

## P5 - Concurrency Control (Part 2)

### Revision

- Q1 There are two companies with 10,000,000 employees. Both companies have the same owner. You would like to know the salaries of all employees of each company. The serial code can be downloaded [here](#). Try to execute the code in parallel by adding the directives below:

```
#pragma omp parallel shared(salaries1, salaries2)
num_threads(16)

#pragma omp for reduction(+:salaries1)

#pragma omp for reduction(+:salaries2)
```

```
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000Salaries1: 1362004000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
1464690000
```

```
Microsoft Visual Studio Debug Console
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries1: 1362004000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
Salaries2: 1464690000
In 0.007391 seconds
```

**Challenge :** Try to make the output to be printed properly, in line by line (right) \* tips: use the **#pragma omp** command that we've learned before.

- Q2 Modify the solution for Q1, so that it displays the salaries once only. (Hint: use **only one line of** reduction clause)

```
Salaries1: 1362004000
Salaries2: 1464690000
```

- Q3 Modify the solution for Q1, so that it displays the salaries once only. (Hint: use **#pragma omp single**)

```
Salaries1: 1362004000
Salaries2: 1464690000
```

## P5 - Concurrency Control (Part 2)

---

### Barrier

A barrier defines a point in the code where all active threads will stop until all threads have arrived at that point. With this, you can guarantee that certain calculations are finished. For instance, in this code snippet, computation of `y` can not proceed until another thread has computed its value of `x`.

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    x[mytid] = some_calculation();
    y[mytid] = x[mytid]+x[mytid+1];
}
```

This can be guaranteed with a barrier pragma:

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    x[mytid] = some_calculation();
    #pragma omp barrier
    y[mytid] = x[mytid]+x[mytid+1];
}
```

Apart from the barrier directive, which inserts an explicit barrier, OpenMP has **implicit barriers** after a load sharing construct. Thus the following code is well defined:

```
#pragma omp parallel
{
    #pragma omp for
        for (int mytid=0; mytid<number_of_threads; mytid++)
            x[mytid] = some_calculation();
    #pragma omp for
        for (int mytid=0; mytid<number_of_threads-1; mytid++)
            y[mytid] = x[mytid]+x[mytid+1];
}
```

## P5 - Concurrency Control (Part 2)

---

### Implicit Barrier

At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an implicit barrier at the end of a parallel region.

```
#pragma omp parallel
{
    some_calculation();
} // Implicit barrier on every end of a parallel region
```

There is some barrier behavior associated with omp for loops and other work sharing constructs barriers. For instance, there is an implicit barrier at the end of the loop. This barrier behavior can be canceled with the clause.

You will often see the idiom

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i=0; i<N; i++)
            a[i] = // some expression
    #pragma omp for
        for (i=0; i<N; i++)
            b[i] = ..... a[i] .....
```

Here the nowait clause implies that threads can start on the second loop while other threads are still working on the first. Since the two loops use the same schedule here, an iteration that uses a[i] can indeed rely on that value has been computed.

### Pros and Cons

The main reason for a barrier in a program is to avoid data races and to ensure the correctness of the program.

Of course there are some downsides. Each synchronization is a threat for performance. When a thread **waits** for other threads, it does not do any useful work and it spends valuable resources.

Another problem might occur if we are not carefully inserting barriers. As soon as one thread reaches the barrier then all threads in the team must reach the barrier. Otherwise, the threads waiting at the barrier will **wait forever**.

## P5 - Concurrency Control (Part 2)

---

### Locks

OpenMP also has the traditional mechanism of a lock. A lock is somewhat similar to a critical section: it guarantees that some instructions can only be performed by one process at a time. However, a critical section is indeed about code; a lock is about **data**. With a lock you make sure that some data elements can only be touched by one process at a time.

One simple example of the use of locks is generation of a histogram . A histogram consists of a number of bins that get updated depending on some data. Here is the basic structure of such a code:

```
int count[100];
float x = some_function();
int ix = (int)x;
if (ix >= 100)
    error();
else
    count[ix]++;
```

It would be possible to guard the last line:

