**Laboratory 2:Multithreaded programming**

**Name: Yufei Zheng Class: CS 1602**

**Date:** April 4, 2019

**Part 1 The introduction**

**1.1The purposes**

(1)To obtain a good understanding of multi-threading;

(2)To practice creating threads and coordinate the running of the threads.

**1.2The principles**

The process tackle with one specific task with the using of threads. Usually, the program aimed for dealing with some less difficult problem use a main thread to execute. But we can definitely use multi-thread which proved to execute faster than single does since the C language provide multifarious convenient functions specifically devised for threads execution.

We will mainly use the following functions provided by C lib: pthread\_create(…),pthread\_exit(..) (note that the parameters the functions need have been omitted, you can get a full understanding of them on the website: <https://www.cnblogs.com/mq0036/p/3710475.html>)

**Part 2 The materials and apparatus**

Material: Function references: <https://www.cnblogs.com/mq0036/p/3710475.html>

Others: The laboratory guideline on the educational platform for undergraduate.

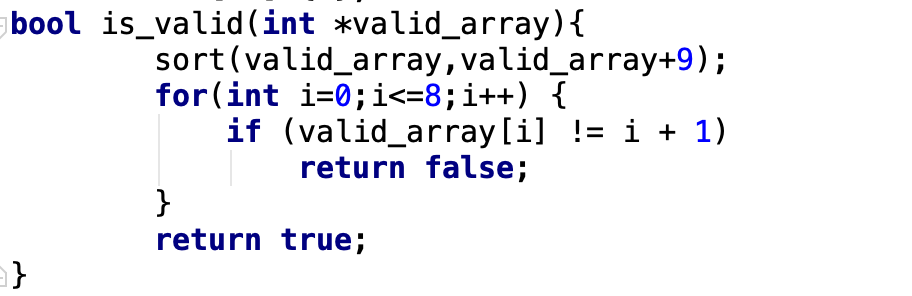
Apparatus：Apple laptop with macOS and C compiling environment.

**Part 3 The procedure**

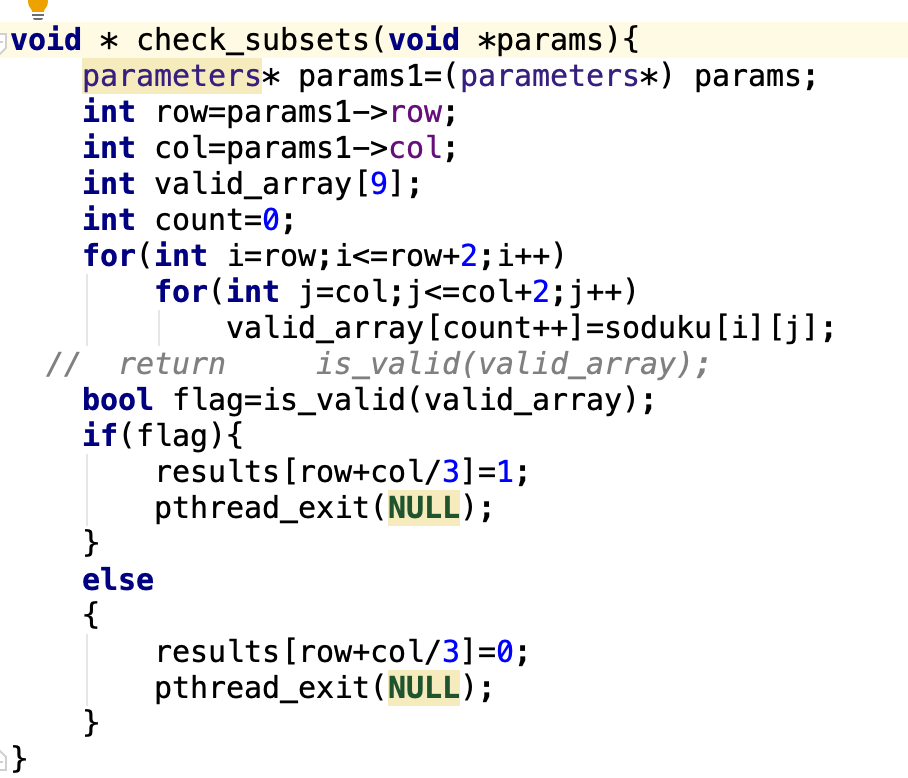
**3.1 operating procedure for part1**

(1) Initialize all the global variables which could be used later such as the soduku, the array of results and the array of the threads.

(2) Finish some the function of judgement of the array which receive the numbers from row, column or subsets of the matrix.



