International Journal of Engineering, Management, Humanities and Social Sciences Paradigms (Volume 31, Special Issue of January 2019)
ISSN (Online): 2347-601X and Website: www.ijemhs.com

Race Conditions: A Case Study

Dr. Prabhash Nanda, [2] Mr. Pradosh Ranjan Parida
[1*] *Associater Professor, Dept. Of Electrical Engineering, NIT BBSR,*
Asst. Professor DEPT. of Electrical Engineering, NIT BBSR,
[1]pranhas@thenalanda.com, pradoshranjan@thenalanda.com

ABSTRACT

There is no effective technique that can aid in the detection of race conditions in a programme since doing so in a multithreaded or multiprocess application is an NP-hard problem. Thus, there are no instructional instruments that are simple to use. The majority of textbooks on concurrent programming and operating systems merely offer a formal definition and a few unimportant examples. This is insufficient for students to learn how to detect race conditions. This paper attempts to fill this gap by presenting a set of well- organized examples, each of which contains one or more race conditions, for instructors to use in the classroom. This set of materials has been classroom tested for two years and the student's reaction has been very positive.

1. INTRODUCTION

Race condition detection is an important topic in an operating systems or concurrent programming course [1,2,5,9-13]. Our experience shows that it is easy to provide students with a formal definition; but it is always difficult for students to pinpoint race conditions in their programs [9,10]. This is largely due to the lack of realistic examples and the dynamic behavior of a multithreaded or multiprocess program. Worse, race conditions cannot be detected at run time because a detection program must monitor every memory access. Additionally, statically detecting race conditions in programs that use multiple semaphores is NP-hard [7], meaning an efficient solution is unlikely. If the synchronization mechanism is weaker than semaphores, an exact and efficient algorithm can be found [6]; otherwise, only heuristic algorithms that scan the source programs statically are available [3,4]. Unfortunately, a heuristic algorithm can only find *potential* race conditions, meaning the detection program may report many race conditions that are *not* actually race conditions. As a result, there are few pedagogical aids designed for teaching students about race conditions. Since there are no reasonable algorithms and universally applicable techniques that can help students pinpoint race conditions, they are left frustrated trying to debug their programs.

A *race condition* is defined as the situation in which multiple threads or processes read and write a shared data item and the final result depends on the order of execution. An obvious example is updating a shared counter as follows:

```
int  count = 0;

Thread_A(…)                     Thread_B(…)
{                               {
   ………                             ………
   count++;                        count--;
   ………                             ………
}                               }
```

Unfortunately, this example only illustrates the most obvious effect of a race condition. Many race conditions that appear in student programs are subtle and very difficult to find. To help students pinpoint race conditions, we have developed a sequence of non-trivial examples. These examples originate from an exam problem that asks students to design a program that permits threads in two groups to exchange integer messages. We anticipated that our students could apply what they learned in class (*e.g.*, the bounded-buffer problem) to solve this problem; however, most of them attempted to reinvent the wheel and came up with all kinds of correct and incorrect solutions. Most incorrect ones are due to race conditions. We believe that discussing these incorrect solutions will provide our students with an opportunity to learn more about pinpointing race conditions. This set of materials has become part of our lecture notes in an introduction to operating systems course for two years with a very positive impact. In this paper, we share these materials with other educators. In the following, Section 2 provides the problem statement, Section 3 to Section 6 discuss four attempts in the order of increasing complexity of the "solution," Section 7 presents the line of thinking using the bounded-buffer problem in order to reach a correct solution, Section 8 polishes this solution to make it more efficient, and, finally,Section 9 contains our conclusion.

