Race Conditions: A Case Study

Steve Carr, Jean Mayo and Ching-Kuang Shene
Department of Computer Science
Michigan Technological University
1400 Townsend Drive
Houghton, MI 49931-1295

Project supported by the National Science Foundation under
grant DUE-9752244 and grant DUE-9984682
What is a Race Condition?

- When *two or more* processes/threads access a *shared* data item, the computed result depends on the order of execution.

- There are three elements here:
  - Multiple processes/threads
  - Shared data items
  - Results may be different if the execution order is altered
A Very Simple Example

Current value of Count is 10

Process #1
Count++;
LOAD Count
ADD #1
STORE Count

Process #2
Count--;
LOAD Count
SUB #1
STORE Count

We have no way to determine what the value Count may have.
Why is *Race Condition* so *Difficult* to Catch?

- *Statically* detecting race conditions in a program using multiple semaphores is NP-complete.
- Thus, no efficient algorithms are available. We have to use our debugging skills.
- It is virtually impossible to catch race conditions *dynamically* because the hardware must examine *every* memory access.
How about our students?

- Normally, they do not realize/believe their programs do have race conditions.
- They claim their programs work, because their programs respond to input data properly.
- It takes time to convince them, because we have to trace their programs carefully.
- So, we developed a series of examples to teach students how to catch race conditions.
Problem Statement

- Two groups, A and B, of threads exchange messages.
- Each thread in A runs a function \( T_A() \), and each thread in B runs a function \( T_B() \).
- Both \( T_A() \) and \( T_B() \) have an infinite loop and never stop.
Threads in group A

T_A()
{
    while (1) {
        // do something
        Ex. Message
        // do something
    }
}

Threads in group B

T_B()
{
    while (1) {
        // do something
        Ex. Message
        // do something
    }
}
What is *Exchange Message*?

- When an instance A makes a message available, it can continue only if it receives a message from an instance of B who has successfully retrieves A’s message.
- Similarly, when an instance B makes a message available, it can continue only if it receives a message from an instance of A who has successfully retrieves B’s message.
- *How about exchanging business cards?*
Watch for Race Conditions

- Suppose thread $A_1$ presents its message for $B$ to retrieve. If $A_2$ comes for message exchange before $B$ retrieves $A_1$’s, will $A_2$’s message overwrites $A_1$’s?
- Suppose $B$ has already retrieved $A_1$’s message. Is it possible that when $B$ presents its message, $A_2$ picks it up rather than $A_1$?
- Thus, the messages between $A$ and $B$ must be well-protected to avoid race conditions.
Students’ Work

- This problem and its variations were used as programming assignments, exam problems, and so on.
- A significant number of students successfully solve this problem.
- The next few slides show how students made mistakes.
First Attempt

Sem A = 0, B = 0;
Int Buf_A, Buf_B;

```
T_A()
{
    int V_a;
    while (1) {
        V_a = ..;
        Signal(B);
        Wait(A);
        Buf_A = V_a;
        V_a = Buf_B;
    }
}

T_B()
{
    int V_b;
    while (1) {
        V_b = ..;
        Signal(A);
        Wait(B);
        Buf_B = V_b;
        V_b = Buf_A;
    }
}
```
**First Attempt: Problem (a)**

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal(B)</td>
<td></td>
</tr>
<tr>
<td>Wait(A)</td>
<td>Signal(A)</td>
</tr>
<tr>
<td></td>
<td>Wait(B)</td>
</tr>
<tr>
<td>Buf_A = V_a</td>
<td></td>
</tr>
<tr>
<td>V_a = Buf_B</td>
<td>Buf_B = V_b</td>
</tr>
</tbody>
</table>

*Buf_B has no value, yet!*  
*Oops, it is too late!*
**First Attempt: Problem (b)**

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₂</th>
<th>B₁</th>
<th>B₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buf_A = .</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buf_B = .</td>
<td></td>
<td></td>
<td>Signal(A)</td>
</tr>
</tbody>
</table>

*Race Condition*
What did we learn?

- If there are shared data items, always protect them properly. Without a proper mutual exclusion, race conditions are likely to occur.