(3) Familiarize with the format of the function invoked by “pthread\_create()”:the return type must be void\* and the formal parameter must be void\* too. The three function, separately for column, row and subsets, should all respect the principles following: Firstly, get the start position; secondly, fill out the new array corresponding to the pointer passed on; thirdly, get the semaphore, namely, flag in this program, to identify whether the nine numbers accord with the demand; fourthly, execute “pthread\_exit”

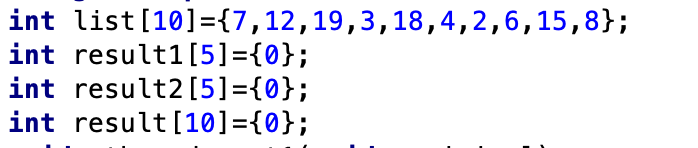


(4) Traverse the soduku using double loop and execute specific function- pthread\_create when the i and j reached some special position, i.e. (i%3==0and j%3==0->subsets),(i==0->column),(j==0->row). Most important, due to the non-synchronization of the multiple threads, it’s necessary for the main thread to wait for moments until all the sub-threads have finished execution.

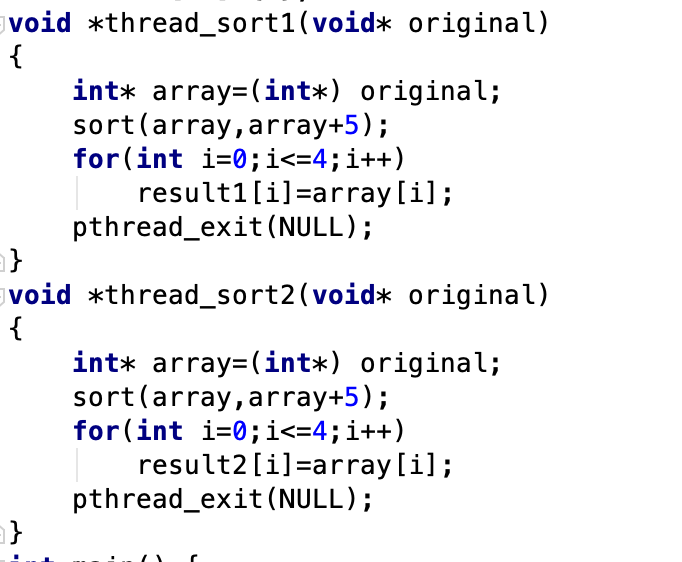


**3.2 Operating procedures for part II**

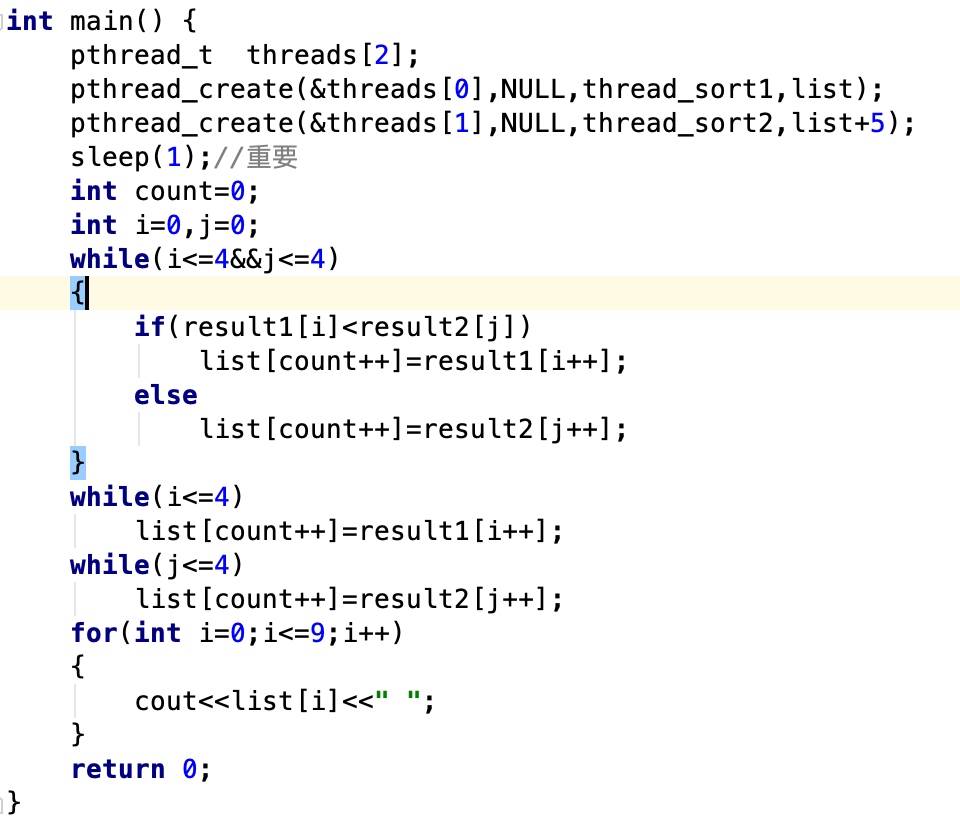
(1) Initialize all the global variables the program needs.



