Operation system HW3 b03705002 林軒逸

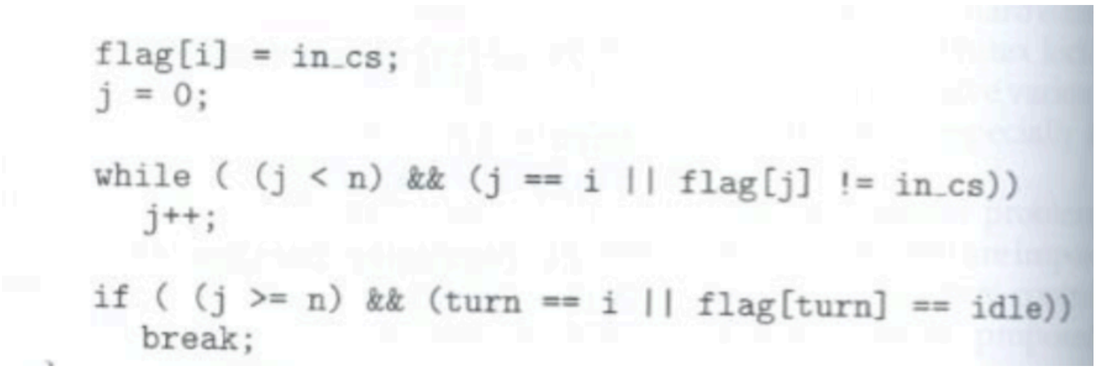
1.

The race of this problem is that if husband withdraw the money from the account and the wife deposits the money concurrently, which means there are two people modifying the information of the account concurrently. Under the situation, we can't and unable to predict what'll happen to the account or the result of their deposit/withdrawal. For instance, one might get the wrong information of bank account, or even cause system error. To prevent such situation, we must ensure the bank system allows only one person accessing and modify the data of specific account.

2.

a. Mutual Exclusion

in the whole "do" part in the first part, we only allow "turn" to execute it's critical section if there is no other flags is in "in_cs" condition, that means if there are any other processes are executing it's critical section, other processes want to enter will get the message that there is a process running in its critical section (in the code, this mean doesn't "break" the whole "do"), and won't change its flag to "in_cs". Thus, by this method, we can ensure mutual exclusion.



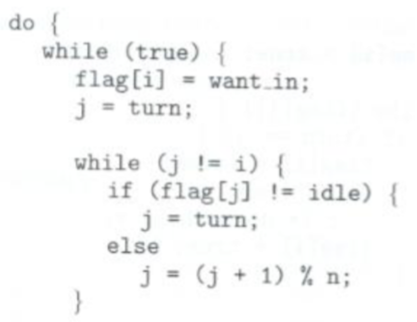
```
flag[i] = in_cs;
j = 0;

while ( (j < n) && (j == i || flag[j] != in_cs))
   j++;

if ( (j >= n) && (turn == i || flag[turn] == idle))
   break;
```

b. Progress

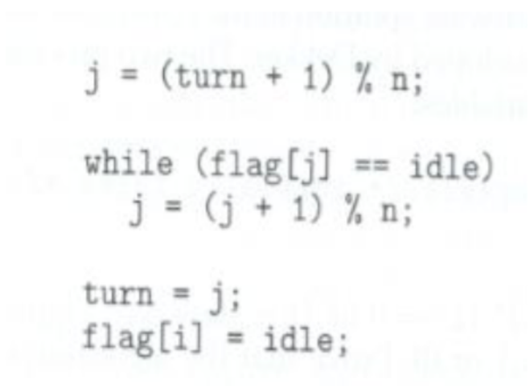
In the upper part in the "do", we can see we'll keep finding a candidate to enter (by turn and whose flag is "want_in". That is, we ensure if there are some processes want to execute its critical section, their flag will be "want_in" and will be executed. Thus, there exist some processes that wish to enter their critical sections.



```
do {
  while (true) {
    flag[i] = want_in;
    j = turn;

    while (j != i) {
      if (flag[j] != idle) {
        j = turn;
      else
        j = (j + 1) % n;
    }
```

Also, we will keep checking whether there is a process in waiting state by the method of exiting protocol. Whenever a process finished its critical section, the flag of itself will be set to idle, so the whole algorithm will get back to the top (finding an flag who has “want_in” state, so the selection won’t be postponed indefinitely.



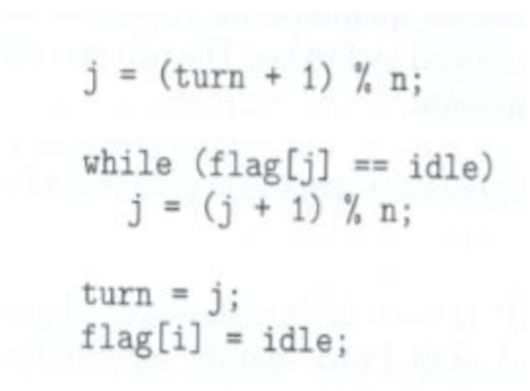
```
j = (turn + 1) % n;

while (flag[j] == idle)
   j = (j + 1) % n;

turn = j;
flag[i] = idle;
```

c. Bounded Waiting

By the same code as above, we can see that the next “turn” (the next process we check whether the flag is in “want_in” status) will be (j+1)%n. Thus, we will check the “next” process for sure. That is, an process will at most wait for n-1 processes to finish their critical sections, and it’ll be able to run its own, which represents a bounded and limited time of waiting.



