Lecture 09: Fair Locks

COSC 273: Parallel and Distributed Computing

Spring 2023

Last Week

A Tale of Two Pups

Mutual Exclusion

Safety Goal:

- Both dogs are not simultaneously out in the yard
  - *mutual exclusion* property

Liveness Goal:

- If a dog needs to go outside, eventually one does
  - *deadlock-freedom* property

A(n Aysmmentric) Protocol I

When Finn needs to go out:

1. Raise flag
2. While Scott's flag is raised, wait
3. Let Finn out
4. When Finn comes in, lower flag

A(n Aysmmentric) Protocol II

When Ru needs to go out:

1. Raise flag
2. While Will's flag is raised:
    1. lower flag
    2. wait until Will's flag is lowered
    3. raise flag
3. When Scott's flag is up and Will's is down, release Ru
4. When Ru returns, lower flag

(Ru) Scott defers to Will (Finn)

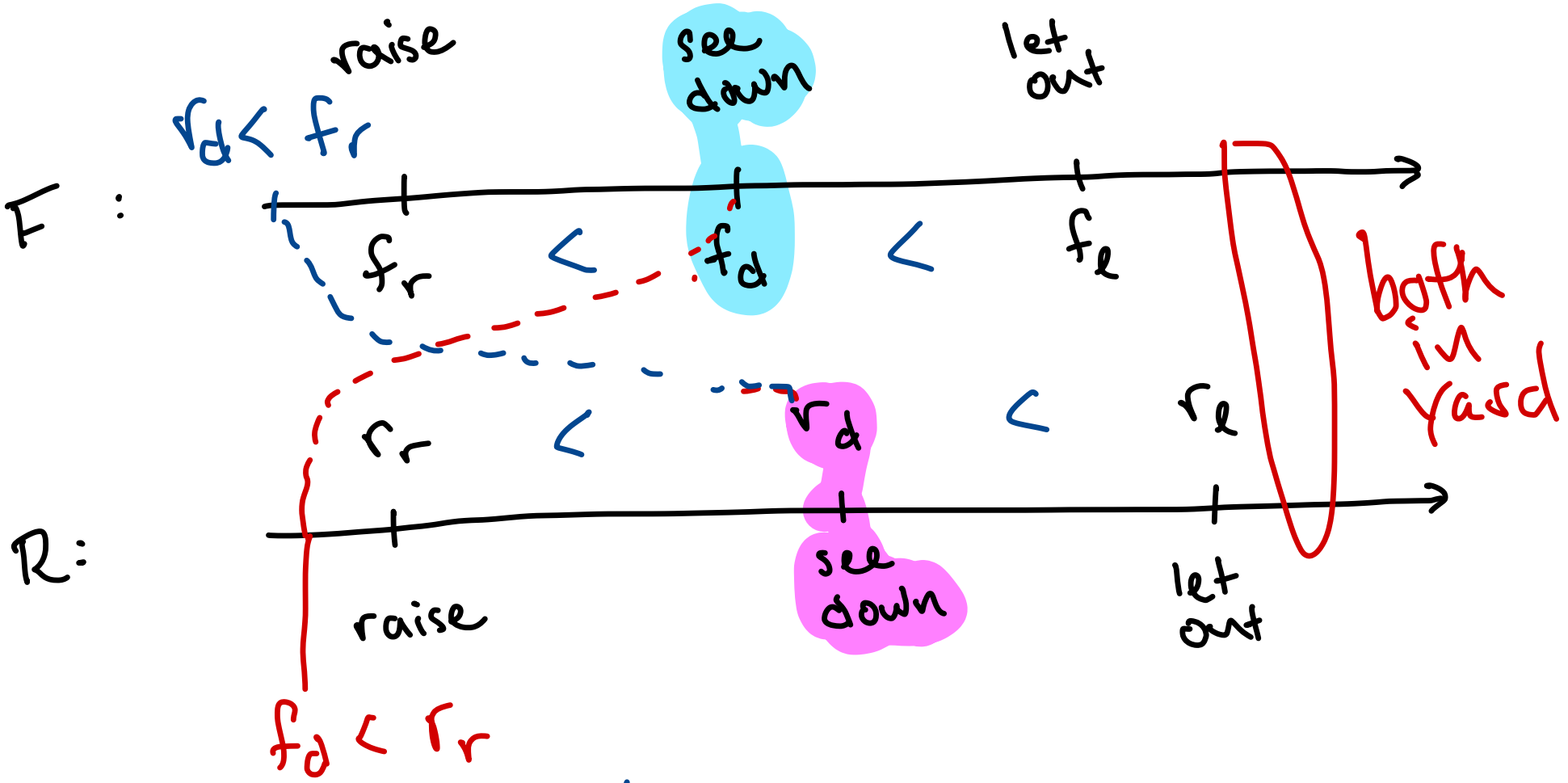
Crucial Observation

Before letting a dog out, both Scott and Will do:

