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

What Will be Covered?

- **This component discusses why detecting race conditions in a concurrent program is difficult.**
- **There are two parts:**
  - **A.** **A few terms in complexity theory will be discussed in a rather intuitive way. These terms illustrate the difficulty in catching race conditions.**
  - **B.** **Then, a set of examples will be presented to illustrate some ideas for catching possible race conditions.**

Race Conditions Revisited

- **If a program produces non-deterministic results, there could be race conditions.**
- **Note that having non-deterministic results does not mean this program has race conditions.**
- **A race condition produces *non-deterministic* results, but producing non-deterministic results does not always indicate the existence of race conditions,**
- **We covered this in an earlier lecture.**
- **See `05-Sync-Basics.pdf` for the details.**

Race Conditions: Definition

- A **Race Condition** occurs, if
  - **two or more** processes/threads manipulate a **shared** resource **concurrently**, and
  - the outcome of the execution **depends on the order** in which the access takes place.
- *Synchronization* is needed to prevent race conditions from happening.

Execution Sequences

- **Always use instruction level interleaving to demonstrate the existence of race conditions**, because
  a) higher-level language statements are not atomic and can be switched in the middle of execution
  b) instruction level interleaving can show clearly the "sharing" of a resource among processes and threads.
  c) **two** execution sequences are needed to show the answer depends on order of execution.

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 carefully, and use debugging skills wisely.**
- **It is virtually impossible to catch race conditions *dynamically* because hardware must examine *every* memory access.**

Terms: P, NP, NP-Hard, etc.

P, NP and NP-Hard: 1/7

- **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, the subset { 4, 1, -3, -2} of { 8, 4, 1, -3, -2, 9 } 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 $\mathscr{P}$ Problems**: If a problem L can be solved in *polynomial time*, L is in class $\mathscr{P}$. This means if there is an algorithm that runs in polynomial time to find the **YES**/**NO** answer, this problem is in $\mathscr{P}$.
- **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.

P, NP and NP-Hard: 3/7

- **Class $\mathcal{NP}$ Problems**: Given a "solution" if we are able to **VERIFY** whether that "solution" is actually a solution in polynomial time, this is a **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 easy to verify if $A \cup B = S$ and $A \cap B = \varnothing$ 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 **Non-deterministic Polynomial** class $\mathcal{NP}$.

