Synchronization

Recap - Peterson's Lock

v3

```
volatile bool flag[2];
volatile int victim;
lock:
    me = tid;
    other = 1 – me;
    flag[me] = true;
    victim = me;
    while (flag[other] &&
                victim == me)
        ;
unlock():
        flag[tid] = false;
```

- Mutual exclusion is guaranteed.
- Does not lead to deadlock.
- The algorithm ensures progress.

- flag indicates if a thread is interested.
- victim = me is "You before me"

What about N threads?

Bakery Algorithm

- Devised by Lamport
- Works with N threads.
- Maintains FCFS using ever-increasing numbers.

```
bool flag[N]; // false
int label[N];  // 0
lock:
    me = tid;
    flag[me] = true;
    label[me] = 1 + max(label);
    while (∃k != me: flag[k] &&
              (label[k], k) < (label[me], me))
        ;

unlock():
    flag[tid] = false;
```

Bakery Algorithm

- Devised by Lamport
- Works with N threads.
- Maintains FCFS using ever-increasing numbers.

```
bool flag[N]; // false
int label[N];  // 0
lock:
    me = tid;
    flag[me] = true;
    label[me] = 1 + max(label);
    while (∃k != me: flag[k] &&
              (label[k], k) < (label[me], me))
        ;
```

- The code works in absence of caches.
- In presence of caches, mutual exclusion is not guaranteed.
- There are variants to address the issue.

```
flag[tid] = false;
```

max is not atomic.

Bakery Algorithm: GPU?

- Across warps is similar to CPU.
- What happens within warp-threads?
- Threads get the same label, **<** prioritizes.

```
bool flag[N]; // false
int label[N];  // 0
lock:
    me = tid;
    flag[me] = true;
    label[me] = 1 + max(label);
    while (∃k != me: flag[k] &&
             (label[k], k) < (label[me], me))
        ;

unlock():
flag[tid] = false;
```

max is not atomic.

Bakery Algorithm: GPU?

- Across warps is similar to CPU.
- What happens within warp-threads?
- Threads get the same label, **<** prioritizes.

- On GPUs, locks are usually prohibited.
- High spinning cost at large scale.
- But locks are feasible!
- Locks can also be implemented using atomics.

Synchronization

- Control + data flow
- Atomics
- Barriers
- ...

atomics

- Atomics are primitive operations whose effects are visible either none or fully (never partially).
- Need hardware support.
- Several variants: atomicCAS, atomicMin, atomicAdd, ...
- Work with both global and shared memory.

atomics

```
__global__ void dkernel(int *x) {
    ++x[0];
}
...
dkernel<<<2, 1>>>(x);
```

After dkernel completes, what is the value of x[0]?

Classwork: What if the kernel configuration is <<<1, 2>>>?

++x[0] is equivalent to:

    Load x[0], R1
    Increment R1
    Store R1, x[0]

Time

Load x[0], R1	Load x[0], R2
Increment R1	Increment R2
	Store R2, x[0]
Store R1, x[0]	

Final value stored in x[0] could be 1 (rather than 2).
What if x[0] is split into multiple instructions? What if there are more threads?

Atomics in ATMs

Twins at ATMs

Twin withdraws 1000 rupees.

System executes the steps:

- Check if balance is >= 1000.
- If yes, reduce balance by 1000 and give cash to the user.
- Otherwise, issue error.

Twins may be able to get 2000 rupees!

Time

Load x[0], R1	Load x[0], R2
Increment R1	Increment R2
	Store R2, x[0]
Store R1, x[0]	

atomics

```
__global__ void dkernel(int *x) {
    ++x[0];
}
...
dkernel<<<2, 1>>>(x);
```

- Ensure all-or-none behavior.
  - e.g., atomicInc(&x[0], ...);
- **dkernel**`<<<K1, K2>>>` would ensure x[0] to be incremented by exactly `K1*K2` – irrespective of the thread execution order.
  - When would this effect be visible?