1. raise flag
2. see other's flag down
3. let dog out

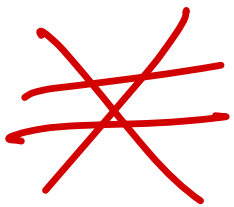
Assume both dogs in yard @ same time
→ Derive contradiction

Mutual Exclusion & Timelines



Everything we know:

$f_r < f_d < r_r < r_{rt} < f_r$



Deadlock Freedom

Claim. If a dog wants to go out, eventually some dog will.

Why?

Case 1: no contention — other flag not raised, one dog out

Case 2: contention — both want to go out

→ Finn goes out

Question

Is the protocol "**fair**?"

→ No:

If both dogs need to go out, Finn could get to go out repeatedly and always block Ru.

↳ convince self this is possible.

Fairness Condition

Safety Goal:

- Both dogs are not simultaneously out in the yard
  - *mutual exclusion* property

Liveness Goals:

- If a dog needs to go outside, eventually one does
  - *deadlock-freedom* property

Fairness Condition

Safety Goal:

- Both dogs are not simultaneously out in the yard
  - *mutual exclusion* property

Liveness Goals:

- If a dog needs to go outside, eventually one does
  - *deadlock-freedom* property
- If a dog needs to go outside, eventually *that dog* does
  - *starvation-freedom* property

Peterson Lock

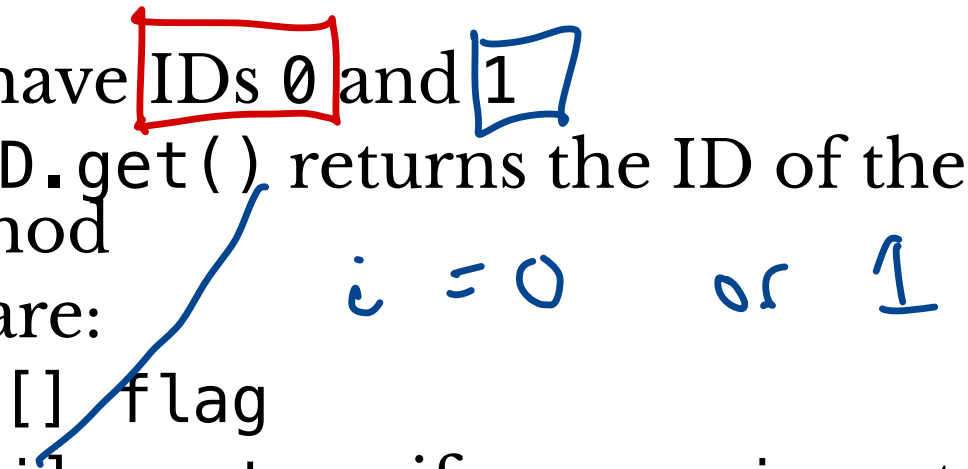
Back to Computers

Two processes (threads) want to access a shared resource

- e.g., increment `Counter` object

Assume:

- processes have IDs `0` and `1`
  - `ThreadID.get()` returns the ID of the thread calling the method
- threads share:
  - `boolean[] flag`
    - `flag[i] == true` if process `i` wants to use resource
  - `int victim`
    - `victim == i` if process `i` is willing to wait (like Ru)



Peterson Lock Idea

Similar to asymmetric protocol with Finn and Ru, but can switch roles.

safe to let my dog out

To obtain lock:

1. indicate intent: set my `flag` to `true`
2. defer to other thread: set myself as `victim`
3. wait until either
   - other thread's flag is `false`, or
   - I am not `victim`

obtained lock when no longer waiting

To release lock:

1. set my `flag` to `false`

Peterson lock Pseudocode

```
public void lock() {
    int i = ThreadID.get(); // get my ID, 0 or 1
    int j = 1 - i;          // other thread's ID

    flag[i] = true;         // set my flag
    victim = i;             // set myself to be victim
    while (flag[j] && victim == i) {
        // wait
    }
}
```