(2) Directly write the corresponding function for “pthread\_create”: Get the pointer passed by the formal parameter. Sort the new formulated array from position 0 to 5.Passed the sorted array to global array result1 or result2 which depends on the location of the part which the function received in the original array.



(3) Invoke the “pthread\_create” in the main function and execute the merge operation in the classical merge sort algorithm.(Don’t forget to let the main thread sleep for a while.)



**Part 4 The results**

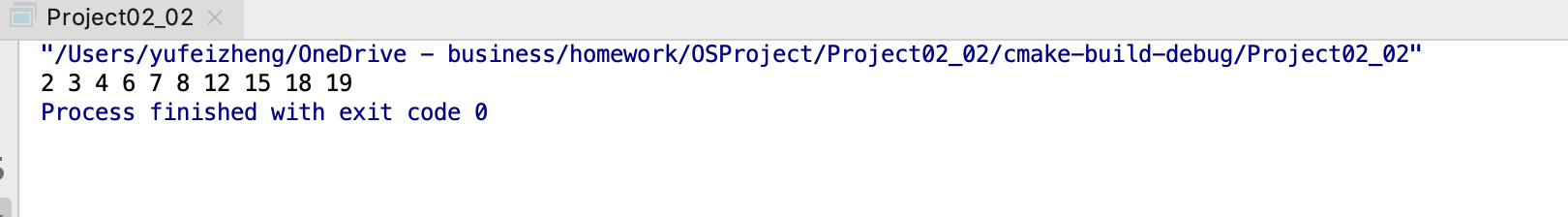
**4.1 Part I**

**The soduku employs the one given by the example.**



**4.2 Part II**

**Original sequence:** 7 12 19 3 18 4 2 6 15 8

****

**Part 5 Discussion**

In this experiment, I understand deeply and precisely about the implementation of the threads by programming oriented towards them. Especially, the notion of synchronization has rooted in my mind due to the bug I confronted in the project：No logical faults occurred in my program, but the results are different separately. Though I haven’t implemented the ideal mechanism of semaphore to avoid fitful execution of multi threads which can lead to terrible results, I use a rather simpler way to tackle this which could only be opportune for invoking small amounts of threads.(The method sleep) After being skilled in semaphore, I definitely would try that advanced methods which are aimed to synchronize different threads.

**Appendix:**

#include <pthread.h>

int pthread\_equal(pthread\_t tid1, pthread\_t tid2);

ret-若相等则返回非0值，否则返回0值

线程标识，可以比较两个线程的ID。

#include <pthread.h>

pthread\_t pthread\_self(void);

ret-调用线程的线程ID

获得线程自身的ID

#include <pthread.h>

int pthread\_create(pthread\_t \*restrict tidp, //返回线程的ID  
  
　　　　　　　　　　　const pthread\_attr\_t \*restrict attr, //线程属性，默认为NULL  
  
 void \*(\*start\_rtn)(void), //线程函数入口地址  
  
 void \*restrict arg); //参数

ret-成功返回0 失败返回错误编号

线程的创建

#include <pthread.h>

void pthread\_exit(void \*rval\_ptr);

#include <pthread.h>

线程的取消，如果只是从启动例程中返回，则返回值是退出码；线程也可以被历程中的其他线程取消。

int pthread\_join(pthread\_t thread, void \*\*rval\_ptr);  
  
ret-成功返回0 否则返回错误编号

注意到exit中的rval\_ptr函数，pthread\_join可以访问到这个函数。返回值将被包含在rval\_ptr中；如果线程被取消，就被设置成PTHREAD\_CANCELED；如果处于分离状态，pthread\_t调用失败，返回EINVAL；如果对其不感兴趣，可以设置为NULL，不获取终止状态。

#include <pthread.h>

int pthread\_cancel(pthread\_t tid);

ret-成功返回0 失败返回错误码

函数使得由tid标识的线程行为表现为如果调用了参数是PTHREAD\_CANCELD的pthread\_exit函数，但是，线程可以选择忽略取消方式或者是控制取消方式。函数并不等待线程终止，它仅仅是提出请求；

#include <pthread.h>

void pthread\_cleanup\_push(void(\*rtn)(void\*), void \*arg);

void pthread\_cleanup\_pop(int execute); //调用删除上次push的清理程序

当线程执行以下动作时调用清理函数，调用者参数为arg，清理函数rtn的调用顺序是由pthread\_cleanup\_pus来安排的。

a. 调用pthread\_exit;

b. 想用取消请求；

c. 用非零的execute参数调用pthread\_cleanup\_pop;

   如果execute=0则函数不被调用；

注意正常从线程返回的时候，不会调用该清理函数；