嵌入式系統軟體設計

Embedded System Software Design

PA2

指導教授: 陳雅淑 教授

學生: M11007308 吳柏翰

Part 1 (Mutex and Barrier):

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

```
/*~~~~~~~~~~~~~~Global Resuource~~~~~~~~~~~~~*/
/*            Your code(Part1~3)             */
float sharedSum = 0;
pthread_mutex_t* ioMutex = new pthread_mutex_t;
pthread_barrier_t* barr = new pthread_barrier_t;
pthread_spinlock_t* spinlock = new pthread_spinlock_t;
/*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
```

首先要創建 Barrier 物件。

```
/*~~~~~~~~~~~~~~~Your code(Part1~3)~~~~~~~~~~~~~~~*/
// Initial the resources (barrier, semaphore) //
if (pthread_barrier_init(barr,NULL,THREAD_NUM) != 0){
    std::cout << "Barrier init has failed." << std::endl;
    return;
}
if (pthread_spin_init(spinlock,0) != 0){
    std::cout << "Spinlock init has failed." << std::endl;
    return;
}
/*~~~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~~~*/
```

初始化 Barrier 設定，因屬性(第二個參數)不需要額外設定，因此設為 NULL 即可，另外 Barrier 需要等待所有執行緒都到齊，所以第三個參數設定成執行緒數量。

```
#if (PART != 2)
        obj->enterCriticalSection(); // Protecting the resource by mutex or spinlock.
        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
```

需要保護的共用資源是 shardSum，因此在此區塊前後加上 CriticalSection 副程式，此副程式內有設定要使用 Mutex 或 Spinlock 保護變數，而此時是使用 Mutex。

```
void
Thread::enterCriticalSection ()
{
#if _ProtectType == MUTEX
  /*~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~*/
  // Implement your mutex ()
  pthread_mutex_lock(ioMutex);
  /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#else
  /*~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~~*/
  // Implement your spinlock
  pthread_spin_lock(spinlock);
  /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#endif
}
```

進入 CriticalSection 時，將 ioMutex 鎖起來，防止其他執行緒對共用資源(shardSum)進行讀寫。

```
void
Thread::exitCriticalSection ()
{
#if _ProtectType == MUTEX
  /*~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~*/
  // Implement your mutex ()
  pthread_mutex_unlock(ioMutex);
  /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#else
  /*~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~~*/
  // Implement your spinlock
  pthread_spin_unlock(spinlock);
  /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#endif
}
```

當結束運算，不再需要保護共用資源時，可將 ioMutex 解鎖，這樣其他執行緒對共用資源就可進行讀寫。

```
#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 //
    /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
    // std::cout << obj->ID << ": is waiting" << std::endl;
    obj->synchronize();     // Synchronize by barrier.
    // std::cout << obj->ID << ": is passed" << std::endl;

    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
```

再來是用 Barrier 進行平行處理，synchronize 副程式內有設定 Barrier。

```
void
Thread::synchronize ()
{
#if _SynType == BARRIER
  /*~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~*/
  // Implement your barrier
  pthread_barrier_wait(barr);
  /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#else
  pthread_mutex_lock (ioMutex);
    std::cout << "Synchronize method not supported." << std::endl;
  pthread_mutex_unlock (ioMutex);
#endif
}
```

Barrier 的意思是屏障，程序中的同步屏障意味著任何執行緒/進程執行到此後就必須等待，直到所有執行緒都到達此點才可繼續執行下去。利用上圖標記的那一行程式即可執行 Barrier，等待所有執行緒到齊。

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

```
brian@brian-pc:~/Downloads/ESSD_PA2$ ./part1.out

======================= 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 : 0.009055

============= Start Single Thread Convolution =============
Single Thread Spend time : 17.504

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 145227    Core : 0
Thread ID : 1   PID : 145228    Core : 1
Thread ID : 2   PID : 145229    Core : 2
Thread ID : 3   PID : 145230    Core : 3
Multi Thread Spend time : 14.1544

======================== checking =========================
Matrix convolution result correct.
brian@brian-pc:~/Downloads/ESSD_PA2$
```

Part 2 (Reentrant Function):

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

```
//=====Turn into reentrant function by using local variable.=====//
float localSum = 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;
    localSum += obj->matrix [i + k][j + l] * obj->mask [k + shift][l + shift];
  } // for (int l...
} // for (int k...
obj->multiResult [i][j] = localSum;
```

不使用全域變數，而是改用區域變數，即可將 Non-reentrant function 變成 Reentrant function，如果當 A 程序在中途運算時發生中斷，執行其他程序 B，當 B 程序結束後回到 A 程序，共用資源也不會因此改變。

```
//=====To make the function reentrant by returning dynamically allocated data.=====//
float *num ;
num = (float*)malloc(MATRIX_SIZE*sizeof(float));

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;
    *num += obj->matrix [i + k][j + l] * obj->mask [k + shift][l + shift];
  } // for (int l...
} // for (int k...
obj->multiResult [i][j] = *num;
```

