SE350: Operating Systems

Lecture 6: Synchronization

Outline

- Atomic operations
- Hardware atomicity primitives
- Different implementations of locks

Synchronization Motivation

- When threads concurrently read from or write to shared memory, program behavior is undefined
  - Two threads write to a variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes over different runs of the same program
- Compiler and hardware reorder instructions
  - Generating efficient code needs control and data dependency analysis
  - E.g., store buffer allows next instruction to execute while store is being completed

Question: Can This Panic?

```
// Thread 1

p = someComputation();
pInitialized = true;
```

```
// Thread 2

While (!pInitialized);
q = someFunc(p);
If (q != someFunc(p))
   panic();
```

Too Much Milk Example

	Roommate A	Roommate B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
01:00		Arrive home, put milk away. Oh no!

Atomic Operations

- Operation that always runs to completion or not at all
  - Indivisible: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block: if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy whole array

Definitions

- Race condition: output of concurrent program depends on order of operations between threads
- Synchronization: using atomic operations to ensure cooperation between multiple concurrent threads
  - For now, only loads and stores are atomic
  - We will see that its hard to build anything useful with only load/store
- Mutual exclusion: ensuring that only one thread does a particular operation at a time
  - One thread excludes others while doing its task
- Critical section: piece of code that only one thread can execute at once
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing same thing

Definitions (cont.)

- Lock: prevent someone from doing something
  - Lock before entering critical section, before accessing shared data
  - Unlock when leaving, after done accessing shared data
  - Wait if locked
    - Important idea: synchronization involves waiting!
- Example: fix milk problem by putting a key on refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants OJ

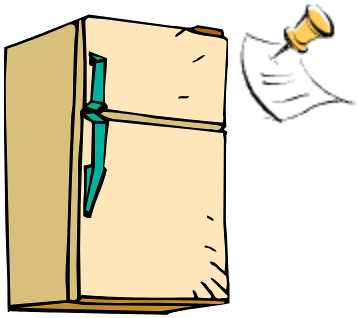
  - Of course, we don't know how to make a lock yet

Too Much Milk: Correctness Properties

- Be careful about correctness of your concurrent programs
  - Behavior could be non-deterministic
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
  - Always write down behavior first
- What are correctness properties of "too much milk" problem?
  - Never more than one person buys
  - Someone buys if needed
- In this lecture, we restrict ourselves to only atomic load/store
- We assume instructions are not reordered by compiler/HW

Too Much Milk (Solution #1)

- Use a note
  - Leave note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Would this work if computer program tries it?
  (remember, only memory load/store are atomic)



```
if (!milk) {
   if (!note) {
      leave note;
      buy milk;
      remove note;
   }
}
```

Solution #1 (cont.)

```
if (!milk) {


    if (!note) {
        leave note;
        buy milk;
        remove note;
    }
}
```

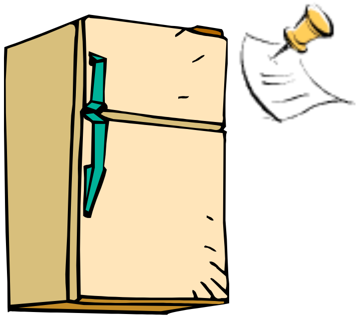
```
if (!milk) {
    if (!note) {






        leave note;
        buy milk;
        remove note;
    }
}
```

Try #1 (cont.)



- Conclusion
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution #1 makes problem worse since it fails intermittently
  - Makes it very hard to debug …
  - Programs must work despite what thread scheduler does!

Too Much Milk (Solution #1½)

- Clearly note is not blocking enough
- Let's try to fix this by placing note first

```
leave note;
if (!milk) {
    if (!note) {
        buy milk;
    }
}
remove note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk (Solution #2)

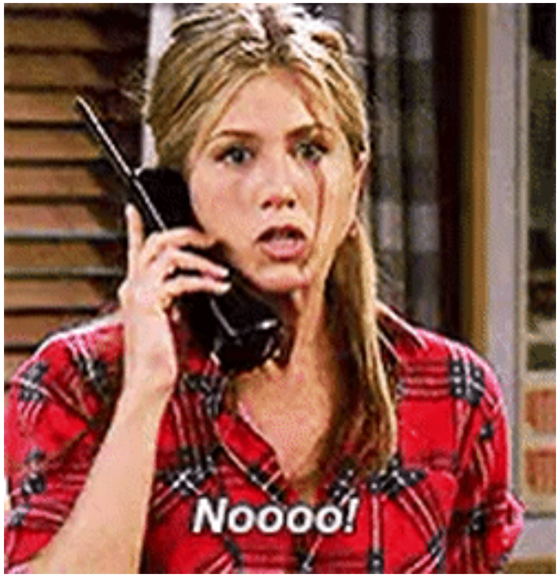
- How about labeled notes?

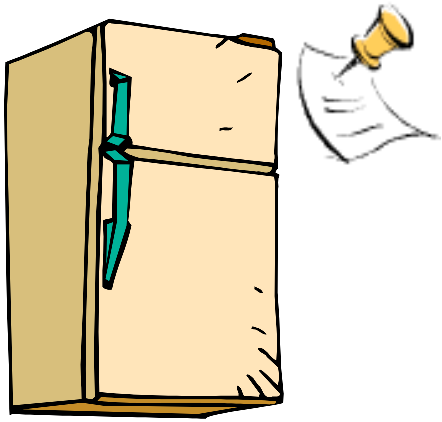
```
// Thread A
leave note A;
if (!note B) {
   if (!milk)
      buy milk;
}
remove note A;
```