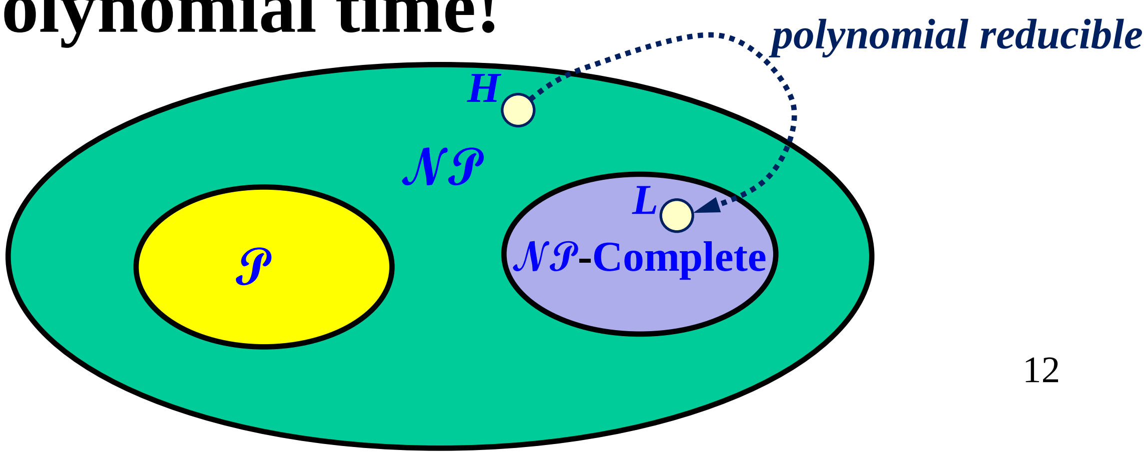
P, NP and NP-Hard: 4/7

- **Obviously, class $\mathcal{P}$ is a subset of class $\mathcal{NP}$ as any problem in $\mathcal{P}$ is already solvable in polynomial time, and hence is in $\mathcal{NP}$ (i.e., $\mathcal{P} \subseteq \mathcal{NP}$).**
- **One of the most challenging questions in computer science is whether $\mathcal{P} = \mathcal{NP}$ holds. If $\mathcal{P} = \mathcal{NP}$ holds, all problems have efficient 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 $\mathcal{NP}$-Complete class if L is in $\mathcal{NP}$ and every problem H in $\mathcal{NP}$ is reducible (or convertible) to L in polynomial time.
- Problems in $\mathcal{NP}$-Complete are the hardest problems. If one solves a $\mathcal{NP}$-Complete problem in polynomial time, all $\mathcal{NP}$-Complete problems are solved in polynomial time!

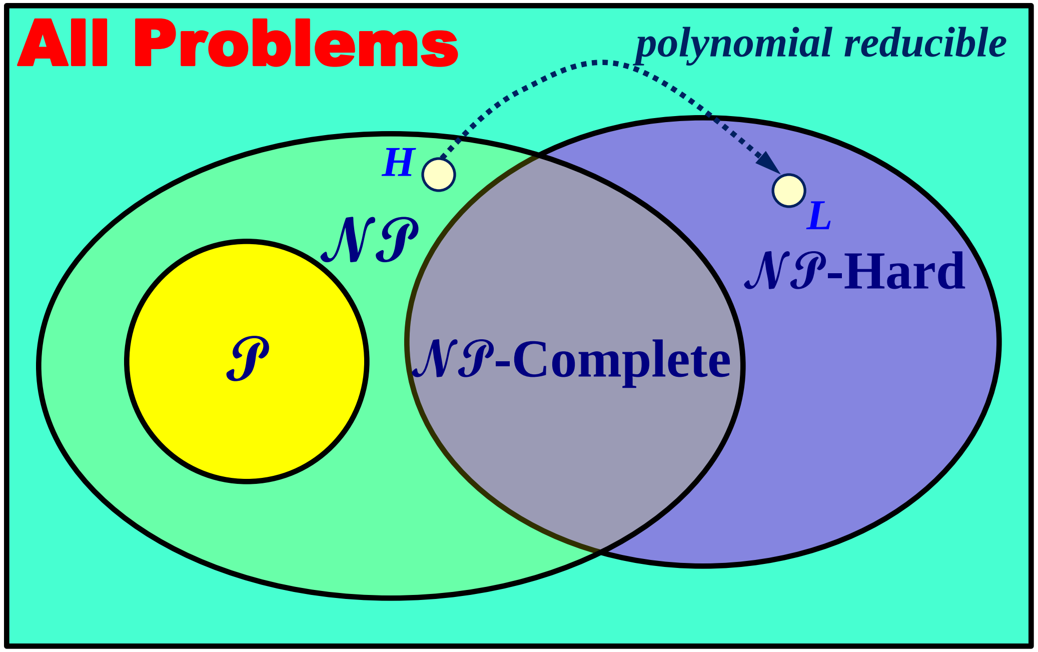
if $\mathcal{P} \neq \mathcal{NP}$:



