微算機系統實習
MICROPROCESSOR SYSTEMS LAB.
SPRING, 2021

Instructor：Yen-Lin Chen(陳彥霖), Ph.D.
Professor
Dept. Computer Science and Information Engineering
National Taipei University of Technology

多執行緒
MULTI-THREADS(PTHREADS)

OUTLINE

- Processes and Threads
- Multi-threading vs Multi-processing
- POSIX Threads 簡介
- 多執行緒分工方法
- 多執行緒同步問題
- Critical Sections Problem
- Mutex & Semaphore

PROCESSES AND THREADS

- 行程(process)意指已經執行並且 load 到記憶體中的 程式(Program)
- 執行緒(Threads)是類似於”輕量化”(light-weight)的行程
  - 可以共享行程的資源
- 在shared memory程式中，一個行程可以擁有及控制多個執行緒

MULTI-THREADING VS MULTI-PROCESSING

- Multi-processing (多行程)
  - 指多個 process 在執行，彼此有各自的資料空間，若有資料需要共用，必須採用特別的方法來傳遞 (視 OS 而定)
  - 由於每個 process 都需要一些資源來工作，所以 Multi-processing 會比 Multi-thread 更消耗資源 (Google Chrome 採用這種設計，因此會消耗不少記憶體)
  - 在Linux使用fork來進行平行運算也是屬於multi-processing的一種

MULTI-THREADING VS MULTI-PROCESSING

- Multi-threading (多執行緒)
  - 指一個 process 裡有多個thread在執行，彼此共用相同的資料空間
  - Multi-threading工作的其中一項特點就是隸屬在同一程式下的所有thread會分享該程式的所有資源，此外各thread彼此間也可以擁有自己私有的資源而不與其它同一程式內的thread共用
  - 這就像是工廠內的公共設備是所有員工可以自由使用一樣，但每個員工也都會有自己的工作空間與置物櫃可以擺放自己私人的物品，其他員工不能隨意侵犯他人的私有領域

MULTI-THREADING VS MULTI-PROCESSING

- 在排程方面
  - multi-thread是全部放在一個process裡面，相對於multi-process更省資源
- 在記憶體方面
  - multi-process的記憶體是分開的，所以要溝通的話必須調用系統函數，而因為multi-thread的記憶體都是放一起的，所以不用呼叫系統函數調用該記憶體

MULTI-THREADING VS MULTI-PROCESSING

- 速度方面
  - 啟用兩個process平行處理，作業系統將運算量平均分配在兩個計算核心之上。 由於2個process各擁獨立的process資源空間，不需另耗資源協調同步工作，故會比multi-thread還快
- 執行方面
  - multi-thread的缺點就是任何一個thread掛掉都可能直接造成整個程式崩潰，但multi-process不會

多執行緒種類(C/C++)

- MPI
- OpenMP
- Pthread
- C++ standard thread
- Qt Q thread
- 除了以上幾種，還有許多的多執行緒實現方法與函式庫

POSIX THREADS 簡介

- POSIX執行緒（英語：POSIX　Threads，常被縮寫為Pthreads）是POSIX的執行緒標準，定義了創建和操縱執行緒的一套API。
- 實現POSIX執行緒標準的函式庫常被稱作Pthreads，一般用於類似Unix的POSIX系統，如Linux，Solaris。但是Microsoft Windows上的實現也存在，例如直接使用Windows API實現的第三方函式庫pthreads-w32;而 利用Windows的SFU / SUA子系統，則可以使用微軟提供的一部分內建POSIX API。

POSIX THREADS 簡介

- Pthreads定義了一套C語言的類型、函數與常數，它以pthread.h標頭文件和一個執行緒庫實現
- 其API可以用於多執行緒的程式開發

POSIX THREADS 簡介

- Pthreads API中大致共有100個函數可呼叫，全都以"pthread_"開頭，並可以分為四類：
  - 執行緒管理，例如創建執行緒，等待(join)執行緒，查詢執行緒狀態等。
  - 互斥鎖(Mutex)：創建、摧毀、鎖定、解鎖、設置屬性等操作
  - 條件變量(Condition Variable)：創建、摧毀、等待、通知、設置與查詢屬性等操作
  - 使用了互斥鎖的執行緒間的同步管理

POSIX THREADS 簡介

- Pthreads範例
  - 主執行緒和子執行緒交替執行
  - 必須要宣告pthread.h
  - 在編譯時要加上 ”-lpthread”，表示要連結器Linker連結Pthread程式庫

```
jiemin@jiemin-D830MT:~$ gcc th1.c -lpthread -o out
jiemin@jiemin-D830MT:~$ ./out
Master
Child
Master
Child
Master
Child
```