- In this first attempt, both global variables Buf_A and Buf_B are shared and should be protected.
Second Attempt

Sem A = B = 0;
Sem Mutex = 1;
Int Buf_A, Buf_B;

T_A()
{ int V_a;
    While (1) {
        Signal(B);
        Wait(A);
        Wait(Mutex);
        Buf_A = V_a;
        Signal(Mutex);
        Signal(B);
        Wait(A);
        Wait(Mutex);
        V_a = Buf_B;
        Signal(Mutex);
    }
}

T_B()
{ int V_b;
    While (1) {
        Signal(A);
        Wait(B);
        Wait(Mutex);
        Buf_B = V_b;
        Signal(Mutex);
        Signal(A);
        Wait(B);
        Wait(Mutex);
        V_b = Buf_A;
        Signal(Mutex);
    }
}

Sem  A = B = 0;
Sem  Mutex = 1;
Int  Buf_A, Buf_B;

protection???
Second Attempt: Problem

<table>
<thead>
<tr>
<th></th>
<th>A₁</th>
<th>A₂</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buf_A = ..</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buf_B = ..</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

race condition

hand shaking with
wrong person
What did we learn?

- Improper protection is no better than no protection, because we have an illusion that data are well-protected.
- We frequently forgot that protection is done by a critical section, which cannot be divided.
- Thus, protecting “here is my card” followed by “may I have yours” separately is unwise.
Third Attempt

Sem Aready = Bready = 1; ready to proceed
Sem Adone = Bdone = 0;
Int Buf_A, Buf_B;

T_A()
{ int V_a;
 while (1) {
   Wait(Aready);
   Buf_A = ..;
   Signal(Adone);
   Wait(Bdone);
   V_a = Buf_B;
   Signal(Aready);
 }
}

T_B()
{ int V_b;
 while (1) {
   Wait(Bready);
   Buf_B = ..;
   Signal(Bdone);
   Wait(Adone);
   V_b = Buf_A;
   Signal(Bready);
 }
}
### Third Attempt: Problem

<table>
<thead>
<tr>
<th></th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buf_A</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>Signal(Adone)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait(Bdone)</td>
<td></td>
<td>Signal(Bdone)</td>
</tr>
<tr>
<td>** loop back **</td>
<td></td>
<td>Wait(Adone)</td>
</tr>
<tr>
<td>... = Buf_B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal(Aready)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>** race condition **</td>
<td>Buf_A = ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ruin the original value of Buf_A</td>
<td>B is a slow thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

- **Buf_A** is assigned to a variable.
- **Signal(Adone)** is used to signal that a condition is met.
- **Wait(Bdone)** waits for a condition to be met.
- **Loop back** refers to a repeated sequence of operations.
- **Race condition** occurs when two or more threads access the same variable at the same time, potentially leading to incorrect results.

**Thread B** is a slow thread, which may affect the overall performance and synchronization in the system.
What did we learn?

- Mutual exclusion for one group may not prevent threads in other groups from interacting with a thread in the group.
- It is common that a student protects a shared item for one group and forgets other possible, unintended accesses.
- Protection must apply *uniformly* to all threads rather than within groups.
Fourth Attempt

Sem  Aready = Bready = 1; \rightarrow \text{ready to proceed}
Sem  Adone = Bdone = 0;
Int  Buf_A, Buf_B;

T_A()
{  int V_a;
   while (1) {
      Wait(Bready);
      Buf_A = ..;
      Signal(Adone);
      Wait(Bdone);
      V_a = Buf_B;
      Signal(Aready);
   }
}

T_B()
{  int V_b;
   while (1) {
      Wait(Aready);
      Buf_B = ..;
      Signal(Bdone);
      Wait(Adone);
      V_b = Buf_A;
      Signal(Bready);
   }
}

\text{I am the only A} \rightarrow \text{Wait(Bready);} \quad \text{Wait(Aready);} \\
\quad \text{Buf}_A = ..; \quad \text{Buf}_B = ..; \\
\text{here is my card} \rightarrow \text{Signal(Adone);} \quad \text{Signal(Bdone);} \\
\text{waiting for yours} \rightarrow \text{Wait(Bdone);} \quad \text{Wait(Adone);} \\
\quad V_a = \text{Buf}_B; \quad V_b = \text{Buf}_A; \\
\text{Job done &} \rightarrow \text{Signal(Aready);} \quad \text{Signal(Bready);} \\
\text{next B please}
## Fourth Attempt: Problem

<table>
<thead>
<tr>
<th>A&lt;sub&gt;1&lt;/sub&gt;</th>
<th>A&lt;sub&gt;2&lt;/sub&gt;</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait(Bready)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buf_A = ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal(Adone)</td>
<td>Buf_B = ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signal(Bdone)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait(Adone)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>... = Buf_A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signal(Bready)</td>
<td></td>
</tr>
<tr>
<td>Wait(Bready)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>......</td>
<td></td>
</tr>
<tr>
<td>Wait(Bdone)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>... = Buf_B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hey, this one is for A<sub>1</sub>!!!
What did we learn?

