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.
**Decision Problems**: A *decision* problem is a problem that needs a **YES** or **NO** answer. By repeatedly answering decision problems, one can transform a non-decision problem to a sequence of decision problems.

**Example 1**: Given a set of positive integers, are there any even (or odd) numbers?

**Example 2**: Given a set of integers (positive, zero and negative), is there a subset that sums to zero? For example, in \{ 8, 4, 1, -3, -2, 9 \} the subset \{ 4, 1, -3, -2 \} sums to 0, and the answer is **YES**. The answer is **NO** with \{ 4, 2, -7, -3 \}.
**P, NP and NP-Hard: 2/7**

- **Class \( \mathcal{P} \) Problems**: If a problem \( L \) can be solved in *polynomial time*, \( L \) is in class \( \mathcal{P} \). This means if there is an algorithm that runs in polynomial time to find the answer, this problem is in \( \mathcal{P} \).

- **Example 1**: Is there an even/odd number in a set of \( n \) positive integers? You can easily design an algorithm to find the answer using \( O(n) \) comparisons.

- **Example 2**: Is a given array of \( n \) elements sorted? An \( O(n) \) algorithm is always possible.

- These are *solvable* problems.
Class NP Problems: Given a “solution” if we are able to \textbf{VERIFY} whether that “solution” is actually a solution in polynomial time, this is a \textbf{verifiable} problem.

Example: Given a set of distinct integers, can it be partitioned into two disjoint sets? Let the given set be \( S \) and let \( A \) and \( B \) be the two possible partitions. It is easily to verify if \( A \cup B = S \) and \( A \cap B = \emptyset \) in polynomial time.

If we are able to guess a solution to a problem \( L \) and verify it in polynomial time, \( L \) is in the \textbf{Non-deterministic Polynomial} class \( \mathcal{NP} \).
P, NP and NP-Hard: 4 of 7

- Obviously, the class $P$ is a subset of class $NP$ as a problem in $P$ is already solvable in polynomial time, and, hence is in $NP$ (i.e., $P \subseteq NP$).

- One of the biggest questions in computer science is whether $P = NP$ holds. If $P = NP$ holds, all problems have easily found solutions.

- This is one of the well-known Millennium Problems: See https://www.claymath.org/millennium-problems/p-vs-np-problem for the details.
**P, NP and NP-Hard: 5/7**

- **NP-Completeness.** A problem $L$ is in the $\text{NP}$-Complete class if $L$ is in $\text{NP}$ and every problem $H$ in $\text{NP}$ is reducible (or convertible) to $L$ in polynomial time.

- Thus, problems in $\text{NP}$-Complete are the hardest problems, and if one solves a $\text{NP}$-Complete problem, all $\text{NP}$-Complete problems are solved!

If $\mathcal{P} \neq \mathcal{NP}$:
P, NP and NP-Hard: 6/7

- **NP-Hardness**: A (decision) problem $L$ is \(NP\)-Hard if every problem in \(NP\) is reducible (or convertible) to $L$. Note that $L$ does not have to be in \(NP\).

\[\text{All Problems} \]

\[\text{NP} \]

\[\text{P} \]

\[\text{NP-Hard} \]

\[\text{NP-Complete} \]

if $P \neq NP$:
**P, NP and NP-Hard: 7/7**

- **NP-Hard** class contains those hardest problems that may not be in **NP**.
- The **NP-Complete** class contains those hardest problems in **NP**.

If $P \neq NP$:

- $NP$-Hard
- $NP$-Complete
- All Problems
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

\text{T\_A()} \\
\{ \\
\quad \text{while (1) \{} \\
\qquad \text{// do something} \\
\qquad \text{Ex. Message} \\
\qquad \text{// do something} \\
\quad \text{\}} \\
\} \\

Processes in group B

\text{T\_B()} \\
\{ \\
\quad \text{while (1) \{} \\
\qquad \text{// do something} \\
\qquad \text{Ex. Message} \\
\qquad \text{// do something} \\
\quad \text{\}} \\
\}
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;
    }
}

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></td>
</tr>
<tr>
<td><code>A.wait()</code></td>
<td><code>A.signal()</code></td>
</tr>
<tr>
<td><code>Buf_A = V_a</code></td>
<td><code>B.wait()</code></td>
</tr>
<tr>
<td><code>V_a = Buf_B</code></td>
<td></td>
</tr>
<tr>
<td><code>Buf_B</code> has no value, yet!</td>
<td><code>Oops, it is too late!</code></td>
</tr>
<tr>
<td></td>
<td><code>Buf_B = V_b</code></td>
</tr>
</tbody>
</table>
First Attempt: Problem (b)

<table>
<thead>
<tr>
<th></th>
<th>A₂</th>
<th>B₁</th>
<th>B₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<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></td>
<td>B.signal()</td>
<td></td>
</tr>
<tr>
<td>A.wait()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.wait()</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>
</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();
  }
}
```
## 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>
</tbody>
</table>

Race condition

Buf_A = ..

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

Buf_B = ..

<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>
</tbody>
</table>

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

Buf_A = ..

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

job done

only one B can proceed
## Third Attempt: Problem

### Table:

<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.signal()</td>
</tr>
<tr>
<td></td>
<td>Adone.wait()</td>
</tr>
<tr>
<td>... = Buf_B</td>
<td></td>
</tr>
<tr>
<td>Aready.signal()</td>
<td></td>
</tr>
<tr>
<td><strong>loops back</strong></td>
<td></td>
</tr>
<tr>
<td>Aready.wait()</td>
<td></td>
</tr>
<tr>
<td>Buf_A = ...</td>
<td></td>
</tr>
<tr>
<td>race condition</td>
<td>... = Buf_A</td>
</tr>
</tbody>
</table>

*ruin the original value of* Buf\_A

*watch for fast runners*

*B is a slow thread*
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

```c
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

Bready.wait();
Buf_A = ..;
Adone.signal();
Bdone.signal();
Bready.wait();
V_a = Buf_B;
Aready.signal();

here is my card
Adone.signal();
Bdone.signal();

wait for yours
Bdone.wait();
Adone.wait();
V_b = Buf_A;
Bready.signal();

job done &
next B please
Aready.signal();
Bready.signal();

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

<table>
<thead>
<tr>
<th></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>Bu_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></td>
<td>Bready.signal()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bready.wait()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>......</td>
<td>Hey, this one is for A₁!!</td>
</tr>
<tr>
<td></td>
<td>Bdone.wait()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>... = Buf_B</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,” A ready 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(...) {
    int Var_A;
    while (1) {
        …
        PUT(Var_A, Buf_A);
        GET(Var_A, Buf_B);
        …
    }
}

Thread_B(...) {
    int Var_B;
    while (1) {
        …
        PUT(Var_B, 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>PUT($Var_A$, $Buf_A$)</td>
<td>PUT($Var_B$, $Buf_B$)</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>$Buf_A$ is empty after this GET and $A_2$ can PUT</td>
</tr>
</tbody>
</table>
We may enforce mutual exclusion to avoid threads starting exchange messages at the same time.

```
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);
        ...... mutual exclusion ......
    }
}

Thread_B(…)
{
    int Var_B;
    while (1) {
        ......
        Wait(Mutex);
        PUT(Var_B, Buf_B);
        GET(Var_B, Buf_A);
        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`!
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(...) {
    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 $A_1$ 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