POSIX THREADS 簡介

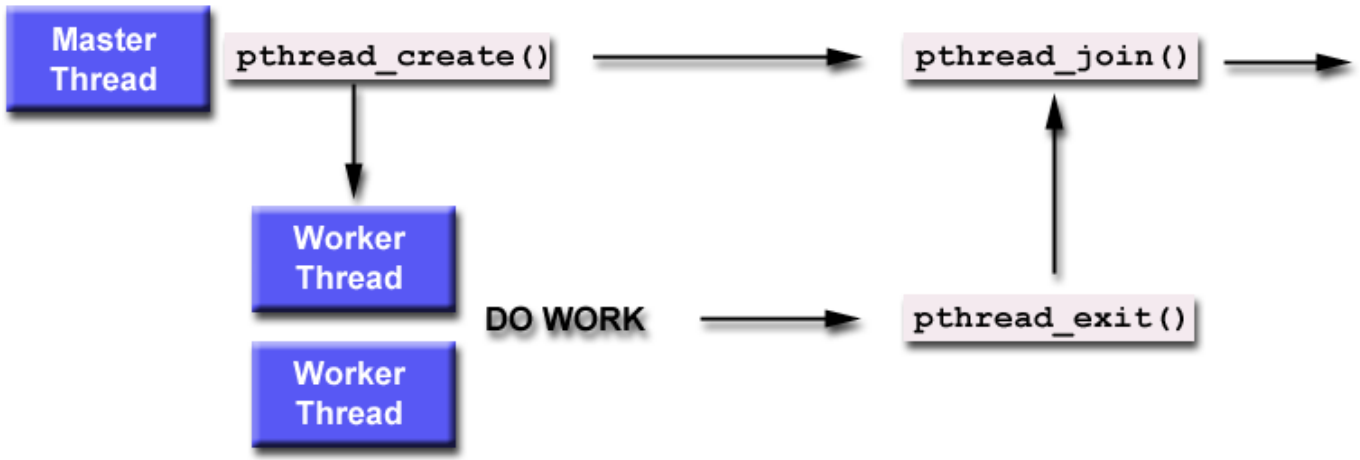
- pthread程式編譯時需要注意 .c 檔必須要放在

  -lpthread的前面，否則不會被參考到

```
lalako@lalako:~/pthread_sample$ gcc -lpthread test1.c -o run.o
/tmp/ccTekAVa.o: 於函式 main:
test1.c:(.text+0x9a): 未定義參考到「pthread_create」
test1.c:(.text+0xb5): 未定義參考到「pthread_create」
test1.c:(.text+0xc6): 未定義參考到「pthread_join」
test1.c:(.text+0xd7): 未定義參考到「pthread_join」
collect2: error: ld returned 1 exit status
lalako@lalako:~/pthread_sample$ gcc test1.c -lpthread -o run.o
lalako@lalako:~/pthread_sample$
```

POSIX THREADS 簡介

- pthread_create() 用於新增一個執行緒
  - 第三個參數為要指定給thread執行的函式
- pthread_join() 會等待執行緒完成工作後才會往下繼續執行程式
- pthread_exit() 程式完成需呼叫主函式
  - 可以填入參數，會回傳於主執行緒



POSIX THREADS 簡介

- 主執行緒的函式

```c
int main() {
  pthread_t t; // 宣告 pthread 變數
  pthread_create(&t, NULL, child, "Child"); // 建立子執行緒

  // 主執行緒工作
  for(int i = 0;i < 3;++i) {
    printf("Master\n"); // 每秒輸出文字
    sleep(1);
  }

  pthread_join(t, NULL); // 等待子執行緒執行完成
  return 0;
}
```

