Programming Assignment 03

by Luke Hanks

Question 1

```c
/*============================================================*/
/* race.c --- for playing with ECE437                         */
/*============================================================*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

struct {
 int balance[2];
} Bank = {{100, 100}};  // global variable defined

void* MakeTransactions() {  // routine for thread execution
 int i, j, tmp1, tmp2, rint;
 double dummy;
 for (i = 0; i < 100; i++) {
   rint = (rand() % 30) - 15;
   if (((tmp1 = Bank.balance[0]) + rint) >= 0 &&
       ((tmp2 = Bank.balance[1]) - rint) >= 0) {
     Bank.balance[0] = tmp1 + rint;
     for (j = 0; j < rint * 1000; j++) {
       dummy = 2.345 * 8.765 / 1.234;
     }  // spend time on purpose
     Bank.balance[1] = tmp2 - rint;
   }
 }
 return NULL;
}

int main(int argc, char** argv) {
 int i;
 void* voidptr = NULL;
 pthread_t tid[2];
 srand(getpid());
 printf("Init balances A:%d + B:%d ⟹ %d!\n", Bank.balance[0],
        Bank.balance[1], Bank.balance[0] + Bank.balance[1]);
 for (i = 0; i < 2; i++) {
   if (pthread_create(&tid[i], NULL, MakeTransactions, NULL)) {
     perror("Error in thread creating\n");
     return (1);
   }
 }
 for (i = 0; i < 2; i++) {
   if (pthread_join(tid[i], (void*)&voidptr)) {
     perror("Error in thread joining\n");
     return (1);
```

```
    }
  }
  printf("Let's check the balances A:%d + B:%d ⟹ %d ?= 200\n",
         Bank.balance[0], Bank.balance[1], Bank.balance[0] + Bank.balance[1]);
  return 0;
 }
```

Compile then run the above code for 10-20 times. Write a paragraph to explain.

`void* MakeTransactions()` makes 100 random transactions under $15 between accounts A and B. The sum of the account balances should not change. But they do change. To understand why consider the following possible (and likely given that dummy for-loop) order of value assignments. Keep in mind that `Bank` is shared, but `temp1` and `temp2` are not.

```
//          Thread 0                        Thread 1
tmp1 = Bank.balance[0]
tmp2 = Bank.balance[1]
Bank.balance[0] = tmp1 + rint
                                        tmp1 = Bank.balance[0]
                                        tmp2 = Bank.balance[1]
                                        Bank.balance[0] = tmp1 + rint
Bank.balance[1] = tmp2 - rint
                                        Bank.balance[1] = tmp2 - rint
```

Thread 1's `temp1` value would reflect Thread 0's partially completed transaction, but Thread 1's `temp2` value would would not. This will result in Thread 0's effect on `Bank.balance[1]` being undone, but not Thread 0's effect on `Bank.balance[0]`. Money will appear to vanish and materialize at random.

Question 2

Use thread library calls (mutex lock and unlock) to modify the code in Q1) to remove any potential race conditions. Show your modification of the code and explain the outcome with your modification.

```
/*============================================================*/
/* raceWithMutex.c                                            */
/*============================================================*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t shared_mutex; //                          ⟵ MODIFICATION

struct {int balance[2];} Bank = {{100, 100}};

void* MakeTransactions() {  // routine for thread execution
  int i, j, tmp1, tmp2, rint;
  double dummy;
  for (i = 0; i < 100; i++) {
    rint = (rand() % 30) - 15;
    pthread_mutex_lock(&shared_mutex); //                 ⟵ MODIFICATION
    if (((tmp1 = Bank.balance[0]) + rint) ⩾ 0 &&
        ((tmp2 = Bank.balance[1]) - rint) ⩾ 0) {
```

```
      Bank.balance[0] = tmp1 + rint;
      for (j = 0; j < rint * 1000; j++) {
        dummy = 2.345 * 8.765 / 1.234;
      }  // spend time on purpose
      Bank.balance[1] = tmp2 - rint;
    }
    pthread_mutex_unlock(&shared_mutex); //                    ⟵ MODIFICATION
  }
  return NULL;
}

int main(int argc, char** argv) {
  int i;
  void* voidptr = NULL;
  pthread_t tid[2];
  srand(getpid());
  pthread_mutex_init(&shared_mutex, NULL); //                  ⟵ MODIFICATION
  printf("Init balances A:%d + B:%d ⟹ %d!\n", Bank.balance[0],
         Bank.balance[1], Bank.balance[0] + Bank.balance[1]);
  for (i = 0; i < 2; i++) {
    if (pthread_create(&tid[i], NULL, MakeTransactions, NULL)) {
      perror("Error in thread creating\n");
      return (1);
    }
  }
  for (i = 0; i < 2; i++) {
    if (pthread_join(tid[i], (void*)&voidptr)) {
      perror("Error in thread joining\n");
      return (1);
    }
  }
  pthread_mutex_lock(&shared_mutex); //                        ⟵ MODIFICATION
  printf("Let's check the balances A:%d + B:%d ⟹ %d ?= 200\n",
         Bank.balance[0], Bank.balance[1], Bank.balance[0] + Bank.balance[1]);
  return 0;
  pthread_mutex_unlock(&shared_mutex); //                      ⟵ MODIFICATION
}
```

The above code always maintains a consistent sum of account balances.

