Lecture 10: More Locks

COSC 273: Parallel and Distributed Computing

Spring 2023

Last Time: Fair Locks

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 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
    }
}
```

Peterson `unlock` Pseudocode

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

Left Off

Showed:

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

Today:

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

Why?

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`.

then never enter loop

→ return
(obtain lock)

Starvation Freedom II

"Wait Free"

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`.

I enter while loop

Then: if B sets flag to false or sets self to victim, A gets lock

Starvation Freedom III

Assumption. After B obtains lock, B calls `unlock()`

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

What then happens to thread A?

- either thread A reads flag[B] is false, obtains lock
- or thread B call lock again, sets victim = B and A obtains lock

Conclusion II

The Peterson lock satisfies starvation freedom!

Semantics of Peterson Lock

- `flag` variable signals *intent* to enter CS (critical section)
  - easily generalizes to more threads
- `victim` variable signals *priority* to enter CS
  - `victim = me` means you have priority
- For more threads
  - more victims?
    - how decide priority among victims?
  - how can this system be fair?

Lamport’s Bakery Algorithm

Locks for more threads!

Lamport’s Inspiration for Priority

Now serving
17

An Attempt

Setup:

- n threads, IDs `0, 1,...,n-1`
- `flag` is Boolean array of size n
  - `flag[i] == true` if thread `i` wants to obtain lock
- `label` is integer array of size n
  - `label[i]` is priority of thread `i`

i's # from dispenser

An Attempt

Setup:

- n threads, IDs `0, 1,...,n-1`
- `flag` is Boolean array of size n
  - `flag[i] == true` if thread `i` wants to obtain lock
- `label` is integer array of size n
  - `label[i]` is priority of thread `i`

Attempt:

- indicate intent: set `flag[i] = true`
- set priority: `label[i] = 1 + max(label[0],...,label[n-1])`
- wait until `label[i]` is smallest label with corresponding `flag` set to `true`

only consider j w/ flag[j] == true

Question

Why won't this attempt work?

→ could have multiple threads w/ same label

Breaking Priority Ties

Two processes may see the same set of tickets and take same label:

- have `label[i] == label[j]` for `i != j`

Breaking Priority Ties

Two processes may see the same set of tickets and take same label:

- have `label[i] == label[j]` for `i != j`

Solution:

Break ties by ID:

- if `label[i] == label[j]` and `i < j`, then `i` has priority

Use **lexicographic order** on pairs `(label[i], i)`

Question About Tie-breaking

Is this process fair?

- Seems we are *always* giving priority to thread `0`…

But each successive setting of labels in strictly increasing

Lamport's Bakery Algorithm

Fields:

- `boolean[] flag`
  - `flag[i] == true` indicates `i` would like enter CS
- `int[] label`
  - `label[i]` indicates "ticket" number held by `i`

Initialization:

- set all `flag[i] = false`, `label[i] = 0`

Locking

Locking Method:

```
public void lock () {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (!hasPriority(i)) {} // wait
}
```

The method `hasPriority(i)` returns `true` if and only if there is no `k` such that

- `flag[k] == true` and
- either `label[k] < label[i]` or `label[k] == label[i]` and `k < i`

Unlocking

Just lower your flag:

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

Bakery Algorithm is Deadlock-Free

```
public void lock () {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (!hasPriority(i)) {} // wait
}
```

Why?

First-come-first-served (FCFS)

- If: *A* writes to `label` before *B* calls `lock()`,
- Then: *A* enters CS before *B*.

```
public void lock () {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (!hasPriority(i)) {} // wait
}
```

Why?

Bakery Algorithm is Starvation-Free

Thread `i` calls `lock()`:

- `i` writes `label[i]`
- By FCFS, subsequent calls to `lock()` by `j != i` have lower priority
- By deadlock-freedom every `k` ahead of `i` eventually releases lock

So:

- `i` eventually served

Bakery Algorithm Satisfies MutEx