2. PROBLEM STATEMENT

Suppose we have two groups of threads **A** and **B**. Each thread in **A** (*resp*., **B**) runs a function `Thread_A()` (*resp*., `Thread_B()`). Both `Thread_A()` and `Thread_B()` contain an infinite loop in which a thread exchanges an integer message with a thread in the other group. Thus, `Thread_A()` and `Thread_B()` have a structure as follows:

```
Thread_A(…)                     Thread_B(…)
{                               {
   while (1) {                     while (1) {
      ………                             ………
      Ex. Message;                    Ex. Message;
      ………                             ………
   }                               }
}                               }
```

There are two important notes. First, once an instance **A** of `Thread_A()` makes a message available, **A** can continue only if it receives a message from an instance **B** of `Thread_B()` who has successfully retrieved **A**'s message. Similarly, an instance **B** of `Thread_B()` can continue only if it receives a message from **A** rather than from any other threads in group **A**. Second, once an instance $\mathbf{A}_1$ of `Thread_A()` makes its message available, we have to make sure that the next instance $\mathbf{A}_2$ of

Thread_A(), which might come a little later, will not overwrite the existing message before it is retrieved by an instance of Thread_B().

Each of the four attempts to be discussed below will contain an execution sequence that can correctly perform a message exchange. However, since there are data items shared by all involved threads, a race condition occurs if we can find an alternative execution sequence that does not correctly exchange messages. Moreover, there is no difference between the use of threads and the use of processes. We choose threads because multithreaded programming is part of our operating systems course [9].

3. FIRST ATTEMPT

The idea of this attempt is quite simple: threads shake hands and exchange messages. It uses two semaphores A and B, with initial values 0. When Thread_A() arrives at the message exchange section, it uses **Signal**(B) to tell Thread_B() that it is ready and then waits for Thread_B()'s reply. Once this signal comes, Thread_A() continues, and Thread_B() should already be there for message exchange. Thus, the **Signal**/**Wait** sequence simulates a hand-shaking protocol. In the message exchange section, Thread_A() copies its message into Buf_A for Thread_B() to retrieve and then copies Thread_B()'s message from Buf_B into its local variable Var_A.

```
semaphore  A = 0, B = 0;
int        Buf_A, Buf_B;

Thread_A(…)                     Thread_B(…)
{                               {
   int  Var_A;                     int  Var_B;

   while (1) {                     while (1) {
      ……….                            ……….
      Var_A = …;                      Var_B = …;
      Signal(B);                      Signal(A);
      Wait(A);                        Wait(B);
      Buf_A = Var_A;                  Buf_B = Var_B;
      Var_A = Buf_B;                  Var_B = Buf_A;
      ……….                            ……….
   }                               }
}                               }
```

The following execution sequence shows a typical race condition, which is caused by grabbing the value of a shared variable too fast *before* it can be filled with a new value. The first two rows indicate that **A** reaches **Wait**(A) and is switched out. Then, **B** comes in, executes **Wait**(B), and is switched out. This causes **A** to continue and move its message from Var_A to Buf_A; **A** then copies **B**'s message from Buf_B to Var_A. However, since **B** has not yet reached the statement that fills

`Buf_B`, the content in `Buf_B` that **A** retrieves is the previous message. This is a race condition.

***Thread* A**	***Thread* B**

Signal(B)	
Wait(A)	
	Signal(A)
	Wait(B)
Buf_A = Var_A	
Var_A = Buf_B	
	Buf_B = Var_B

The following execution sequence shows another typical race condition in which two threads in group **A** may exchange messages with the same thread in group **B**. As a result, we cannot be sure what message thread **B** will receive. $\mathbf{A}_1$'s signal causes $\mathbf{B}_1$ to pass through **Wait**(B), and $\mathbf{B}_1$'s signal makes $\mathbf{A}_1$ pass through **Wait**(A). Thus, $\mathbf{A}_1$ and $\mathbf{B}_1$ have a match and are supposed to exchange their messages. However, right after these two waits, $\mathbf{A}_2$ comes into the scene and executes **Signal**(B) and **Wait**(A), which makes $\mathbf{B}_2$ execute **Signal**(A) to release $\mathbf{A}_2$ from **Wait**(A). Thus, $\mathbf{A}_1$ and $\mathbf{A}_2$ can put different messages into the shared variable Buf_A and we have a race condition. By changing the order of execution, one can easily find other race conditions.