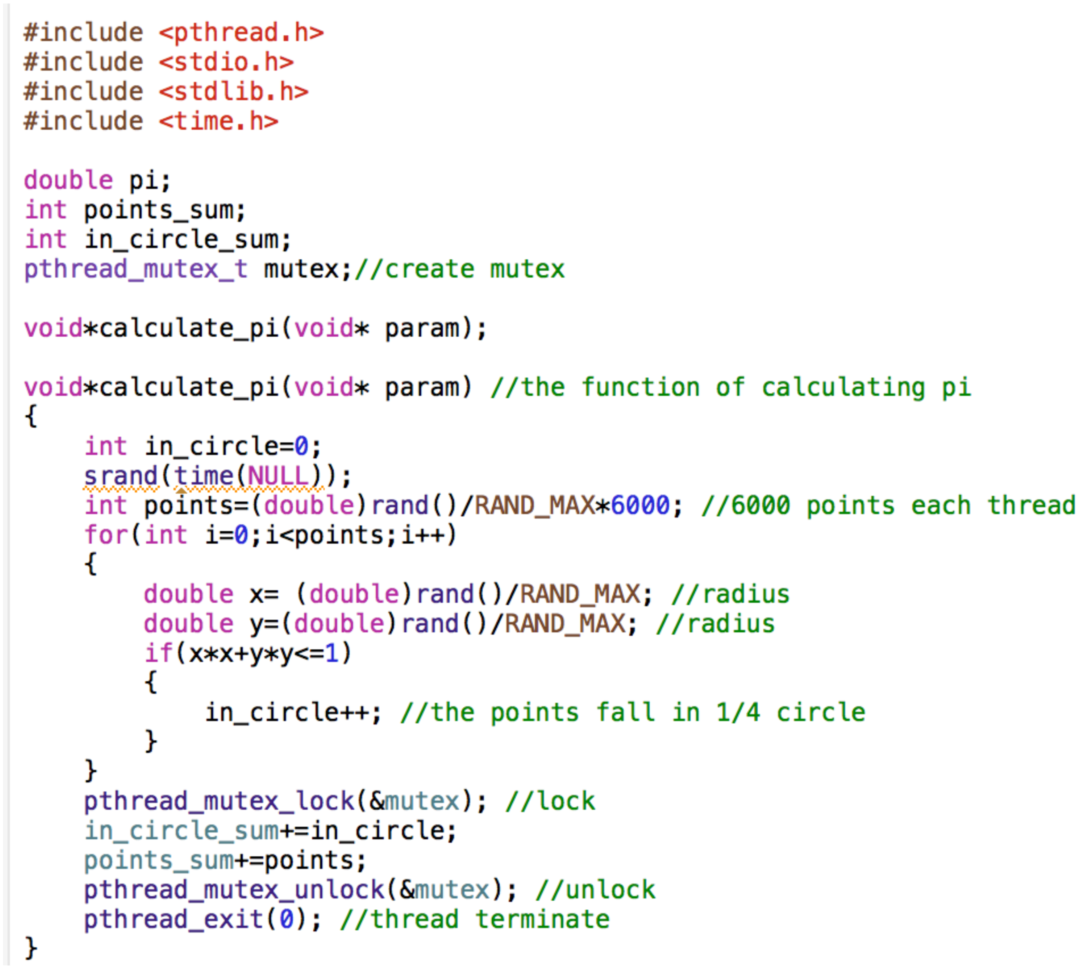
```
j = (turn + 1) % n;

while (flag[j] == idle)
   j = (j + 1) % n;

turn = j;
flag[i] = idle;
```

By a, b, c, we can say all three requirements of critical sections are satisfied, and the algorithm of Eisenburg and McGruein is correct and valid.

