

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.*

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

Processes in group B

```
T_A()
```

```
{
```

```
while (1) {
```

```
    // do something
```

```
    .....  
    Ex. Message  
    .....
```

```
    // do something
```

```
}
```

```
}
```

```
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;
```

```
T_A()  
{
```

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

```
}
```

```
T_B()  
{
```

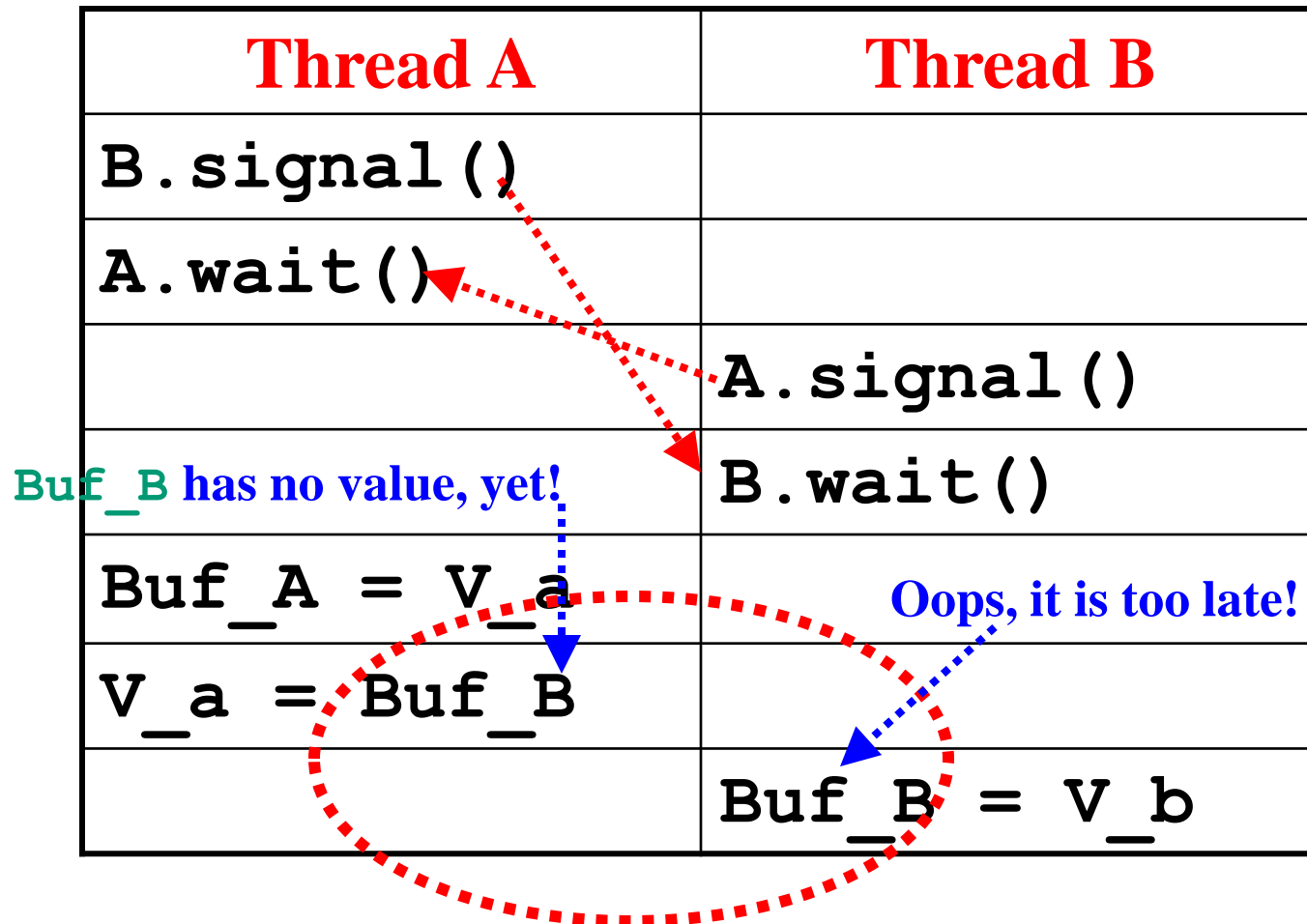
```
    int V_b;  
    while (1) {  
        V_b = ..;  
        A.signal();  
        B.wait();  
        Buf_B = V_b;  
        V_b = Buf_A;
```

```
}
```

I am ready

Wait for your card!

