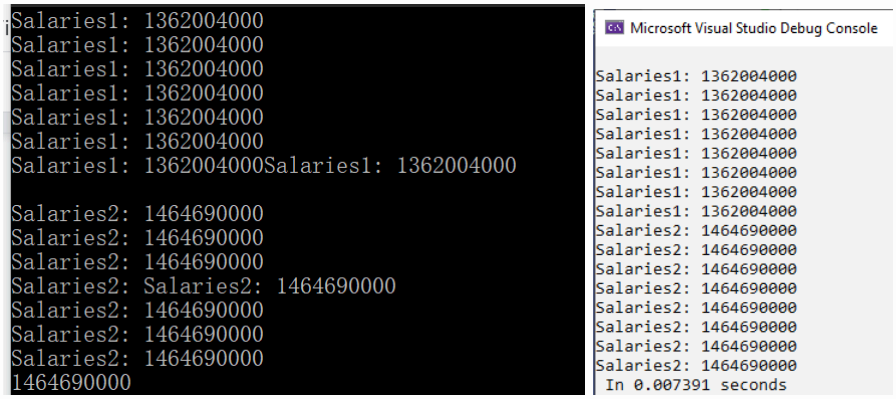
Revision

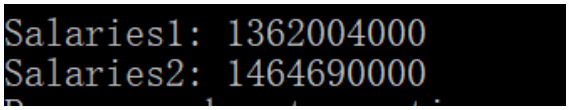
Q1 There are two companies with 10,000,000 employees. Both companies have the same owner. You would like to know the salaries of all employees of each company. The serial code can be downloaded here. Try to execute the code in parallel by adding the directives below:

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

#pragma omp for reduction(+:salaries1)

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

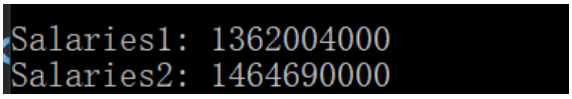


Challenge : Try to make the output to be printed properly, in line by line (right) * tips: use the *#pragma omp* command that we've learned before.

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



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

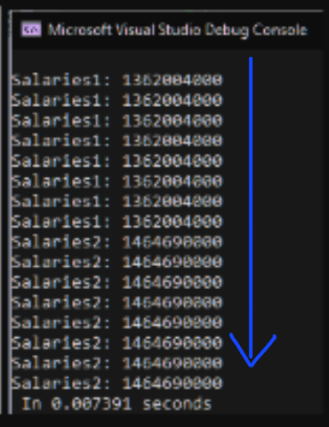


01_OMP_Atomic_For Reduction_Single.CPP

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
1464690000
```



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.

Critical

Solution

```
int company_salaries_q1()
{
    int salaries1 = 0;
    int salaries2 = 0;
    double runtime;
    runtime = omp_get_wtime();
#pragma omp parallel shared(salaries1, salaries2) num_threads(16)
    {
#pragma omp for reduction(+ : salaries1)
        for (int employee = 0; employee < MAX_EMPLOYEE; employee++)
            salaries1 += fetchTheSalary(employee, Co::Company1);

#pragma omp critical
        {
            std::cout << "Salaries1: " << salaries1 << std::endl;
            // printf("Salaries1: %d\n", salaries1); // printf is thread safe
        }

#pragma omp for reduction(+ : salaries2)
        for (int employee = 0; employee < MAX_EMPLOYEE; employee++)
            salaries2 += fetchTheSalary(employee, Co::Company2);

#pragma omp critical
        {
            std::cout << "Salaries2: " << salaries2 << std::endl;
            // printf("Salaries2: %d\n", salaries2);
        }
    }
    runtime = omp_get_wtime() - runtime;
    std::cout << " In " << runtime << " seconds" << std::endl;
    return 0;
}
```

↓ initial setup

challenge:
↓ avoid race condition

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
```

Solution

```
int company_salaries_q2()
{
    int salaries1 = 0;
    int salaries2 = 0;
    double runtime;
    runtime = omp_get_wtime();
#pragma omp parallel shared(salaries1, salaries2) num_threads(16)
    {
#pragma omp for reduction(+ : salaries1, salaries2)
        for (int employee = 0; employee < MAX_EMPLOYEE; employee++)
        {
            salaries1 += fetchTheSalary(employee, Co::Company1);
            salaries2 += fetchTheSalary(employee, Co::Company2);
        }
        std::cout << "Salaries1: " << salaries1 << std::endl;
        std::cout << "Salaries2: " << salaries2 << std::endl;
    }
    runtime = omp_get_wtime() - runtime;
    std::cout << " In " << runtime << " seconds" << std::endl;
    return 0;
}
```

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

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

Solution

```
int company_salaries_q3()
{
    int salaries1 = 0;
    int salaries2 = 0;
    double runtime;
    runtime = omp_get_wtime();
#pragma omp parallel shared(salaries1, salaries2) num_threads(16)
    {
#pragma omp for reduction(+ : salaries1, salaries2)
        for (int employee = 0; employee < MAX_EMPLOYEE; employee++)
        {
            salaries1 += fetchTheSalary(employee, Co::Company1);
            salaries2 += fetchTheSalary(employee, Co::Company2);
        }
#pragma omp single
        {
            std::cout << "Salaries1: " << salaries1 << std::endl;
            std::cout << "Salaries2: " << salaries2 << std::endl;
        }
    }
    runtime = omp_get_wtime() - runtime;
    std::cout << " In " << runtime << " seconds" << std::endl;
    return 0;
}
```

