which is defined to be the smallest integer k such that $k \geq \lceil n/p \rceil$. Clearly, $k = n/p$ if p divides n, or else, $k = (n \text{ div } p) + 1$.

The maximum speedup therefore is n/k.

It is instructive to consider the speedup for various values of n and p. This is given in the following table. We have assumed p = 10 throughout.

n	$\lceil n/p \rceil$	$n/\lceil n/p \rceil$
10	1	10
100	10	10
9	1	9
19	2	9.5
109	11	9.9
11	2	5.5
21	3	7.0
101	11	9.2

Finally, what would be the speedup if the actual number of processors is much less than the number of processes? In this case, the operating system will assign the processes to the available number of processors. Assuming all processors are busy, if m is the number of processors (with m # p), then each processor would have to run p/m processes, assuming p/m is an integer. If not, as in the above analysis, the maximum speedup will be $n/(\lceil p/m \rceil . \lceil n/p \rceil)$. As we would expect, if the quantities inside the ceiling functions are all integers, the maximum speedup is m, the actual number of processors present.

3.2 Self-scheduling, contention, and spin-locks

In a loop that is split as follows,

```
for (i = id; i < n; i=i+p)
    {Code to be executed here};
```

as we just saw in the previous section, the loop may not be split evenly among the various processors and processes. Even if we knew the number of processors available and took care to ensure that all processes did identical number of loop iterations, it is still likely that they do unequal amount of work, that is, the work load may not be balanced. This would occur if the code to be executed does not take a fixed amount of time, is dependent upon fulfillment of other conditions, and which varies from iteration to iteration. This would result in some processes being busy for much longer time than others. Ideally, one should be able to let any process (when it is not busy) choose the "next available" value of the loop counter and execute the code. In other words, processes have the freedom to choose whichever value of the loop counter is available and not be forced into executing the code for fixed values (dependent upon process id) of the counter. This procedure of allocating work is usually called *self-scheduling*.

It is immediate that the values of the loop counter for which code still needs to be executed, must be visible to all the processes, and which they must share. This also means that such values will be updated by the various process which process the loop and this would lead to contention!

Consider the following example in which a vector is multiplied by a constant using self-scheduling. However, this program has a bug, and may not run as intended!

```
#include <iostream.h>
const int size = 25, procs = 8;
float vector[size+1], c = (float)3;
int i, id, *nextindex = NULL, temp = size;
float *A = NULL;
int shareid[2];
extern "C"{                          //link c functions from util.o
int process_fork(int nproc);
void process_join(int nproc,int id);
int *shareint (int size, int& id);
int *sharerealarray (int size, int& id);
void clean_up_shared (int id);
}


main()
{
      int k;

      nextindex = shareint(sizeof(float*), shareid[0]);
      A = (float*)sharerealarray(sizeof(float*), shareid[1]);

                  //set A to start of vector
      for (i = 0; i < size; i++){         //populate the array
            A[i] = (float) i;
      }
      cout << "\nA Initial:" << endl;
      for(k = 0; k < size; k++)
            cout << " A[" << k << "] = " << A[k]<<endl;
      cout << endl;
      *nextindex = size-1;

      id = process_fork(procs);

      while ((*nextindex) >= 0){
            i = *nextindex          // Statement A
            (*nextindex)--;         // Statement B
            A[i] *= c;              // Statement C

      }
      process_join(procs, id);

      cout << "A multiplied by c:" << endl;     //BUG IN PROGRAM!
      for(k = 0; k < size; k++)
            cout << " A[" << k << "] = " << A[k]<<endl;

     for (k=0;k<2;k++)
         clean_up_shared(shareid[k]);

            return 0;
}
```

In this program, self-scheduling is attempted inside the *while loop*. The array entries are multiplied by the constant c, beginning with the last index. Each process picks up the next

available index (from the end), decrements by one, and then multiplies the array entry by c. This way, the number of array indices handled by a process is simply dependent upon the speed of the process.

As you can see, the variable **nextindex* is shared and is sure to cause contention. In fact, consider the following scenario immediately after the fork. The labels A, B and C refer to the corresponding labeled statements inside the *while loop*:

1. Process 0 executes (A) and idles. Note that i = 1000, *nextindex = 1000.
2. Process 1 executes both (A) and (B)and then idles. Here i = 1000, *nextindex = 999.
3. Process 0 executes (B) and idles. Again i = 1000, but *nextindex = 998.
4. Process 2 executes (A) through (C). Here i = 998, *nextindex = 997.

Assuming no interference while executing (C), evidently A[999] will remain unchanged. The cause of the problem again is contention.

3.3 Mutual exclusion, critical section

When a shared variable is changed by several processes, the final value depends on the order in which it is accessed. In particular, the changes may be unintended! This cannot be checked by program testing. To avoid this, we define a *critical section*.