POSIX THREADS 簡介

- 子執行緒的函式

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// 子執行緒函數
void* child(void* data) {
  char *str = (char*) data; // 取得輸入資料
  for(int i = 0;i < 3;++i) {
    printf("%s\n", str); // 每秒輸出文字
    sleep(1);
  }
  pthread_exit(NULL); // 離開子執行緒
}
```

POSIX THREADS 簡介

- Pthreads範例
  - 將要計算的數字傳至Thread且回傳計算結果至主執行緒

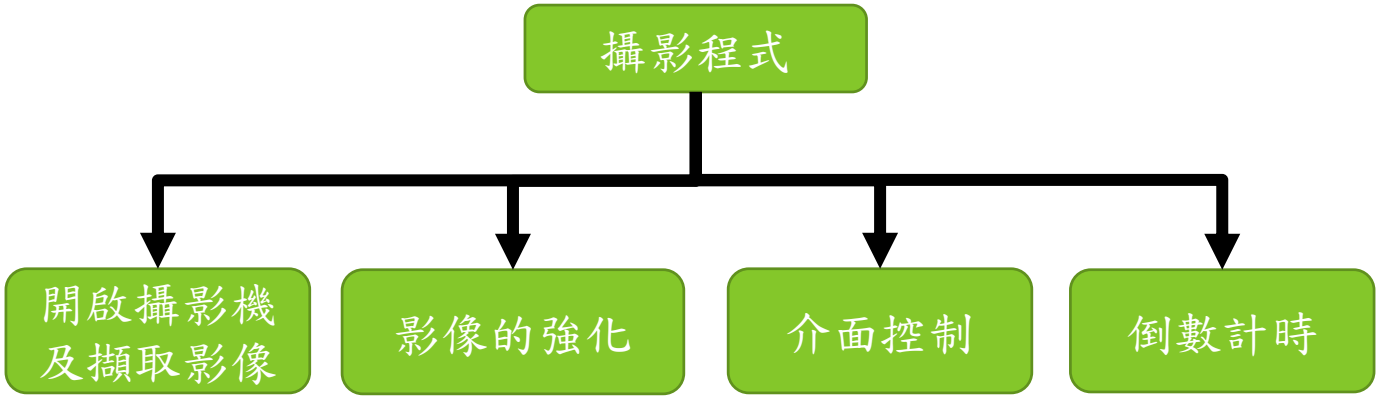
```
jiemin@jiemin-D830MT:~$ gcc th1.c -lpthread -o out
jiemin@jiemin-D830MT:~$ ./out
1 + 2 = 3
```

POSIX THREADS 簡介

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// 子執行緒函數
void *child(void *arg) {
   int *input = (int *) arg; // 取得資料
   int *result = malloc(sizeof(int) * 1); // 配置記憶體
   result[0] = input[0] + input[1]; // 進行計算
   pthread_exit((void *) result); // 傳回結果
}

int main() {
   pthread_t t;
   void *ret; // 子執行緒傳回值
   int input[2] = {1, 2}; // 輸入的資料

   // 建立子執行緒，傳入 input 進行計算
   pthread_create(&t, NULL, child, (void*) input);

   // 等待子執行緒計算完畢
   pthread_join(t, &ret);

   // 取得計算結果
   int *result = (int *) ret;

   // 輸出計算結果
   printf("%d + %d = %d\n", input[0], input[1], result[0]);

   // 釋放記憶體
   free(result);

   return 0;
}
```

多執行緒分工方法

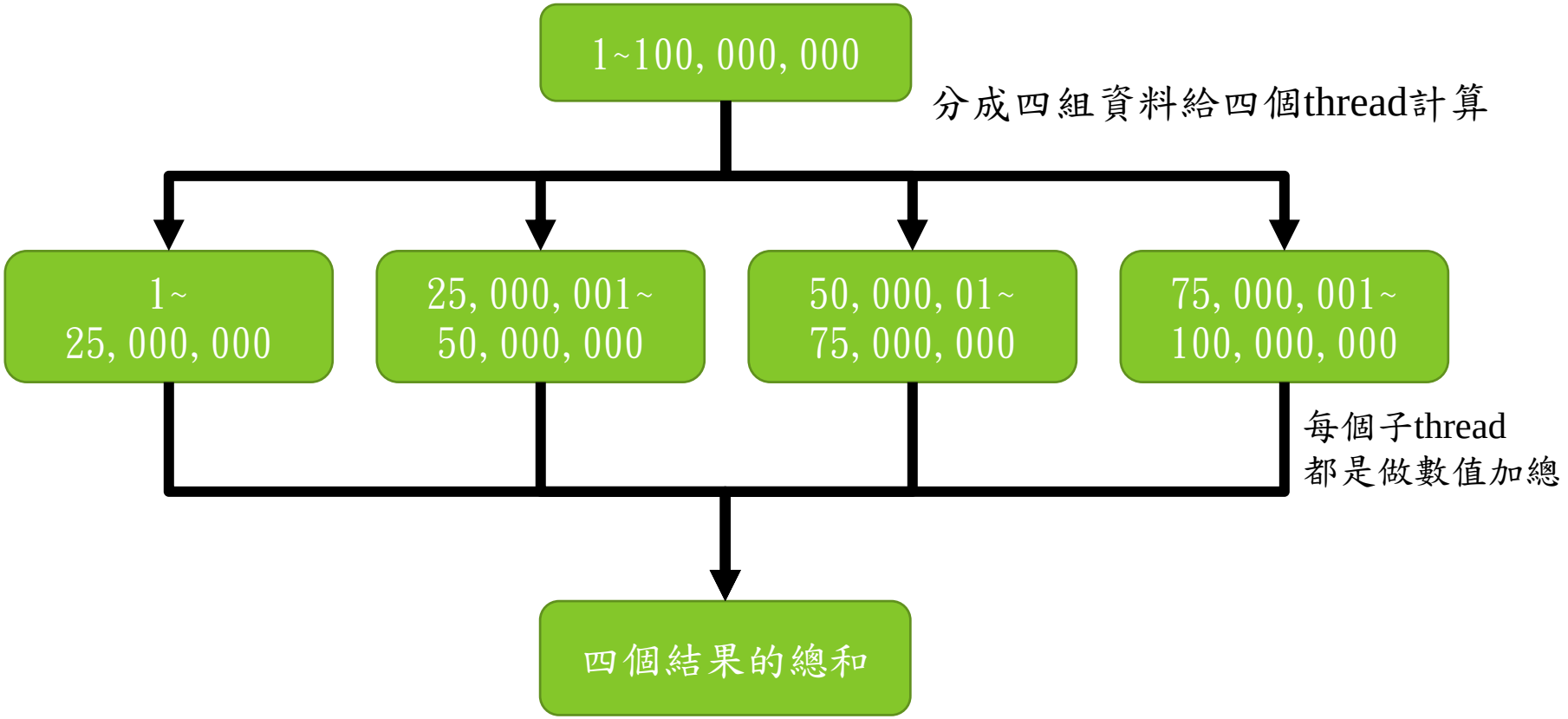
- 多執行緒在做工作分配時可以分為兩種類型
  - 任務平行(Task parallelism)，每一個執行緒執行一個分配到的任務，各自完成不同的任務
    - 如: 一個攝影程式，在多執行緒的分工上，可以讓一個執行緒做開啟攝影機及擷取攝影機影像的工作，第二個做影像的強化，第三個介面控制，倒數計時的計數器等等的分工處理模式



