1. **Part1 (Mutex and Barrier):**

   (a). Describe how to protect share resources and how to synchronize threads in your report.

code	描述

`/*~~~~~~~~~~~~Global Resuource~~~~~~~~~~~*/`
`/*            Your code(Part1~3)          */`
`float sharedSum = 0;`
`pthread_mutex_t* ioMutex = new pthread_mutex_t;`
`pthread_mutex_t* count_mutex = new pthread_mutex_t;`
`pthread_barrier_t *barr = new pthread_barrier_t;`
`pthread_spinlock_t *lock = new pthread_spinlock_t;`
 	
`/* Global Resource */`
`extern float sharedSum;`
`extern pthread_mutex_t* ioMutex;`
`extern pthread_mutex_t *count_mutex;`
`extern pthread_spinlock_t *lock;`
`extern pthread_barrier_t *barr;`
`extern sem_t *sem;`
	首先，在 system.cpp 和 thread.h 的 global Resourse 都建立 mutex 和 barr

`/*~~~~~~~~~~~~~Your code(Part1~3)~~~~~~~~~~~~~*/`
`// Initial the resources (barrier, semaphore) //`
`pthread_barrier_init ( barr, NULL, THREAD_NUM );`
	在 system.cpp 的 Initial the resources 地方加入初始化 barrier 的函式，函式的最前面放入變數名稱，中間應該是 barrier 的屬性，最後則是數的次數。

```
void
Thread::enterCriticalSection ()
{
#if _ProtectType == MUTEX
    /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~~*/
  // Implement your mutex ()
  // ...
  // ...
    pthread_mutex_lock ( count_mutex );

Thread::exitCriticalSection ()
{
#if _ProtectType == MUTEX
    /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~~*/
  // Implement your mutex ()
  // ...
  // ...
    pthread_mutex_unlock ( count_mutex );
    /*~~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~~*/

void
Thread::synchronize ()
{
#if _SynType == BARRIER
  /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~~*/
  // Implement your barrier
  pthread_barrier_wait (barr);

    /*~~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~~*/
```

在 thread.cpp 的 enterCriticalSection ()放入上鎖的函式 pthread_mutex_lock，並在下面的 exitCriticalSection ()放入解鎖的函式 pthread_mutex_unlock，最後，在 synchronize () 內放入 pthread_barrier_wait 函式，這個函式在 thread 時會數一次，並且在數到初始化時給的值之前，thread 不能繼續往下執行，因此能夠讓先執行完的 thread 等待其他 thread 都處理好。

```
#if (PART != 2)
        obj->enterCriticalSection();
        sharedSum = 0;

        for (int k = -shift; k <= shift; k++) {
          for (int l = -shift; l <= shift; l++) {
            if ( i + k < 0 ||  i + k >= MATRIX_SIZE || j + l < 0 ||  j + l >= MATRIX_SIZE)
              continue;
            sharedSum += obj->matrix [i + k][j + l] * obj->mask [k + shift][l + shift];

          } // for (int l...

        } // for (int k...

        obj->multiResult [i][j] = sharedSum;
        obj->exitCriticalSection();

#else
```

由於這邊的 sharedSum 是 global 的變數，因此必須受到保護，在用到 sharedSum 前放上鎖的函式，並在最後 sharedSum 的值被儲存後才解鎖。

```
#if (CONVOLUTION_TIMES > 1)
    /*~~~~~~~~~~~~~~~~~~HINT~~~~~~~~~~~~~~~~~~*/
    // We use the previous convolution re- //
    // -sult to be the next round's input. //
    // So here we copy the result to input //
    /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
    obj->synchronize();
    for (int i = obj->startCalculatePoint; i < obj->endCalculatePoint; i++)
        memcpy (obj->matrix [i], obj->multiResult [i], MATRIX_SIZE * sizeof (float));

    obj->synchronize();

  } // for (int round...
#endif
```

在這邊第一個 synchronize ()，是為了避免有 thread()還沒算完，而 mask 若有覆蓋到已經先被 copy 進去值的話，會導致用到已經 convolution 的值，造成錯誤，因此放在這邊讓全部的 thread 都做完再

	進行複製，而第二個 synchronize ()則是為了避免執行較快的 thread 在其他 thread 還沒完成複製前就計算 convolution，造成錯誤。

(b). Show the execution result in Figure 2 in your report.

```
======================= System Info =======================
numThread: 4
maskSize: 31 x 31
matrixSize: 1000 x 1000
Protect Shared Resource: Mutex
Synchronize: Barrier

=================== Generate Matrix Data ==================
Generate Date Spend time : 0.008636

============= Start Single Thread Convolution =============
Single Thread Spend time : 15.1229

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 94        Core : 0
Thread ID : 3   PID : 97        Core : 3
Thread ID : 2   PID : 96        Core : 2
Thread ID : 1   PID : 95        Core : 1
Multi Thread Spend time : 14.1175

======================== checking =========================
Matrix convolution result correct.
```