P, NP and NP-Hard: 6/7

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

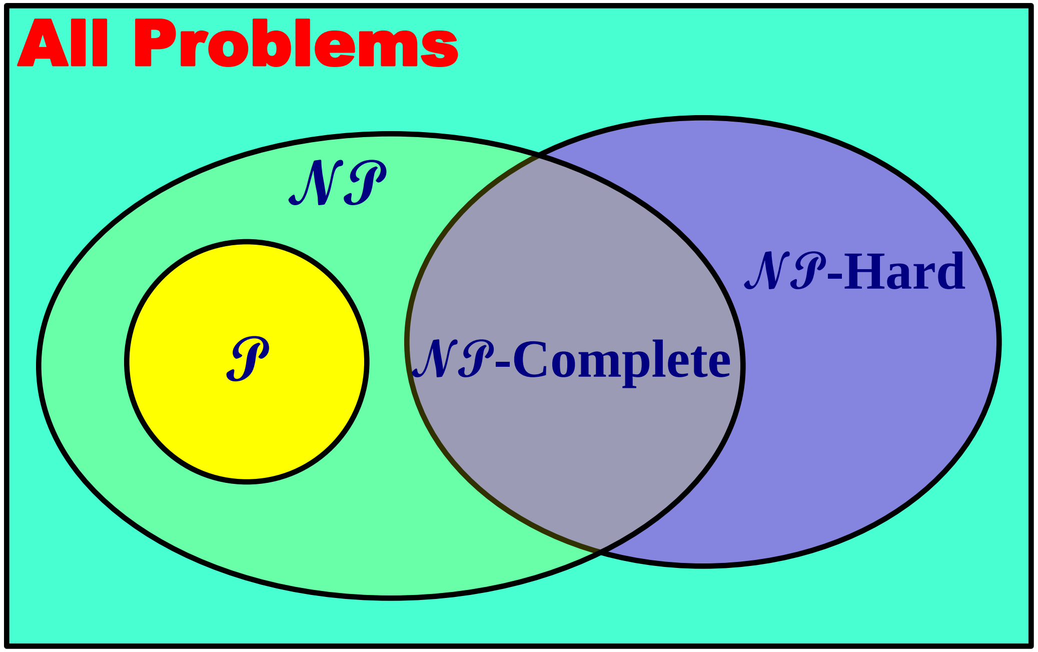
if $\mathcal{P} \neq \mathcal{NP}$:



P, NP and NP-Hard: 7/7

- $\mathcal{NP}$-Hard class contains those hardest problems that may not be in $\mathcal{NP}$.
- The $\mathcal{NP}$-Complete class contains those hardest problems in $\mathcal{NP}$.

if $\mathcal{P} \neq \mathcal{NP}$:



Examples

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 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;
```

I am ready

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

Wait for your card!

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

First Attempt: Problem (a)

Thread A	Thread B
`B.signal()`	
`A.wait()`	
	`A.signal()`
	`B.wait()`
`Buf_A = V_a`	
`V_a = Buf_B`	
	`Buf_B = V_b`

`Buf_B` has no value, yet!

Oops, it is too late!

First Attempt: Problem (b)

A_1	A_2	B_1	B_2
`B.signal()`			
		`A.signal()`	
		`B.wait()`	
	`B.signal()`		
	`A.wait()`		
		`Buf_B = .`	
			`A.signal()`
`A.wait()`			
`Buf_A = .`			
	`Buf_A = .`		

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;
```

shake hands

protection???

```
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();
  }
}
```

offer my card

Second Attempt: Problem

A_1	A_2	B
`B.signal()`		
`A.wait()`		
		`A.signal()`
		`B.wait()`
`Buf_A = ..`		
		`Buf_B = ..`
	`B.signal()`	
	`A.wait()`	
		`A.signal()`
		`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

job done

ready to proceed

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

only one A can proceed

here is my card let me have yours

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

only one B can proceed

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

Third Attempt: Problem

Thread A	Thread B
`Buf_A = …`	
`Adone.signal()`	
`Bdone.wait()`	`Buf_B = …`
	`Bdone.signal()`
	`Adone.wait()`
`… = Buf_B`	
`Aready.signal()`	
**** loops back ****	
`Aready.wait()`	
`Buf_A = …`	
race condition	`… = Buf_A`

ruin the original value of `Buf_A`

B is a slow thread

watch for fast runners

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

ready to proceed

job done

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

wait/signal switched

I am the only A

here is my card

wait for yours

job done & next B please

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

what would happen if `Aready=1` and `Bready=0`?

Fourth Attempt: Problem

A_1	A_2	B
`Bready.wait()`		
`Buf_A = …`		
`Adone.signal()`		`Buf_B = …`
		`Bdone.signal()`
		`Adone.wait()`
		`… = Buf_A`
		`Bready.signal()`
	`Bready.wait()`	
	……	Hey, this one is for A_1!!!
	`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.