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];
}
```

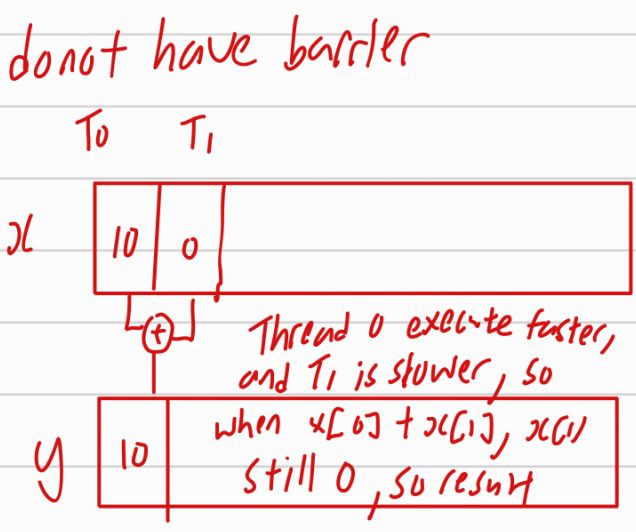
Apart from the barrier directive, which inserts an explicit barrier, OpenMP has **implicit barriers** after a load sharing construct. Thus the following code is well defined:

```
#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];
}
```

02_ Barrier.cpp

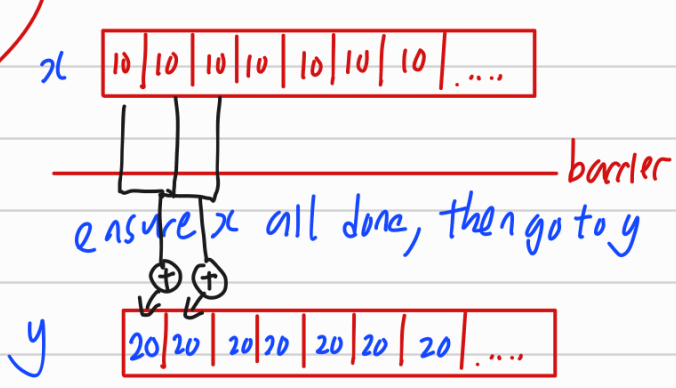
```cpp
#include <iostream>
#include <omp.h>

using namespace std;

int some_calculation()
{
    return 10;
}

const int MAX_THREADS = 16;

void barrier_exercise()
{
    int x[MAX_THREADS] = {0}, y[MAX_THREADS] = {0};

#pragma omp parallel num_threads(MAX_THREADS)
    {
        int mytid = omp_get_thread_num();
        x[mytid] = some_calculation();
#pragma omp barrier
        y[mytid] = x[mytid] + x[mytid + 1];
    }
    for (int i = 0; i < MAX_THREADS; i++)
        cout << "y[" << i << "] = " << y[i] << endl;
}

int main()
{
    barrier_exercise();
    return 0;
}
```

if donot have barrier




```
y[0] = 10
y[1] = 10
y[2] = 10
y[3] = 20
y[4] = 10
y[5] = 20
y[6] = 20
y[7] = 10
y[8] = 20
y[9] = 10
y[10] = 20
y[11] = 20
y[12] = 10
y[13] = 10
y[14] = 20
y[15] = 10
```

after use barrier




```
y[0] = 20
y[1] = 20
y[2] = 20
y[3] = 20
y[4] = 20
y[5] = 20
y[6] = 20
y[7] = 20
y[8] = 20
y[9] = 20
y[10] = 20
y[11] = 20
y[12] = 20
y[13] = 20
y[14] = 20
y[15] = 30
```

Implicit Barrier

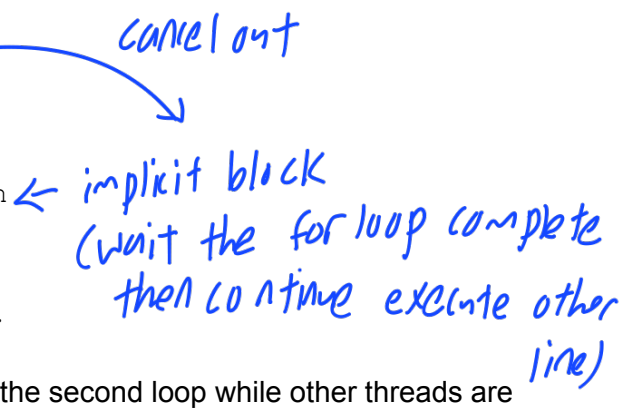
At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an implicit barrier at the end of a parallel region.

```
#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**.

