Race Conditions Revisited
Catching Race Conditions: An Extremely Difficult Task

- *Statically* detecting race conditions exactly in a program using multiple semaphores is NP-hard.
- Thus, no efficient algorithms are available. We have to use debugging skills.
- It is virtually impossible to catch race conditions *dynamically* because hardware must examine *every* memory access.
- We shall use a few examples to illustrate some subtle race conditions.
Problem Statement

- Two groups, \( A \) and \( B \), of processes exchange messages.
- Each process in \( A \) runs function \( T_A() \), and each process in \( B \) runs function \( T_B() \).
- Both \( T_A() \) and \( T_B() \) have an infinite loop and never stop.
- In the following, we show execution sequences that cause race conditions. You can always find an execution sequence without race conditions.
Processes in group A

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

Processes in group B

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

- When a process in A makes a message available, it can continue only if it receives a message from a process in B who has successfully retrieved A’s message.

- Similarly, when a process in B makes a message available, it can continue only if it receives a message from a process in A who has successfully retrieved B’s message.

- *How about exchanging business cards?*
Watch for Race Conditions

• Suppose process $A_1$ presents its message for $B$ to retrieve. If $A_2$ comes for message exchange before $B$ can retrieve $A_1$’s, will $A_2$’s message overwrite $A_1$’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 by $A_1$?

• Thus, the messages between $A$ and $B$ must be well-protected to avoid race conditions.
First Attempt

sem A = 0, B = 0;
int Buf_A, Buf_B;

T_A()
{
    int V_a;
    while (1) {
        V_a = ..;
        B.signal();
        A.wait();
        Buf_A = V_a;
        V_a = Buf_B;
    }
}

T_B()
{
    int V_b;
    while (1) {
        V_b = ..;
        A.signal();
        B.wait();
        Buf_B = V_b;
        V_b = Buf_A;
    }
}

Wait for your card!

I am ready
# First Attempt: Problem (a)

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.signal()</td>
<td></td>
</tr>
<tr>
<td>A.wait()</td>
<td>A.signal()</td>
</tr>
<tr>
<td></td>
<td>B.wait()</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>B.signal()</td>
<td>A.wait()</td>
<td>A.signal()</td>
<td>Buf_B = .</td>
</tr>
<tr>
<td>A.wait()</td>
<td>A.signal()</td>
<td>B.wait()</td>
<td></td>
</tr>
<tr>
<td>B.signal()</td>
<td>B.wait()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.wait()</td>
<td>Buf_A = .</td>
<td>Buf_A = .</td>
<td></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) {
    B.signal();
    A.wait();
    Mutex.wait();
    Buf_A = V_a;
    Mutex.signal();
  }
}

T_B()
{ int V_b;
  while (1) {
    A.signal();
    B.wait();
    Mutex.wait();
    Buf_B = V_b;
    Mutex.signal();
  }
}

shake hands

protection???

offer
My card
Second Attempt: Problem

<table>
<thead>
<tr>
<th></th>
<th>A₁</th>
<th>A₂</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B.signal()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A.wait()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B.wait()</td>
<td>A.signal()</td>
</tr>
<tr>
<td>Buf</td>
<td>Buf_A = ..</td>
<td></td>
<td>Buf_B = ..</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B.signal()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A.wait()</td>
<td>A.signal()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B.wait()</td>
</tr>
<tr>
<td>Buf</td>
<td>Buf_A = ..</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 it gives us an *illusion* that data have been well-protected.
- We frequently forget 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 not a good idea.
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) {
        Aready.wait();
        Buf_A = ..;
        Adone.signal();
        Bdone.wait();
        V_a = Buf_B;
        Aready.signal();
    }
}

T_B()
{ int V_b;
    while (1) {
        Bready.wait();
        Buf_B = ..;
        Bdone.signal();
        Adone.wait();
        V_b = Buf_A;
        Bready.signal();
    }
}
**Third Attempt: Problem**

<table>
<thead>
<tr>
<th></th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
</table>
| Buf_A = ... |          | Buf_B = ...
| Adone.signal() |          | Bdone.signal() |
| Bdone.wait() |          | Buf_B = ...
|                  | Bdone.signal() | Adone.wait() |
| ... = Buf_B      |          |                |
| Aready.signal() |          |                |
| **loops back**   |          | B is a slow thread |
| Aready.wait()    |          |                |
| Buf_A = ...      |          |                |
| race condition   |          |                |

B is a slow thread may ruin the original value of Buf_A.
What Did We Learn?

