Report by B09902043 沈竑文

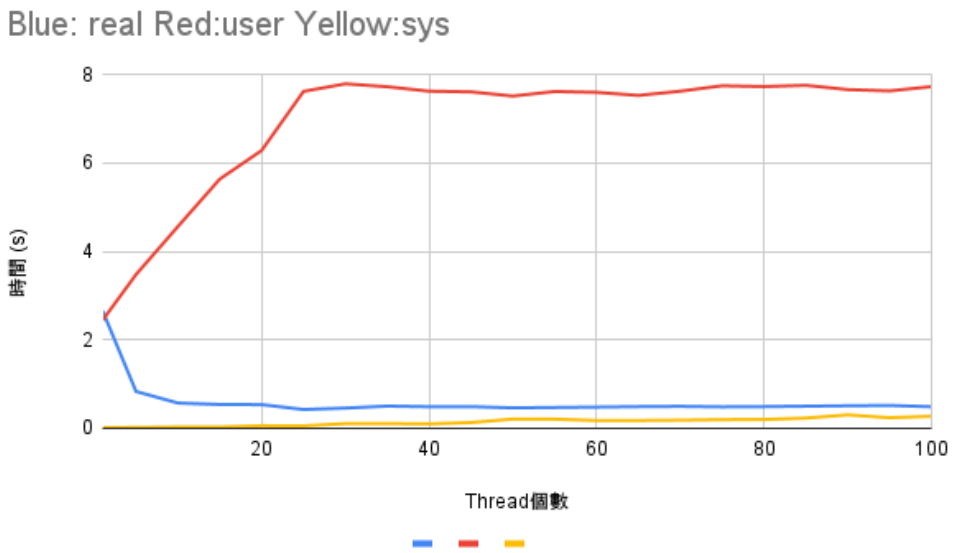
第四題的 comment 與結果: 20 個 thread 的速度顯著地高於 2 個 thread。因為將 board 上的區域切成 20 個，將每個區域分給不同 thread 處理時，一個 thread 分到的 cell 數少於分給 2 個 thread 時的狀況。而 thread model 有點類似平行運作，大家同時各自處理較少的 cell，因此 20 個 thread 處理完整個盤面會比 2 個 thread 快得多。

```
b09902043@linux1 [~/sp_test/programming-hw4-JRSmel0For3] time ./main -t 20 ./largeCase_in.txt ./output.txt

real    0m0.406s
user    0m4.965s
sys     0m0.134s
b09902043@linux1 [~/sp_test/programming-hw4-JRSmel0For3] time ./main -t 2 ./largeCase_in.txt ./output.txt

real    0m1.252s
user    0m2.379s
sys     0m0.001s
```

第五題的 comment 與折線圖:



Real time: 在 thread 少時，增加 thread 的效益非常顯著，因為同時工作的 thread 變多了所以明顯變快。然而資源有限，一次能同時執行的 thread 數目也有上限，因此在 thread 個數高時速度趨於平穩不再增加，甚至有時還因為 overhead 變多而有微幅的增加。

User time: thread 上升會帶來 user time 的增加，因為 user time 每條 thread 都要算。然而在 thread 數目達可平行處理的上限後便趨於平穩，因為有被考量入時間計算的 thread 就那麼多。

System time: pthread_create()以及 pthread_join()是 system call，而越多 thread 呼叫這些 system call 的需求就越多，因此 system time 隨 thread 數量增加而上升。

第六題的 comment:

```
b09902043@linux1 [~/sp_test/programming-hw4-JRSmel0For3] time ./main -p 2 ./largeCase_in.txt ./outoof2.txt

real    0m1.525s
user    0m2.846s
sys     0m0.011s
b09902043@linux1 [~/sp_test/programming-hw4-JRSmel0For3] time ./main -t 2 ./largeCase_in.txt ./outoof2.txt

real    0m1.645s
user    0m2.998s
sys     0m0.000s
```