```
// Thread B
leave note B;
if (!note A) {
   if (!milk)
      buy milk;
}
remove note B;
```

- Does this work?
  - It is still possible that neither of threads buys milk
  - This is extremely unlikely, but it's still possible

Problem with Solution #2





- I thought *you* had the milk! But I thought *you* had the milk!
- This kind of lockup is called "starvation!"

Too Much Milk (Solution #3)

```
// Thread A
leave note A;
while (note B) // (X)
   do nothing;
if (!milk)
   buy milk;
remove note A;
```

```
// Thread B
leave note B;
if (!note A) { // (Y)
   if (!milk)
      buy milk;
}
remove note B;
```

- Does this work?
  - Yes! It can be guaranteed that it is safe to buy, or others will buy: it is ok to quit
- At (X)
  - If no note from B, safe for A to buy
  - Otherwise, wait to find out what will happen
- At (Y)
  - If no note from A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Case 1.a

- A leaves note A before B checks

```
// Thread A
leave note A;
while (note B) // (X)
    do nothing;

if (!milk)
    buy milk;
remove note A;
```

```
// Thread B
leave note B;
if (!note A) { // (Y)
    if (!milk)
        buy milk;
}
remove note B;
```

If A checks note B before B leaves the note, then A goes ahead and buys milk

Case I.b

- A leaves note before B checks

```
// Thread A
leave note A;
while (note B) // (X)
    do nothing;



if (!milk)
    buy milk;
remove note A;
```

```
// Thread B
leave note B;
if (!note A) { // (Y)
    if (!milk)
        buy milk;
}
remove note B;
```

If A checks note B <u>after</u> B leaves the note, then A waits to see what happens

Case I.b (cont.)

- A leaves note before B checks

```
// Thread A
leave note A;
while (note B) // (X)
    do nothing;

if (!milk)
    buy milk;
remove note A;
```

```
// Thread B
leave note B;
if (!note A) { // (Y)
    if (!milk)
        buy milk;
}
remove note B;
```

B will not buy milk!

Case 2

- B checks note A before A leaves it

```
// Thread A
leave note A;
while (note B) // (X)
    do nothing;

if (!milk)
    buy milk;
remove note A;
```

```
// Thread B
leave note B;
if (!note A) { // (Y)
    if (!milk)
        buy milk;
}
remove note B;
```

Solution #3: Discussion

- Our solution protects single critical section for each thread

```
if (!milk) {
   buy milk;
}
```

- Solution #3 works, but it's very unsatisfactory
  - Way too complex – even for this simple example
    - It's hard to convince yourself that this really works
    - Reasoning is even harder when modern compilers/hardware reorder instructions
  - A's code is different from B's – what if there are lots of threads?
    - Code would have to be slightly different for each thread (see Peterson's algorithm)
  - A is busy-waiting – while A is waiting, it is consuming CPU time
- There's a better way
  - Have hardware provide higher-level primitives other than atomic load/store
  - Build even higher-level programming abstractions on this hardware support

Too Much Milk (Solution #4)

- Suppose we have some sort of implementation of a lock
  - `lock.Acquire()` – wait until lock is free, then grab
  - `lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our "too much milk" problem is easy to solve

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

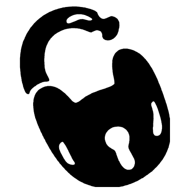
- Code between **`Acquire()`** and **`Release()`** is called critical section
- This could be even simpler: what if we are out of ice cream instead of milk
  - Skip the test since you always need more ice cream ;-)

Where Are We Going with Synchronization?

Programs	Shared Programs			
Higher-level API	Locks	Semaphores	Monitors	Send/Receive
Hardware	Load/Store	Disable Interrupts	Test&Set	Compare&Swap

- We will see how we can implement various higher-level synchronization primitives using atomic operations
  - Everything is quite painful if load/store are the only atomic primitives
  - Hardware needs to provide more primitives useful at user-level

How to Implement Locks?



- Locks are used to prevent someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: synchronization involves waiting
    - Busy-waiting is wasteful (should sleep if waiting for a long time)
- With only atomic load/store we get solutions like "Solution #3"
  - Too complex and error prone
- Is hardware lock instruction good idea?
  - What about putting threads to sleep?
    - How does hardware interact with OS scheduler?
  - What about complexity?
    - Adding each extra feature makes HW more complex and slower

Naïve Implementation of Locks

- Goal: building multi-instruction atomic operations
- Recall: dispatcher gets control in two ways
  - Internal: thread does something to relinquish CPU
  - External: interrupts cause dispatcher to take CPU
- On uniprocessors, we can avoid context-switching by
  - Avoiding internal events (virtual memory is tricky, more on this later)
  - Preventing external events by disabling interrupts
- Consequently, naïve implementation of locks in uniprocessors