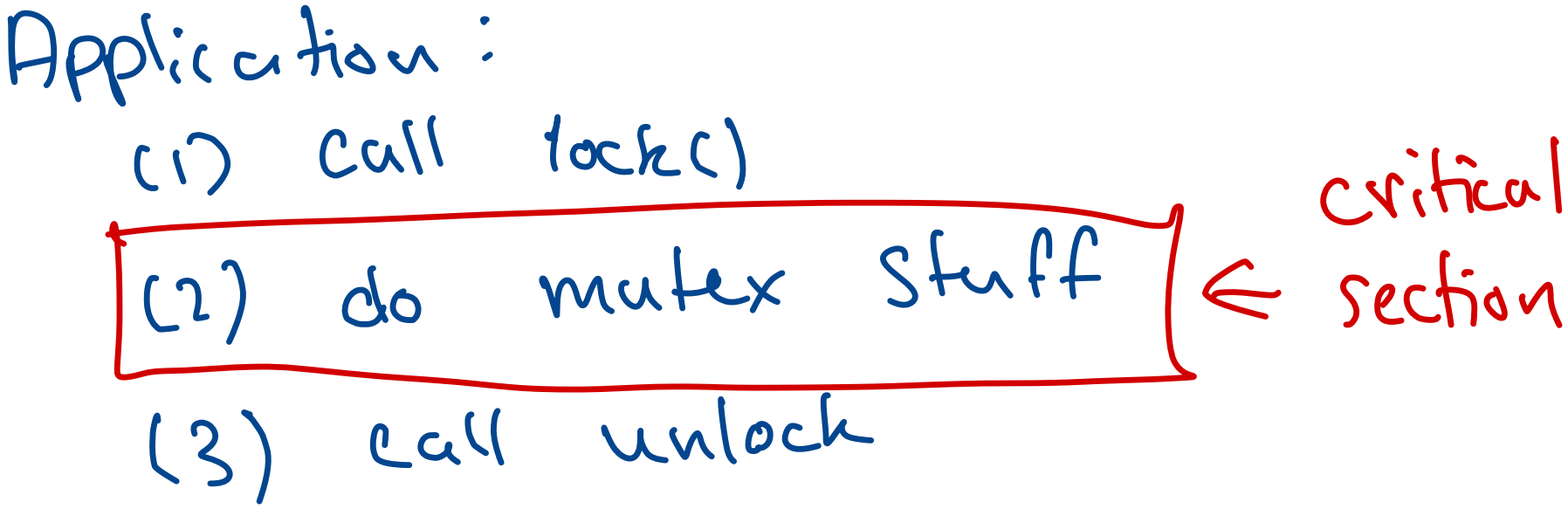
other flag is up

i am victim

"obtained lock" when lock() returns

Peterson unlock Pseudocode

```
public void unlock() {
    int i = ThreadID.get();
        flag[i] = false;
}
```



Goals

Mutual Exclusion. If both threads concurrently call `lock()`, then both cannot return until other calls `unlock()`.

Starvation Freedom. If thread `i` calls `lock()` then eventually thread `i` returns.

Question. Why does Peterson lock achieve these properties?

Proof of Mutual Exclusion I

Suppose not...

- *A* and *B* concurrently call `lock()`
- both return before other calls `unlock()`

In this case we say both threads enter **critical section**

Want to avoid

Proof of Mutual Exclusion II

Atomic operations:

