Lec 8

Condition Variables

Locks (recap)

- busy or free
- at most one thread "holds" the lock
- acquire(): wait until lock is free
  then atomically take it (→ busy)
- release(): set lock to free

Question: why no method to check if free?
↳ what would we know if that method returned true?

Properties of Locks

Mutual Exclusion: ≤ 1 thread holds lock

Progress: If no thread holds lock, and a thread tries to acquire it, eventually some thread gets the lock.

Bounded Waiting: If thread T attempts to acquire, then there is a bound on how many times other threads acquire lock before T.

not: FIFO

Using locks in code

- for each piece of data that can be concurrently accessed by >1 thread (and is not immutable)

  choose a lock that "protects" that data.

- whenever you want to access that data, first acquire the lock

- this "synchronization discipline" guarantees there will never be concurrent access to data

Monitor Pattern

idea: organize the data with its protecting lock

meh:

```
Lock l = new Lock();
int x=0; // protected by l
⋮
l.acquire();
x++;
l.release();
```

better:

```
class Counter {
    Lock l;
    int x;

    Counter() {
        l = new Lock();
        x=0;
    }

    void inc() {
        l.acquire()
        x++;
        l.release()
    }
}
```

```
main() {
    Counter c = new ...
    ⋮
    c.inc()
}
```

Java Monitors

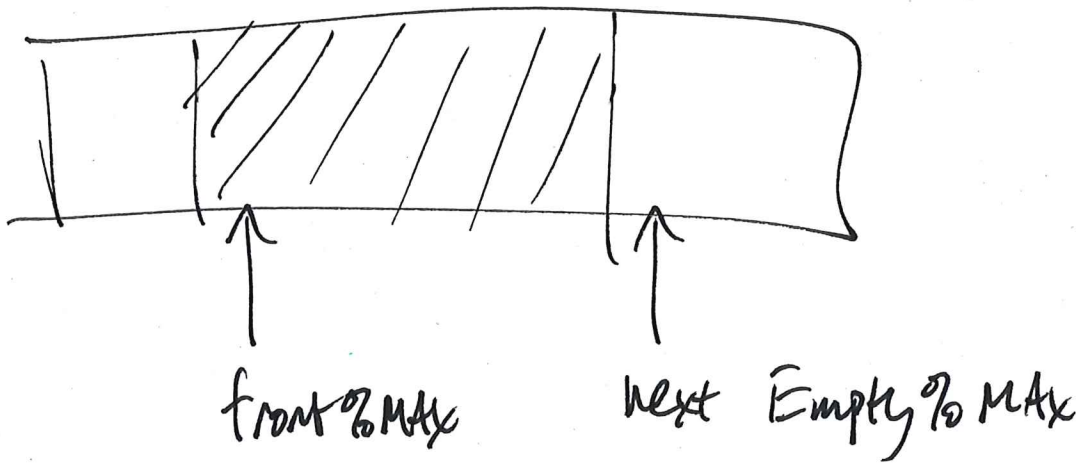
- every object "comes with" a lock

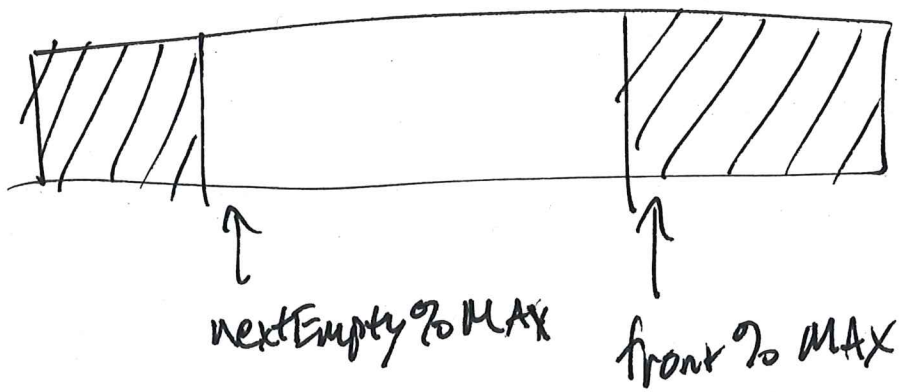
```
class Counter {
    int x

    Counter () { x = 0; }

synchronized void inc() {
        x++;
    }

}
```