多執行緒分工方法

- 資料平行(Data parallelism)，將資料分配到不同的執行緒中，資料平行把大的任務化解成若干個相同的子任務，加速執行
  - 如:將一個龐大的運算，將一加到一億的加法，分成四組資料，各別都去做相同的加法計算，得到四組各別的和，最後再加總得到解

多執行緒分工方法


1~100, 000, 000
分成四組資料給四個thread計算
1~
25, 000, 000
25, 000, 001~
50, 000, 000
50, 000, 01~
75, 000, 000
75, 000, 001~
100, 000, 000
每個子thread
都是做數值加總
四個結果的總和

多執行緒同步問題

- 在多執行緒程式中，兩條執行緒可能同時會對同一公共資源（比如全域變數）進行讀寫，可能會導致一執行緒在讀取完後，被作業系統暫停，使另外一個執行緒插入執行並寫入新的資料，而原先讀取的值就會變成錯誤的舊資料，最終資料處理的結果便產生問題
- 範例為同時讓兩個執行緒對一全域變數SUM各別累加10000次的1，下圖結果為錯誤資料，正確預期結果為20000

```
lalako@lalako:~/pthread_sample$ gcc test1.c -o run.o -lpthread
lalako@lalako:~/pthread_sample$
lalako@lalako:~/pthread_sample$ ./run.o
 SUM : 13406
lalako@lalako:~/pthread_sample$ ./run.o
 SUM : 12910
lalako@lalako:~/pthread_sample$ ./run.o
 SUM : 14481
lalako@lalako:~/pthread_sample$
```

多執行緒同步問題

test1.c (~/pthread_sample) - gedit

開啟(O) 儲存(S)

```c
#include <stdio.h>
#include <pthread.h>

/*pthread*/
pthread_mutex_t mutex;

int sum;

void * thread_run(void * arg){
        int i = 0;
        for(i = 0; i < 10000; i++){
        //      pthread_mutex_lock(&mutex);
                sum = sum + 1;
        //      pthread_mutex_unlock(&mutex);
        }
        pthread_exit(NULL);
}

int main(int argc, char * argv[]){
        pthread_t thread1, thread2;
        pthread_mutex_init(&mutex, 0);
        pthread_create(&thread1, NULL, thread_run, 0);
        pthread_create(&thread2, NULL, thread_run, 0);
        pthread_join(thread1, 0);
        pthread_join(thread2, 0);
        printf("sun: %d\n", sum);
        pthread_mutex_destroy(&mutex);
        return 0;
}
```