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);
```

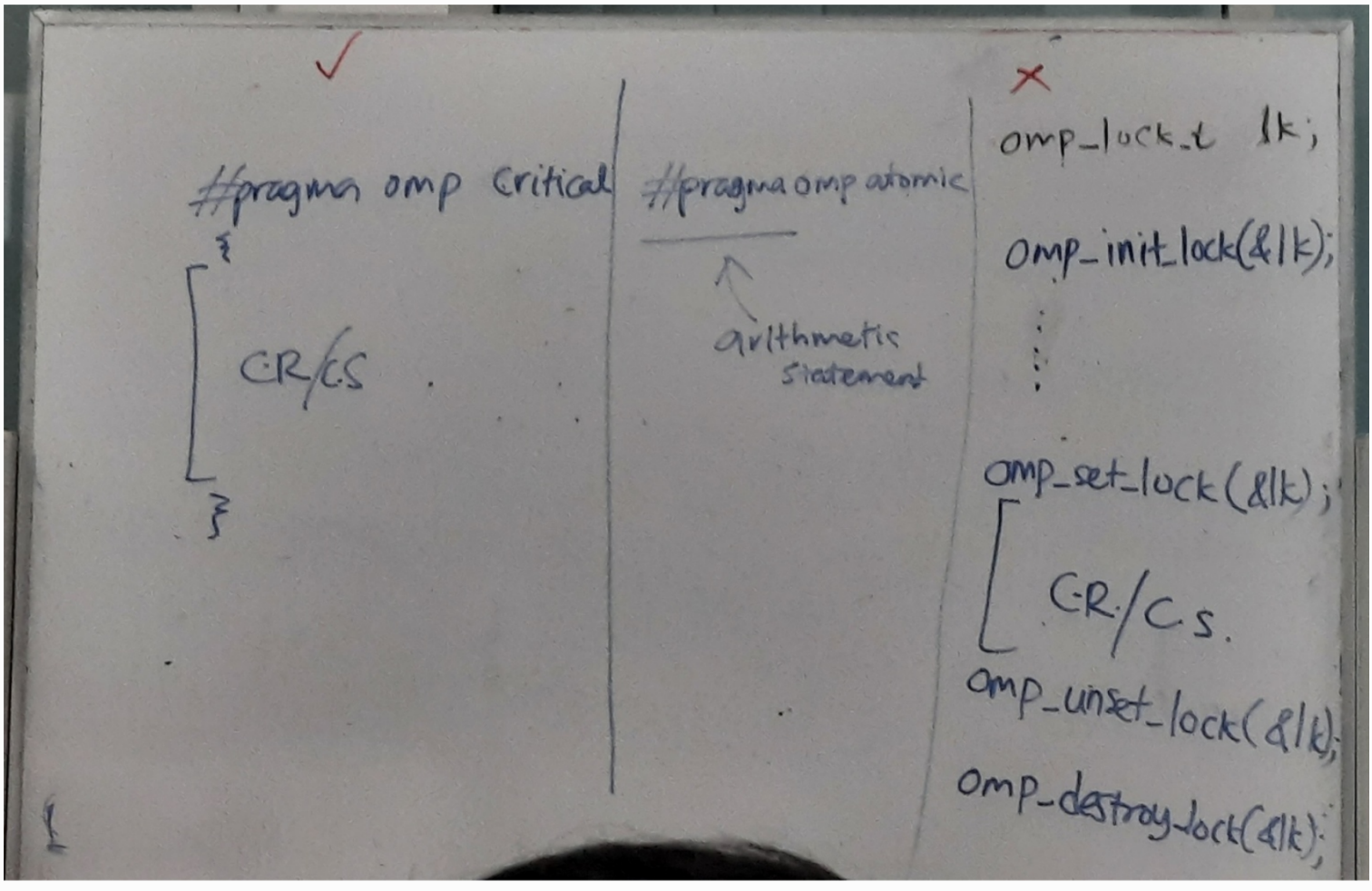
Set and release:

```
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**.



Lock

✓ flexibly can unlock and lock in a if else case

✗ difficult to use

✗ error prone

```
if()
  lock()
else()
  unlock()
```

Thread Library

↳ concurent computing

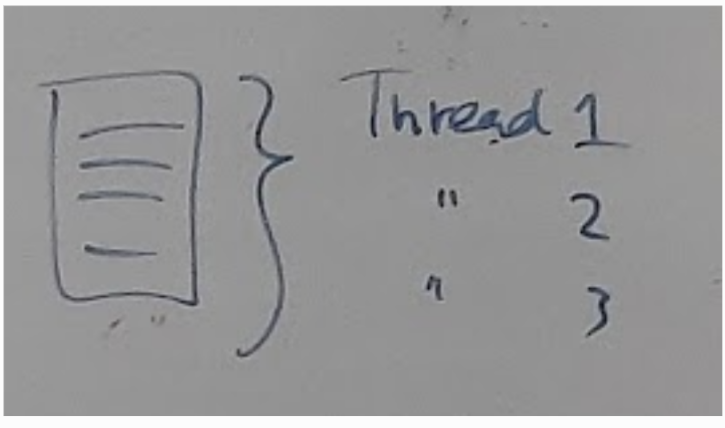
↳ no guarantee to use multiple core

↳ cannot expect performance gain

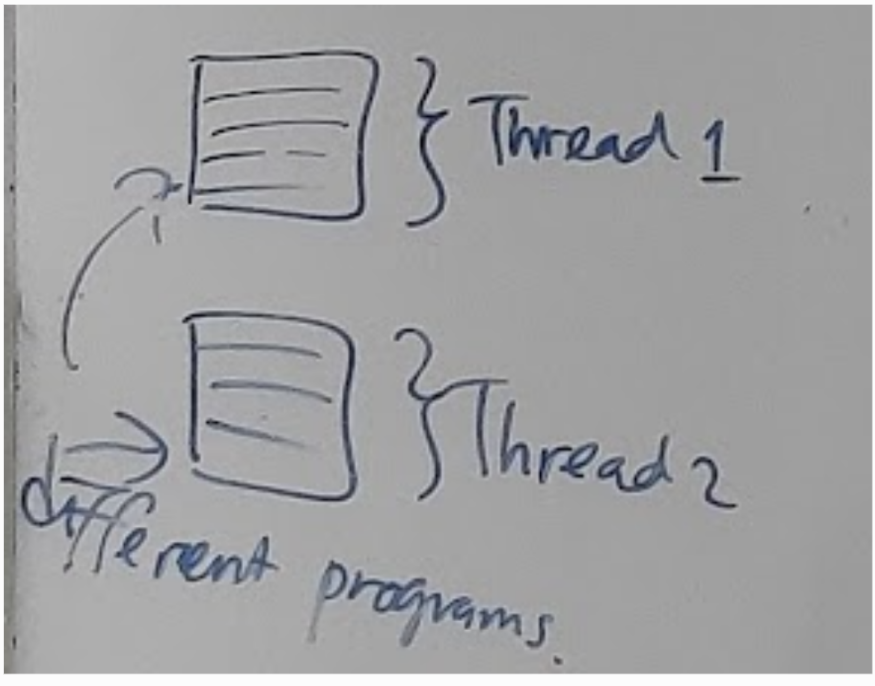
OMP

↳ parallel computing

↳ expect performance gain due to omp use multiple cores.



