Problem 4

	2	20
Real	0.790	0.221
User	1.478	2.474
sys	0.017	0.055

Real time is less for 20 threads since threads run simultaneously, and there are less values to calculate for each thread.

Problem 5

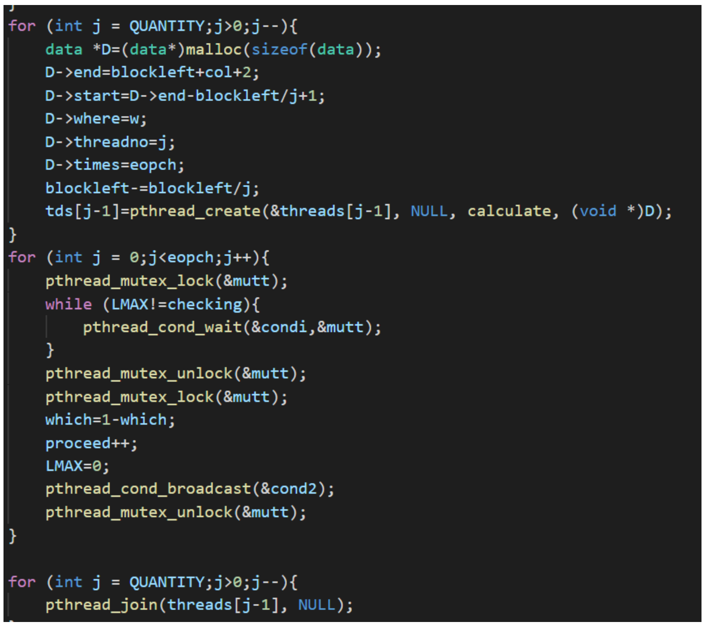
Real time decreases as thread increases, since threads run simultaneously, and there are less values to calculate for each thread.
User time increases as thread increases, since it is the sum of the time of all threads.
Sys time increases as thread increases, since there are more threads to create, join, etc.

Problem 6

	process	Thread
Real	0.679	0.790
User	1.231	1.478
sys	0.006	0.017

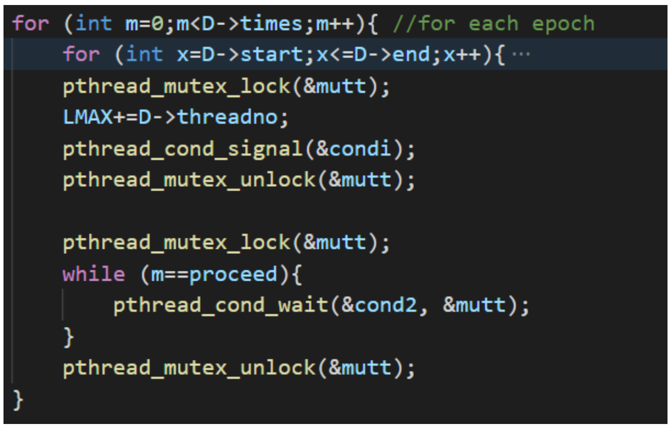
The use of `mmap` makes it such that the 2 processes share most of the memory, so that it does not need to allocate and copy much resources.

Problem 7



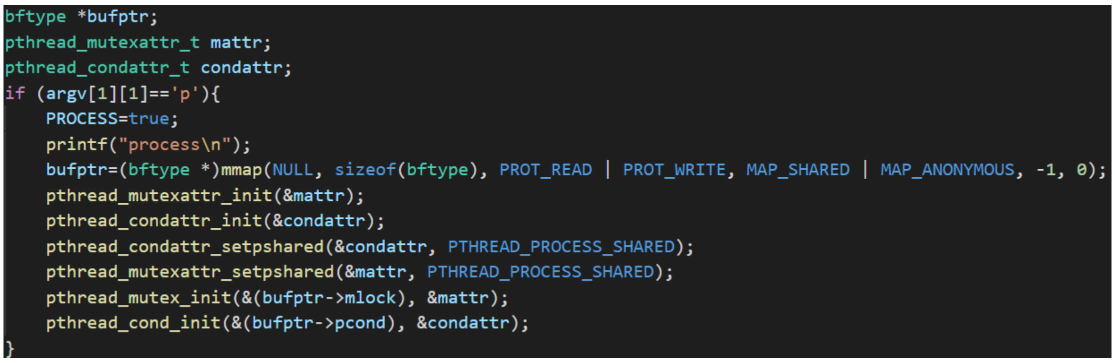
```
for (int j = QUANTITY;j>0;j--){
    data *D=(data*)malloc(sizeof(data));
    D->end=blockleft+col+2;
    D->start=D->end-blockleft/j+1;
    D->where=w;
    D->threadno=j;
    D->times=eopch;
    blockleft-=blockleft/j;
    tds[j-1]=pthread_create(&threads[j-1], NULL, calculate, (void *)D);
}
for (int j = 0;j<eopch;j++){
    pthread_mutex_lock(&mutt);
    while (LMAX!=checking){
        pthread_cond_wait(&condi,&mutt);
    }
    pthread_mutex_unlock(&mutt);
    pthread_mutex_lock(&mutt);
    which=1-which;
    proceed++;
    LMAX=0;
    pthread_cond_broadcast(&cond2);
    pthread_mutex_unlock(&mutt);
}

for (int j = QUANTITY;j>0;j--){
    pthread_join(threads[j-1], NULL);
```