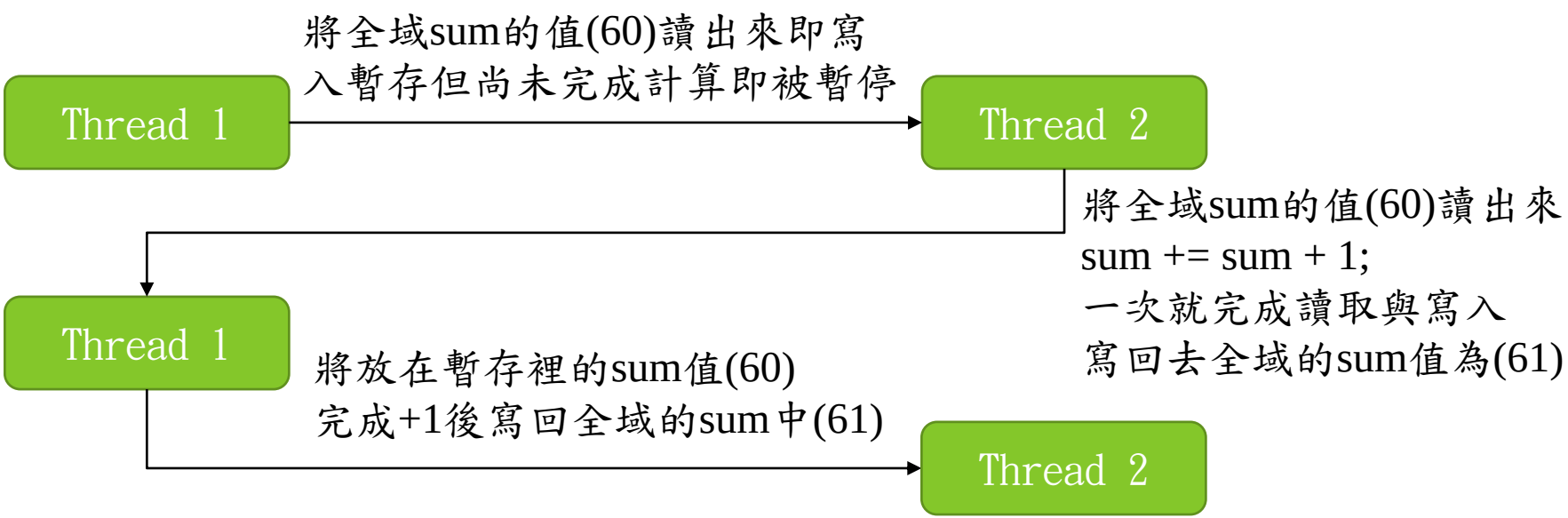
會造成記憶體覆蓋的全域變數

C Tab 寬度：8 第 29 行第 2 字 插入

多執行緒同步問題

```
int sum;

void * thread_run(void * arg){
        int i = 0;
        //pthread_mutex_lock(&mutex);
        for(i = 0; i < 10000; i++){
                sum = sum + 1;
        }
        //pthread_mutex_unlock(&mutex);
        return 0;
}
```

- 如thread1, thread2兩個執行緒在對同一個全域變數sum進行加法，在作業系統的執行上，可能會發生這種情況，假定sum值目前為60

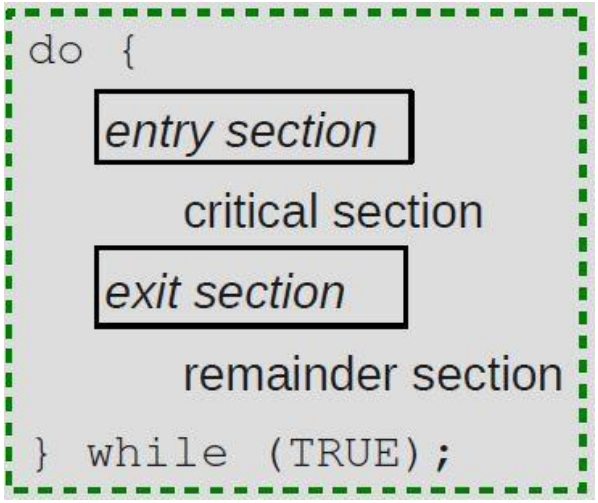


CRITICAL SECTIONS PROBLEM

- 提供對共享變數存取的互斥控制，確保資料的正確性
- entry section
  - 每個thread要進去critical section之前都要在entry section 詢問是否允許進入，進入後便會上鎖禁止其他thread使用
- critical section
  - process中對共享變數進行存取的敘述之集合區間

CRITICAL SECTIONS PROBLEM

- exit section
  - 移除鎖，可以再讓其他thread進入critical section內
- remainder section
  - 剩餘需要執行的程式部分



SOLUTION TO CRITICAL SECTION PROBLEM

- Mutual exclusion
  - 任一時間點，只允許一個 thread 進入他自己的 critical section 內活動
- Progress
  - 不想進入 critical section 的 thread 不可以阻礙其它 thread 進入 critical section，即不可參與進入 critical section 的決策過程
  - 必須在有限的時間從想進入 critical section 的 thread 中，挑選其中一個 thread 進入 critical section，隱含No Deadlock

SOLUTION TO CRITICAL SECTION PROBLEM

- Bound waiting
  - 自 thread 提出進入 critical section 的申請到獲准進入 critical section 的等待時間是有限的。即若有 n 個 threads 想進入，則任一 thread 至多等待 n-1 次即可進入，隱含 No starvation

互斥鎖 (MUTEX)