- Mutual exclusion for group A may not prevent processes in group B from interacting with a process in group A, and vice versa.
- It is common that we protect a shared item for one group and forget other possible, unintended accesses.
- Protection must be applied *uniformly* to all processes rather than within groups.
Fourth Attempt

sem Aready = Bready = 1; \textit{ready to proceed}

sem Adone = Bdone = 0;

int Buf_A, Buf_B;

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

T_B()
{ 
  int V_b;
  while (1) {
    Aready.wait();
    Buf_B = ..;
    Bdone.wait();
    Adone.signal();
    Bdone.signal();
    Bready.signal();
  }
}
**Fourth Attempt: Problem**

<table>
<thead>
<tr>
<th></th>
<th>A_1</th>
<th>A_2</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>Bready.wait()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Buf_A = ...</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Adone.signal()</code></td>
<td><code>Buf_B = ...</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Bdone.signal()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Adone.wait()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>... = Buf_A</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Bready.signal()</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Bready.wait()</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>......</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Bdone.wait()</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>... = Buf_B</code></td>
</tr>
</tbody>
</table>

*Hey, this one is for A_1!!!*
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 process A but released by a process B. This is risky!
- In this case, a pure lock is more natural than a binary semaphore.
A Good Attempt: 1/7

- This message exchange problem is actually a variation of the producer-consumer problem.
- A thread is a producer (resp., consumer) when it deposits (resp., retrieves) a message.
- Therefore, a complete “message exchange” is simply a deposit followed by a retrieval.
- We may use a buffer `Buf_A` (resp., `Buf_B`) for a thread in `A` (resp., `B`) to deposit a message for a thread in `B` (resp., `A`) to retrieve.
Based on this observation, we have the following. Does it work?

```c
bounded_buffer Buf_A, Buf_B;

Thread_A(...)                Thread_B(...)
{                        }
    int Var_A;                   int Var_B;

    while (1) {
        ....

        PUT(Var_A, Buf_A);
        GET(Var_A, Buf_B);

        exchange message ...
    }

}                          }
```
A Good Attempt: 3/7

- Unfortunately, this is **NOT** a correct solution!
- Thread $A_1$’s message may be retrieved by thread $B$, and thread $B$’s message may be retrieved by thread $A_2$, a wrong message exchange!

<table>
<thead>
<tr>
<th>Thread $A_1$</th>
<th>Thread $A_2$</th>
<th>Thread $B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT(Var$_A$, Buf$_A$)</td>
<td></td>
<td>PUT(Var$_B$, Buf$_B$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GET(Var$_B$, Buf$_A$)</td>
</tr>
<tr>
<td>PUT(Var$_A$, Buf$_A$)</td>
<td>GET(Var$_A$, Buf$_B$)</td>
<td></td>
</tr>
<tr>
<td>GET(Var$_A$, Buf$_B$)</td>
<td></td>
<td>Buf$_A$ is empty here and $A_2$ can PUT</td>
</tr>
</tbody>
</table>
We may enforce mutual exclusion to avoid threads starting exchange messages at the same time.

```c
bounded_buffer Buf_A, Buf_B;
semaphore Mutex = 1;

Thread_A(...)                Thread_B("
{                          {
  int Var_A;                int Var_B;

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

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

---

**mutual exclusion**
Once a thread enters the critical section and finishes executing **PUT**, it will be blocked in **GET**!

```c
bounded_buffer Buf_A, Buf_B;
semaphore Mutex = 1;

Thread_A(…)
{
    int Var_A;
    while (1) {
        ……
        Wait(Mutex);
        PUT(Var_A, Buf_A);
        GET(Var_A, Buf_B);
        Signal(Mutex);
    }
}

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

*mutual exclusion*
In this particular case, mutual exclusion does not have to extend to the other group.

```c
bounded_buffer Buf_A, Buf_B;
semaphore A_Mutex = 1, B_Mutex = 1;

Thread_A(...)
{
    int Var_A;

    while (1) {
        ....
        Wait(A_Mutex);
        PUT(Var_A, Buf_A);
        GET(Var_A, Buf_B);
        Signal(A_Mutex);
        .... mutual exclusion for A
    }
}

Thread_B(...)
{
    int Var_B;

    while (1) {
        ....
        Wait(B_Mutex);
        PUT(Var_B, Buf_B);
        GET(Var_B, Buf_A);
        Signal(B_Mutex);
        .... mutual exclusion for B
    }
}
```
A Good Attempt: 7/7

- Is this solution correct? Yes, it is!
- Before a thread in A finishes its message exchange (i.e., PUT and GET), no other threads in A can start a message exchange.
- If A₁ PUTs a message and B has a message available, it is impossible for any A₂ to retrieve B’s message.
- If A₂ can retrieve B’s message, A₂ must be in the critical section while A₁ is about to execute GET. This is impossible because A₁ is already in the critical section!
What Did We Learn?

- The most important lesson is that classical problems (e.g., dining philosophers, producers-consumers and readers-writers) can serve as models to solve other problems.
- Many problems are variations or extensions of the classical problems.
- Check ThreadMentor’s tutorial pages for simplified solutions.
Conclusions

- Detecting race conditions is difficult as it is an NP-hard problem.
- Hence, detecting race conditions is heuristic.
- Incorrect mutual exclusion is no better than no mutual exclusion.
- Race conditions are sometimes very subtle. They may appear at unexpected places.
- Check the ThreadMentor tutorial pages for more details and correct solutions.