`shared_mutex` is a static variable so all the threads share it. The first thread to call `pthread_mutex_lock(&shared_mutex)` locks `shared_mutex` and executes its transaction. While that transaction is happening, the second thread to call `pthread_mutex_lock(&shared_mutex)` waits for `shared_mutex` to be unlocked. After the first thread finishes its transaction, it calls `pthread_mutex_unlock(&shared_mutex)` which unlocks `shared_mutex`. The second thread which has been waiting immediately locks `shared_mutex` and does its transaction.

Question 3

> Rewrite your code in Q1) replacing threads by processes.
>
> - Instead of creating two threads to call “MakeTransactions”, you will use fork() to create a child process. Both parent and child processes will call procedure “MakeTransactions”.
> - Since two processes will not share a common address space, you will need to rewrite code to allocate “Bank” as a shared variable (by applying shared memory IPC, see Slide M02c).
> - Other parts (i.e., set up initial values, print the initial values and balance, and print the ending values and balance) stay the same.

> Show your implementation code in the written report, compile then run your new process-based code for 10-20 times. Write a paragraph to explain if the race condition still exists.

Code

```c
/*============================================================*/
/* raceWithMutexAndProcesses.c                                */
/*============================================================*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

pthread_mutex_t shared_mutex;

struct {
  int balance[2];
} * Bank;  // global variable defined

void* MakeTransactions() {  // routine for thread execution
  int i, j, tmp1, tmp2, rint;
  double dummy;
  for (i = 0; i < 100; i++) {
    rint = (rand() % 30) - 15;
    pthread_mutex_lock(&shared_mutex);
    if (((tmp1 = Bank->balance[0]) + rint) >= 0 &&
        ((tmp2 = Bank->balance[1]) - rint) >= 0) {
      Bank->balance[0] = tmp1 + rint;
      for (j = 0; j < rint * 1000; j++) {
        dummy = 2.345 * 8.765 / 1.234;
      }  // spend time on purpose
      Bank->balance[1] = tmp2 - rint;
    }
    pthread_mutex_unlock(&shared_mutex);
  }
  return NULL;
}

int main(int argc, char** argv) {
  int i, shmid;
  void* voidptr = NULL;
  pthread_t tid[2];
  srand(getpid());
  pthread_mutex_init(&shared_mutex, NULL);

  if ((shmid = shmget(1234, 4, IPC_CREAT | 0666)) == -1) {
    perror("Error in getting shared memory segment\n");
    return 1;
  }
  Bank = shmat(shmid, NULL, 0);
  if (Bank == (void*)-1) {
```

```
    perror("Error in shared memory attach");
    return 1;
  }
  Bank->balance[0] = 100;
  Bank->balance[1] = 100;
  printf("Init balances A:%d + B:%d ⟹ %d!\n", Bank->balance[0],
         Bank->balance[1], Bank->balance[0] + Bank->balance[1]);
  pid_t pid = fork();
  if (pid < 0) {
    // Error
    perror("Error in forking\n");
    return (1);
  } else if (pid == 0) {
    MakeTransactions();
  } else {
    MakeTransactions();
    pthread_mutex_lock(&shared_mutex);
    printf("Let's check the balances A:%d + B:%d ⟹ %d ≥ 200\n",
           Bank->balance[0], Bank->balance[1],
           Bank->balance[0] + Bank->balance[1]);
    pthread_mutex_unlock(&shared_mutex);
  }
  if (shmdt(Bank) == -1) {
    perror("Error in shared memory detach");
    return 1;
  }
  return 0;
}
```