Thread $\mathbf{A}_1$	***Thread*** $\mathbf{A}_2$	***Thread*** $\mathbf{B}_1$	***Thread*** $\mathbf{B}_2$
Signal(B)			
Wait(A)			
		Signal(A)	
		Wait(B)	
	Signal(B)		
	Wait(A)		
		Buf_B = …	
			Signal(A)
Buf_A = …			
	Buf_A = …		

Lesson learned: When a variable is shared by many threads, without a proper mutual exclusion protection, race conditions are likely to occur. In both execution sequences above, messages received may not be the correct ones.

4. SECOND ATTEMPT

Let us use a semaphore Mutex, with initial value 1, to protect the shared variables. This makes sure that the access to Buf_A and Buf_B is mutually exclusive. Before a thread can exchange a message, it follows the hand-shaking protocol in the first attempt, and adds its own message into a shared variable. Then, it performs a second hand-shaking protocol to receive a message from a thread in the other group.

```
semaphore  A = 0, B = 0;
semaphore  Mutex = 1;
int        Buf_A, Buf_B;
```

```
Thread_A(…)                    Thread_B(…)
{                              {
   int  Var_A;                    int  Var_B;
```

```
        while (1) {                   while (1) {
            .........                     .........
            Signal(B);                    Signal(A);
            Wait(A);                      Wait(B);
               Wait(Mutex);                  Wait(Mutex);
                  Buf_A = Var_A;                Buf_B = Var_B;
               Signal(Mutex);                Signal(Mutex);
            Signal(B);                    Signal(A);
            Wait(A);                      Wait(B);
               Wait(Mutex);                  Wait(Mutex);
                  Var_A = Buf_B;                Var_B = Buf_A;
               Signal(Mutex);                Signal(Mutex);
            .........                     .........
        }                             }
    }                             }
```

The use of semaphore `Mutex` prevents two threads in group **A** from accessing `Buf_A` and `Buf_B` at the same time. However, this protection is inadequate. Once **A** and **B** complete the first stage of message exchange and signal each other, the values of semaphores `A` and `B` are both 1s. Consequently, we cannot be sure if (1) **A** and **B** will continue with the second stage of message exchange, (2) another pair of threads will start their first stage, or (3) one of the current pair will continue and exchange a message with a newcomer in the other group. All of these possibilities can cause race conditions. The following execution sequence shows a race condition of (3). Right after $\mathbf{A}_1$ and **B** make their messages available, $\mathbf{A}_2$ starts its first stage and signals and waits. Then, **B** enters its second stage and signals and waits. This may release $\mathbf{A}_2$ rather than $\mathbf{A}_1$. As a result, $\mathbf{A}_2$'s message overwrites $\mathbf{A}_1$'s and we have a race condition.

Thread $\mathbf{A}_1$	*Thread* $\mathbf{A}_2$	*Thread* **B**
Signal(B)		
Wait(A)		
		Signal(A)
		Wait(B)
Buf_A = …		
		Buf_B = …
	Signal(B)	
	Wait(A)	
		Signal(A)
		Wait(B)
	Buf_A = …	

Lesson learned: Protecting each shared variable separately may be insufficient if the use of that variable is part of a long execution sequence. Protect the whole execution rather than each individual variable.