A *critical section* is a section of code which can only be executed by a single process at a time. When a process begins to execute the code in a critical section, it will be allowed to finish the execution before another process is allowed access inside the critical section. In particular, in the example above, the lines

```
i = *nextindex
(*nextindex)--;
```

should be in a critical section (it involves a few assembly language instructions). Critical sections should be relatively short to avoid other processes being denied access for too long a time resulting in performance degradation.

The term *mutual exclusion* is sometimes also used in conjunction with critical section. It refers to allowing access to the critical section, for precisely one process. It is instructive to consider a software solution. Assume that we have two processes. Consider the following attempted "solution" in pseudocode. We use "(shared)" to indicate a shared variable:

Attempted Solution 1

```
var Door : (open, closed); (shared);
begin
    Door := Open;

        Process 1                   Process 2

        repeat                      repeat
        {continue testing}      {continue testing}
        until Door = Open;          until Door = Open;
        Door := closed;             Door := closed;
            {CRITICAL                   {CRITICAL
             SECTION}                    SECTION}
        Door := Open;           Door := Open;

....
end.
```

In this code, a boolean variable *Door* is used to check by each process to see if it can enter the critical section. The idea is that if it finds it open, it will close it immediately and get inside the critical section. After finishing the work, it sets Door to open. Unfortunately, this solution cannot work. It is possible that both processes find Door open, and both will enter the critical section. For instance, once Process 1 notes Door is open, it idles. Process 2, also noticing that Door is

open, enters the critical section. Now Process 1 wakes up and proceeds. Obviously we have failed to provide mutual exclusion.

Attempted Solution 2

We could order the processes to enter their critical sections in a specified way: First Process 1 and then Process 2, and so on.

```
var Turn : 1..2; (shared);

begin

    Turn := 1; {Initialize so that it is the turn of Process 1}

        Process 1                  Process 2

        repeat                     repeat
        {continue testing}      {continue testing}
        until Turn = 1;         until Turn = 2;

            {CRITICAL                  {CRITICAL
             SECTION}                   SECTION}

        Turn := 2;                 Turn : = 1;

....
end.
```

This solution works! But it places severe constraints on the processes, namely that they proceed in the order 1, 2, 1, 2. One disadvantage is that we no longer have self-scheduling. More seriously, if Process 1 goes slower (or is idled), so must Process 2.

Attempted Solution 3

This is a combination of the last two. Each process has its own temporary area (think of this as a "vestibule", or entrance hall), before entry to the critical section. Each process also knows whether another is inside or outside this temporary area . A process would then enter the critical

section only if it knows that the other is *not* in its own vestibule. The idea is that each process would better coordinate its own entry into the critical section by knowing the status of the other.

```
var Proc :array [1..2] of (inside, outside); (shared);
begin
    Process 1                        Process 2

    {Initially both are outside critical section}

    Proc [1] := Outside;             Proc [2] := Outside;
    repeat                           repeat
        Proc [1] := Inside;          Proc [2] := Inside; {Attempting to enter}
        if Proc [2] = Inside then        if Proc [1 = Inside then {If the other is also inside}
            Proc [1] := Outside;             Proc [2] := Outside {get out}
        repeat                           repeat
        until Proc [2] = Outside;        until Proc [1] = Inside;
    until Proc [1] = Inside;         until Proc [2] = Inside;

        {CRITICAL                    {CRITICAL
         SECTION}                     SECTION}

    Proc [1] := Outside;             Proc [2] := Outside;

....
end.
```

This is an interesting attempt: A process that finds that the other is inside, changes its own and waits until the other is done. Unfortunately, this may not work either! If the two processes go in lock-step (both in, both out), they will "spin" in their respective repeat loops. This is similar to two people calling each other at the same time, both hang up and try again - still finding busy signals.

The first person to solve the mutual exclusion problem for two processes was Dekker. Since then, this has been generalized to several processes. We give below Dekker's solution; the solution for several processes is complicated.

Dekker's solution for mutual exclusion (two processes)

This is a combination of the previous attempted solutions. Both "vestibules" and "turns" are used.

```
var
    Proc : array [1..2] of (inside, outside) ; (shared);
    Turn = 1..2; (shared);

begin
    Turn := 1;  Proc [1] := Outside;  Proc [2] := Outside;

    Process 1                           Process 2

    Proc [1] := Inside;             Proc [2] := Inside;
    if Proc  [2] = Inside then          if Proc [1] = Inside then
        begin                               begin
            if Turn = 2 then                    if Turn = 1 then
                begin                               begin
                    Proc [1]:=Outside                   Proc [2] := Outside;
                    repeat                              repeat
                    until Turn = 1;                 until Turn = 2;
                    Proc [1] := Inside;             Proc [2] := Inside;
                end;                                end;
            repeat                              repeat
            until Proc [2] = Outside;           until Proc [1] = Outside;
        end;                                end;

    {CRITICAL SECTION}                      {CRITICAL SECTION}

    Turn := 2;                          Turn := 1;
    Proc [1] := Outside;                Proc [2] := Outside;

    ...........                     ...............
end.
```

3.4 Semaphores and Spinlocks

We have just seen that enforcement of mutual exclusion to a critical section, solely by means of software can be cumbersome and complicated. Fortunately, UNIX provides the *semaphore* construct. These constructs are fairly complex. Essentially, semaphores control access to critical sections. However, they can provide such access to groups of processes, not just one.