```
Acquire { disable interrupts; }
Release { enable interrupts; }
```

Problems with Naïve Implementation of Locks

- OS cannot let users use this!

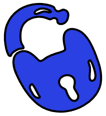
```
Acquire();
while(TRUE) {;}
```

- In real-time systems, there is no guarantees on timing!
  - Critical sections might be arbitrarily long
  - What happens with I/O or other important events?
    - "Reactor about to meltdown. Help?"

Better Implementation of Locks

- Key idea: maintain lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```



```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        go_to_sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

```
Release() {
    disable interrupts;
    if (threads on wait queue) {
        take one off wait queue
        place it on ready queue;
    } else {
        value = FREE;
    }
    enable interrupts;
}
```

New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise, two threads could think that they both have lock

```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        go_to_sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

Critical Section

- Unlike previous solution, critical section (inside `Acquire()`) is very short
  - User of lock can take as long as they like in their own critical section (doesn't impact global machine behavior)
  - Critical interrupts taken in time!

Re-Enabling Interrupts

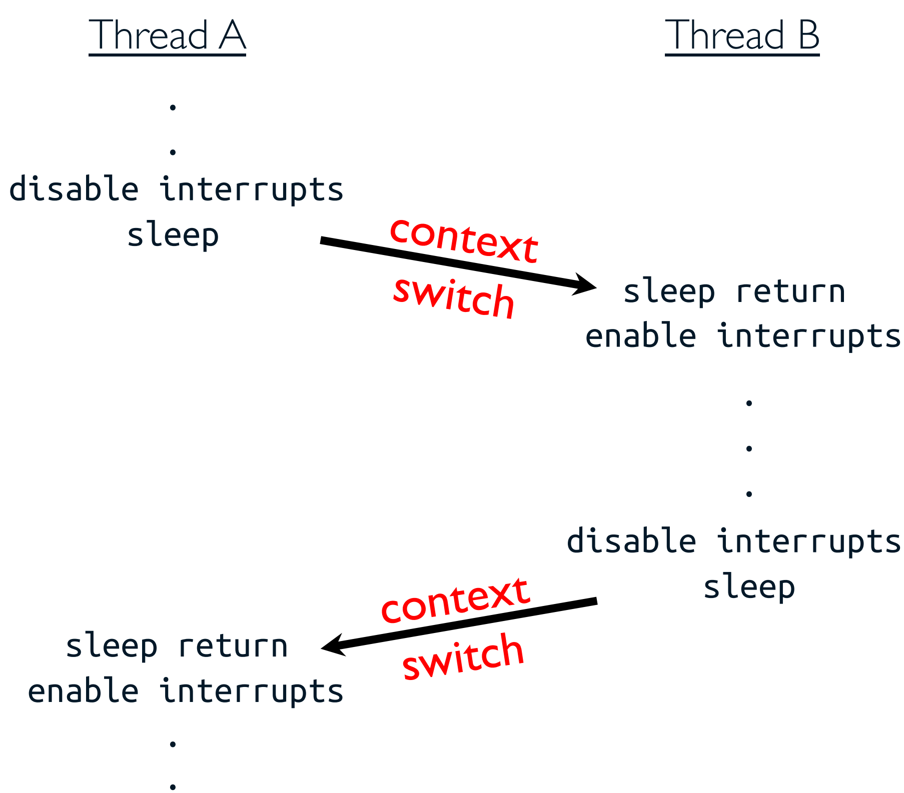
enable here? →

```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        go_to_sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

- Before putting thread on wait queue?
  - Release can check waiting queue and not wake up thread
- After putting thread on wait queue?
  - Release puts thread on ready queue, but thread still thinks it needs to go to sleep!
  - Thread goes to sleep while holding lock (deadlock!)
- After `go_to_sleep()`? But – how?