5. THIRD ATTEMPT

Because a thread may come in and ruin a message before the previous message exchange completes, we need to expand the critical section so that it can cover the *complete* message exchange section. In the attempt below, semaphore Aready (*resp*., Bready), with initial value 1, is used to block any other **A**'s (*resp*., **B**'s) from performing a message exchange if there is a **A** (*resp*., **B**) exchanging a message. We cannot use only one semaphore in both groups, because a deadlock may occur (Section 7). Semaphore Adone (*resp*., Bdone) is used to inform a **B** (*resp*., **A**) that a message is there. Thus, a thread **A** waits until Buf_A is available, deposits a message, informs a **B** that a message is there with **Signal**(Adone), waits on semaphore Bdone until a **B** deposits its message, takes the message, and finally releases the message exchange critical section.

```
semaphore Aready = 1, Bready = 1;
semaphore Adone  =  0,  Bdone  =  0;
int  Buf_A, Buf_B;

Thread_A(…)                       Thread_B(…)
{                                 {
   int  Var_A;                       int  Var_B;

   while (1) {                       while (1) {
      ……….                              ……….
      Wait(Aready);                     Wait(Bready);
         Buf_A = Var_A;                    Buf_B = Var_B;
         Signal(Adone);                    Signal(Bdone);
         Wait(Bdone);                      Wait(Adone);
         Var_A = Buf_B;                    Var_B = Buf_A;
      Signal(Aready);                   Signal(Bready);
      ……….                              ……….
   }                                 }
}                                 }
```

Does this attempt work? No! Suppose both **A** and **B** successfully deposit their messages and reach the second wait. At this point, semaphores Adone and Bdone are both 1's. Assume that **A** passes through **Wait**(Bdone), takes the message from Buf_B, executes **Signal**(Aready) to indicate the completion of a message exchange of **A**, and then loops back. If this **A** or another **A** is lucky enough to pass through this **Wait**(Aready) and deposits a new message into Buf_A before any **B** can retrieve the previous one, we lose a message and a race condition occurs.

***Thread* A**	***Thread* B**
Buf_A = …	
Signal(Adone)	
Wait(Bdone)	
	Signal(Bdone)
	Wait(Adone)
… = Buf_B	
Signal(Aready)	
…. loop back …	
Wait(Aready)	

`Buf_A = …`	
	`… = Buf_A`

Lesson learned: If we have a number of cooperating thread groups, mutual exclusion guaranteed for one group may not prevent threads in other groups from interacting with a thread in the group, even though the latter thread still is in its critical section. Think globally when setting up mutual exclusion.

6. FOURTH ATTEMPT

The critical sections in the third attempt are not good enough because they cannot block threads in the same group from rushing in and overwriting the existing message before it is taken. So, we might want to force a thread in group **A** (*resp*., group **B**) to wait until a thread in group **B** (*resp*., group **A**) completes its task. The following is an attempt similar to the previous one, except that a different hand-shaking protocol is used and that message exchange happens *within* this hand-shaking protocol.

```
semaphore Aready = 1, Bready = 1;
semaphore Adone = 0, Bdone = 0;
int Buf_A, Buf_B;
```

```
Thread_A(…)
{
   int Var_A;

   while (1) {
      ……….
      Wait(Bready);
         Buf_A = Var_A;
         Signal(Adone);
         Wait(Bdone);
         Var_A = Buf_B;
      Signal(Aready);
      ……….
   }
}
```

```
Thread_B(…)
{
   int Var_B;

   while (1) {
      ……….
      Wait(Aready);
         Buf_B = Var_B;
         Signal(Bdone);
         Wait(Adone);
         Var_B = Buf_A;
      Signal(Bready);
      ……….
   }
}
```

In the following execution sequence, right after $\mathbf{A}_1$ deposits its message into `Buf_A` and informs **B**, **B** retrieves the message, and signals the semaphore `Bready`. This permits $\mathbf{A}_2$ to start a message exchange. However, $\mathbf{A}_2$ may run faster than $\mathbf{A}_1$ does and retrieve the message that is supposed to be retrieved by $\mathbf{A}_1$. Therefore, we have a race condition.

Thread $\mathbf{A}_1$	*Thread* $\mathbf{A}_2$	*Thread* **B**
Wait(Bready)		
Buf_A = …		
Signal(Adone)		
		Signal(Bdone)
		Wait(Adone)
		… = Buf_A
		Signal(Bready)

	Wait(Bready)	
	………	
	Wait(Bdone)	
	… = Buf_B	

Lesson learned: Mutual exclusion is important! If the lock for mutual exclusion is not released by its owner, race conditions are likely to occur. In the above, the lock `Bready` (*resp*., `Aready`) is acquired by a thread in group **A** (*resp*., **B**) and released by a thread in group **B** (resp., **A**). This is a very risky programming practice if mutual exclusion is the central concern.