Semaphores are created through system calls (you guessed it, code is in C!) and shared amongst the various processes. A general semaphore has an internal integer (∃ 0) counter which is preset to a given value (usually the number of processes that should be allowed access).

Before entering a critical section, a process checks the status of the semaphore counter. If the counter is positive, it is decremented by one, and the process allowed access. Otherwise, the process waits. Likewise, a process already inside the critical section, before leaving the section sends a call to increment the counter by one.

A *binary semaphore* has exactly two states. Its counter value is either zero or one. A process checks the status of the semaphore, and if it is one, is allowed entry to the critical section; otherwise it waits until a process already inside the section increments the semaphore counter.

It is important to note that, in general, mutual exclusion is *not* enforced by the operating system, or hardware (it could be). It is enforced mutually by all the processes which agree to abide by the status of the semaphore or similar construct, and by software.

For our programs, we will ensure mutual exclusion by constructing *spin-locks.* A spin-lock is a construct which uses a binary semaphore. Such a construct, with appropriate system calls, gives the semaphore an id, initializes its state and changes it at our option. These are accomplished by the following three functions which must be placed in your Pascal program as external C programs since their codes are all in C (provided in the Appendix). As before, the file containing them must be precompiled before (just as in the case of process_fork, process_join and shared-memory).

```
void spin_lock_init(int& lock, int condition);
void spin_lock(int lock);
void spin_unlock(int lock);
```

Here, the parameter *lock* is the id of the binary semaphore manipulated by these functions. The parameter *condition* describes the status of the semaphore. Its values are zero (unlocked, or permission to enter granted) or one (locked, wait until condition is unlocked). The id of the semaphore is needed in "clean up" as in the case of shared memory. There are only a fixed number of semaphores and since a semaphore is shared by all processes, you must free them at the end of the program (the code to do this is given in the Appendix). The clean up function is

void clean_up_sem (int lock);

We illustrate the use of these functions in the following program, which is simply a revised version of the program on page 3-2, using spinlocks.

```
#include <iostream.h>
const int size = 100, procs = 10, unlocked = 0, locked = 1;

float realmax;
int i;
int lockword1 = 5, lockword2 = 6; // semaphore id's ?
// Operating system assigns the actual id's


float *pra = NULL;
int *nindex;
float *amax;
int shareid[4] ;
//Operating system reassigns these id's

extern "C"{                              //link c functions from util.o
int process_fork(int nproc);
void process_join(int nproc,int id);
int *shareint (int size, int& id);
int *sharerealarray (int size, int& id);
int *sharereal (int size, int& id);
void spin_lock_init(int& lock, int condition);
void spin_lock(int lock);
void spin_unlock(int lock);
void clean_up_shared(int id);
void clean_up_sem(int id);
}



// header for function splitmax
void splitmax(float *pra, int *nindex, float* amax);
```

```
main()
{
      spin_lock_init(lockword1, unlocked);
      spin_lock_init(lockword2, unlocked);
      // lockword1 and lockword2 are assigned semaphore id's

      pra = (float*)sharerealarray(size*sizeof(float*), shareid[1]);
      nindex = shareint(sizeof (int*), shareid[2]);
      *nindex = size-1;

      amax = (float*)sharereal(sizeof(float*), shareid[3]);

      for (i = 0; i < size; i++){         //populate the array
            pra[i] = (float)(i * 2.2);

      }

      pra[8] = 9999.99;  // Verify that pra[8] is the maximum.
      *amax = pra[0];

      splitmax (pra, nindex, amax);
      realmax = *amax;

      cout << "Max of this array = " << realmax << endl;


// Remember to clean up and release resources

 clean_up_sem(lockword1);
 clean_up_sem(lockword2);     // semaphores
for (i=1;i<4;i++)
 clean_up_shared(shareid[i]);    //shared memory pointers

cout << "check if all resources all released by ipcs command";

      return 0;

}

void splitmax(float *pra, int *nindex, float* amax){

      int j, id;
      float tempmax;
      id = process_fork(procs);
      tempmax = 0;
      while (*nindex >= 0){

            spin_lock(lockword1);         //get lock to enter critical section
            j = *nindex;                  //decremented in critical section
            *nindex = *nindex - 1;
            spin_unlock(lockword1); //free spinlock

            cout << id<< "  " << j<<endl; // check self scheduling of j
            if ((pra[j]) >= tempmax)
                  tempmax = (pra[j]);
```

```
        }

        spin_lock(lockword2);               //a different spinlock to update max

        if (tempmax >= (*amax))
              *amax = tempmax;
        spin_unlock(lockword2);
        cout << id << "    "<<tempmax<<endl;
        process_join(procs, id);

}
```