```
public void lock () {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (!hasPriority(i)) {} // wait
}
```

Suppose not:

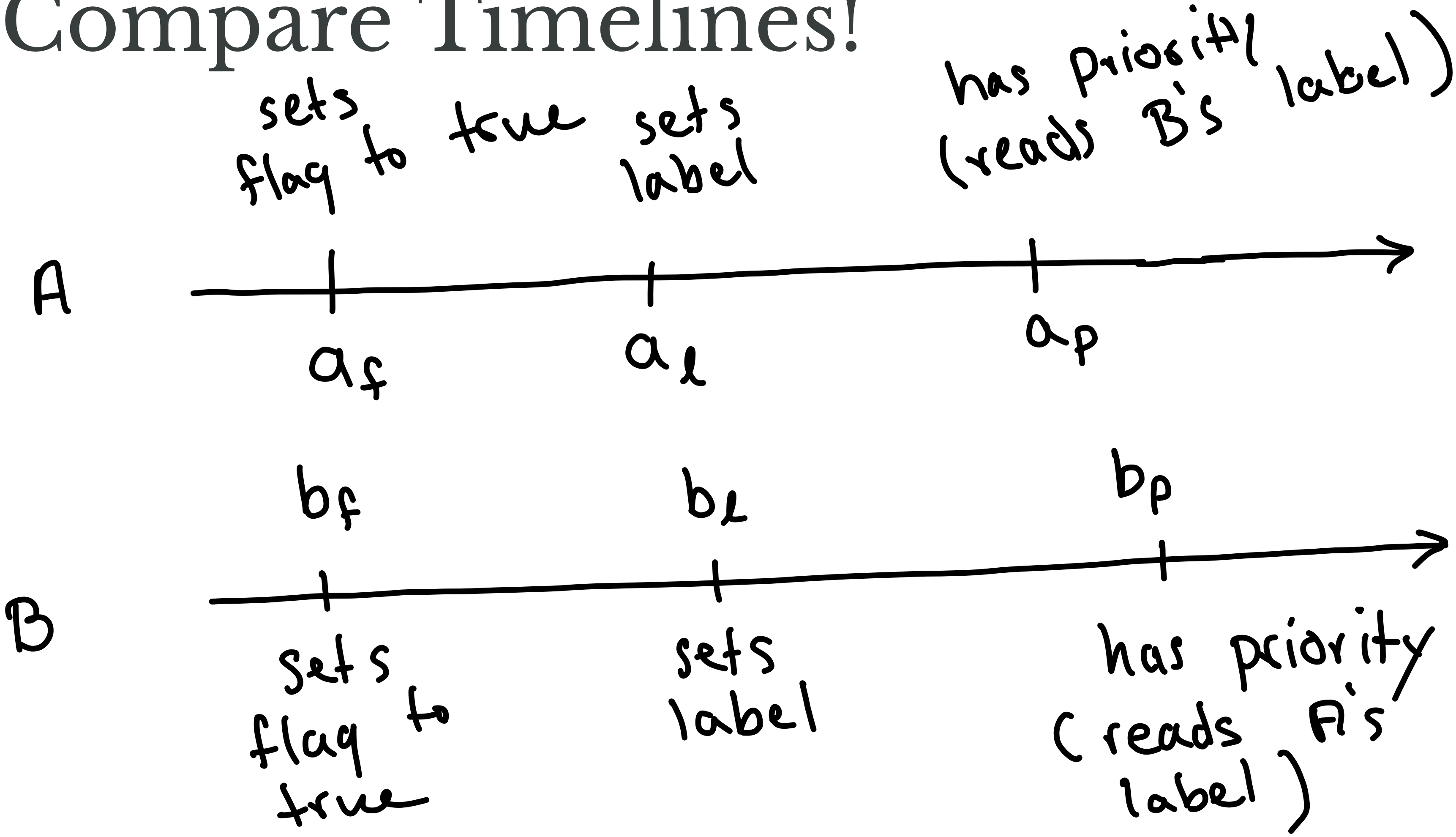
- A and B concurrently in CS
- Assume: $(\text{label}(A), A) < (\text{label}(B), B)$ while both in CS

Proof (Continued)

Since B entered CS:

- Must have read
  - $(\text{label}(B), B) < (\text{label}(A), A)$, or
  - $\text{flag}[A] == \text{false}$
- Former can not happen: labels strictly increasing
- So B read $\text{flag}[A] == \text{false}$

Compare Timelines!



Conclusion

Lamport's Bakery Algorithm:

1. Works for any number of threads
2. Satisfies MutEx and starvation-freedom

Is the bakery algorithm practical?

Two Issues:

1. For n threads, need arrays of size n
   - `hasPriority` method is costly
   - what if we don't know how many threads?
2. Assume threads have sequential IDs `0, 1,...`
   - not the case with Java!
   - thread IDs are essentially random `long` values

Homework 2 will have questions that address these issues.

Remarkably

We cannot do better:

- If n threads want to achieve mutual exclusion + deadlock-freedom, must have n read/write registers (variables)
- This is really bad if we have a lot of threads!
  - 1,000 threads means each call to `lock()` requires 1,000s of reads
  - each call to `hasPriority` requires either 1,000s of reads or a more advanced data structure
- Things are messy!

A Way Around the Bound

- Argument relies crucially on fact that the *only* atomic operations are `read` and `write`
- Modern computers offer more powerful atomic operations
- In Java, `AtomicInteger` class
  - `getAndIncrement()` is supported **atomic** operation

Homework 2 Use `AtomicIntegers` to get a cleaner and more efficient realization of Lamport's bakery idea.

Next Week

Vector operations!