Catching Race Conditions
An Extremely Difficult Task

- ***Statically*** **detecting race conditions in a program using multiple semaphores is NP-hard.**
- **Thus, no efficient algorithms are available. We have to use human debugging skills.**
- **It is virtually impossible to catch race conditions *dynamically* because hardware must examine *every* memory access.**
- **So, 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 a function `T_A()`, and each process in B runs a function `T_B()`.
- Both `T_A()` and `T_B()` have an infinite loop and never stop.
- In the following, we show execution sequences that cause race conditions. You can always find a correct execution sequence 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 retrieves 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 retrieves B's message.**
- ***How about exchanging business cards?***

Watch for Race Conditions

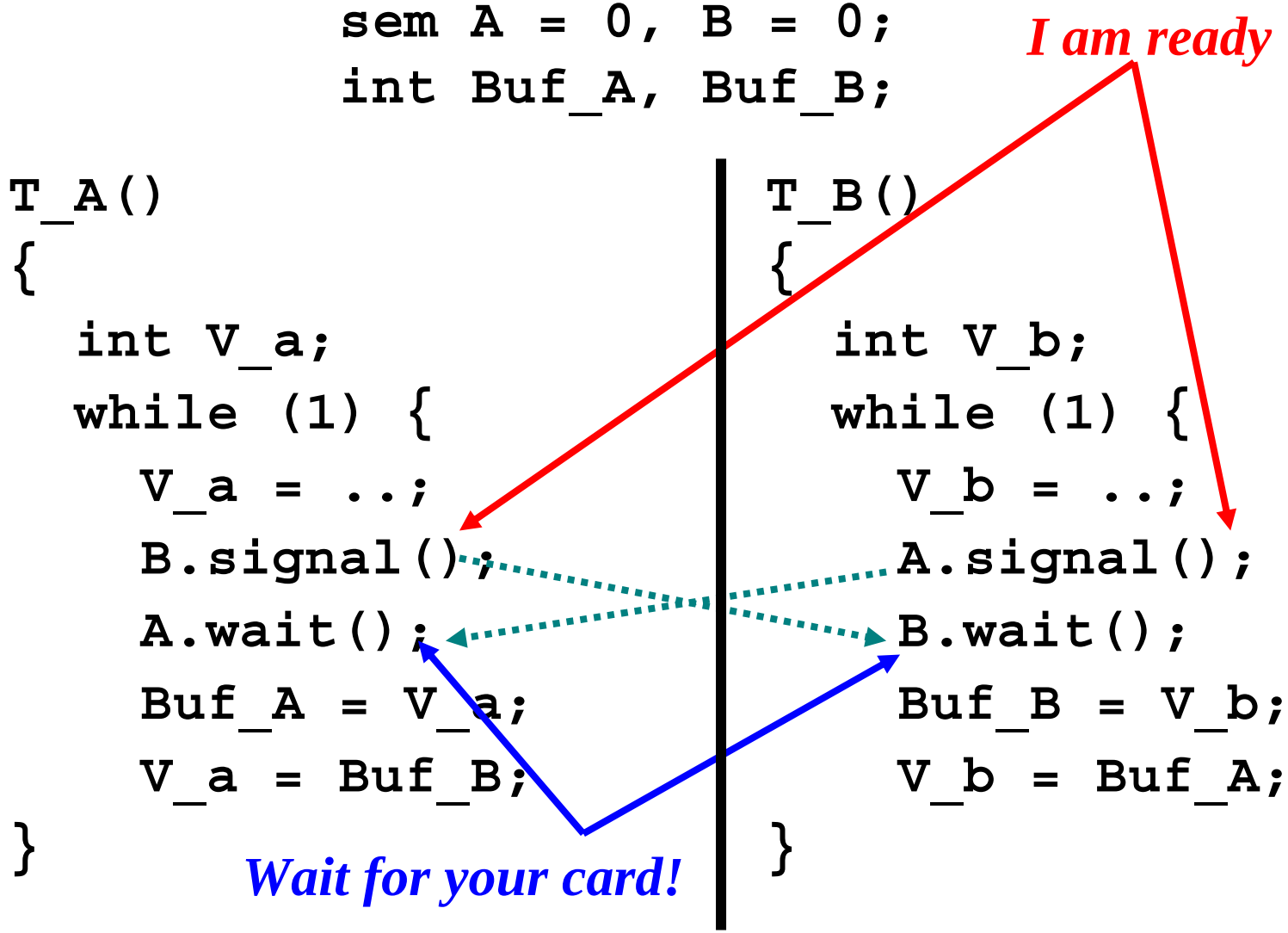
- **Suppose process A_1 presents its message for B to retrieve. If A_2 comes for message exchange before B retrieves A_1's, will A_2's message overwrites A_1's?**
- **Suppose B has already retrieved A_1's message. Is it possible that when B presents its message, A_2 picks it up rather than A_1?**
- **Thus, 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;
}
```



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

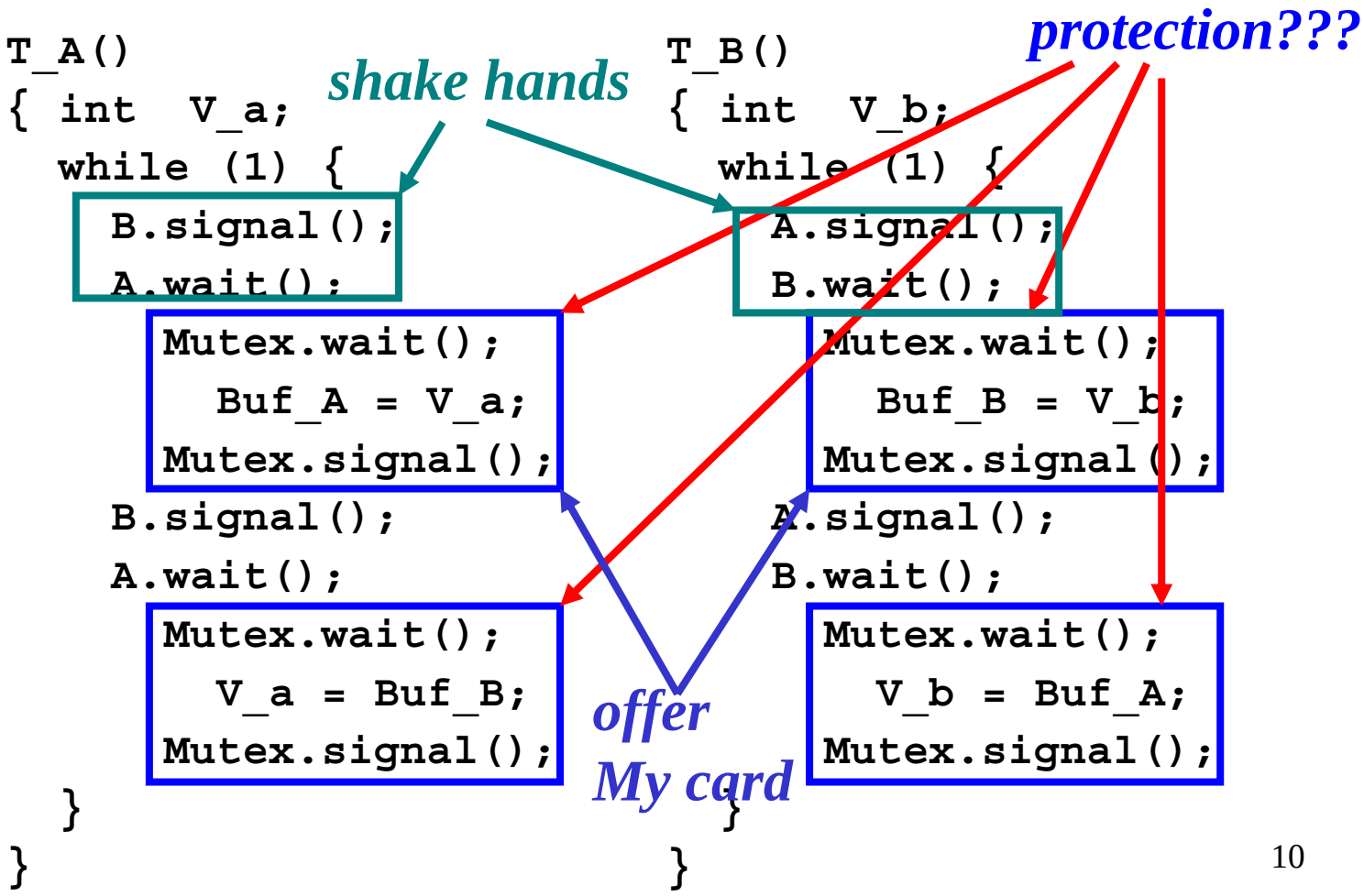
Race Condition

What did we learn?