↑ assign different thread on same program



assign diff thread on diff program

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;
}
```

How many threads?

3 threads

a) main thread

b) main thread spawn first thread

c) main thread spawn second thread

if donot include this, after main thread end, first and second thread will get destroy at the same time when mutex is getting destroy

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/

03 - Thread Library

```cpp
// thread example
#include <iostream> // std::cout
#include <thread>   // std::thread
// #include <windows.h> // Sleep() function is only available in Window
#include <chrono> // Protable, can use in both window and linux environment

void foo()
{
    // do stuff...
    // Sleep only available in Window (Because from window.h)
    // Sleep(4000);
    std::chrono::milliseconds timespan(4000);
    std::this_thread::sleep_for(timespan);
}

void bar(int x)
{
    // do stuff...
    // Sleep(5000);
    std::chrono::milliseconds timespan(4000);
    std::this_thread::sleep_for(timespan);
}

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;
}
```

5000

output:

```
main, foo and bar now execute concurrently...
foo and bar completed.
```

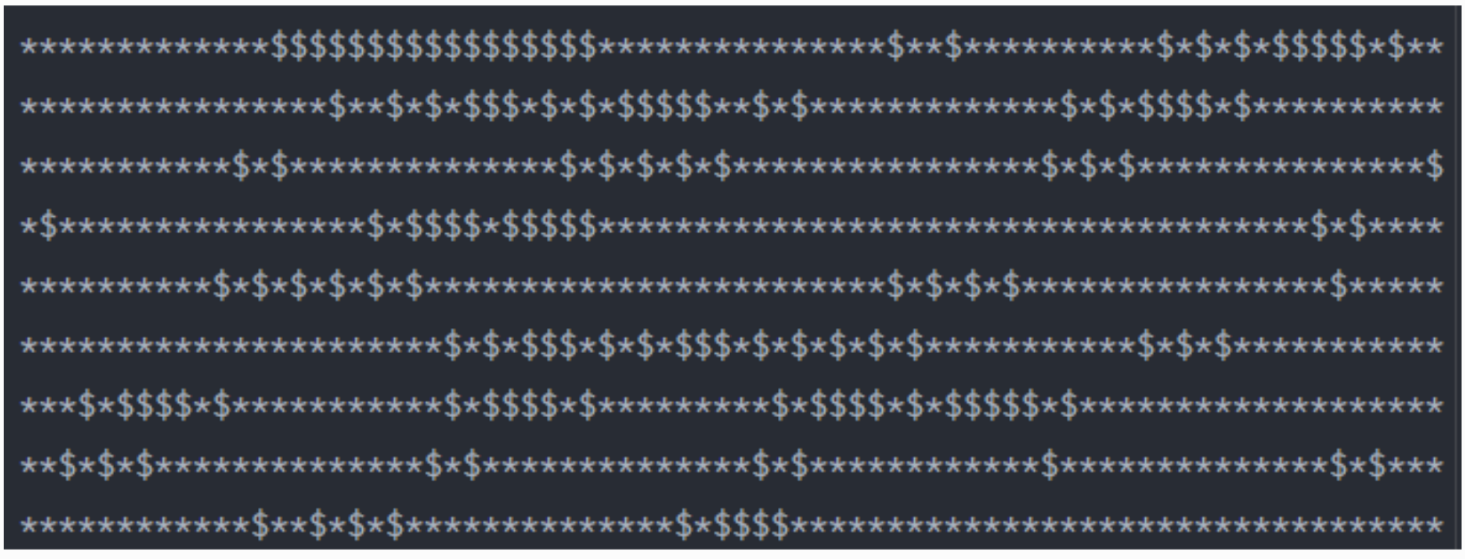
← wait 5 sec

04_Mutex.cpp

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

std::mutex mtx; // mutex for critical section

void print_block(int n, char c)
{
    // critical section (exclusive access to std::cout signaled by locking mtx):
    mtx.lock();
    for (int i = 0; i < n; ++i)
    {
        std::cout << c;
    }
    std::cout << '\n';
    mtx.unlock();
}

int main()
{
    std::thread th1(print_block, 10000, '*');
    std::thread th2(print_block, 10000, '$');

    th1.join();
    th2.join();

    return 0;
}
```

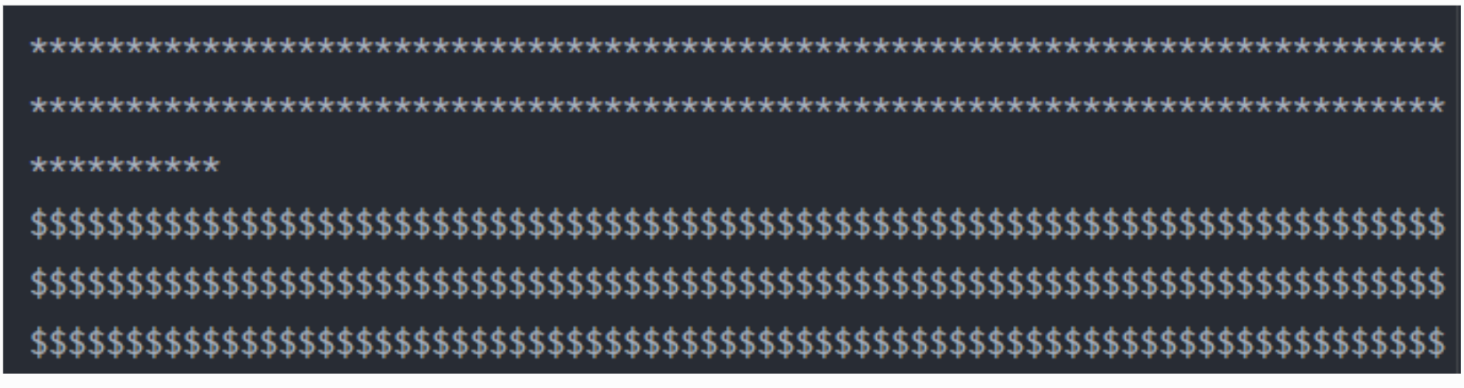
output:

without mutex:



