Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
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 design programs properly and carefully, and use debugging skills wisely.
- 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 can cause race conditions. You may always find other execution sequences without race conditions.
Processes in group A

\[
T_A() \\
\{
\quad \text{while (1) { \\
\quad \quad \text{// do something} \\
\quad \quad \text{Ex. Message} \\
\quad \quad \text{// do something} \\
\quad \} \\
\}
\]

Processes in group B

\[
T_B() \\
\{
\quad \text{while (1) { \\
\quad \quad \text{// do something} \\
\quad \quad \text{Ex. Message} \\
\quad \quad \text{// do something} \\
\quad \} \\
\}
\]
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 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 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>A.signal()</td>
</tr>
<tr>
<td>A.wait()</td>
<td>B.wait()</td>
</tr>
<tr>
<td>Buf_A = V_a</td>
<td>Buf_B = V_b</td>
</tr>
<tr>
<td>V_a = Buf_B</td>
<td>Buf_B has no value, yet!</td>
</tr>
</tbody>
</table>

**Oops, it is too late!**

Buf_B has no value, yet!
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></td>
<td>A.signal()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B.wait()</td>
<td></td>
</tr>
<tr>
<td>B.signal()</td>
<td>A.wait()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.wait()</td>
<td></td>
<td></td>
<td>Buf_B = .</td>
</tr>
<tr>
<td></td>
<td>A.wait()</td>
<td></td>
<td>A.signal()</td>
</tr>
<tr>
<td>Buf_A = .</td>
<td></td>
<td></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();
    B.signal();
    A.wait();
    Mutex.wait();
    V_a = Buf_B;
    Mutex.signal();
  }
}

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

sem A = B = 0;
sem Mutex = 1;
int Buf_A, Buf_B;
### Second Attempt: Problem

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₂</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.signal()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.wait()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buf_A = ..</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Race condition

B.signal() A.signal() A.wait() B.wait() Buf_A = ..

---

Race condition

hand shaking with a 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*. That is, execution in the protected critical section must be atomic.

- Thus, protecting “here is my card” followed by “may I have yours” separately is not a good idea.
Third Attempt

```c
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();
  }
}
```

only one A can proceed

here is my card, let me have yours

only one B can proceed
### Third Attempt: Problem

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buf_A = ...</td>
<td>Buf_B = ...</td>
</tr>
<tr>
<td>Adone.signal()</td>
<td>Bdone.signal()</td>
</tr>
<tr>
<td>Bdone.wait()</td>
<td>Bdone.wait()</td>
</tr>
<tr>
<td>... = Buf_B</td>
<td>** loops back **</td>
</tr>
<tr>
<td>Aready.signal()</td>
<td>** loops back **</td>
</tr>
<tr>
<td>Aready.wait()</td>
<td>Aready.wait()</td>
</tr>
<tr>
<td>Buf_A = ...</td>
<td>... = Buf_A</td>
</tr>
</tbody>
</table>

**race condition**

- ruin the original value of Buf_A
- B is a slow thread
- watch for fast runners

---

15
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; /* ready to proceed */
sem Adone = Bdone = 0;
int Buf_A, Buf_B;

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

T_B()
{  int V_b;
    while (1) {
        Aready.wait();
        Buf_B = ..;
        Bdone.signal();
        Adone.wait();
        V_b = Buf_A;
        Bready.signal();
    }
}
```

To understand what would happen if `Aready = 1` and `Bready = 0`, let's analyze the code step by step:

1. **Initial State**: `Aready = 1` and `Bready = 0`.
2. **Condition Check**: Since `Aready` is 1, `Bready.wait()` will block `T_A()` until `Bready` becomes 1.
3. **Thread Behavior**:
   - **T_A()** waits until `Bready` is 1.
   - **T_B()** continues to run, updating `Buf_B` and `V_b = Buf_A`.

**Result**: `T_A()` remains blocked, while `T_B()` proceeds with its tasks.

**Question**: What would happen if `Aready = 1` and `Bready = 0`? (What would happen if `Aready = 1` and `Bready = 0`?)

**Answer**: `T_A()` would be blocked until `Bready` becomes 1, while `T_B()` continues its operations. The synchronization between the threads is maintained through the semaphores, ensuring that only one thread can access the buffer at a time.

---

**Notes**:
- The code snippet demonstrates the use of semaphores for synchronization between two threads (`T_A()` and `T_B()`).
- The variables `Buf_A` and `Buf_B` are used for buffering between the processes.
- The `wait()` and `signal()` functions are used to control the flow of execution between the threads.
- The code is designed to ensure that only one thread is allowed to access the buffer at a time, preventing conflicts.

---

**Further Considerations**:
- The `sem` statement initializes the semaphores, indicating that both `Aready` and `Bready` are ready to proceed.
- The `while (1)` loop in both functions indicates an infinite loop, allowing the threads to continue indefinitely until explicitly interrupted.

---

**Conclusion**:
- The scenario described is a classic example of thread synchronization using semaphores, ensuring that the system remains stable and predictable.
- The specific question about the state transition between threads highlights the importance of understanding the basic mechanisms of thread synchronization.

---

**Additional Resources**:
- **Synchronization**
- **Semaphores**
- **Thread Synchronization in Operating Systems**

---

**Disclaimer**:
- This explanation is based on the provided code snippet and theoretical understanding of thread synchronization.
- It is not a direct translation of the code but rather an interpretation to aid in comprehension.
## Fourth Attempt: Problem

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A₁</strong></td>
<td><strong>A₂</strong></td>
<td><strong>B</strong></td>
</tr>
<tr>
<td>Bready.wait()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Buf_A = ... | | Buf_B = ...
| Adone.signal() | Buf_B = ... | Bdone.signal()
| | | Adone.wait() |
| | | ... = Buf_A |
| | | Bready.signal() |
| | Bready.wait() | |
| | | |
| | ...... | Hey, this one is for A₁!! |
| | Bdone.wait() | |
| | | ...
| | ... = Buf_B |
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 in *A* but released by a process in *B*. This is risky!
- In this case, a pure lock is more natural than a binary semaphore.
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.
A Good Attempt: 2/7

- 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);         PUT(Var_B, Buf_B);
        GET(Var_A, Buf_B);         GET(Var_B, Buf_A);
        ......                         ......
        exchange message            exchange message
        ......                         ......
    }
}
```
A Good Attempt: 3/7

- Unfortunately, this is an **incorrect** 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>PUT($Var_A$, $Buf_A$)</td>
<td>GET($Var_B$, $Buf_A$)</td>
</tr>
<tr>
<td></td>
<td>GET($Var_A$, $Buf_B$)</td>
<td></td>
</tr>
</tbody>
</table>

$Buf_A$ is empty after this GET and $A_2$ can PUT.
A Good Attempt: 4/7

- 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);          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);
      ......                 ......
  }
}
```

Is this solution correct?
A Good Attempt: 5/7

- **Deadlock! Deadlock! Deadlock!**

```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);
        ......
    }
}
```

if a thread passes PUT, it will be blocked by GET!
In fact, mutual exclusion does not have to extend to the other group as **PUT** and **GET** sync accesses.

```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);
    }
}

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 A*  
*... 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 A_1 PUTs a message and B has a message available, it is impossible for any A_2 to retrieve B’s message.
- If A_2 can retrieve B’s message, A_2 must be in the critical section while A_1 is about to execute GET. This is impossible because A_1 is already in the critical section!
What Did We Learn?

- The most important lessen is that classical problems (e.g., dinning 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 using bounded buffers.
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.
The End