Comments

1. Each process does two things: One, it processes as many index values as possible (in the *repeat loop*) to find the maximum of the array elements it has examined. This may not be the true maximum, since the process may not have examined the entire array. Second, when no more array entries remain to be examined, the process compares the temporary maximum it has with the maximum found by others and updates it (*amax*^) .
2. We have used two critical sections, one for updating the last index processed, and the second to update the maximum found thus far. Note that both variables, *nindex* and *amax*^ are shared, hence they must be in critical sections.
3. The spinlocks are initialized in the main program. When initialized, the parameter *condition* describes the initial status of the lock, here set to zero, indicating unlocked. Besides the function to initialize the spinlock, the actual work in protecting the critical section is done by two other functions: *spin_lock* and *spin_unlock*. The first of these, called in beginning of the critical section, does the most important work of checking whether to allow access to the critical section or not. The second function, called at the end of the critical section, returns the spinlock to unlocked state.

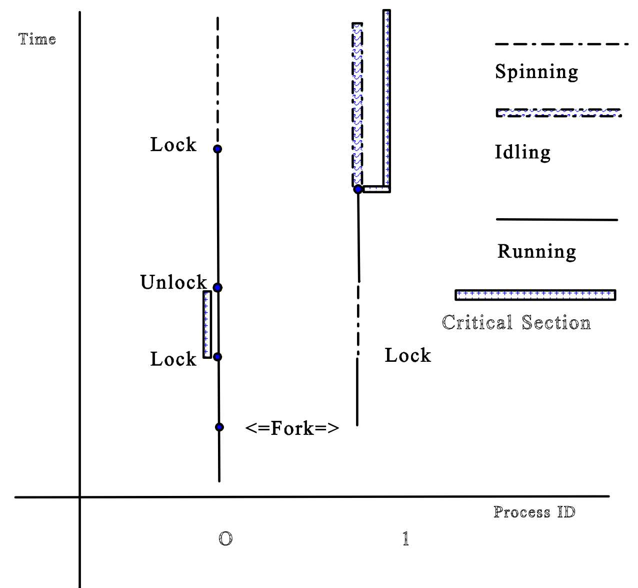
4. Since all processes call function *spin_lock*, it is interesting to examine the effect of this call. It is important to note that *a mere call to this fimction does not allow entrance to the critical section!* When a process calls *spin_lock*, a check is made to see if the lock is open or unlocked. If it is, the process, enters the critical region and *the condition of the lock is immediately changed to locked.*

5. The condition of a spinlock is controlled by the underlying semaphore. The changes in the value of the semaphore (zero or one) determines the status of the lock. Note that the change in the value of the semaphore is *atomic*. Note also that this done through software. What happens when a process calls *spin_lock,* as it eventually must? If the spinlock is unlocked, it is changed to lock (atomic) and the process is allowed access to the protected section. However, if the process finds that the condition of the spinlock is locked (meaning another process is already in the critical section), the calling process *must wait until the spinlock's condition changes to unlocked*. In essence, it is executing one of the following loop structures:

```
repeat  until (condition = 0) ;
        or
 while (condition =1) ;
```

The waiting process is said to be "spinning its wheels", hence the name spinlock. Any number of processes could be waiting at the spinlock at a given time. The instant spinlock changes its condition, one of the waiting process will be allowed inside the critical section. However, amongst the various waiting process, it is not known which one will be allowed.

6. A process waiting for the spinlock to open is not idling; it is in a waiting state which could change any instant. See figure on right. Since waiting processes do not do any useful work, *spinlocks must be used efficiently.* In particular, if the code in the protected region is very small, and a process which is inside is idled by the operating system, the other process will be spinning doing no work. This is probably more efficiently done by loop splitting. In general, code inside a critical region must be fairly long and involved to make self-scheduling inefficient.



Spinlocks provide an important mechanism for *inter-process-communication* in shared memory systems. Programs where there is little communication between processes are called coarse-grained. They are relatively easier to write. On the other hand, fine-grained programs where processes need to communicate with each other for exchange of data are usually harder. What happens when a process is idled in a critical section (as in the case of Process 1 in the above diagram)? How does it affect the performance of others and the program? These questions are subjects for the next chapter.