↳race condition

with mutex



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();
```

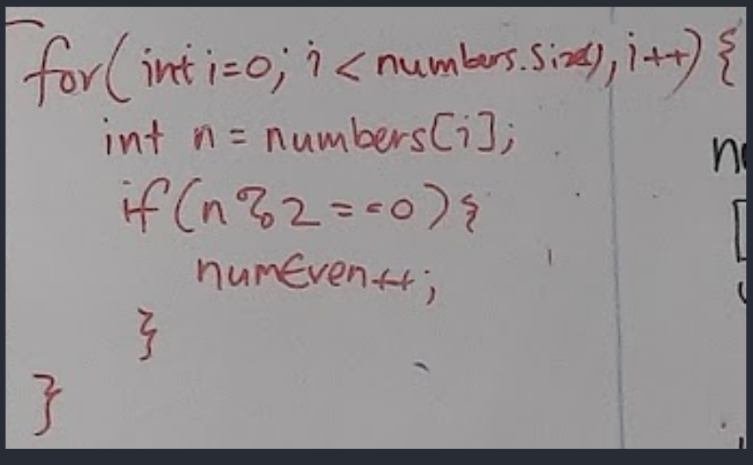
* Add

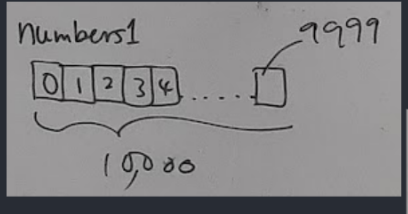
```
std::cout << "thread: " << numEven << std::endl;
```

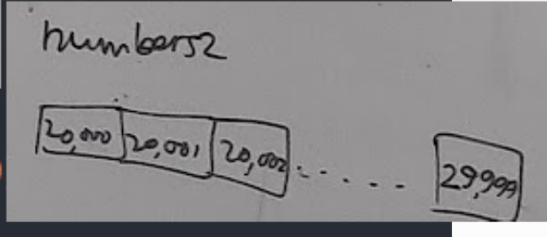
after the increment of `numEven` to see the difference of implementing / not implementing lock mutex.

os_mutex_NumEven.cpp

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

std::mutex increment;

void countEven(const std::vector<int> &numbers,
               int &numEven)
{
    increment.lock();
    for (const auto n : numbers)
    {
        if (n % 2 == 0)
        {
            numEven++;
        }
    }
    increment.unlock();
}
void insert_numbers_into_vector(std::vector<int> &v, int start_val, int len)
{
    for (int i = start_val; i < start_val + len; i++)
        v.insert(v.end(), i);
}

int main()
{
    // std::vector<int> numbers1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    // std::vector<int> numbers2 = { 11, 12, 13, 14, 15, 16, 17, 18 };
    std::vector<int> numbers1;
    std::vector<int> numbers2;

    int n = 0;

    insert_numbers_into_vector(numbers1, 0, 10000);
    insert_numbers_into_vector(numbers2, 20000, 10000

    std::thread t1(countEven, std::ref(numbers1), std::ref(n));
    std::thread t2(countEven, std::ref(numbers2), std::ref(n));

    t1.join();
    t2.join();

    std::cout << "Total even numbers is: " << n << std::endl;
    return 0;
}
```







Output.

With mutex

```
Total even numbers is: 10000
```

without mutex

```
Total even numbers is: 9914
```

↳ race condition

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/

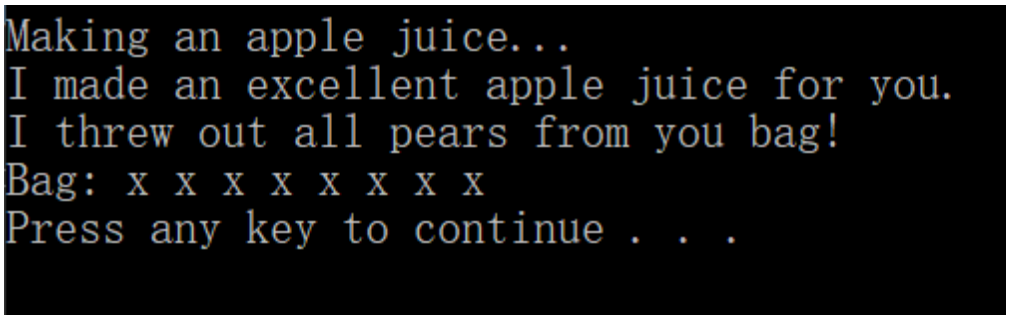
Deadlock

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);
```



```cpp
std::mutex m_print;
std::mutex m_bag;

void makeAppleJuice(std::vector<std::string> &bag)
{
    std::lock_guard<std::mutex> bagGuard(m_bag);

    std::lock_guard<std::mutex> printGuard(m_print);

    std::cout << "Making an apple juice..." << std::endl;

    // std::lock_guard<std::mutex> bagGuard(m_bag);

    int numApples = 0;
    for (std::size_t i = 0; i != bag.size(); i++)
    {
        if (bag[i] == "a")
        {
            numApples++;
            bag[i] = "x";
        }
    }

    if (numApples < 5)
    {
        std::cout << "I can not make an apple juice." << std::endl;
    }
    else
    {
        std::cout << "I made an excelent apple juice for you." << std::endl;
    }
}
```

need to follow fixed order if not it will cause deadlock bag > print

CR for printGuard

CR for bagGuard

```cpp
void throwOutPear(std::vector<std::string> &bag)
{
    // std::lock_guard<std::mutex> printGuard(m_print);

    std::lock_guard<std::mutex> bagGuard(m_bag);

    for (std::size_t i = 0; i != bag.size(); i++)
    {
        if (bag[i] == "p")
        {
            bag[i] = "x";
        }
    }

    std::lock_guard<std::mutex> printGuard(m_print);

    std::cout << "I threw out all pears from your bag!" << std::endl;
}
```

need to in fixed order if not deadlock

```cpp
void printItem(const std::string &fruit)
{
    std::cout << fruit << " ";
}
```

```cpp
int main()
{
    std::vector<std::string> bag = {"a", "a", "a", "p", "p", "a", "p", "a"};

    std::thread t1(makeAppleJuice, std::ref(bag));
    std::thread t2(throwOutPear, std::ref(bag));
    t1.join();
    t2.join();

    std::cout << "Bag: ";
    std::for_each(bag.begin(), bag.end(), printItem);
    std::cout << std::endl;

    return 0;
}
```

What makeAppleJuice function does?