7. A GOOD ATTEMPT

Some may notice that this problem is a variation of the bounded-buffer problem, also known as the producer-consumer problem, because a thread in group **A** puts an integer into a buffer for a thread **B** to retrieve and then waits for a message from a thread in group **B**. This is a good observation; however, we still have two questions that need to be answered. First, how many buffers are required? Second, how many slots are in each buffer? An obvious answer to the first question is two buffers, one for a thread in **A** (producer) sending an integer to a thread in **B** (consumer) and the other for a thread in **B** (producer) sending an integer to a thread in **A** (consumer). As for the second question, consider the way of sending and receiving a message. Because there is no ordering assumption for releasing threads from a synchronization primitive, if a buffer has more than one slot, we cannot guarantee that the message sent by a thread in group **B**, who received a message from a thread in group **A**, will be received by that thread in group **A**. Therefore, the number of slots in each buffer should be exactly one!

```
int  Buf_A, Buf_B;

Thread_A(…)                        Thread_B(…)
{                                  {
   int  Var_A;                        int  Var_B;

   while (1) {                        while (1) {
      ……….                               ……….
      PUT(Var_A,Buf_A);                  PUT(Var_B,Buf_B);
      GET(Var_A,Buf_B);                  GET(Var_B,Buf_A);
      ……….                               ……….
   }                                  }
}                                  }
```

The above code reflects this idea, where `PUT(a,b)` means adding the value of `a` into a one-slot buffer `b` and `GET(a,b)` means retrieving a value from a one-slot buffer `b` into `a`. However, this is *not* a correct solution as demonstrated by the following execution sequence: (1) $\mathbf{A}_1$ and **B** both successfully execute their `PUT()` calls, (2) **B** executes its `GET()` to retrieve $\mathbf{A}_1$'s message, which causes $\mathbf{A}_2$ to execute its `PUT()` call, and (3) $\mathbf{A}_2$ continues and retrieves the message which is supposed to be received by $\mathbf{A}_1$. A critical section may be used to make sure that while **A** and **B** are exchanging messages no other threads can enter (the third attempt). There are two possibilities: (1) a single semaphore to enforce mutual exclusion for all threads in *both* groups **A** and **B**, or (2) two semaphores, one for each group. The first option is not a good idea as shown below.

```
semaphore  Mutex = 1;
int        Buf_A, Buf_B;
```

```
Thread_A(…)                        Thread_B(…)
{                                  {
   int  Var_A;                        int  Var_B;

   while (1) {                        while (1) {
      ……….                               ……….
      Wait(Mutex);                       Wait(Mutex);
         PUT(Var_A,Buf_A);                  PUT(Var_B,Buf_B);
         GET(Var_A,Buf_B);                  GET(Var_B,Buf_A);
      Signal(Mutex);                     Signal(Mutex);
      ……….                               ……….
   }                                  }
}                                  }
```

Suppose thread **A** successfully passes through `Wait(Mutex)` and calls `PUT(Var_A, Buf_A)` to deposit its message. Because **A** is the only thread that owns the lock (*i.e.*, in its critical section), no other **A**'s and **B**'s can enter, and, as a result, `Buf_B` contains no message from a thread in group **B**. Hence, **A** and none of the other threads can continue and the whole system locks up. Because of this problem, we use two semaphores:

```
semaphore  Amutex = 1, Bmutex = 1;
int        Buf_A, Buf_B;

Thread_A(…)                        Thread_B(…)
{                                  {
   int  Var_A;                        int  Var_B;

   while (1) {                        while (1) {
      ……….                               ……….
      Wait(Amutex);                      Wait(Bmutex);
         PUT(Var_A,Buf_A);                  PUT(Var_B,Buf_B);
         GET(Var_A,Buf_B);                  GET(Var_B,Buf_A);
      Signal(Amutex);                    Signal(Bmutex);
      ……….                               ……….
   }                                  }
}                                  }
```