First Attempt: Problem (a)



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 =		

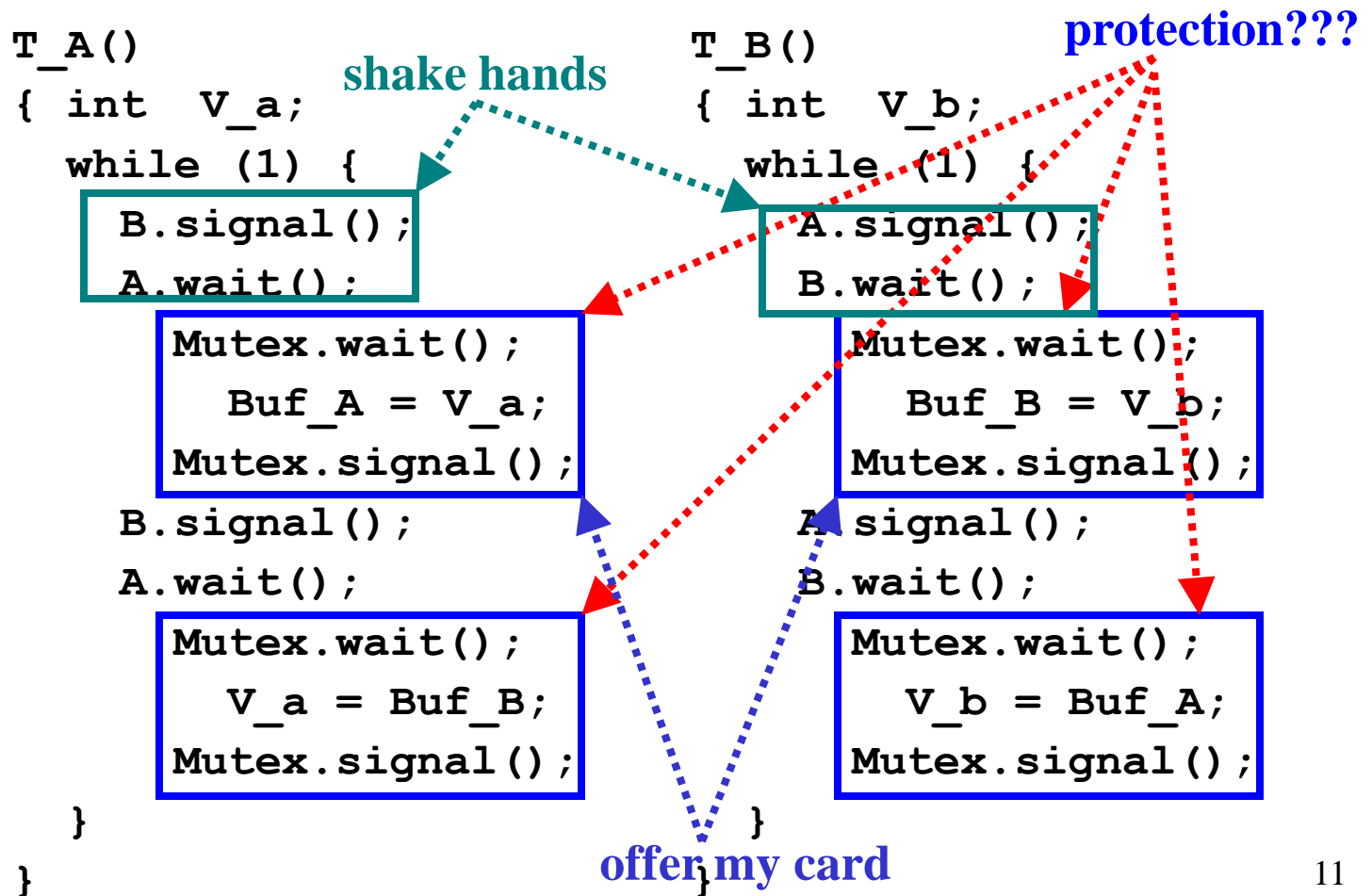
Race Condition

What Did We Learn?