或是使用動態陣列來儲存計算結果，每次 Iteration 中都會分配新的記憶體，這樣就不會有改動到共用資源的問題，但這樣會使用較多的記憶體，計算時間也較長。其他改成 Reentrant function 的方法還有 Mutex、Semaphore、關閉中斷或排程等。

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

```
brian@brian-pc:~/Downloads/ESSD_PA2$ ./part2.out

====================== 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 : 0.00811

============= Start Single Thread Convolution =============
Single Thread Spend time : 16.1673

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 145815    Core : 0
Thread ID : 3   PID : 145818    Core : 3
Thread ID : 2   PID : 145817    Core : 2
Thread ID : 1   PID : 145816    Core : 1
Multi Thread Spend time : 2.79786

======================== checking ========================
Matrix convolution result correct.
brian@brian-pc:~/Downloads/ESSD_PA2$
```

(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)

Part1.(b)與 Part2.(b)的 Multi-thread 結果做比較，以下是執行時間的比較。

	Non-reentrant	**Reentrant**
Time (sec)	14.154	2.798

從上表可以發現 Non-reentrant 的執行時間約是 Reentrant 的 5 倍，Reentrant 的效能比 Non-reentrant 好，原因是 Non-reentrant function 會用到 global variables 或 static variable，而在不同的執行緒之間需要用 Mutex 或 Spinlock 保護變數，以免更動到共用資源，這樣導致 Mutex 會鎖死當下的 thread，等到做完之後，才會讓其他 thread 動作，而 Reentrant function 沒有這樣的問題，因此執行時間比 Reentrant function 長。

Part 3 (Spinlock):

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

```
/*~~~~~~~~~~~~~Global Resuource~~~~~~~~~~~*/
/*            Your code(Part1~3)           */
float sharedSum = 0;
pthread_mutex_t* ioMutex = new pthread_mutex_t;
pthread_barrier_t* barr = new pthread_barrier_t;
pthread_spinlock_t* spinlock = new pthread_spinlock_t;
/*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
```

首先要創建 Spinlock 物件。

```
/*~~~~~~~~~~~~~~Your code(Part1~3)~~~~~~~~~~~~~~*/
// Initial the resources (barrier, semaphore) //
if (pthread_barrier_init(barr,NULL,THREAD_NUM) != 0){
    std::cout << "Barrier init has failed." << std::endl;
    return;
}
if (pthread_spin_init(spinlock,0) != 0){
    std::cout << "Spinlock init has failed." << std::endl;
    return;
}
/*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
```

初始化 Spinlock 設定，第二個參數 pshared 設定成 0，單執行緒可以設置成 PTHREAD_PROCESS_SHARED。

```
#if (PART != 2)
        obj->enterCriticalSection(); // Protecting the resource by mutex or spinlock.
        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
```

需要保護的共用資源是 shardSum，因此在此區塊前後加上 CriticalSection 副程式，此副程式內有設定要使用 Mutex 或 Spinlock 保護變數，而此時是使用 Spinlock。

```
void
Thread::enterCriticalSection ()
{
#if _ProtectType == MUTEX
  /*~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~*/
  // Implement your mutex ()
  pthread_mutex_lock(ioMutex);
  /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#else
  /*~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~*/
  // Implement your spinlock
  pthread_spin_lock(spinlock);
  /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#endif
}
```

進入 Critical section 時，Spinlock 防止其他執行緒對共用資源(shardSum)進行讀寫。Spinlock 的中文稱做自旋鎖，功能與 Mutex 相同，Spinlock 可以用來保護 Critical section，如果執行緒沒有獲取鎖，則會進入迴圈直到獲得上鎖的資格，因此叫做自旋鎖。

```
void
Thread::exitCriticalSection ()
{
#if _ProtectType == MUTEX
  /*~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~*/
  // Implement your mutex ()
  pthread_mutex_unlock(ioMutex);
  /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#else
  /*~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~*/
  // Implement your spinlock
  pthread_spin_unlock(spinlock);
  /*~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~*/
#endif
}
```

當結束運算，不再需要保護共用資源時，Spinlock 解鎖，這樣其他執行緒對共用資源就可進行讀寫。

(b) Show the execution result in Figure 4.

```
brian@brian-pc:~/Downloads/ESSD_PA2$ ./part3.out

====================== 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 : 0.008721

============= Start Single Thread Convolution =============
Single Thread Spend time : 17.4417

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 146009    Core : 0
Thread ID : 1   PID : 146010    Core : 1
Thread ID : 3   PID : 146012    Core : 3
Thread ID : 2   PID : 146011    Core : 2
Multi Thread Spend time : 11.0936

======================== checking ========================
Matrix convolution result correct.
brian@brian-pc:~/Downloads/ESSD_PA2$
```

(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.)