實際運行起來的速度並沒有差很多，我推測是由於 thread、process 數目很少，所以 create thread 跟 process 的 cost 差距並沒有帶來巨大的影響。

重點截圖:

切工作範圍:

```
int start = 0;
for(int i = 0;i < thread_num;i++){
    pthread_create(ids + i, NULL, play, (void *)start);
    start += row * col / thread_num + ((i < row * col % thread_num)? 1:0);
}
```

```
void *play(void *arg){
    int start = (int) arg, next_round;
    int n = row * col / thread_num + ((start / (row * col / thread_num) < row * col % thread_num)? 1:0);
    for(int t = 0;t < epoch;t++){
        next_round = (Round + 1) % 2;
        for(int i = start;i < start + n;i++){
            update(i, next_round);
        }
```

以上兩圖為 thread 的範圍分工，start function 的參數為這個 thread 要處理的開始位置(我把 board 拉成一維)，而 n 則是該 thread 分到多少 cell。

若(格子總數 % thread 個數 != 0)，代表前(row * col % thread_num)個 thread 應該要多拿 1 個 cell 才會剛好。

```
if(fork() == 0){
    play_p(ptr, (row * col / 2 + row * col % 2));
    pthread_mutexattr_destroy(&mattr);
    pthread_condattr_destroy(&cattr);
}
else{
    play_p(ptr, 0);
    pthread_mutexattr_destroy(&mattr);
    pthread_condattr_destroy(&cattr);
    wait(NULL);

    for(int i = 0;i < row;i++){
        if(Round == 0) for(int j = 0;j < col;j++) fputc(ptr->board_p0[i * col + j], fp_out);
        else for(int j = 0;j < col;j++) fputc(ptr->board_p1[i * col + j], fp_out);
        if(i < row - 1) fputc('\n', fp_out);
    }
}
```

```
void play_p(Data *ptr, int start){
    int next_round;
    int n = row * col / 2 + ((start == 0)? (row * col % 2):0);
    for(int t = 0;t < epoch;t++){
        next_round = (Round + 1) % 2;
        for(int i = start;i < start + n;i++){
            update_p(i, ptr);
        }
```

以上兩圖為 2 process 的範圍分工，原理及意義與上方提到的 thread 作法類似，只是分的人數固定為 2。

處理同步:

```
pthread_mutex_lock(&lock);
done++;
if(done < thread_num)
    pthread_cond_wait(&cond, &lock);
else{
    done = 0;
    Round = next_round;
    pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&lock);
```

```
void play_p(Data *ptr, int start){
    int next_round;
    int n = row * col / 2 + ((start == 0)? (row * col % 2):0);
    for(int t = 0;t < epoch;t++){
        next_round = (Round + 1) % 2;
        for(int i = start;i < start + n;i++){
            update_p(i, ptr);
        }
```

左圖為處理 thread 同步，右圖為處理 2 process 同步。方法都是會利用所有人能 access 到的 mutex, condition variable 來保護 resource 及控制等待、醒來，並以全域變數 done 紀錄做完當前回合的人數，若是此回合最後一個完成的人就要重製 done 並 broadcast()叫醒其他所有人。pthread_cond_wait()時使用 if 而非 while 是因為叫醒大家時 done 已經重製，要避免回去睡。

Shared memory:

```
char *board[2], mode;
int row, col, epoch, Round, thread_num, done;
pthread_t ids[100];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
pthread_mutexattr_t mattr;
pthread_condattr_t cattr;
Data *ptr = (Data *) mmap(NULL, sizeof(Data), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
ptr->Done = 0;
pthread_mutexattr_init(&mattr);
pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&(ptr->mlock), &mattr);
pthread_condattr_init(&cattr);
pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);
pthread_cond_init(&(ptr->cv), &cattr);
```

上圖為 thread 及 2 process 達成 shared memory 的過程。Thread 是利用 global variables 會共享的特性來達成目的，2 process 則是利用 mmap 以及調整 mutex, cond 的 attribute 來達成 shared memory。