```
Making an apple juice...
I made an excelent apple juice for you.
Bag: x x x p p x p x
```

numApple >5

what does throwOutPear Does

```
I threw out all pears from your bag!
Bag: a a a x x a x a
```

output if does not follow lock order:

Deadlock

```
└─ ./06_Deadlock
Making an apple juice...
```

↳ but sometime deadlock won't happen

```
Making an apple juice...
I made an excelent apple juice for you.
I threw out all pears from your bag!
Bag: x x x x x x x x
```

If follow order guarantee, no deadlock

```
Making an apple juice...
I made an excelent apple juice for you.
I threw out all pears from your bag!
Bag: x x x x x x x x
```

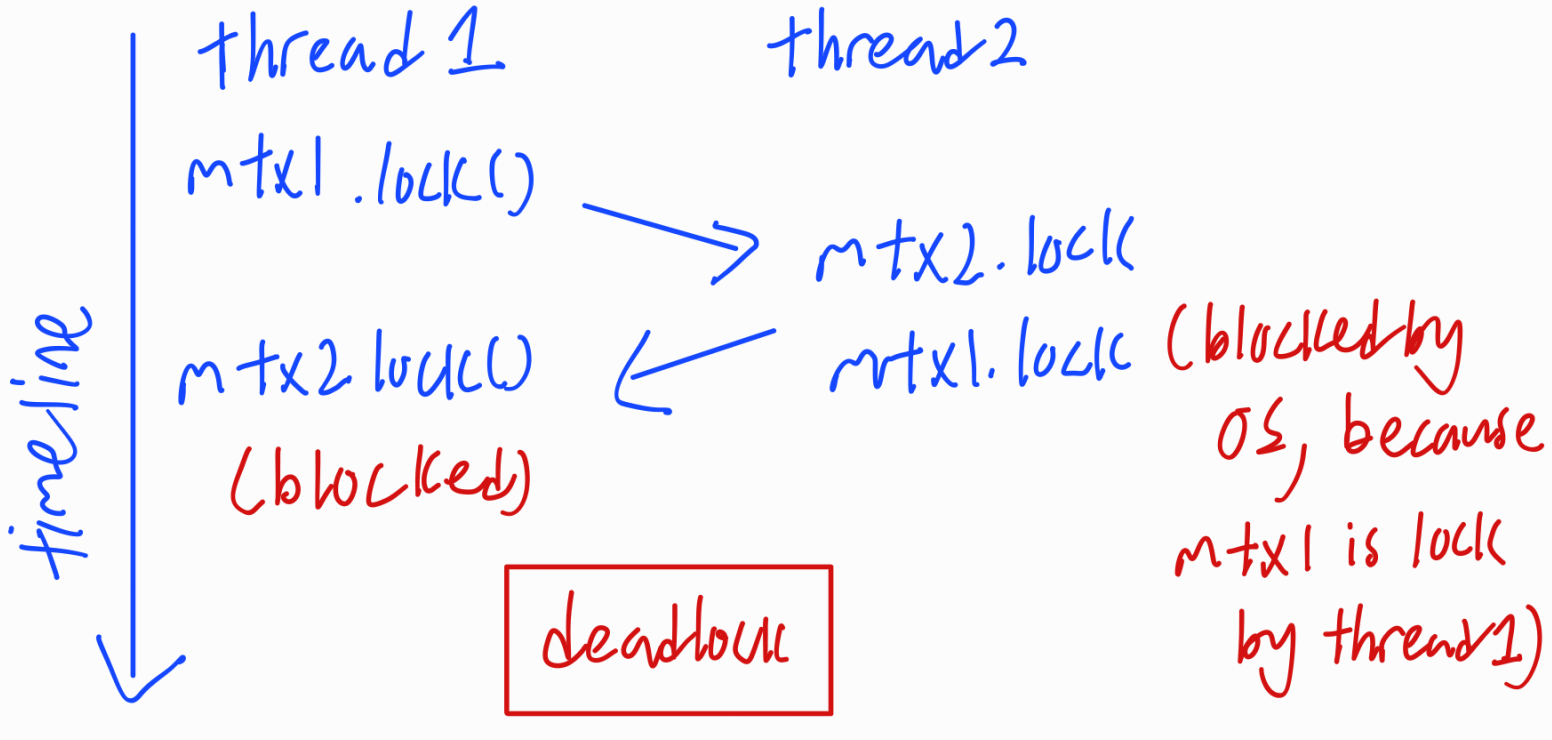
Mutex mtx1, mtx2;

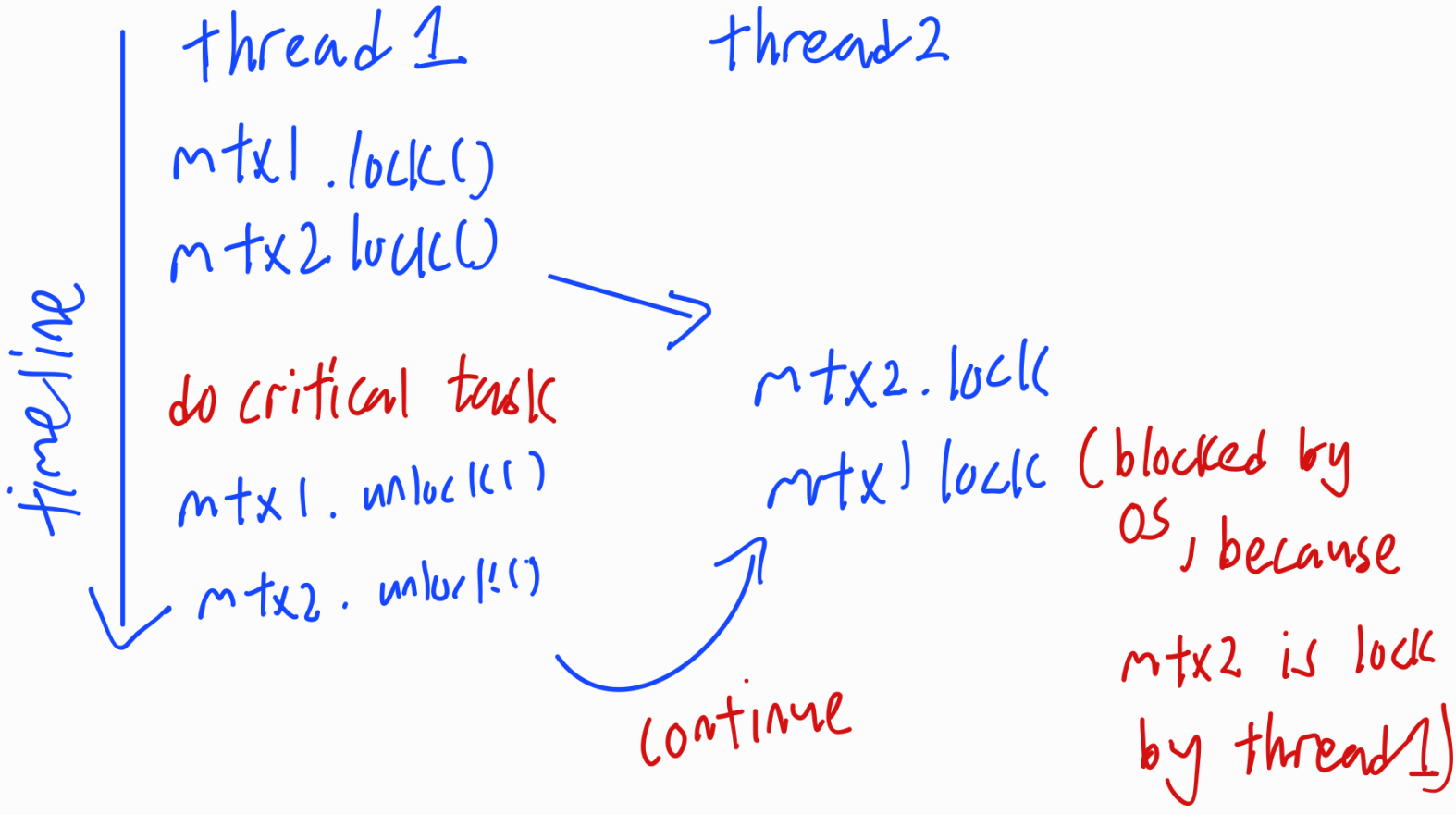
Thread 1