```
#pragma omp critical
count[ix]++;
```

but that is unnecessarily restrictive. If there are enough bins in the histogram, and if the `some_function` takes enough time, there are unlikely to be conflicting writes. The solution then is to create an array of locks, with one lock for each count location.

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Since the set call is blocking, there is also

```
omp_test_lock(omp_lock_t *lock);
```

Unsetting a lock needs to be done by the thread that set it. Lock operations implicitly have a **flush**.

## P5 - Concurrency Control (Part 2)

---

Besides using OpenMP locks, the C++ Thread class can be utilized to perform parallel concurrent processing using multiple threads.

### Thread

Class to represent individual threads of execution.

<https://m.cplusplus.com/reference/thread/thread/>

Example of code with Threads process

```
// thread example
#include <iostream>          // std::cout
#include <thread>             // std::thread

void foo(){
    // do stuff...
}

void bar(int x){
    // do stuff...
}

int main(){
    std::thread first (foo);    // spawn new thread that calls foo()
    std::thread second (bar,0); // spawn new thread that calls bar(0)

    std::cout << "main, foo and bar now execute concurrently...\n";

    // synchronize threads:
    first.join();               // pauses until first finishes
    second.join();              // pauses until second finishes

    std::cout << "foo and bar completed.\n";

    return 0;
}
```

### Mutex

A *mutex* is a [lockable object](#) that is designed to signal when critical sections of code need exclusive access, preventing other threads with the same protection from executing concurrently and accessing the same memory locations.

<https://m.cplusplus.com/reference/mutex/mutex/>

## P5 - Concurrency Control (Part 2)

---

### Lock mutex

Locks all the objects passed as arguments, blocking the calling thread if necessary.

<https://m.cplusplus.com/reference/mutex/lock/>

Q4 Besides using OpenMP locks, the C++ <mutex> library also allows the program to initialize locks to prevent data races. Declare `std::mutex` increment and then implement the methods below for the [code](#) that count the number of even numbers:

```
increment.lock();
```

```
increment.unlock();
```

\* Add

```
std::cout << "thread: " << numEven << std::endl;
```

after the increment of `numEven` to see the difference of implementing / not implementing lock mutex.

## P5 - Concurrency Control (Part 2)

---

### Lock guard

A *lock guard* is an object that manages a *mutex object* by keeping it always locked.

[https://m.cplusplus.com/reference/mutex/lock\\_guard/](https://m.cplusplus.com/reference/mutex/lock_guard/)

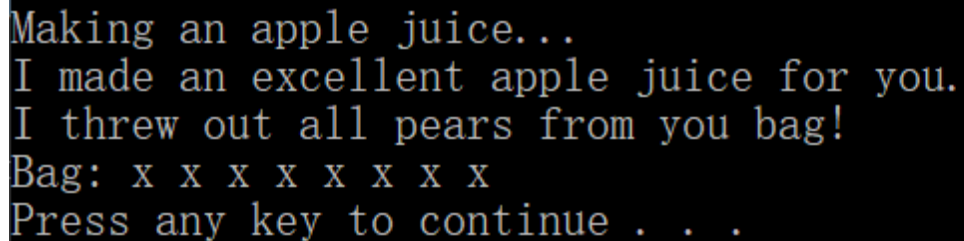
### Deadlock

Q5 Download the demo [code](#) for deadlock. Implement locks at the correct section to prevent deadlock. To avoid the deadlock:

- The simplest solution is to always lock the mutexes in the same order.
- Use individual mutex lock and unlock for any critical sections.
- The `std::lock` function can lock 2 or more mutexes at once without risk of a deadlock.

Place the locks below at the correct code section.

```
// std::lock(m_print, m_bag);  
// put the lock_guard in the correct order  
std::lock_guard<std::mutex> printGuard(m_print,  
std::adopt_lock);  
std::lock_guard<std::mutex> bagGuard(m_bag, std::adopt_lock);
```



```
Making an apple juice...  
I made an excellent apple juice for you.  
I threw out all pears from you bag!  
Bag: x x x x x x x x  
Press any key to continue . . .
```