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.

A Good Attempt: 2/7

- **Based on this observation, we have the following. Does it work?**

```
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!

Thread A_1	Thread A_2	Thread B
`PUT(Var_A,Buf_A)`		`PUT(Var_B,Buf_B)`
		`GET(Var_B,Buf_A)`
	`PUT(Var_A,Buf_A)`	
	`GET(Var_A,Buf_B)`	

`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.

```
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);
    ……                                   …
  }                                    }
}                                    }
```

Is this solution correct?

mutual exclusion

A Good Attempt: 5/7

- **Deadlock! Deadlock! Deadlock!**

if a thread passes PUT, it will be blocked by GET!

```
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

A Good Attempt: 6/7

- In fact, mutual exclusion does not have to extend to the other group as `PUT` and `GET` sync accesses.

```
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 `PUT`s 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 (i.e., `A_Mutex`)!**

Constructing A Solution: 1/5

This Is a Solution to the Bounded Buffer Problem

number of slots

```
semaphore NotFull=n, NotEmpty=0, Mutex=1;
```

producer

```
while (1) {
  NotFull.wait();
    Mutex.wait();
      Buf[in] = x;
      in = (in+1)%n;
    Mutex.signal();
  NotEmpty.signal();
}
```

consumer

```
while (1) {
  NotEmpty.wait();
    Mutex.wait();
      x = Buf[out];
      out = (out+1)%n;
    Mutex.signal();
  NotFull.signal();
}
```

notifications

critical section

Constructing A Solution: 2/5

This Is a Solution to the Bounded Buffer Problem

only one slots

```
semaphore NotFull=1, NotEmpty=0, Mutex=1;
```

only 1 buffer slot needed

producer

```
while (1) {
  NotFull.wait();
    Mutex.wait();
      Buf[in] = x;
      in = (in+1)%n;
    Mutex.signal();
  NotEmpty.signal();
}
```

consumer

```
while (1) {
  NotEmpty.wait();
    Mutex.wait();
      x = Buf[out];
      out = (out+1)%n;
    Mutex.signal();
  NotFull.signal();
}
```

notifications

no need to update `in` and `out`

critical section

Constructing A Solution: 3/5

```
semaphore NotFull_A=1, NotEmpty_A=0, Mutex_A=1; // for Buf_A
semaphore NotFull_B=1, NotEmpty_B=0, Mutex_B=1; // for Buf_B
Semaphore Amutex = 1,  Bmutex = 1;
```

PUT(Var_A, Buf_A)

```
while (1) {
  Wait(Amutex);
    Wait(NotFull_A);
      Wait(Mutex_A);
        Buf_A = Var_A;
      Signal(Mutex_A);
    Signal(NotEmpty_A);

    Wait(NotEmpty_B);
      Wait(Mutex_B);
        Var_A = Buf_B;
      Signal(Mutex_B);
    Signal(NotFull_B);
  Signal(Amutex);
}
```

GET(Var_A, Buf_B)

PUT(Var_B, Buf_B)

```
while (1) {
  Wait(Bmutex);
    Wait(NotFull_B);
      Wait(Mutex_B);
        Buf_B = Var_B;
      Signal(Mutex_B);
    Signal(NotEmpty_B);

    Wait(NotEmpty_A);
      Wait(Mutex_A);
        Var_B = Buf_A;
      Signal(Mutex_A);
    Signal(NotFull_A);
  Signal(Bmutex);
}
```

GET(Var_B, Buf_A)

There are 2 critical sections protected by Mutex_A and Mutex_B.

Are they needed?

Constructing A Solution: 4/5

```
semaphore NotFull_A=1, NotEmpty_A=0, Mutex_A=1;
semaphore NotFull_B=1, NotEmpty_B=0, Mutex_B=1;
Semaphore Amutex = 1,  Bmutex = 1;
```

None of these two mutexes are needed.

PUT(Var_A, Buf_A)

```
while (1) {
  Wait(Amutex);
    Wait(NotFull_A);
      Wait(Mutex_A);
        Buf_A = Var_A;
      Signal(Mutex_A);
    Signal(NotEmpty_A);