This solution is an extension to the third attempt (Section 5). Does this solution work? Yes, it works. To prove this, we need to address the following issues: (1) the message exchange section is mutually exclusive within the thread group, (2) once two threads enter their critical section, they exchange messages without the interference from any other threads, and (3) after one thread exits its critical section, no thread in the same group can rush in and ruin the existing message. Because of the use of semaphores `Amutex` and `Bmutex`, (1) holds. Once **A** and **B** execute `Signal(Amutex)` and `Signal(Bmutex)`, their messages have been exchanged successfully. This addresses point (2). As for (3), assume that **A** exits its critical section while **B** is still in its critical section. Because **A** exits, **A** must have retrieved **B**'s message, meaning **B** has completed its `PUT()` call. However, before **B** can successfully complete its `GET()` call, buffer `Buf_A` is not empty, and, consequently, any new **A** that passes through `Wait(Amutex)` is blocked by the `PUT()` call until **B** completes its `GET()` call. Therefore, no other **A**'s can ruin the message before it is retrieved by **B**. We shall call this the *symmetric* solution.

The following is a *non-symmetric* solution. It forces a sequential execution of the following activities: (1) **A** deposits a message, (2) **B** receives **A**'s message, (3) **B** deposits a message, and (4) **A** receives **B**'s message. Because **A** and **B** are in their critical sections, this sequence is not interrupted by any other threads in **A** and in **B**. Thus, before the completion of a message exchange, threads **A** and **B** that are exchanging messages will not be interrupted, and message exchange is correctly implemented.

```
semaphore  Amutex = 1, Bmutex = 1;
int        Buf_A, Buf_B;

Thread_A(…)                         Thread_B(…)
{                                   {
   int  Var_A;                         int  Var_B, Temp;

   while (1) {                         while (1) {
      ………                                 ………
      Wait(Amutex);                       Wait(Bmutex);
         PUT(Var_A,Buf_A);                   GET(Temp,Buf_A);
         GET(Var_A,Buf_B);                   PUT(Var_B,Buf_B);
      Signal(Amutex);                     Signal(Bmutex);
      ………                                 ………
   }                                   }
}                                   }
```

Lesson learned: Review the solutions to well-known problems, because a correct solution to the problem in hand may be a variation of a well-known problem. Classic problems are designed to illustrate frequently encountered problems and their solutions.

8. A MINOR VARIATION BUT MORE EFFICIENT DESIGN

The solutions discussed in the previous section look simple; however, it is not very efficient. We can count at least three semaphore waits for each message exchange, one for waiting on semaphore `Amutex`, one for waiting on the first buffer until it is not full, and one for waiting on the second buffer until it is not empty. Since more semaphore wait means less program efficiency, we shall polish these solutions to make them more efficient in the following two subsections.

Polishing the Symmetric Solution

For buffer `Buf_A` (*resp*., `Buf_B`), two semaphores `NotFull_A` and `NotEmpty_A` (*resp*., `NotFull_B` and `NotEmpty_B`) are required. Semaphore `NotFull_A` (*resp*., `NotFull_B`) blocks threads (*i.e*., producers) when buffer `Buf_A` (*resp*., `Buf_B`) is full, and semaphore `NotEmpty_A` (*resp*., `NotEmpty_B`) blocks threads (*i.e*., consumers) when buffer `Buf_A` (*resp*., `Buf_B`) is empty.