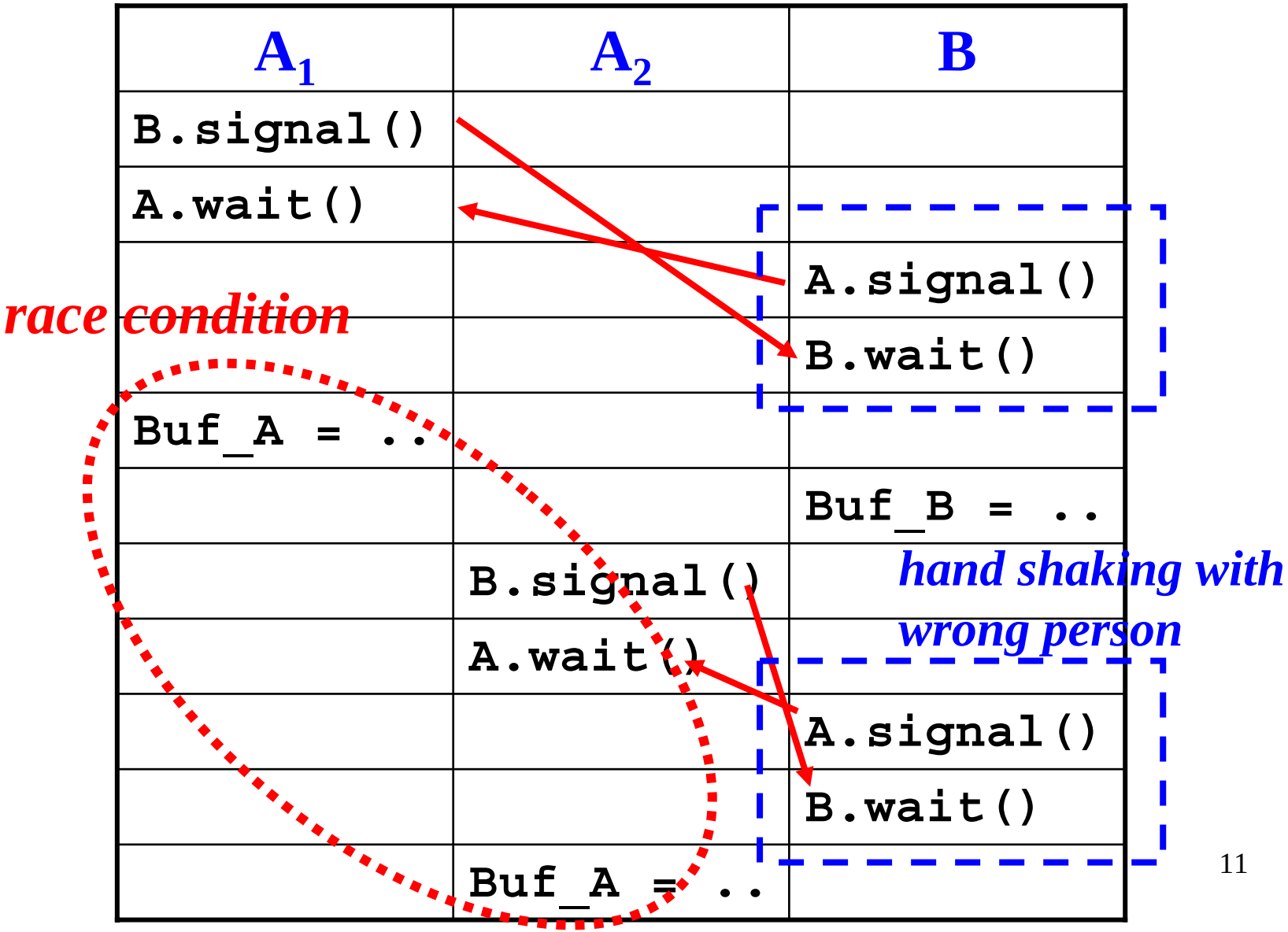
- **If there are shared data items, always protect them properly. Without a proper mutual exclusion, race conditions are likely to occur.**
- **In this first attempt, both global variables `Buf_A` and `Buf_B` are shared and should be protected.**

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 forgot that protection is done by a critical section, which *cannot be divided*.**
- **Thus, protecting "here is my card" followed by "may I have yours" separately is not good enough.**

Third Attempt

job done →

```