- 互斥鎖(MUTEX)是為了保護資料完整性，防止兩條執行緒同時對同一個公共資源(比如全域變數)進行讀寫的機制
- 當一個執行緒對全域變數進行讀寫時，此時上鎖互斥鎖，其餘執行緒若要讀寫同一全域變數便會被阻塞，直到原執行緒完成讀寫操作並將互斥鎖解鎖後，其餘執行緒才可以存取全域變數。

互斥鎖 (MUTEX)

test1.c (~/pthread_sample) - gedit

開啟(O) 儲存(S)

```c
#include <stdio.h>
#include <pthread.h>

/*pthread*/
pthread_mutex_t mutex;

int sum;

void * thread_run(void * arg){
        int i = 0;
        for(i = 0; i < 10000; i++){
                pthread_mutex_lock(&mutex);
                sum = sum + 1;
                pthread_mutex_unlock(&mutex);
        }
        pthread_exit(NULL);
}

int main(int argc, char * argv[]){
        pthread_t thread1, thread2;
        pthread_mutex_init(&mutex, 0);
        pthread_create(&thread1, NULL, thread_run, 0);
        pthread_create(&thread2, NULL, thread_run, 0);
        pthread_join(thread1, 0);
        pthread_join(thread2, 0);
        printf("sum: %d\n", sum);
        pthread_mutex_destroy(&mutex);
        return 0;
}
```

上鎖

解鎖

初始化mutex

建立執行緒，以函數指標指定子執行緒所要執行的函數

等待子執行緒計算完畢

C Tab 寬度：8 第 12 行第 17 字 插入

互斥鎖 (MUTEX)

- 該程式如同多執行緒同步問題範例，即同時讓兩個執行緒讀寫同一個全域變數，將sum各別累加10000次的1，加入了互斥鎖機制，可得到正確預期結果20000

```
lalako@lalako:~/pthread_sample$ gcc test1.c -o run.o -lpthread
lalako@lalako:~/pthread_sample$ ./run.o
 SUM : 20000
lalako@lalako:~/pthread_sample$ ./run.o
 SUM : 20000
lalako@lalako:~/pthread_sample$ ./run.o
 SUM : 20000
lalako@lalako:~/pthread_sample$
```

SEMAPHORE (號誌)

- Semaphore(號誌)是一個用來解決Critical Section(臨界區間)及Synchronization(同步)問題的資料型別
- 為同步物件，用於保持在0至指定最大值之間的一個計數值
- 提供2種操作，分別是**signal()**增加計數值和**wait()**減少計數值
- 當執行緒執行wait()時，計數值減1並進入至critical section；當執行緒執行完critical section後，接續執行signal()將計數值加1
- 當計數值為0的時候，若有執行緒要執行wait()，表示該執行緒要去執行critical section，則會被阻塞，直至其餘執行緒執行signal()將計數值加1，該執行緒才可進入critical section

SEMAPHORE (號誌)

- semaphore物件的計數值大於0，為signaled狀態；計數值等於0，為nonsignaled狀態
- 當計數值為0，則執行緒被阻塞等待該semaphore物件，直至該semaphore物件變成signaled狀態
- 若semaphore只有二進位的0或1，又稱為binary semaphore (二進位號誌)

SEMAPHORE (號誌)

- include semaphore.h (引入號誌函式庫)

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

sem_t semaphore; // 號誌
int counter = 0;

// 子執行緒函數
void* child() {
  for(int i = 0;i < 5;++i) {
    sem_wait(&semaphore); // 等待工作
    printf("Counter = %d\n", ++counter);
    sleep(1);
  }
  pthread_exit(NULL);
}
```

SEMAPHORE (號誌)

sem_init 初始化號誌，其第2個參數是指定是否要讓其他的process共用號誌，這裡我們是單一process、多執行緒的程式，所以第2個參數設定為 0 即可；第3個參數則是設定計數值的初始值。

```c
// 主程式
int main(void) {

  // 初始化號誌，僅用於本行程，初始值為 0
  sem_init(&semaphore, 0, 0);

  pthread_t t;
  pthread_create(&t, NULL, child, NULL);

  // 送出兩個工作
  printf("Post 2 jobs.\n");
  sem_post(&semaphore);
  sem_post(&semaphore);
  sleep(4);

  // 送出三個工作
  printf("Post 3 jobs.\n");
  sem_post(&semaphore);
  sem_post(&semaphore);
  sem_post(&semaphore);

  pthread_join(t, NULL);

  return 0;
}
```

SEMAPHORE (號誌)

- 在這個程式中，主執行緒負責派送工作，工作有時候多、有時候少，而子執行緒則是以每秒處理一個工作的速度，消化接收到的工作