The first for loop creates threads and assign each thread a section.
For a board of $n*m$, I keep 2 1D array of size $(n+2)(m+2)$, and split evenly among all thread or processes.
The second for loop is a main thread communicating to worker threads. Main thread waits on `condi` for `LMAX`, which is for checking if all threads are done with an epoch. Then main thread "flip" which side of the board to calculate, and reset some variables.



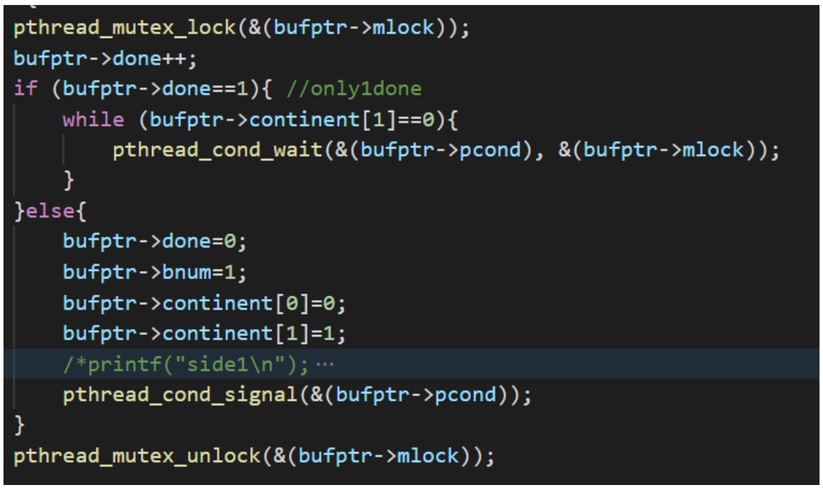
```
for (int m=0;m<D->times;m++){ //for each epoch
    for (int x=D->start;x<=D->end;x++){⋯
    pthread_mutex_lock(&mutt);
    LMAX+=D->threadno;
    pthread_cond_signal(&condi);
    pthread_mutex_unlock(&mutt);

    pthread_mutex_lock(&mutt);
    while (m==proceed){
        pthread_cond_wait(&cond2, &mutt);
    }
    pthread_mutex_unlock(&mutt);
}
```

The above is the worker thread. After worker has finished an epoch, it tells main thread via `LMAX`. Then they wait for main thread to give signal to calculate next epoch



```
bftype *bufptr;
pthread_mutexattr_t mattr;
pthread_condattr_t condattr;
if (argv[1][1]=='p'){
    PROCESS=true;
    printf("process\n");
    bufptr=(bftype *)mmap(NULL, sizeof(bftype), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    pthread_mutexattr_init(&mattr);
    pthread_condattr_init(&condattr);
    pthread_condattr_setpshared(&condattr, PTHREAD_PROCESS_SHARED);
    pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&(bufptr->mlock), &mattr);
    pthread_cond_init(&(bufptr->pcond), &condattr);
}
```

I use `mmap` to share `bufptr` between the 2 processes.



```
pthread_mutex_lock(&(bufptr->mlock));
bufptr->done++;
if (bufptr->done==1){ //only1done
    while (bufptr->continent[1]==0){
        pthread_cond_wait(&(bufptr->pcond), &(bufptr->mlock));
    }
}else{
    bufptr->done=0;
    bufptr->bnum=1;
    bufptr->continent[0]=0;
    bufptr->continent[1]=1;
    /*printf("side1\n");⋯
    pthread_cond_signal(&(bufptr->pcond));
}
pthread_mutex_unlock(&(bufptr->mlock));
```

The above is the communication between process. `bufptr->done` refers to how many processes have finished an epoch. If the process is the first to finish, then it waits for a signal. If it is the last to finish, then it resets some variables, 'flip' the board, and send signal to other process.