Output

```
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:25 + B:157 ⟹ 182 ≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:1 + B:205 ⟹ 206 ≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:187 + B:61 ⟹ 248 ≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:118 + B:140 ⟹ 258 ≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:156 + B:94 ⟹ 250 ≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:64 + B:97 ⟹ 161 ≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:30 + B:139 ⟹ 169 ≥ 200
./raceWithMutexAndProcesses
```

```
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:101 + B:116 ⟹ 217 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:8 + B:221 ⟹ 229 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:204 + B:79 ⟹ 283 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:14 + B:156 ⟹ 170 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:41 + B:172 ⟹ 213 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:288 + B:3 ⟹ 291 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:17 + B:176 ⟹ 193 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:9 + B:164 ⟹ 173 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:25 + B:195 ⟹ 220 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:296 + B:29 ⟹ 325 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:220 + B:59 ⟹ 279 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:141 + B:92 ⟹ 233 ?≥ 200
./raceWithMutexAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:233 + B:38 ⟹ 271 ?≥ 200
```

The race condition exists because the static `pthread_mutex_t shared_mutex` is not shared across processes, just threads.

Question 4

> Use semaphore system calls to modify your code in Q3 in order to remove any potential race conditions. Show your modification of the code and explain the outcome with your modification.

Code

```c
/*============================================================*/
/* raceWithSemaphoresAndProcesses.c                           */
/*============================================================*/
#include <fcntl.h>
#include <pthread.h>
```

```c
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

sem_t* shared_mutex;

struct {
  int balance[2];
} * Bank;  // global variable defined

void* MakeTransactions() {  // routine for thread execution
  int i, j, tmp1, tmp2, rint;
  double dummy;
  for (i = 0; i < 100; i++) {
    rint = (rand() % 30) - 15;
    if (sem_wait(shared_mutex) < 0) {
      perror("Error in sem_wait()");
      continue;
    }
    if (((tmp1 = Bank->balance[0]) + rint) >= 0 &&
        ((tmp2 = Bank->balance[1]) - rint) >= 0) {
      Bank->balance[0] = tmp1 + rint;
      for (j = 0; j < rint * 1000; j++) {
        dummy = 2.345 * 8.765 / 1.234;
      }  // spend time on purpose
      Bank->balance[1] = tmp2 - rint;
    }
    while (sem_post(shared_mutex) < 0) {
      perror("Error in sem_post()");
    }
  }
  return NULL;
}

int main(int argc, char** argv) {
  int i, shmid_bank, shmid_mutex;
  void* voidptr = NULL;
  pthread_t tid[2];
  srand(getpid());

  shared_mutex = sem_open("/mutex_sem", O_CREAT, 0666, 1);

  if ((shmid_bank = shmget(1234, 4, IPC_CREAT | 0666)) < 0) {
    perror("Error in getting shared memory segment\n");
    return 1;
  }
  Bank = shmat(shmid_bank, NULL, 0);
  if (Bank == (void*)-1) {
    perror("Error in shared memory attach");
    return 1;
  }
```

```
  Bank->balance[0] = 100;
  Bank->balance[1] = 100;
  printf("Init balances A:%d + B:%d ⟹ %d!\n", Bank->balance[0],
         Bank->balance[1], Bank->balance[0] + Bank->balance[1]);
  pid_t pid = fork();
  if (pid < 0) {
    perror("Error in forking\n");
    return (1);
  } else if (pid == 0) {
    shared_mutex = sem_open("/mutex_sem", O_RDWR);
    MakeTransactions();
  } else {
    shared_mutex = sem_open("/mutex_sem", O_RDWR);
    MakeTransactions();
    if (sem_wait(shared_mutex) < 0) {
      perror("Error in sem_wait()");
    }
    printf("Let's check the balances A:%d + B:%d ⟹ %d ≥ 200\n",
           Bank->balance[0], Bank->balance[1],
           Bank->balance[0] + Bank->balance[1]);
    while (sem_post(shared_mutex) < 0) {
      perror("Error in sem_post()");
    }
  }

  if (shmdt(Bank) < 0) {
    perror("Error in shared memory detach");
    return 1;
  }
  sem_unlink("/mutex_sem");
  sem_destroy(shared_mutex);
  return 0;
}
```

Output

```
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:25 + B:175 ⟹ 200 ≥ 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:4 + B:196 ⟹ 200 ≥ 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:13 + B:187 ⟹ 200 ≥ 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:137 + B:63 ⟹ 200 ≥ 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:38 + B:162 ⟹ 200 ≥ 200
```

```
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:30 + B:170 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:26 + B:174 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:197 + B:3 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:5 + B:195 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:103 + B:97 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:52 + B:148 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:174 + B:26 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:22 + B:178 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:4 + B:196 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:45 + B:155 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:61 + B:139 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:158 + B:42 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:121 + B:79 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:18 + B:182 ⟹ 200 ?= 200
./raceWithSemaphoresAndProcesses
Init balances A:100 + B:100 ⟹ 200!
Let's check the balances A:97 + B:103 ⟹ 200 ?= 200
```