```
jiemin@jiemin-D830MT:~$ gcc th1.c -lpthread -o out
jiemin@jiemin-D830MT:~$ ./out
Post 2 jobs.
Counter = 1
Counter = 2
Post 3 jobs.
Counter = 3
Counter = 4
Counter = 5
```

AI深度學習-多執行緒
延伸補充資料

AI深度學習概念 – NVIDIA GPU ARCHITECTURE

- GPU最基本的處理單位是*SP(Streaming Processor)*
- 數個SP 附加其他單元組成*SM(Streaming Multiprocessor)*
- GPU 擁有許多SM可同時進行運算

AI深度學習概念-
SM, BLOCK, WARP

- SM(Streaming Multiprocessor)

  –GPU硬體運算時會把CUDA中的thread block分配給SM進行運算

- block(thread block)

  –Thread block中實際進行運算的threads以*warp* 為單位一起執行同一道指令

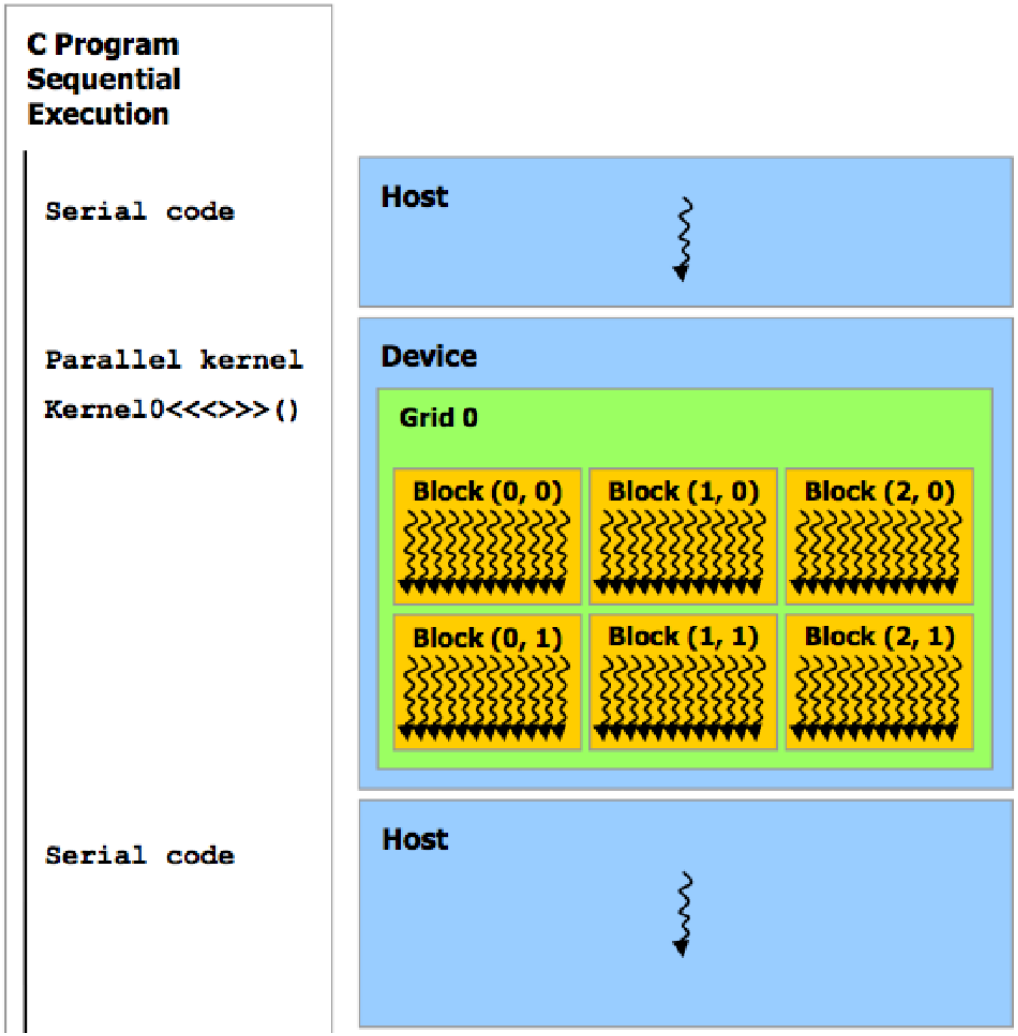
- Warp

  –1 warp = 32 threads

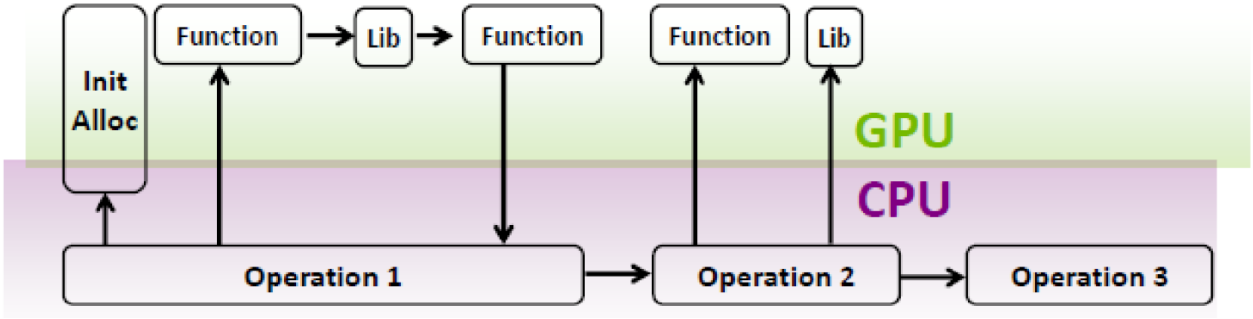
AI深度學習概念 - CUDNN

- NVIDIA cuDNN 是一個為深層神經網路設計的 GPU 加速原式函式庫，提供經過調校的常式建置方法，這些常式則常用於 DNN 應用
- cuDNN具有執行緒安全的功能，並提供情境式API，可輕鬆進行多執行緒存取和提供與CUDA的互通性。舉例來說，這可讓開發人員在使用多個主執行緒和多重GPU時能夠明確監控函式庫的設定，另外確保一個特定GPU裝置可用於特定的主執行緒。

CUDNN CUDA程式執行流程圖



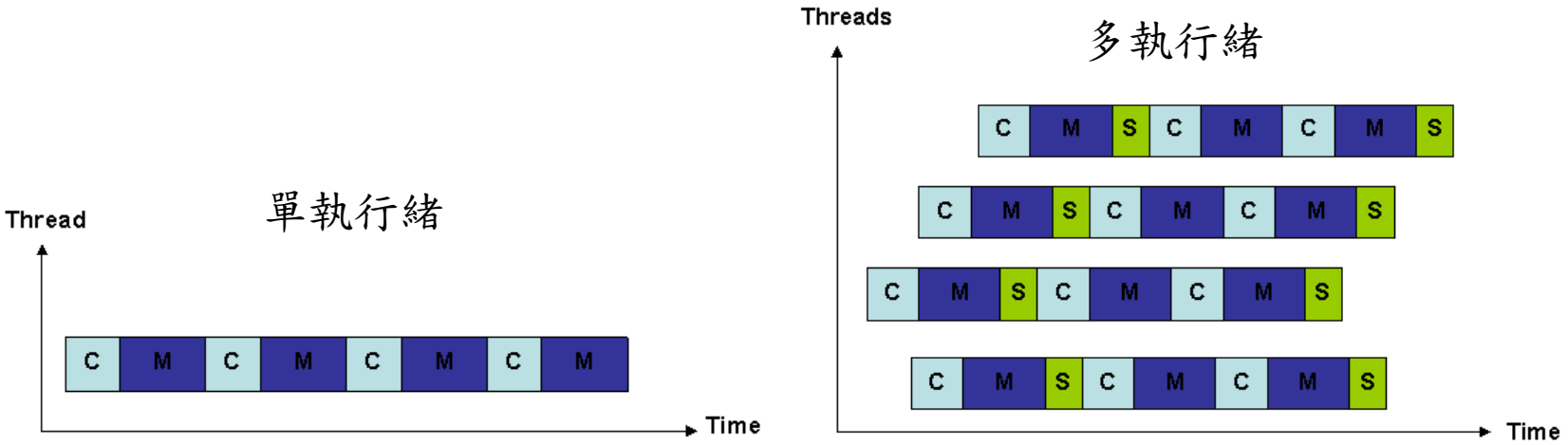