3. Code:

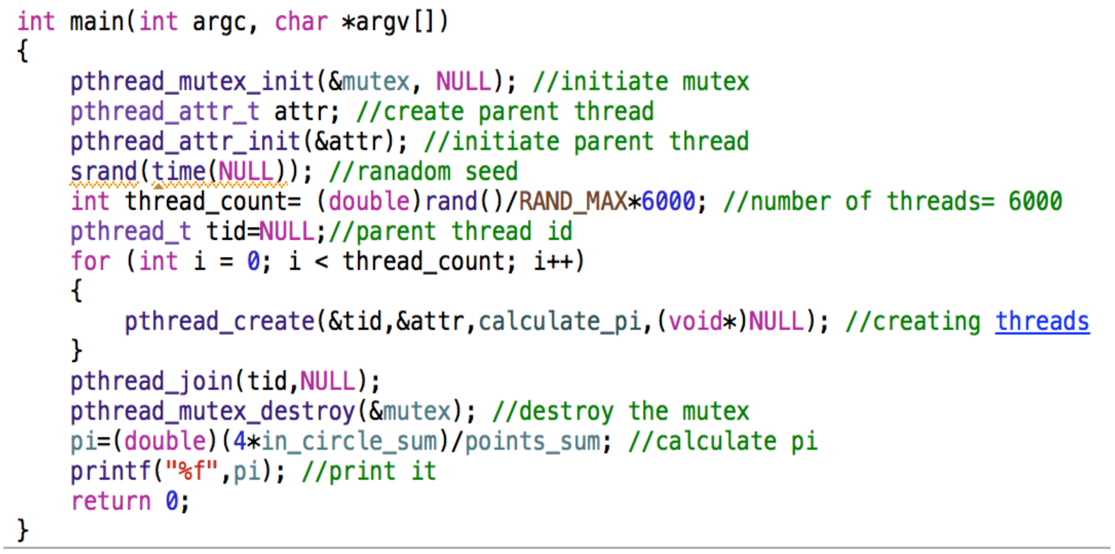


```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double pi;
int points_sum;
int in_circle_sum;
pthread_mutex_t mutex;//create mutex

void*calculate_pi(void* param);

void*calculate_pi(void* param) //the function of calculating pi
{
    int in_circle=0;
    srand(time(NULL));
    int points=(double)rand()/RAND_MAX*6000; //6000 points each thread
    for(int i=0;i<points;i++)
    {
        double x= (double)rand()/RAND_MAX; //radius
        double y=(double)rand()/RAND_MAX; //radius
        if(x*x+y*y<=1)
        {
            in_circle++; //the points fall in 1/4 circle
        }
    }
    pthread_mutex_lock(&mutex); //lock
    in_circle_sum+=in_circle;
    points_sum+=points;
    pthread_mutex_unlock(&mutex); //unlock
    pthread_exit(0); //thread terminate
}
```



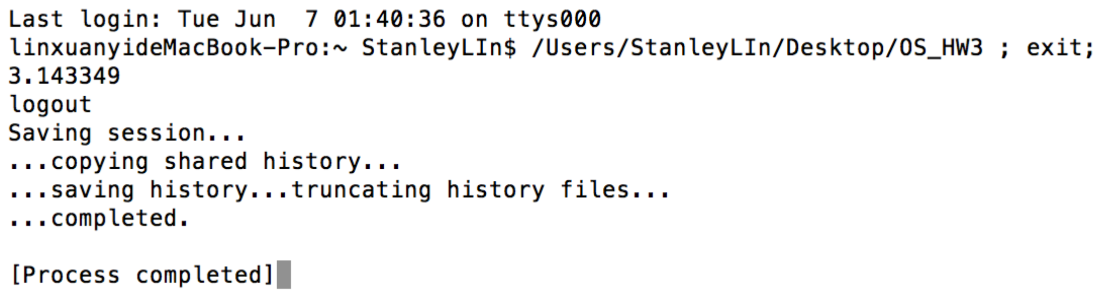
```c
int main(int argc, char *argv[])
{
    pthread_mutex_init(&mutex, NULL); //initiate mutex
    pthread_attr_t attr; //create parent thread
    pthread_attr_init(&attr); //initiate parent thread
    srand(time(NULL)); //ranadom seed
    int thread_count= (double)rand()/RAND_MAX*6000; //number of threads= 6000
    pthread_t tid=NULL;//parent thread id
    for (int i = 0; i < thread_count; i++)
    {
        pthread_create(&tid,&attr,calculate_pi,(void*)NULL); //creating threads
    }
    pthread_join(tid,NULL);
    pthread_mutex_destroy(&mutex); //destroy the mutex
    pi=(double)(4*in_circle_sum)/points_sum; //calculate pi
    printf("%f",pi); //print it
    return 0;
}
```

Explanation:

First, I used the method of thread as we used before to creating parent thread and some other child threads. Also, by the mutex lock method to ensure the synchronization problem. I created a function calculate_pi by the Monte Carlo method. We generate random numbers by srand and rand, and check whether the point falls in the ¼ circle. By adding all the values to the global variable point_sum and circle_sum. Also, for each thread, there will be 6000 points.

In the main function, we first create parent thread. After that, we create 6000 child threads to generate the “calculate_pi” function. After that, we calculate pi by continuing the Monte Carlo method using the global variables, and eventually print out the pi we calculated.

Run Result:



```
Last login: Tue Jun  7 01:40:36 on ttys000
linxuanyideMacBook-Pro:~ StanleyLIn$ /Users/StanleyLIn/Desktop/OS_HW3 ; exit;
3.143349
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
```