Part III
Synchronization
Race Conditions - Revisited

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.

Donald Knuth
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()
{
    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

```c
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;
    }
}
```

I am ready

Wait for your card!
**First Attempt: Problem (a)**

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>B.signal()</code></td>
<td><code>A.signal()</code></td>
</tr>
<tr>
<td><code>A.wait()</code></td>
<td><code>B.wait()</code></td>
</tr>
<tr>
<td><code>Buf_A = V_a</code></td>
<td><code>Buf_B = V_b</code></td>
</tr>
<tr>
<td><code>V_a = Buf_B</code></td>
<td><code>Buf_B has no value, yet!</code></td>
</tr>
</tbody>
</table>

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></td>
<td>A.signal()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A.wait()</td>
<td>B.wait()</td>
</tr>
<tr>
<td>B.signal()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.wait()</td>
<td></td>
<td></td>
<td>Buf_B = .</td>
</tr>
<tr>
<td>A.wait()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buf_A = .</td>
<td></td>
<td></td>
<td>Buf_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 \texttt{Buf\_A} and \texttt{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;

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();
}
}
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>A.signal()</td>
</tr>
<tr>
<td>A.wait()</td>
<td></td>
<td>B.wait()</td>
</tr>
<tr>
<td>Buf_A = ..</td>
<td></td>
<td>Buf_B = ..</td>
</tr>
<tr>
<td>B.signal()</td>
<td></td>
<td>A.signal()</td>
</tr>
<tr>
<td>A.wait()</td>
<td></td>
<td>B.wait()</td>
</tr>
<tr>
<td>Buf_A = ..</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

- Ruin the original value of Buf_A
- B is a slow thread
- Watch for fast runners
- Race condition
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();
    }
}

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

I am the only A
here is my card
wait for yours
job done &
next B please

wait/signal switched

what would happen if Aready=1 and Bready=0?
### Fourth Attempt: Problem

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₂</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bready.wait()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buf_A = …</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adone.signal()</td>
<td>Buf_B = …</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bdone.signal()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adone.wait()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>... = Buf_A</td>
<td></td>
</tr>
<tr>
<td>Bready.wait()</td>
<td>Bready.signal()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hey, this one is for A₁!!</td>
<td></td>
</tr>
<tr>
<td>Bdone.wait()</td>
<td></td>
<td>... = Buf_B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $\text{Buf}_A$ (resp., $\text{Buf}_B$) 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

...
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>\text{PUT}(\text{Var}_A,\text{Buf}_A)</td>
<td>\text{PUT}(\text{Var}_B,\text{Buf}_B)</td>
<td>\text{GET}(\text{Var}_B,\text{Buf}_A)</td>
</tr>
<tr>
<td>\text{PUT}(\text{Var}_A,\text{Buf}_A)</td>
<td>\text{PUT}(\text{Var}_A,\text{Buf}_A)</td>
<td>\text{GET}(\text{Var}_A,\text{Buf}_B)</td>
</tr>
</tbody>
</table>

$\text{Buf}_A$ is empty after this \text{GET} and $A_2$ can \text{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) {                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);
        ......                     ......                     mutual exclusion
    }                          }
}                          }
```

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);
        …
    }
}
```

*mutual exclusion*

If a thread passes **PUT**, it will be blocked by **GET**!
A Good Attempt: 6/7

- 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(…)                Thread_B(…)
{                          {
  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
  }
}

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 \( A_1 \) 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 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 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