Xi Xia
2016213482
2016215117

5.7. Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring.
Possible race condition:
Assume there is 100 in the account. When the husband deposit 10: Before it overwrite the balance to 110. the calculation in the withdraw process executes based on the original balance 100. And it overwrites the balance after the overwriting of husband's deposit. So, the operation of deposit is omitted.

husband	wife	balance
Read 100		100
100+10	Read 100	100
Write 110	100-20	110
	Write 80	80

To avoid that, a RW lock should be added, which makes the transactions executed in a serial order.

5.8. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1 , share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (i == 0 or 1) is shown in Figure 5.21. The other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```

Figure 5.21 The structure of process P_i in Dekker's algorithm.

Xi Xia
2016213482
2016215117

1. **mutual exclusion**: there will only be at most one process that is in the critical section. Others need to wait.
2. **progress**: It is implemented by the two variable “flag” and “return”. If one process wants to enter the critical section, it will set its flag as true and then wait for its turn.
3. **Bounded waiting**: Assume the two processes' flags are both true. If one process enters its critical section first, the other one will wait. When the first process exits, it’ll be the next process’s turn to execute. It won’t wait timelessly.

5.39. Exercise 4.22 asked you to design a multithreaded program that estimated pi using the Monte Carlo technique. In that exercise, you were asked to create a single thread that generated random points, storing the result in a global variable. Once that thread exited, the parent thread performed the calculation that estimated the value of pi. Modify that program so that you create several threads, each of which generates random points and determines if the points fall within the circle. Each thread will have to update the global count of all points that fall within the circle. Protect against race conditions on updates to the shared global variable by using mutex locks.

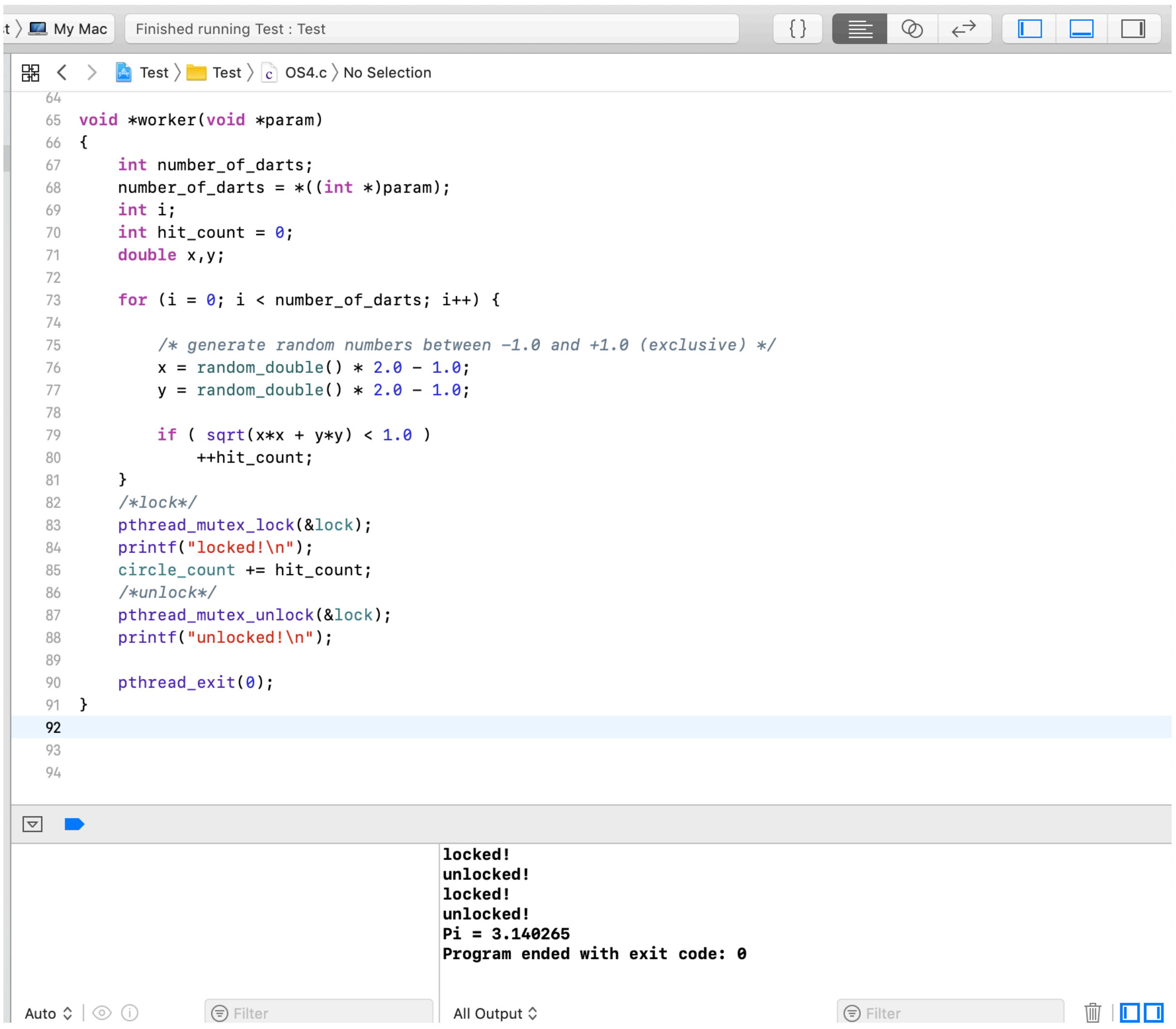
```c
//
//  OS5.c
//  Test
//
//  Created by 夏曦 on 2019/4/16.
//  Copyright © 2019 夏曦. All rights reserved.
//

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

void *worker(void *param);

#define NUMBER_OF_DARTS      50000000
#define NUMBER_OF_THREADS    2

/* the number of hits in the circle */
int circle_count = 0;
pthread_mutex_t lock;

/*
 * Generates a double precision random number
 */
double random_double()
{
    return random() / ((double)RAND_MAX + 1);
    //every call to rand gives a sudo random number between 0 and RAND_MAX
}

int main (int argc, const char * argv[]) {
    int darts_per_thread = NUMBER_OF_DARTS/ NUMBER_OF_THREADS;
    int i;

    double estimated_pi;

    /*mutex clock*/
    pthread_mutex_init(&lock,NULL);