Replacing the `PUT()` and `GET()` calls yields the following solution:

```
semaphore  Amutex = 1, Bmutex = 1;
semaphore  NotFull_A  = NotFull_B  = 1;
```

```
semaphore  NotEmpty_A = NotEmpty_B = 0;
int        Buf_A, Buf_B;

Thread_A(…)                  Thread_B(…)
{                            {
  int  Var_A;                  int  Var_B;

  while (1) {                  while (1) {
    ……….                         ……….
    Wait(Amutex);                Wait(Bmutex);
      Wait(NotFull_A);             Wait(NotFull_B);
        Buf_A = Var_A;               Buf_B = Var_B;
        Signal(NotEmpty_A);          Signal(NotEmpty_B);
      Wait(NotEmpty_B);            Wait(NotEmpty_A);
        Var_A = Buf_B;               Var_B = Buf_A;
        Signal(NotFull_B);           Signal(NotFull_A);
    Signal(Amutex);              Signal(Bmutex);
    ……….                         ……….
  }                            }
}                            }
```

Polishing the Non-Symmetric Solution

Since **A** deposits its message into a buffer for **B** to retrieve, once **B** takes this message, **B** can deposit its message into the same buffer for **A** to retrieve. Thus, one buffer is sufficient, and is denoted as `Shared`.

Let us first concentrate on the buffer operations. A thread in group **A** needs a semaphore `NotFull` to control whether `Shared` is full or not. Because threads in group **A** are allowed to deposit messages first, the initial value of `NotFull` is one. After depositing a message, a thread in group **A** must notify a thread in group **B** to proceed. We shall use a semaphore `NotEmpty_A` for this purpose. Because threads in group **B** must wait until notified, the initial value of `NotEmpty_A` must be zero. After this notification, this thread in group **A** must wait until **B**'s message becomes available. Thus, a third semaphore `NotEmpty_B`, with initial value zero, is used for this purpose. On the other hand, a thread in group **B** waits until **A**'s message arrives, retrieves this message, deposits its own message, and notifies the waiting **A** to continue. Thus, the message exchange sections look like the following:

```
Wait(NotFull);
   Shared = Var_A;
   Signal(NotEmpty_A);   Wait(NotEmpty_A);
                            Temp = Shared;
                            Shared = Var_B;
   Wait(NotEmpty_B);     Signal(NotEmpty_B);
   Var_A = Shared;
Signal(NotFull);
```

Are semaphores `Amutex` and `Bmutex` necessary? Because the initial value of `NotFull` is one, only one thread in group **A** can pass through **`Wait`**`(NotFull)` and hence mutual exclusion among threads in group **A** is guaranteed. Because the initial value of `NotEmpty_A` is zero and `NotEmpty_A` is only signaled once by a thread in group **A** in its critical section, the value of `NotEmpty_A` is either zero or one. Hence, no more than one thread in group **B** can pass through

`Wait(NotEmpty_A)` and mutual exclusion among threads in group **B** is also guaranteed. As a result, semaphore `Amutex` and `Bmutex` are redundant. The following is a complete solution:

```
semaphore  NotFull = 1, NotEmpty_A = NotEmpty_B = 0;
int        Shared;

Thread_A(…)                        Thread_B(…)
{                                  {
   int  Var_A;                        int  Var_B, Temp;

   while (1) {                        while (1) {
      ………                                ………
      Wait(NotFull);
         Shared = Var_A;
         Signal(NotEmpty_A);             Wait(NotEmpty_A);
                                            Temp = Shared;
                                            Shared = Var_B;
         Wait(NotEmpty_B);               Signal(NotEmpty_B);
         Var_A = Shared;
      Signal(NotFull);
      ………                                ………
   }                                  }
}                                  }
```

A Simple Comparison

Both correct solutions, especially the non-symmetric one, are no more complex than the incorrect ones. The symmetric version has six statements in each critical section, and the non-symmetric one has four in `Thread_A()`'s critical section and two in `Thread_B()`'s. However, because the statements in the non-symmetric version are executed sequentially, there are actually six statements. Hence, in terms of statements count, both versions are similar. Since the symmetric solution has three waits and the non-symmetric one has only two, in terms of synchronization efficiency, the non-symmetric version is better. On the other hand, since the message exchange sections are identical in both thread groups, the symmetric version may be easier to understand.