```
task1 {
mtx1.lock();
mtx2.lock();
do_the_critical_task1();
mtx1.unlock();
mtx2.unlock();
}
```

Thread 2

```
task2 {
mtx2.lock();
mtx1.lock();
do_the_critical_task2();
mtx1.unlock();
mtx2.unlock();
```





→ not always got deadlock

↳ got chance happen

Mutex mtx1, mtx2;

Thread 1

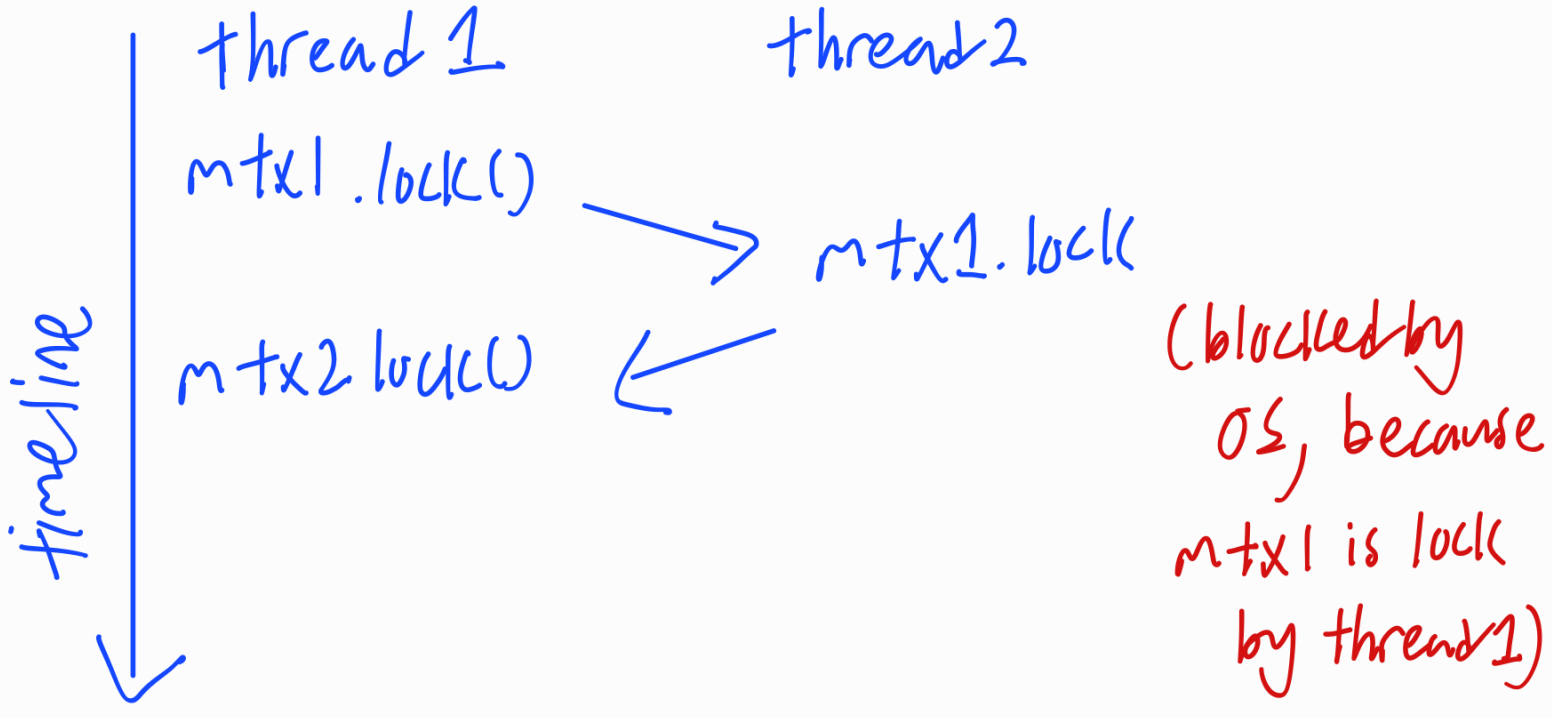
```
task1 {
mtx1.lock();
mtx2.lock();
do_the_critical_task1();
mtx1.unlock();