- synchronized method acquires lock at top and releases on exit

- synchronized block does same

```
    ⋮
    ⋮
    Synchronized (myObj) {
        ⋮
    }
```


front % MAX
next Empty % MAX


nextEmpty % MAX
front % MAX

Example: Bounded Queue

```
class Queue {
    int items[MAX];
    int front;
    int nextEmpty;

    Queue() {
        front = 0;
        nextEmpty = 0;
    }

    sync. tryInsert(int item) {
        if (nextEmpty - front < MAX) {
            items[nextEmpty++ % MAX] = item;
            return true;
        }
        return false;
    }

    sync. tryRemove() {
        if (front < nextEmpty) {
            int item = items[front++];
            return (true, item);
        }
        return (false, -1);
    }
}
```

Reasoning about Concurrent ADTs

- an ADT impl. maintains certain invariants
  - e.g. front $\leq$ nextEmpty
    nextEmpty - front $\leq$ MAX

  "representation invariants"

- in a concurrent ADT, these invariants are true only when no thread holds the lock
  ↳ if I acquire the lock, I know they are true
  ↳ must re-establish before releasing the lock

Improving the Queue

- we want to wait until there is room
  and then insert
  or wait until non-empty and then remove

- one idea: "polling"
  keep trying in a loop

  simple

  inefficient: keeps CPU busy
  may prevent other threads from running!

- better idea: give up CPU until some condition is true

Condition Variables

- synchronization primitive that lets a thread wait (efficiently) until some condition is true
- each condition variable is associated with a lock

API:

wait(): atomically release the lock and add this thread to the CV's waiting list. suspend thread. when thread is resumed, re-acquire the lock.

signal(): remove one thread from waiting list and mark it RUNNABLE. if no threads waiting, do nothing.

broadcast(): remove <u>all</u> threads from waiting list.

must hold associated lock to call these methods!

8.10

Names are hard

wait / signal / broadcast

Java: wait / notify / notifyAll

xk: sleep / - / wakeup

note: not the same as unix wait/sleep/signal!

8.11

```
class Queue {
    CV notEmpty;
    CV notFull;

    int items[MAX]
    int front = 0;
    int nextEmpty = 0;

    sync. insert(int item) {
        while (nextEmpty - front == MAX) {
            notFull.wait()
        }
        items[nextEmpty++ % MAX]
                = item;
        notEmpty.signal();
    }

    sync. remove() {
        while (front == nextEmpty) {
            notEmpty.wait()
        }
        int item = items[front++];
        notFull.signal();
        return item;
    }
```

8.12

Condition Variable Patterns

- every condition variable should have a documented **condition** "what are you waiting for?"
- every mutable variable in the condition must be protected by the lock
- wait atomically puts thread on waiting list and releases lock.
- signal marks a thread RUNNABLE
  - does not atomically switch to that thread

↳ when wait returns, all you know is you hold lock

↳ must re-check predicate

8.13

Warning: always call wait in a loop

```
while (! condition) {
    cv. wait();
}
```

```
if (! condition) {
    cv. wait();
}
```

If you do this, I will find you

- spurious wakeups
- broadcasts
- other threads

Synchronization in XK

- Spinlock: disable interrupts + spin

- sleeplock: give up processor until lock free

```
// Mutual exclusion lock.
struct spinlock {
  uint locked; // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint64_t pcs[10]; // The call stack (an array of program counters)
                     // that locked the lock.
};

// Acquire the lock. Loops (spins) until the lock is acquired.
void acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock.
  if (holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while (xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

8.16

```
// Release the lock.
void release(struct spinlock *lk) {
  if (!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m"(lk->locked) :);

  popcli();
}
```

8.17

```
static inline uint xchg(volatile uint *addr, uint newval) {
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1"
               : "+m"(*addr), "=a"(result)
               : "1"(newval)
               : "cc");
  return result;
}
```

8.17½

```
// Long-term locks for processes
struct sleeplock {
  uint locked;         // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name; // Name of lock.
  int pid;    // Process holding lock
};

// a sleeping lock relinquishes the processor if the lock is busy
// note mesa semantics: process can wakeup and find the lock still busy.
// NOTE: no spinlocks should be held while calling `acquiresleep`
// (since if `acquiresleep` tries to sleep while a spinlock is held,
// `sched`'s contract will be violated).
void acquiresleep(struct sleeplock *lk) {
  acquire(&lk->lk);

  // We should never try to acquire sleeplock while we already hold it
  if (lk->locked && lk->pid == myproc()->pid) {
    panic("Already holding sleeplock!");
  }

  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

8.18

```
// a sleeping lock wakes up a waiting process, if any, on lock release.
void releasesleep(struct sleeplock *lk) {
  acquire(&lk->lk);

  // Assert that the lock is actually locked.
  if (!lk->locked) {
    panic("releasesleep: sleeplock is not locked!");
  }

  // Assert that we are the one holding the lock.
  if (lk->pid != myproc()->pid) {
    panic("releasesleep: sleeplock is not held by current proc!");
  }

  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

8.19