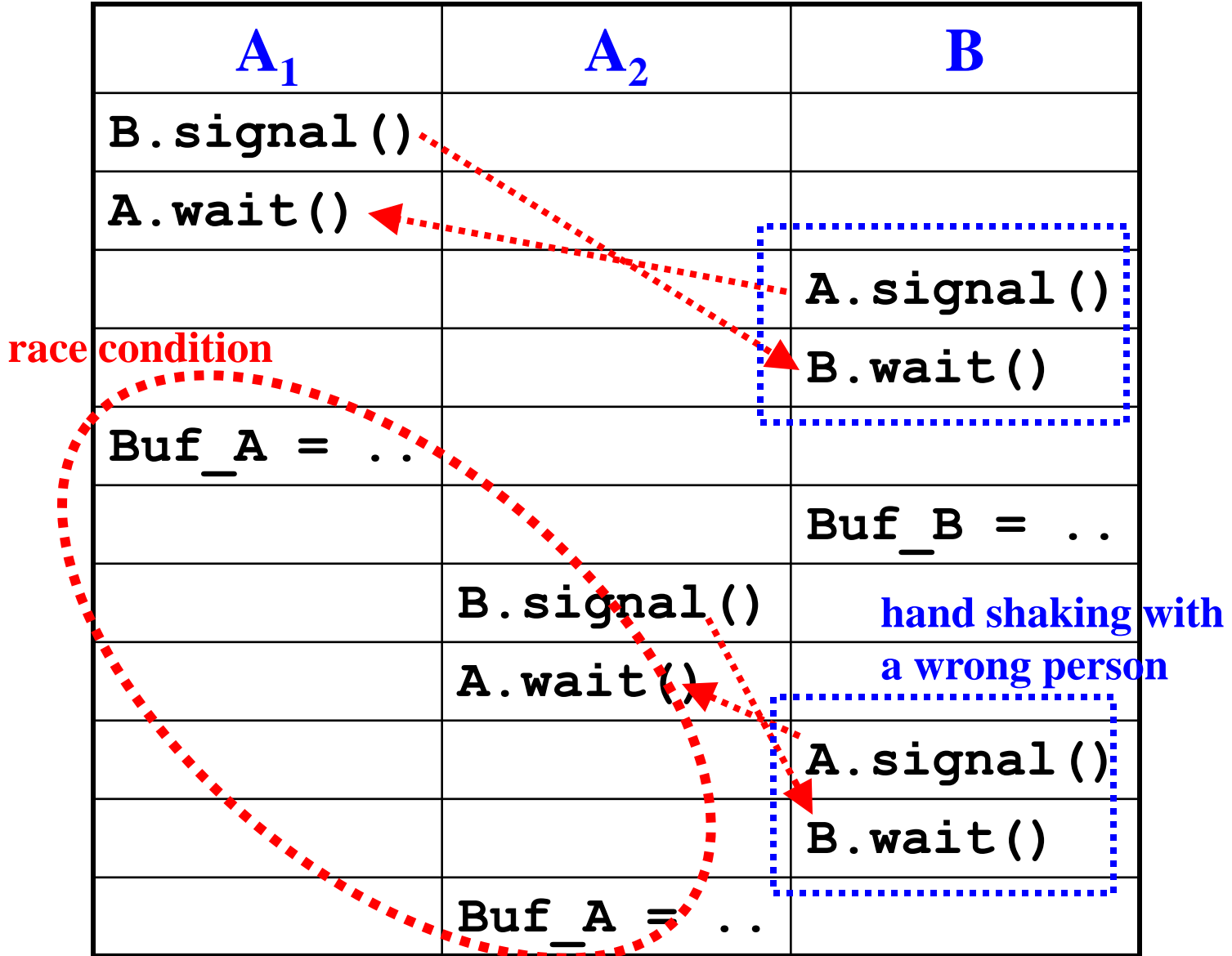
- If there are shared data items, always protect them properly. Without a proper mutual exclusion, race conditions are likely to occur.
- In this first attempt, both global variables `Buf_A` and `Buf_B` are shared and should be protected.