```

Xi Xia
2016213482
2016215117

```c
    /*pthread_t is used to initialise the thread ID*/
    pthread_t workers[NUMBER_OF_THREADS];


    /* seed the random number generator */
    srandom((unsigned)time(NULL));

    /*creat threads*/
    for (i = 0; i < NUMBER_OF_THREADS; i++)
        pthread_create(&workers[i], 0, worker, &darts_per_thread);

    /*wait for the end of a thread, do resource recycling*/
    for (i = 0; i < NUMBER_OF_THREADS; i++)
        pthread_join(workers[i],NULL);

    /* estimate Pi */
    estimated_pi = 4.0 * circle_count / NUMBER_OF_DARTS;

    printf("Pi = %f\n",estimated_pi);

    return 0;
}

void *worker(void *param)
{
    int number_of_darts;
    number_of_darts = *((int *)param);
    int i;
    int hit_count = 0;
    double x,y;

    for (i = 0; i < number_of_darts; i++) {

        /* generate random numbers between -1.0 and +1.0 (exclusive) */
        x = random_double() * 2.0 - 1.0;
        y = random_double() * 2.0 - 1.0;

        if ( sqrt(x*x + y*y) < 1.0 )
            ++hit_count;
    }
    /*lock*/
    pthread_mutex_lock(&lock);
    printf("locked!\n");
    circle_count += hit_count;
    /*unlock*/
    pthread_mutex_unlock(&lock);
    printf("unlocked!\n");

    pthread_exit(0);
}
```

The highlight part is how the mutex lock works.
Results are shown as below:

Xi Xia
2016213482
2016215117



My Mac | Finished running Test : Test

Test 〉 Test 〉 OS4.c 〉 No Selection

```c

void *worker(void *param)
{
    int number_of_darts;
    number_of_darts = *((int *)param);
    int i;
    int hit_count = 0;
    double x,y;

    for (i = 0; i < number_of_darts; i++) {

        /* generate random numbers between -1.0 and +1.0 (exclusive) */
        x = random_double() * 2.0 - 1.0;
        y = random_double() * 2.0 - 1.0;

        if ( sqrt(x*x + y*y) < 1.0 )
            ++hit_count;
    }
    /*lock*/
    pthread_mutex_lock(&lock);
    printf("locked!\n");
    circle_count += hit_count;
    /*unlock*/
    pthread_mutex_unlock(&lock);
    printf("unlocked!\n");

    pthread_exit(0);
}
```

```
locked!
unlocked!
locked!
unlocked!
Pi = 3.140265
Program ended with exit code: 0
```

Auto | Filter | All Output | Filter