sem Aready = Bready = 1;   ← ready to proceed
sem Adone = Bdone = 0;
int Buf_A, Buf_B;
```

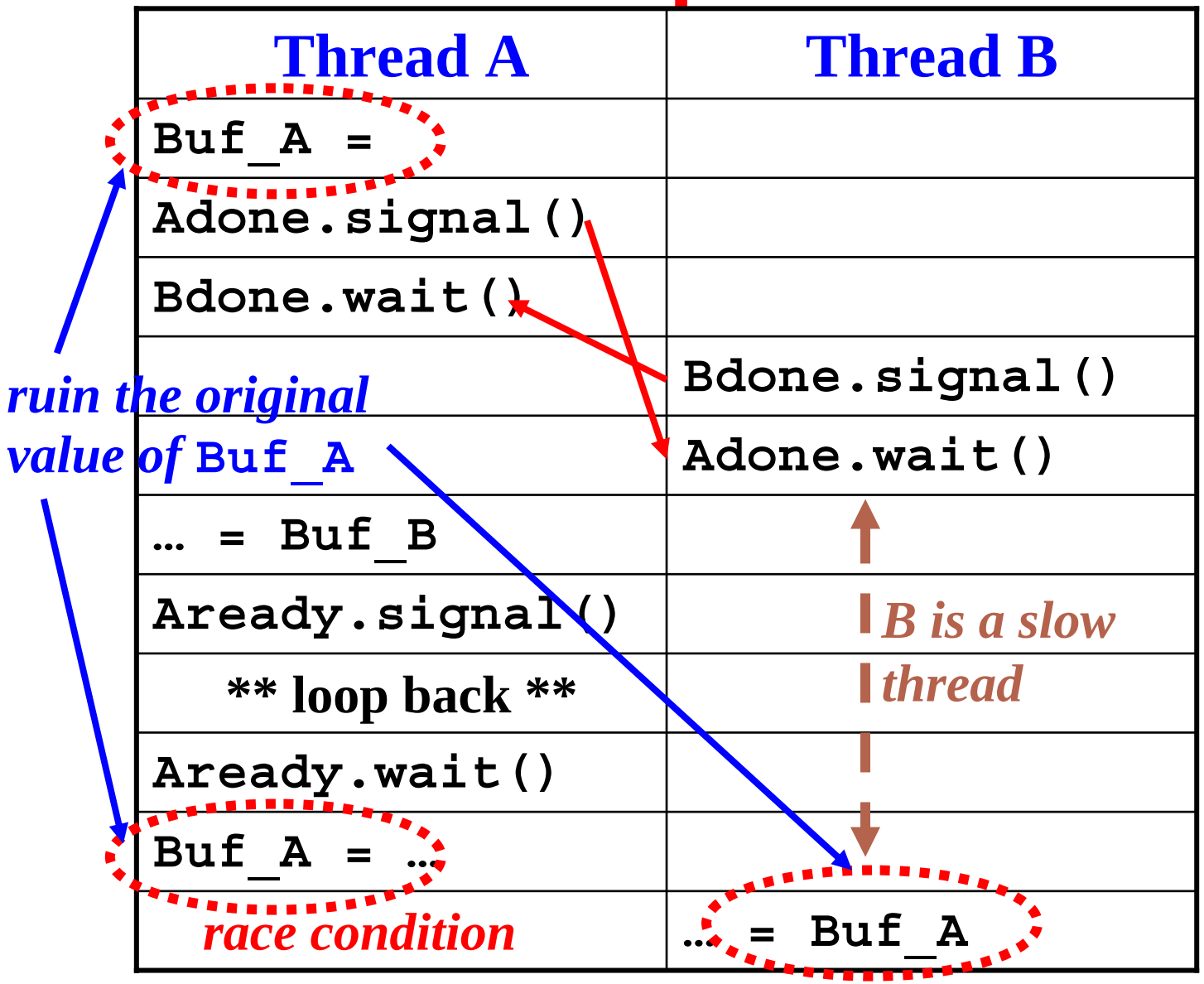
```
T_A()
{ int V_a;
  while (1) {
    Aready.wait();
      Buf_A = ..;
      Adone.signal();
      Bdone.wait();
      V_a = Buf_B;
    Aready.signal();
  }
}
```

only one A can Pass this point

here is my card, let me have yours

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

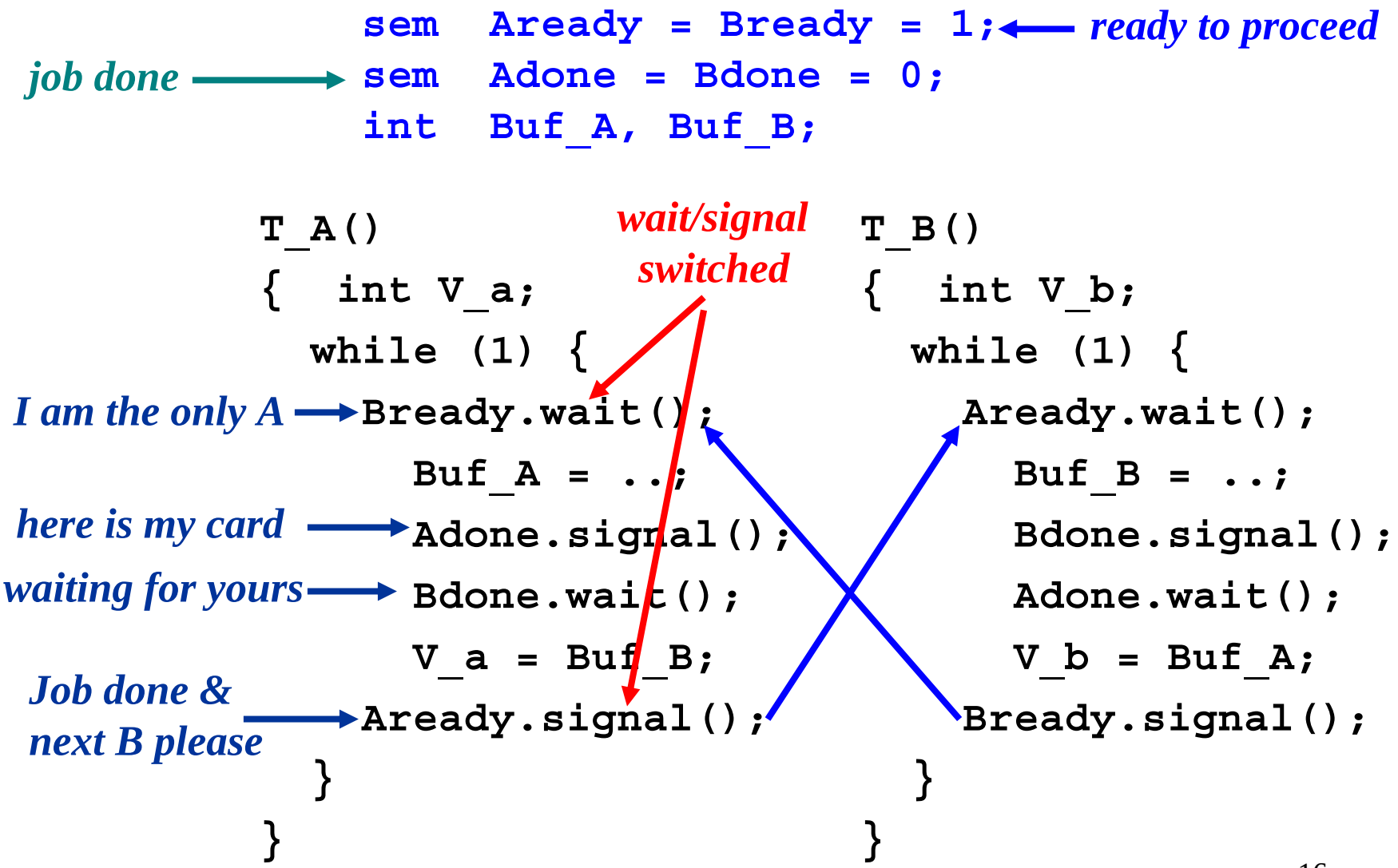
Third Attempt: Problem



What did we learn?

- **Mutual exclusion for one group may not prevent processes in other groups from interacting with a process in this group.**
- **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



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()`	
	`……`	
	`Bdone.wait()`	
	`… = Buf_B`	

Hey, this one is for A_1!!!

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 **A** but released by a process **B**. This is risky!
- In this case, a pure lock is more natural than a binary semaphore.

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.
- Check the ThreadMentor tutorial pages for more details and correct solutions.