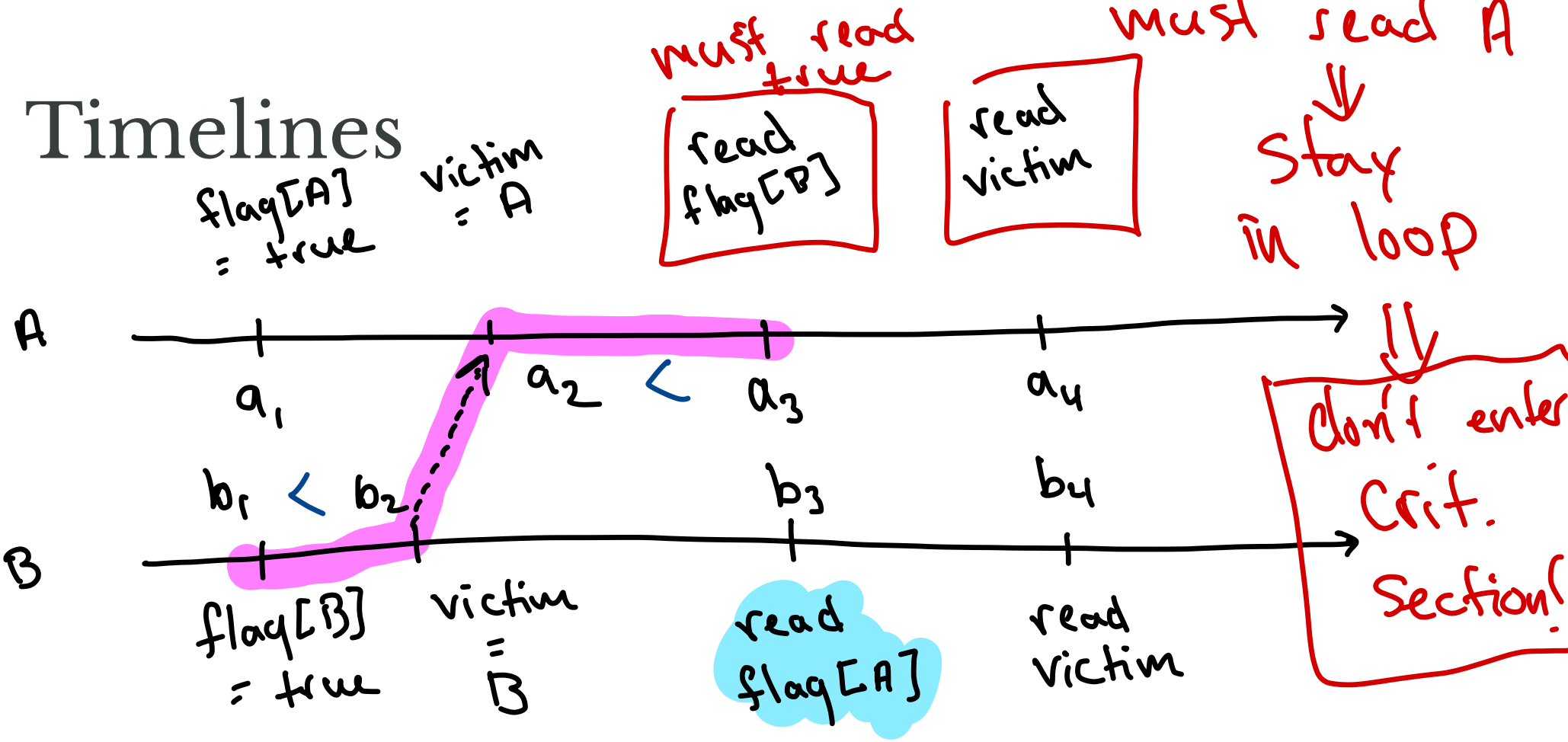
- Actions of A:
  - (A.1) writes `flag[A] = true`
  - (A.2) writes `victim = A`
  - (A.3) reads `flag[B]`
  - (A.4) reads `victim`
- Actions of B:
  - (B.1) writes `flag[B] = true`
  - (B.2) writes `victim = B`
  - (B.3) reads `flag[A]`
  - (B.4) reads `victim`

Proof of Mutual Exclusion III

Suppose $(B.2) \rightarrow (A.2)$:

- i.e., A wrote to `victim` last
- if not, continue argument with roles of A and B reversed

Timelines



$\Rightarrow$ both threads do not return (enter C.S.) before other unlocks.

Conclusion I

The Peterson lock satisfies mutual exclusion!

Starvation Freedom I

Claim. If thread *A* calls `lock()`, eventually the method will return.

```
public void lock() {
   int i = ThreadID.get(); // get my ID, 0 or 1
   int j = 1 - i;          // other thread's ID
   flag[i] = true;         // set my flag
   victim = i;             // set myself to be victim
   while (flag[j] && victim == i) { /*wait*/ }
}
```

Case 1. *A* reads `flag[B] == false` or `victim == B`.

Starvation Freedom II

Claim. If thread A calls `lock()`, eventually the method will return.

```
public void lock() {
   int i = ThreadID.get(); // get my ID, 0 or 1
   int j = 1 - i;          // other thread's ID
   flag[i] = true;         // set my flag
   victim = i;             // set myself to be victim
   while (flag[j] && victim == i) { /*wait*/ }
}
```

Case 2. A reads `flag[B] == true` and `victim == A`.

Starvation Freedom III

Assumption. Once thread `B` obtains lock, eventually `B` calls `unlock()`

```
public void unlock() {
    int i = ThreadID.get();
        flag[i] = false;
}
```

What then happens to thread `A`?

Conclusion II

The Peterson lock satisfies starvation freedom!

Next Time

Locks for more threads!