mtx2.unlock();
}
```

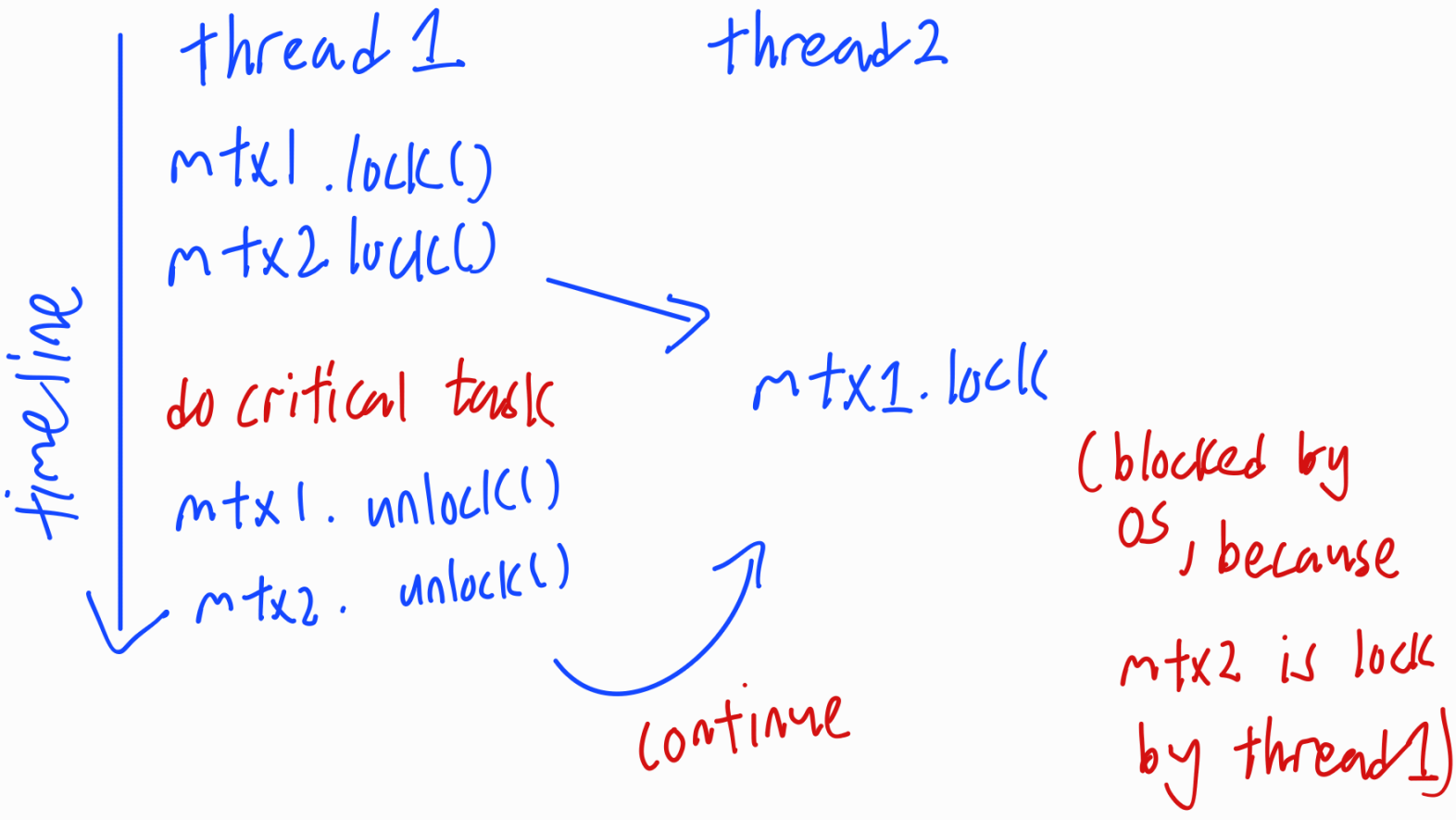
Thread 2

```
task2 {
mtx(1).lock();
mtx(2).lock();
do_the_critical_task2();
mtx1.unlock();
mtx2.unlock();
```

In order to prevent circular wait, the mutex lock need to be same order

---




thread 1
thread 2
mtx1.lock()
mtx2.lock()
do critical task
mtx1.unlock()
mtx2.unlock()
mtx1.lock
(blocked by OS, because mtx2 is lock by thread 1)
continue
timeline