Second Attempt

```
sem  A = B = 0;  
sem  Mutex = 1;  
int  Buf_A, Buf_B;
```



Second Attempt: Problem



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
job done→ 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();

}

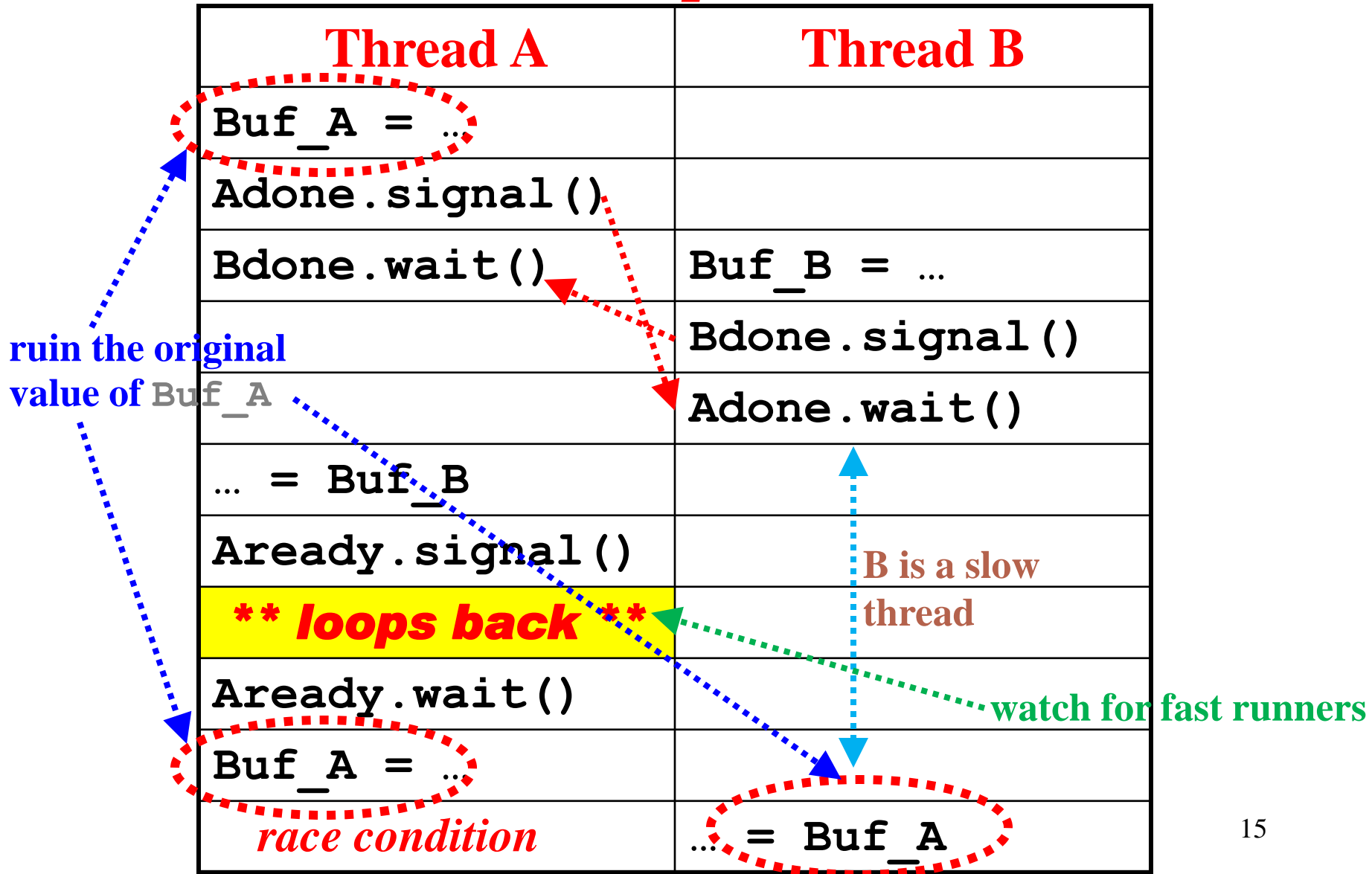
}

only one B
can proceed

only one A
can proceed

here is my card:
let me have
yours

Third Attempt: Problem

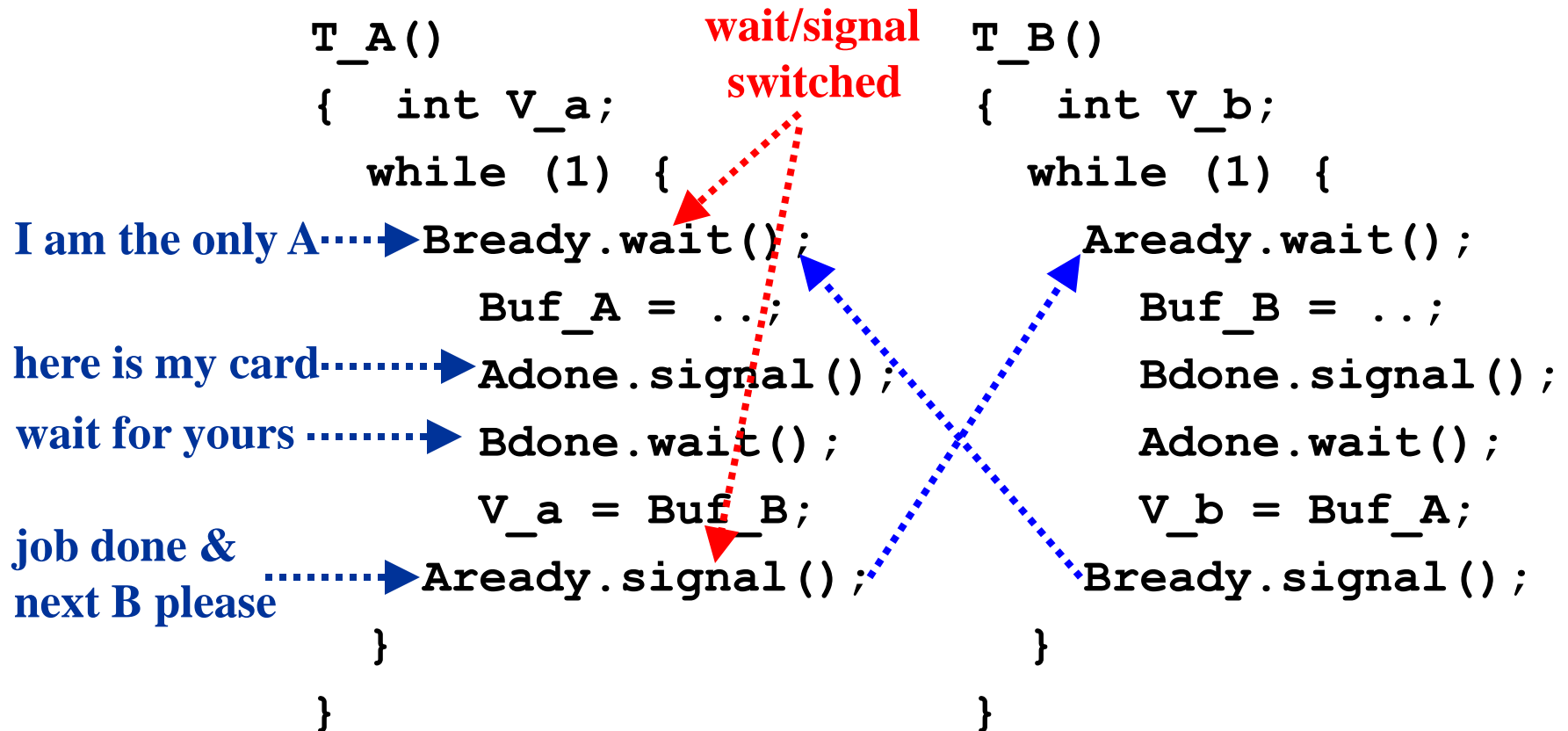


What Did We Learn?

- 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;



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,” **Already** 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)	

The diagram illustrates a sequence of operations for three threads: Thread A_1 , Thread A_2 , and Thread B. The operations are as follows:

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

Red dotted arrows indicate that Thread A_1 's message is retrieved by Thread B and Thread A_2 's message is retrieved by Thread B. Gray dotted arrows indicate that Thread B's message is retrieved by Thread A_2 and Thread A_1 's message is retrieved by Thread A_2 .

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;
```

Is this solution correct?

```
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: 5/7

■ **Deadlock! Deadlock! Deadlock!**

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

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

```
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 **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 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