Part1.(b)和 Part3.(b)的 Multi-thread 結果做比較，以下是 Mutex 與 Spinlock 的執行時間之比較。

	Mutex	**Spinlock**
Time (sec)	14.154	11.094

從上表可以發現 Mutex 執行時間較長，Spinlock 的效能優於 Mutex，原因是在遇到 Critical section 時，Mutex 會先切換到其他 task 去執行，這樣會增加 Context switch overhead，而 Spinlock 會採用 Busy waiting 方式，會一直停留在同個 task 直到解鎖，在繼續執行下去。結論是 Spinlock 適用於保護單一全域變數，而 Mutex 比較適合保護一段程式區塊，以此作業而言，算是保護單一全域變數，因此 Spinlock 的效能表現較佳。

(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 your 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.)

```
src > h config.h > ...
1  #ifndef _CONFIG_H_
2  #define _CONFIG_H_
3
4  #include <sched.h>
5
6  #define PART 3
7
8  // Hardware dependency parameter
9  #define CORE_NUM 4
10 #define THREAD_NUM 4
11
12 // Workload parameter
13 #define MASK_SIZE 3
14 #define MATRIX_SIZE 64
15 #define CONVOLUTION_TIMES 3
16
17 // Protecte shared resource setting
18 #define MUTEX 0
19 #define SPINLOCK 1
20
21 #define _ProtectType SPINLOCK
22
23 // Synchronize method setting
24 #define BARRIER 0
25 #define SEMAPHORE 1
26
27 #define _SynType BARRIER
28
29
30 #endif
```

當 MASK_SIZE 與 MATRIX_SIZE 分別調整成 3 和 64 時，Mutex 的執行時間小於 Spinlock 的時間。

```
brian@brian-pc:~/Downloads/ESSD_PA2$ ./part1.out

======================== System Info ========================
numThread: 4
maskSize: 3 x 3
matrixSize: 64 x 64
Protect Shared Resource: Mutex
Synchronize: Barrier

=================== Generate Matrix Data ===================
Generate Date Spend time : 0.000197

============= Start Single Thread Convolution =============
Single Thread Spend time : 0.000802

============== Start Multi-Thread Convolution =============
Thread ID : 2   PID : 153084    Core : 2
Thread ID : 0   PID : 153082    Core : 0
Thread ID : 1   PID : 153083    Core : 1
Thread ID : 3   PID : 153085    Core : 3
Multi Thread Spend time : 0.002354

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

```
brian@brian-pc:~/Downloads/ESSD_PA2$ ./part3.out

======================== System Info ========================
numThread: 4
maskSize: 3 x 3
matrixSize: 64 x 64
Protect Shared Resource: Spinlock
Synchronize: Barrier

=================== Generate Matrix Data ===================
Generate Date Spend time : 0.000198

============= Start Single Thread Convolution =============
Single Thread Spend time : 0.004018

============== Start Multi-Thread Convolution =============
Thread ID : 0   PID : 153140    Core : 0
Thread ID : 1   PID : 153141    Core : 1
Thread ID : 2   PID : 153142    Core : 2
Thread ID : 3   PID : 153143    Core : 3
Multi Thread Spend time : 0.006226

======================== checking ========================
Matrix convolution result correct.
brian@brian-pc:~/Downloads/ESSD_PA2$
```

經過反覆的測試，下表是調整參數的數據。

	Mutex	Spinlock
Mask=31,Matrix=1000(原始)	14.154	11.094
Mask=31,Matrix=500	3.602	2.749
Mask=15,Matrix=1000	4.412	2.712
Mask=61,Matrix=1000	47.032	43.718
Mask=61,Matrix=1500	106.319	98.646

Mask=121,Matrix=1000	171.318	167.694
Mask=121,Matrix=500	41.379	40.077
Mask=15,Matrix=250	0.264	0.167
Mask=7,Matrix=125	0.0253	0.0119
Mask=3,Matrix=64	0.00235	0.00623

從上表可發現，當 Mask 與 Matrix 越大，運算量越大，花費的時間更長，而 Mutex 的執行時間都是比 Spinlock 來的大，原因是 Mutex 的 Context switch overhead 會越大，因此若要 Mutex 的執行時間小於 Spinlock 的，就要降低運算量，最後調整成 Mask=3,Matrix=64 才符合需求，當運算量變低，Mutex 的 Context switch 次數變少，Overhead 隨之變小，而 Spinlock 採用 Busy waiting 方式，會一直停留在同個 task 直到解鎖，雖然運算量降低，執行時間也跟著縮短，但隨著運算量一直降低，到了某個程度，Mutex 執行時間的下降幅度會超過 Spinlock 時間的下降幅度。