- We use locks for mutual exclusion.
- The owner, the one who locked the lock, should unlock the lock.
- In the above “solution,” Aready is acquired by a thread A but released by a thread B. This is risky!
- In this case, a pure lock is more natural than a binary semaphore.
A Good Attempt

How about the use of a bounded buffer?

```c
int Buf_A, Buf_B; /* Buffer variables */

T_A()
{
    int V_a;
    while (1) {
        PUT(V_a, Buf_A);
        GET(V_a, Buf_B);
    }
}

T_B()
{
    int V_b;
    while (1) {
        PUT(V_b, Buf_B);
        GET(V_b, Buf_A);
    }
}
```

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₂</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>PUT</td>
<td>GET</td>
</tr>
<tr>
<td>GET</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Good Attempt
Protection still makes sense

Sem Mutex = 1;
int Buf_A, Buf_B;

T_A()
{  int V_a;
   while (1) {
      Wait(Mutex);
      PUT(V_a, Buf_A);
      GET(V_a, Buf_B);
      Signal(Mutex);
   }
}

T_B()
{  int V_b;
   while (1) {
      Wait(Mutex);
      PUT(V_b, Buf_B);
      GET(V_b, Buf_A);
      Signal(Mutex);
   }
}

System will lock up when A or B enters its critical section.
A Good Attempt: Make It Right

Sem Amutex = Bmutex = 1;
int Buf_A, Buf_B;

T_A()
{  int V_a;
while (1) {
    Wait(Amutex);
    PUT(V_a, Buf_A);
    GET(V_a, Buf_B);
    Signal(Amutex);
}
}

T_B()
{  int V_b;
while (1) {
    Wait(Bmutex);
    PUT(V_b, Buf_B);
    GET(V_b, Buf_A);
    Signal(Bmutex);
}
}

This solution works, even though each group has its own protection. The PUT and GET make a difference.
A Good Attempt: Symmetric

Sem Amutex = Bmutex = 1;
Sem NotFul_A=NotFul_B=1;  Sem NotEmp_A=NotEmp_B=0;
int Buf_A, Buf_B;

T_A()
{
    int V_a;
    while (1) {
        Wait(Amutex);
        Wait(NotFul_A);
        Buf_A = V_a;
        Signal(NotEmp_A);
        Wait(NotEmp_B);
        V_a = Buf_B;
        Signal(NotFul_B);
        Wait(Amutex);
    }
}

T_B()
{
    int V_b;
    while (1) {
        Wait(Bmutex);
        Wait(NotFul_B);
        Buf_B = V_b;
        Signal(NotEmp_B);
        Wait(NotEmp_A);
        V_b = Buf_A;
        Signal(NotFul_A);
        Signal(Bmutex);
    }
}
A Good Attempt: Another Version

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

T_A()
{
    int V_a;
    while (1) {
        Wait(Amutex);
        PUT(V_a, Buf_A);
        GET(V_a, Buf_B);
        Signal(Amutex);
    }
}

T_B()
{
    int V_b, T;
    while (1) {
        Wait(Bmutex);
        GET(T, Buf_A);
        PUT(V_b, Buf_B);
        Signal(Bmutex);
    }
}
```

Note that the PUTs and GETs also provide mutual exclusion.
A Good Attempt: Non-Symmetric

Sem NotFull = 1, NotEmp_A = NotEmp_B = 0;
int Shared;

T_A()
{
    int V_a;
    while (1) {
        Wait(NotFull);
        Shared = V_a;
        Signal(NotEmp_A);
        Wait(NotEmp_B);
        V_a = Shared;
        Signal(NotFull);
    }
}

T_B()
{
    int V_b, T;
    while (1) {
        Wait(NotEmp_A);
        T = Shared;
        Shared = V_b;
        Signal(NotEmp_B);
        Wait(NotEmp_B);
        V_a = Shared;
        Signal(NotFull);
    }
}

this is a lock

no B can be here without A’s Signal
What did we learn?

- Understand the solutions to the classical synchronization problems, because they are *useful*.
- The problem in hand could be a variation of some classical problems.
- Combine, apply and/or simplify the classical solutions.
- Thus, classical problems are not toy problems! They have their meaning.
Conclusions

- Detecting race conditions is difficult as it is an NP-hard problem.
- Detecting race conditions is also difficult to teach as there is no theory. It is heuristic.
- Incorrect mutual exclusion is no better than no mutual exclusion.
- Use solutions to classical problems as models.
- The examples have been classroom tested, and are useful, helpful and well-received.