2. Part2 (Reentrant Function):

(a). Describe how to modify the non-reentrant function to the reentrant function in your report

code	描述

```
#else

    /*~~~~~~~~~~~~Your code(Part2)~~~~~~~~~~~*/
    // Turn this non-reentrant function into //
    // reentrant function which means no re- //
    // sources contention issue happen.      //
    float sharedSum2 = 0;

    for (int k = -shift; k <= shift; k++) {
      for (int l = -shift; l <= shift; l++) {
        if ( i + k < 0 ||  i + k >= MATRIX_SIZE || j + l < 0 ||  j + l >= MATRIX_SIZE)
          continue;
        sharedSum2 += obj->matrix [i + k][j + l] * obj->mask [k + shift][l + shift];

      } // for (int l...

    } // for (int k...

    obj->multiResult [i][j] = sharedSum2;
    /*~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/

#endif
```

在 Part1 的時候，因為 sharedSum 是全域變數，導致 code 變成是 non-reentrant，因此把 sharedSum 改成不是全域變數的 sharedSum2 就可以讓程式變成 reentrant 了。

(b). Show the execution result of multi-thread with reentrant function as Figure 3.

```
======================= System Info =======================
numThread: 4
maskSize: 31 x 31
matrixSize: 1000 x 1000
Protect Shared Resource: Mutex
Synchronize: Barrier

=================== Generate Matrix Data ==================
Generate Date Spend time : 0.008454

============= Start Single Thread Convolution =============
Single Thread Spend time : 14.7708

============== Start Multi-Thread Convolution =============
Thread ID : 0    PID : 121       Core : 0
Thread ID : 1    PID : 122       Core : 1
Thread ID : 2    PID : 123       Core : 2
Thread ID : 3    PID : 124       Core : 3
Multi Thread Spend time : 3.08202

======================== checking =========================
Matrix convolution result correct.
```

(c). Analyze the execution time of the non-reentrant function and reentrant function, and compare them. (Please use your experimental result to support your discussion)

	non-reentrant	reentrant
time	14.1175	3.08202
Analyze	從實驗結果中，reentrant 的結果比 non-reentrant 還快上許多，這是因為 non-reentrant 在遇到 mutex 時，只有一個 thread 能夠計算 convolution，而 reentrant 則是每個 thread 都可	

	以進行計算，而 non-reentrant 所花費的時間差不多也是 reentrant 所花的 4 倍左右。

3. Part3 (Spinlock):

(a). Describe how to protect the shared resource by using spinlock.

<table>
<tr><th>code</th><th>描述</th></tr>
<tr><td><pre>
/*~~~~~~~~~~~~Global Resuource~~~~~~~~~~~*/
/*   Your code(Part1~3)   */
float sharedSum = 0;
pthread_mutex_t* ioMutex = new pthread_mutex_t;
pthread_mutex_t* count_mutex = new pthread_mutex_t;
pthread_barrier_t *barr = new pthread_barrier_t;
pthread_spinlock_t *lock = new pthread_spinlock_t;

/* Global Resource */
extern float sharedSum;
extern pthread_mutex_t* ioMutex;
extern pthread_mutex_t *count_mutex;
extern pthread_spinlock_t *lock;
extern pthread_barrier_t *barr;
extern sem_t *sem;
</pre></td><td>首先，在 system.cpp 和 thread.h 的 global Resourse 建立 spinlock</td></tr>
<tr><td><pre>
/*~~~~~~~~~~~~~~Your code(Part1~3)~~~~~~~~~~~~~*/
// Initial the resources (barrier, semaphore) //
pthread_barrier_init (barr, NULL, THREAD_NUM);
pthread_spin_init (lock, PTHREAD_PROCESS_PRIVATE);
</pre></td><td>在 system.cpp 內進行初始化，函式的前面代表變數名稱，後面則是屬性。</td></tr>
<tr><td><pre>
/*~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~*/
// Implement your spinlock
// ...
// ...
   pthread_spin_unlock (lock);
   /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/