    Wait(NotEmpty_B);
      Wait(Mutex_B);
        Var_A = Buf_B;
      Signal(Mutex_B);
    Signal(NotFull_B);
  Signal(Amutex);
}
```

GET(Var_A, Buf_B)

PUT(Var_B, Buf_B)

```
while (1) {
  Wait(Bmutex);
    Wait(NotFull_B);
      Wait(Mutex_B);
        Buf_B = Var_B;
      Signal(Mutex_B);
    Signal(NotEmpty_B);

    Wait(NotEmpty_A);
      Wait(Mutex_A);
        Var_B = Buf_A;
      Signal(Mutex_A);
    Signal(NotFull_A);
  Signal(Bmutex);
}
```

GET(Var_B, Buf_A)

Only one `A` can pass `NotFull_A`.
Only one `B` can pass `NotFull_B`.

A `B` can reach `Mutex_A` only after an `A` signals `NotEmpty_A`.

Hence, `A` and `B` cannot reach the same critical section `Mutex_A` at the same time.

Constructing A Solution: 5/5

Hence, `Mutex_A` and `Mutex_B` can be removed.

This is a symmetric solution.

```
semaphore NotFull_A=1, NotEmpty_A=0;
semaphore NotFull_B=1, NotEmpty_B=0;
Semaphore Amutex = 1,  Bmutex = 1;
```

```
while (1) {
  Wait(Amutex);
    Wait(NotFull_A);
      Buf_A = Var_A;
    Signal(NotEmpty_A);

    Wait(NotEmpty_B);
      Var_A = Buf_B;
    Signal(NotFull_B);
  Signal(Amutex);
}
```

```
while (1) {
  Wait(Bmutex);
    Wait(NotFull_B);
      Buf_B = Var_B;
    Signal(NotEmpty_B);

    Wait(NotEmpty_A);
      Var_B = Buf_A;
    Signal(NotFull_A);
  Signal(Bmutex);
}
```

Think Differently: 1/3

1. A solution does not have to be symmetric.
2. Let A be active, and B be passive.
3. B waits for A's message, gets it, and offers its message.
4. Then, A gets this (i.e., B's) message.

Asymmetric Version

```
Semaphore Amutex = 1,    Bmutex = 1;

while (1) {                while (1) {
  Wait(Amutex);              Wait(Bmutex);
    PUT(Var_A, Buf_A);
                               GET(Temp, Buf_A);
                               PUT(Var_B, Buf_B);
                               Var_B = Temp;
    GET(Var_A, Buf_B);
  Signal(Amutex);            Signal(Bmutex);
}                          }
```

Think Differently: 2/3

```
Semaphore NotFull = 1;
Semaphore NotEmpty_A = 0, NotEmpty_B = 0;
Semaphore Amutex = 1,     Bmutex = 1;
```

```
while (1) {                  while (1) {
  Wait(Amutex);                Wait(Bmutex);
    Wait(NotFull);
      Shared = Var_A;
      Signal(NotEmpty_A);
                               Wait(NotEmpty_A);
                                 Temp   = Shared;
                                 Shared = Var_B;
                               Signal(NotEmpty_B);
      Wait(NotEmpty_B);
      Var_A = Shared;
    Signal(NotFull);
  Signal(Amutex);              Signal(Bmutex);
}                            }
```

Think Differently: 3/3

- The symmetric solution has **six** statements in each critical section, and the asymmetric solution has **four** in `Thread_A()`'s critical section and **two** in `Thread_B()`'s.
- Because statements in the asymmetric version are executed sequentially, there are **six** statements. In terms of statement count, both versions are similar.
- Because the symmetric version has **four** waits and the asymmetric one has **two**, in terms of efficiency, the asymmetric version seems to be better.
- On the other hand, because the message exchange sections are identical in both group, the symmetric version may be easier to understand.

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.**
- **Thus, analyzing your work in hand and finding similarity with one or more classical problems is an important skill, so that you don't have to reinvent the wheel.**

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