SIMPLE 5 STEPS TO PROGRAM GPU WITH CUDA



- Step 1 : Allocate device memory
- Step 2 : Upload input data from host to device memory
- Step 3 : Call CUDA kernel(s)
  ….. (can call many kernels to manipulate data in device memory)
- Step 4 : Download output data from device to host memory
- Step 5 : Free device memory

AI深度學習概念 - 平行運算

- 分工器是將CPU的資源進行共享他會依照開發人員所設定的執行緒優先權(Thread Priority)，將原本需要一筆一筆序列化進行的工作，藉由分工器不斷的在執行緒間快速的切換工作權，讓每個工作執行緒輪流使用CPU資源，達到「近乎同時」進行運算的目的



參考資料

- https://qing-yao.blogspot.com/2016/08/writeByMind-2.html
- https://zh.wikipedia.org/zh-tw/POSIX%E7%BA%BF%E7%A8%8B
- https://blog.gtwang.org/programming/pthread-multithreading-programming-in-c-tutorial/
- http://www.cc.ntu.edu.tw/chinese/epaper/0012/20100320_1205.htm
- https://codertw.com/%E7%A8%8B%E5%BC%8F%E8%AA%9E%E8%A8%80/517694/?fbclid=IwAR3BZdRLb8lF7PpE4tuIeSIsDlpPl7MyDVQXjgFR1xETDJqeqtH1ixFr2RM