9. CONCLUSIONS

Despite the fact that learning how to identify race conditions is a crucial component of operating systems and concurrent programming courses, the majority of textbooks just offer a concept and a few unimportant examples without going any further. Furthermore, no software can precisely identify those race situations in a programme since detecting race conditions is an NP-hard job. As a result, it might be challenging for students to identify potential, particularly subtle, racial conditions in their programmes. This paper gives a series of well-organized examples based on a straightforward problem to handle this issue. Each of these examples includes one or more race conditions.

These examples share a single design merit: how to set up mutual exclusion among threads in different groups. The first example shows a naive idea that a simple hand-shaking can solve the problem without noticing that the shared buffers are the best places for race conditions to occur. The second example tries to protect each individual buffer. This is a common beginner problem that only focuses on the shared data rather than the combined effects of the program execution and the shared data items. After two attempts, students might realize that the protection should be extended to cover the whole message exchange section. However, reaching a correct solution is still not easy as shown by the third and fourth attempts. So, what is the major problem? Students learn the solutions to classic problems such as the bounded-buffer problem, the smoker problem, the philosophers problem, the readers-writers problem and so on without recognizing the merit of each solution. As a result, each of these solutions remains *the* solution of that particular problem instead of using it as a vehicle for solving other problems (*i.e.*, seeing the trees without seeing the forest). Once we point out that this message exchange problem has a very simple solution that they have already learned in class, many students can immediately solve this problem without any difficulty. This is exactly the lesson we want to tell our students: understand the solutions to the classic problems and extract the idea and merit so that these solutions can be used in other applications. Finally, we analyze a simple idea (Section 7) and polish its solutions to use semaphores only. Then, students realize that they reinvented the wheel and that some of their incorrect solutions are very close to the correct ones except for the presence of race conditions. Through the use of these materials, our students have become more confident and more capable in dealing with race conditions in their programs. We hope this set of materials will help other educators who are teaching similar courses and are looking for more examples.

This paper is part of our on-going NSF concurrent computing project. The interested readers can find more information and software tools for teaching multithreaded programming at `http://www.cs.mtu.edu/~shene/NSF-3/index.html`.

REFERENCES

1. Gregory R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.
2. Alan Burns and Geoff Davies, *Concurrent Programming*, Addison-Wesley, 1993.
3. Jong-Deok Choi and Sang Lyul Min, RACE FRONTIER: Reproducing Data Races in Parallel-Program Debugging, *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, April 1991, pp. 145-154.
4. Anne Dinning and Edith Schonberg, Detecting Access Anomalies in Programs with Critical Sections, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991, pp. 85-96.
5. Stephen J. Hartley, *Concurrent Programming: The Java Programming Language*, Oxford University Press, 1998.

6. Robert H. B. Netzer and S. Ghosh, Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization, *International Conference on Parallel Processing*, August 1992, pp. II242-II246.
7. Robert H. B. Netzer and Barton P. Miller, On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions, *International Conference on Parallel Processing*, August 1990, pp. II93-II97.
8. Robert H. B. Netzer and Barton P. Miller, Improving the Accuracy of Data Race Detection, *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, April 1991, pp. 133-144.
9. Ching-Kuang Shene, Multithreaded Programming in an Introduction to Operating Systems Course, *ACM Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, February 1998, pp. 242-246.
10. Ching-Kuang Shene and Steve Carr, The Design of a Multithreaded Programming Course and Its Accompanying Software Tools, *The Journal of Computing in Small College*, Vol. 14 (1998), No. 1, pp. 12-24.
11. Abraham Silberschatz and Peter B. Galvin, *Operating System Concepts*, 5^{th} edition, Addison-Wesley, 1998.
12. William Stallings, *Operating Systems*, 4^{th} edition, Prentice Hall, 2001.
13. Andrew S. Tanenbaum, *Modern Operating Systems*, 2^{nd} edition, Prentice Hall, 2001.