  /*~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~*/
  // Implement your spinlock
  // ...
  // ...
     pthread_spin_unlock (lock);
     /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#endif
</pre></td><td>在 thread.cpp 的 enterCriticalSection ()放入上鎖的函式 pthread_mutex_lock，並在下面的 exitCriticalSection ()放入解鎖的函式 pthread_mutex_unlock。</td></tr>
</table>

```
#if (PART != 2)
        obj->enterCriticalSection();
        sharedSum = 0;

        for (int k = -shift; k <= shift; k++) {
          for (int l = -shift; l <= shift; l++) {
            if ( i + k < 0 ||  i + k >= MATRIX_SIZE || j + l < 0 ||  j + l >= MATRIX_SIZE)
              continue;
            sharedSum += obj->matrix [i + k][j + l] * obj->mask [k + shift][l + shift];

          } // for (int l...

        } // for (int k...

        obj->multiResult [i][j] = sharedSum;
        obj->exitCriticalSection();

#else
```

跟 Part1 一樣，保需要被保護的全域變數 sharedSum 前放鎖住資源的函式，在最後儲存 sharedSum 的結果後放入解鎖資源的函式。

(b). Show the execution result in Figure 4

```
======================= System Info =======================
numThread: 4
maskSize: 31 x 31
matrixSize: 1000 x 1000
Protect Shared Resource: Spinlock
Synchronize: Barrier

=================== Generate Matrix Data ==================
Generate Date Spend time : 0.011452

============= Start Single Thread Convolution =============
Single Thread Spend time : 15.211

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 148       Core : 0
Thread ID : 1   PID : 149       Core : 1
Thread ID : 2   PID : 150       Core : 2
Thread ID : 3   PID : 151       Core : 3
Multi Thread Spend time : 12.2374

======================== checking =========================
Matrix convolution result correct.
```

(c). Compare to part 1, please observe which method could obtain better performance under the benchmark we provided and explain why. Please use the execution results to support your discussion. (Show both the execution results of using mutex and spinlock respectively to support your discussion.)

	mutex	spinlock
time	14.1175	12.374
Analyze	從實驗結果中發現，spinlock 比 mutex 還要快了一點，spinlock 跟 mutex 最大的差異在於 spinlock 在等待時是使用 busy-waiting，也就是不會去進行其他事情的處理，就一直在鎖資源	

	的地方等待，而 mutex 是使用 sleep-waiting，在等待時會先去做其他事情，等資源解鎖後再回來處理，因此在這裡 spinlock 會快一點可能是因為 convolution 在這個配置時的處理速度很快，使用 busy-waiting 的效益更好。

(d). Following (c), please modify the configuration of the benchmark (maskSize and matrixSize) such that the performance results are opposite to the results of (c). Show the configuration of our benchmark by the screenshot of a file (config.h) and describe the property of the configuration. (Show both the execution results of using mutex and spinlock respectively to support your discussion.)

Mutex:

```
======================= System Info =======================
numThread: 4
maskSize: 101 x 101
matrixSize: 5000 x 5000
Protect Shared Resource: Mutex
Synchronize: Barrier

=================== Generate Matrix Data ==================
Generate Date Spend time : 0.259716

============= Start Single Thread Convolution =============
Single Thread Spend time : 4056.82

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 276       Core : 0
Thread ID : 1   PID : 277       Core : 1
Thread ID : 2   PID : 278       Core : 2
Thread ID : 3   PID : 279       Core : 3
Multi Thread Spend time : 3064.65

======================== checking =========================
Matrix convolution result correct.
```

Spinlock

```
======================= System Info =======================
numThread: 4
maskSize: 101 x 101
matrixSize: 5000 x 5000
Protect Shared Resource: Spinlock
Synchronize: Barrier

=================== Generate Matrix Data ==================
Generate Date Spend time : 0.262388

============= Start Single Thread Convolution =============
Single Thread Spend time : 4058.15

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 303       Core : 0
Thread ID : 1   PID : 304       Core : 1
Thread ID : 2   PID : 305       Core : 2
Thread ID : 3   PID : 306       Core : 3
Multi Thread Spend time : 3169.17

======================== checking =========================
Matrix convolution result correct.
```

<table>
<tr><th></th><th>mutex</th><th>spinlock</th></tr>
<tr><td>configuration</td><td colspan="2">// Workload parameter
#define MASK_SIZE 101
#define MATRIX_SIZE 5000
#define CONVOLUTION_TIMES 3

矩陣 size 設為 5000,mask 大小設為 101</td></tr>
<tr><td>Results</td><td>3064.65</td><td>3169.17</td></tr>
<tr><td>Analyze</td><td colspan="2">根據(c)的分析，mutex 和 spinlock 的差異在於等待的模式，因此將矩陣和 mask 的大小都提高，讓計算量更多，BUSY-Waiting 的效益就會小於 SLEEP-Waiting，達到 mutex 時間小於 spinlock 的結果。</td></tr>
</table>