How to Re-Enable After go_to_sleep()?

- Make it responsibility of next thread to re-enable interrupts
- When sleeping thread wakes up, returns to **`Acquire()`** and re-enables interrupts



Problem with
Implementing Locks Using Interrupts

- Cannot give lock implementation to users
- Doesn't work well on multiprocessor
  - Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative solution: atomic read-modify-write instructions
  - Read value from an address and then write new value to it *atomically*
  - Make HW responsible for implementing this correctly
    - Uniprocessors (not too hard)
    - Multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, this can be used in both uniprocessors and multiprocessors

Examples of Read-Modify-Write Instructions

- ```
  test&set (&address) {            /* most architectures */
        result = M[address];        /* return result from
        M[address] = 1;                “address” and set value at
        return result;                 “address” to 1 */
  }
  ```

- ```
  swap (&address, register) {      /* x86 */
        temp = M[address];          /* swap register’s value to
        M[address] = register;         value at “address” */
        register = temp;
  }
  ```

- ```
  compare&swap (&address, reg1, reg2) {   /* 68000 */
        if (reg1 == M[address]) {
            M[address] = reg2;
            return success;
        } else {
            return failure;
        }
  }
  ```
  ```

Implementing Locks Using test&set

- Simple implementation

```
int value = 0;                              // Free
Acquire() {
    while (test&set(value));                // while busy
}
Release() {
    value = 0;
}
```

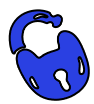
- Free lock: `test&set` reads 0 and sets `value = 1`
- Busy lock: `test&set` reads 1 and sets `value = 1` (no change)
- What is wrong with this implementation?
  - Waiting threads consume cycles while busy-waiting

Locks with Busy-Waiting: Discussion

- Upside?
  - Machine can receive interrupts
  - User code can use this lock
  - Works on multiprocessors
- Downside?
  - This is very wasteful as threads consume cycles waiting
  - Waiting threads may take cycles away from thread holding lock (no one wins!)
  - Priority inversion: if busy-waiting thread has higher priority than thread holding lock $\Rightarrow$ no progress!
- In semaphores and monitors, threads may wait for arbitrary long time!
  - Even if busy-waiting was OK for locks, it's not ok for other primitives
  - Exam solutions should avoid busy-waiting!

Better Implementation of Locks Using test&set

- Can we build **test&set** locks without busy-waiting?
  - We cannot eliminate busy-waiting, but we can minimize it!
  - Idea: only busy-wait to atomically check lock value



```
int guard = 0;
int value = FREE;
Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
        put thread on wait queue;
        go_to_sleep() & guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}
```

```
Release() {
    // Short busy-wait time
    while (test&set(guard));
    if (threads on wait queue) {
        take one off wait queue
        place it on ready queue;
    } else {
        value = FREE;
    }
    guard = 0;
}
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Locks Using Interrupts vs. test&set

```
int value = FREE;


Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        go_to_sleep() & enable interrupts;
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

```
int guard = 0;
int value = FREE;

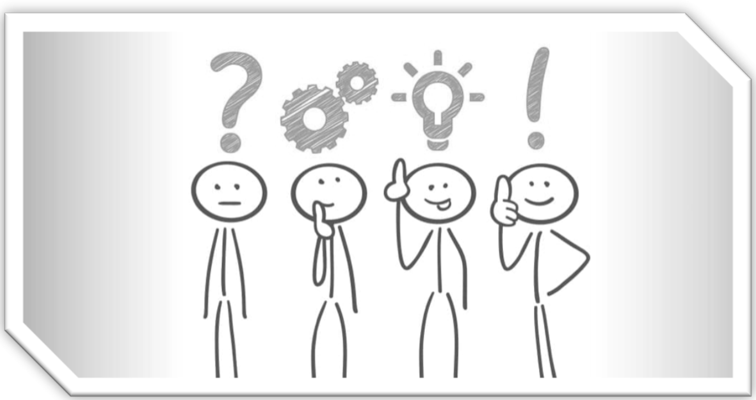
Acquire() {
    while (test&set(guard));
    if (value == BUSY) {
        put thread on wait queue;
        go_to_sleep() & guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}
```

- Replace
  - `disable interrupts;` ⇒ `while (test&set(guard));`
  - `enable interrupts` ⇒ `guard = 0;`

Summary

- Atomic operations
  - Operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Hardware atomicity primitives
  - Disabling of Interrupts, test&set, swap, compare&swap
- Several implementation of Locks
  - Must be very careful not to waste/tie up machine resources
    - Shouldn't disable interrupts for long
    - Shouldn't busy-wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny