CS162
Operating Systems and
Systems Programming
Lecture 7

Synchronization 2: Semaphores (Con’t)
Lock Implementation, Atomic Instructions

February 9th, 2021
Profs. Natacha Crooks and Anthony D. Joseph
http://cs162.eecs.Berkeley.edu
Recall: Multithreaded Stack Example

• Consider the following code blocks:

```plaintext
proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
```

• Suppose we have 2 threads:
  – Threads S and T

Thread S's switch returns to Thread T's (and vice versa)
Recall: Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions

Timer Interrupt routine:

```c
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```
Hardware Context Switch Support in x86

• Syscall/Intr (U → K)
  – PL 3 → 0;
  – TSS ← EFLAGS, CS:EIP;
  – SS:ESP ← k-thread stack (TSS PL 0);
  – push (old) SS:ESP onto (new) k-stack
  – push (old) eflags, cs:eip, <err>
  – CS:EIP ← <k target handler>

• Then
  – Handler then saves other regs, etc
  – Does all its works, possibly choosing other threads, changing PTBR (CR3)
  – kernel thread has set up user GPRs

• iret (K → U)
  – PL 0 → 3;
  – Eflags, CS:EIP ← popped off k-stack
  – SS:ESP ← popped off k-stack

Figure 7-1. Structure of a Task

Pintos: tss.c, intr-stubs.S
Pintos: Kernel Crossing on Syscall or Interrupt
Pintos: Context Switch – Scheduling

User:
- Code
  - user stack
- Stack
  - cs:eip
  - ss:esp

Kernel:
- Code
  - kernel thread stack
- Stack
  - cs:eip
  - ss:esp

Syscall / Interrupt:
- Saves
- TCB
- PTBR

Time:
- Schedule
- Processing
- ready to resume

Switch kernel threads

Pintos: switch.S
Recall: Fix banking problem with Locks!

• Identify critical sections (atomic instruction sequences) and add locking:

```java
Deposit(acctId, amount) {
    Lock.acquire() // Wait if someone else in critical section!
    acct = GetAccount(actId);
    acct->balance += amount;
    StoreAccount(acct);
    Lock.release() // Release someone into critical section
}
```

• Must use SAME lock with all of the methods (Withdraw, etc….)
Recall: Red-Black tree example

- Here, the Lock is associated with the root of the tree
  - Restricts parallelism but makes sure that tree *always* consistent
  - No races at the operation level
- Threads are exchange information through a consistent data structure
- Could you make it faster with one lock per node? Perhaps, but must be careful!
  - Need to define invariants that are always true despite many simultaneous threads...

Thread A
Insert(3) {
    acquire(&treelock)
    Tree.Insert(3)
    release(&treelock)
}

Thread B
Insert(4) {
    acquire(&treelock)
    Tree.insert(4)
    release(&treelock)
}

Get(6) {
    acquire(&treelock)
    Tree.search(6)
    release(&treelock)
}
**Producer-Consumer with a Bounded Buffer**

- **Problem Definition**
  - Producer(s) put things into a shared buffer
  - Consumer(s) take them out
  - Need synchronization to coordinate producer/consumer

- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- **Example 1:** GCC compiler
  - `cpp | cc1 | cc2 | as | ld`

- **Example 2:** Coke machine
  - Producer can put limited number of Cokes in machine
  - Consumer can't take Cokes out if machine is empty

- Others: Web servers, Routers, ….
Circular Buffer Data Structure (sequential case)

```c
typedef struct buf {
    int write_index;
    int read_index;
    <type> *entries[BUFSIZE];
} buf_t;
```

- Insert: write & bump write ptr (enqueue)
- Remove: read & bump read ptr (dequeue)
- How to tell if Full (on insert) Empty (on remove)?
- And what do you do if it is?
- What needs to be atomic?
Circular Buffer – first cut

mutex buf_lock = <initially unlocked>

Producer(item)
    acquire(&buf_lock);
    while (buffer full) {} // Wait for a free slot
    enqueue(item);
    release(&buf_lock);
}

Consumer()
    acquire(&buf_lock);
    while (buffer empty) {} // Wait for arrival
    item = dequeue();
    release(&buf_lock);
    return item

Will we ever come out of the wait loop?
Circular Buffer – 2\textsuperscript{nd} cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
    acquire(&buf_lock);
    while (buffer full) {release(&buf_lock); acquire(&buf_lock);} 
    enqueue(item);
    release(&buf_lock);
}

Consumer() {
    acquire(&buf_lock);
    while (buffer empty) {release(&buf_lock); acquire(&buf_lock);} 
    item = dequeue();
    release(&buf_lock);
    return item
}
```

What happens when one is waiting for the other?
- Multiple cores?
- Single core?

Circular Buffer – 2\textsuperscript{nd} cut
Higher-level Primitives than Locks

• What is right abstraction for synchronizing threads that share memory?
  – Want as high a level primitive as possible

• Good primitives and practices important!
  – Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  – UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs

• Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  – This lecture and the next presents some ways of structuring sharing
Recall: Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - \textbf{Down()} or \textbf{P()}: an atomic operation that waits for semaphore to become positive, then decrements it by 1
    » Think of this as the wait() operation
  - \textbf{Up()} or \textbf{V()}: an atomic operation that increments the semaphore by 1, waking up a waiting \textbf{P}, if any
    » This of this as the signal() operation
- Note that \textbf{P()} stands for “proberen” (to test) and \textbf{V()} stands for “verhogen” (to increment) in Dutch
Semaphores Like Integers Except…

• Semaphores are like integers, except:
  – No negative values
  – Only operations allowed are P and V – can’t read or write value, except initially
  – Operations must be atomic
    » Two P’s together can’t decrement value below zero
    » Thread going to sleep in P won’t miss wakeup from V – even if both happen at same time
• POSIX adds ability to read value, but technically not part of proper interface!
• Semaphore from railway analogy
  – Here is a semaphore initialized to 2 for resource control:
Two Uses of Semaphores

Mutual Exclusion (initial value = 1)
- Also called “Binary Semaphore” or “mutex”.
- Can be used for mutual exclusion, just like a lock:
  ```
  semaP=&mysem);
  // Critical section goes here
  semaV(&mysem);
  ```

Scheduling Constraints (initial value = 0)
- Allow thread 1 to wait for a signal from thread 2
  - thread 2 schedules thread 1 when a given event occurs
- Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:
  ```
  Initial value of semaphore = 0
  ThreadJoin {
    semaP(&mysem);
  }
  ThreadFinish {
    semaV(&mysem);
  }
  ```
Revisit Bounded Buffer: Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)

- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine

- General rule of thumb: Use a separate semaphore for each constraint
  - Semaphore fullBuffers; // consumer’s constraint
  - Semaphore emptyBuffers; // producer’s constraint
  - Semaphore mutex; // mutual exclusion
Full Solution to Bounded Buffer (coke machine)

Semaphore fullSlots = 0; // Initially, no coke
Semaphore emptySlots = bufSize; // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    semaP(&emptySlots); // Wait until space
    semaP(&mutex); // Wait until machine free
    Enqueue(item);
    semaV(&mutex);
    semaV(&fullSlots); // Tell consumers there is more coke
}

Consumer() {
    semaP(&fullSlots); // Check if there’s a coke
    semaP(&mutex); // Wait until machine free
    item = Dequeue();
    semaV(&mutex);
    semaV(&emptySlots); // tell producer need more
    return item;
}
Discussion about Solution

• Why asymmetry?
  – Producer does: `semaP(&emptyBuffer), semaV(&fullBuffer)`
  – Consumer does: `semaP(&fullBuffer), semaV(&emptyBuffer)`

• Is order of P’s important?

• Is order of V’s important?

• What if we have 2 producers or 2 consumers?
Where are we going with synchronization?

<table>
<thead>
<tr>
<th>Programs</th>
<th>Shared Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-level API</td>
<td>Locks  Semaphores  Monitors  Send/Receive</td>
</tr>
<tr>
<td>Hardware</td>
<td>Load/Store  Disable Ints  Test&amp;Set  Compare&amp;Swap</td>
</tr>
</tbody>
</table>

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level
Motivating Example: “Too Much Milk”

- Great thing about OS’s – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Arrive home, put milk away</td>
</tr>
</tbody>
</table>
Recall: What is a lock?

- **Lock**: prevents 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: all synchronization involves waiting
- For example: fix the milk problem by putting a key on the 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
  - Let’s see if we can answer this question!
Too Much Milk: Correctness Properties

• Need to be careful about correctness of concurrent programs, since 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 the correctness properties for the “Too much milk” problem???
  – Never more than one person buys
  – Someone buys if needed

• First attempt: Restrict ourselves to use only atomic load and store operations as
  building blocks
Too Much Milk: Solution #1

• Use a note to avoid buying too much milk:
  – Leave a note before buying (kind of “lock”)
  – Remove note after buying (kind of “unlock”)
  – Don’t buy if note (wait)
• Suppose a computer tries this (remember, only memory read/write are atomic):
  
  ```java
  if (noMilk) {
    if (noNote) {
      leave Note;
      buy milk;
      remove note;
    }
  }
  ```
Too Much Milk: Solution #1

• Use a note to avoid buying too much milk:
  – Leave a note before buying (kind of “lock”)
  – Remove note after buying (kind of “unlock”)
  – Don’t buy if note (wait)

• Suppose a computer tries this (remember, only memory read/write are atomic):

```java
Thread A
if (noMilk) {
  if (noNote) {
    leave Note;
    buy Milk;
    remove Note;
  }
}

Thread B
if (noMilk) {
  if (noNote) {
    leave Note;
    buy Milk;
    remove Note;
  }
  leave Note;
  buy Milk;
  remove Note;
}
```
Too Much Milk: Solution #1

• Use a note to avoid buying too much milk:
  – Leave a note before buying (kind of “lock”)  
  – Remove note after buying (kind of “unlock”)  
  – Don’t buy if note (wait)

• Suppose a computer tries this (remember, only memory read/write are atomic):
  
  ```java
  if (noMilk) {
    if (noNote) {
      leave Note;
      buy milk;
      remove note;
    }
  }
  ```

• Result?
  – Still too much milk but only occasionally!
  – Thread can get context switched after checking milk and note but before buying milk!

• Solution makes problem worse since fails intermittently
  – Makes it really hard to debug…
  – Must work despite what the dispatcher does!
Too Much Milk: Solution #1½

• Clearly the Note is not quite blocking enough
  – Let’s try to fix this by placing note first
• Another try at previous solution:

```java
leave Note;
if (noMilk) {
  if (noNote) {
    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?
  - Now we can leave note before checking
- Algorithm looks like this:

  ```
  Thread A                          Thread B
  leave note A;
  if (noNote B) {
    if (noMilk) {
      buy Milk;
    }
  } else {
    remove note A;
  }

  leave note B;
  if (noNoteA) {
    if (noMilk) {
      buy Milk;
    }
  } else {
    remove note B;
  }
  ```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely this would happen, but will at worse possible time
  - Probably something like this in UNIX
Too Much Milk Solution #2: problem!

- I'm not getting milk, You're getting milk
- This kind of lockup is called “starvation!”
Too Much Milk Solution #3

• Here is a possible two-note solution:

  Thread A                      Thread B
  leave note A;                leave note B;
  while (note B) {  
    do nothing;            if (noNote A) {  
               \ X \ Y
    }                if (noMilk) {
    }                   buy milk;
        if (noMilk) { 
      buy milk;             }
    }                   remove note B;
  remove note A;

• Does this work? Yes. Both can guarantee that:
  – It is safe to buy, or
  – Other will buy, ok to quit

• At X:
  – If no note B, safe for A to buy,
  – Otherwise wait to find out what will happen

• At Y:
  – If no note A, safe for B to buy
  – Otherwise, A is either buying or waiting for B to quit
Case 1

• “leave note A” happens before “if (noNote A)"

```
leave note A;
while (note B) { \X
do nothing;
};

if (noMilk) {
  buy milk;
}
remove note B;

if (noMilk) {
  buy milk;
}
remove note A;
```
Case I

• “leave note A” happens before “if (noNote A)”

```c
leave note A;
while (note B) {
  do nothing;
};

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

```c
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;

if (noMilk) {
  buy milk;
}
remove note A;
```
Case 1

- “leave note A” happens before “if (noNote A)”
Case 2

• “if (noNote A)” happens before “leave note A”
Case 2

• “if (noNote A)” happens before “leave note A”

```java
leave note B;
if (noNote A) {
  \ Y

  if (noMilk) {
    buy milk;
  }
}
remove note B;

leave note A;
while (note B) {
  \ X
do nothing;
}

if (noMilk) {
  buy milk;
}
remove note A;
```
Case 2

• “if (noNote A)” happens before “leave note A”
This Generalizes to $n$ Threads...

- Leslie Lamport’s “Bakery Algorithm” (1974)

A New Solution of Dijkstra’s Concurrent Programming Problem

Leslie Lamport
Massachusetts Computer Associates, Inc.

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate.
Solution #3 discussion

• Our solution protects a single “Critical-Section” piece of code for each thread:
  
  ```java
  if (noMilk) {
    buy milk;
  }
  ```

• Solution #3 works, but it is really unsatisfactory
  – Really complex – even for this simple an example
    » Hard to convince yourself that this really works
  – A’s code is different from B’s – what if lots of threads?
    » Code would have to be slightly different for each thread
  – While A is waiting, it is consuming CPU time
    » This is called “busy-waiting”

• There must be a better way!
  – Have hardware provide higher-level primitives than atomic load & store
  – Build even higher-level programming abstractions on this hardware support
Too Much Milk: Solution #4?

• Recall our target lock interface:
  – `acquire(&milklock)` – wait until lock is free, then grab
  – `release(&milklock)` – Unlock, waking up anyone waiting
  – These must be atomic operations – if two threads are waiting for the lock and both see it is free, only one succeeds to grab the lock

• Then, our milk problem is easy:

```c
acquire(&milklock);
if (nomilk)
  buy milk;
release(&milklock);
```
Back to: How to Implement Locks?

- **Lock**: prevents 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: all synchronization involves waiting
    » Should sleep if waiting for a long time

- **Atomic Load/Store**: yields a solution like Milk #3
  - Pretty complex and error prone

- **Hardware Lock instruction**
  - Is this a good idea?
  - What about putting a task to sleep?
    » What is the interface between the hardware and scheduler?
  - Complexity?
    » Done in the Intel 432
    » Each feature makes HW more complex and slow
Naïve use of Interrupt Enable/Disable

• How can we build multi-instruction atomic operations?
  – Recall: dispatcher gets control in two ways.
    » Internal: Thread does something to relinquish the CPU
    » External: Interrupts cause dispatcher to take CPU
  – On a uniprocessor, can avoid context-switching by:
    » Avoiding internal events (although virtual memory tricky)
    » Preventing external events by disabling interrupts

• Consequently, naïve Implementation of locks:
  
  LockAcquire { disable Ints; }
  LockRelease { enable Ints; }

• Problems with this approach:
  – Can’t let user do this! Consider following:
    
    LockAcquire();
    While(TRUE) {;;}

  – Real-Time system—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 by Disabling Interrupts

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

```c
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 (anyone on wait queue) {
        take thread off wait queue
        Place 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

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

- Note: unlike previous solution, the 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!
Interrupt Re-enable in Going to Sleep

• What about re-enabling ints when going to sleep?
  
  ```
  Acquire() {
    disable interrupts;
    if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
    } else {
      value = BUSY;
    }
    enable interrupts;
  }
  ```
Interrupt Re-enable in Going to Sleep

• What about re-enabling ints when going to sleep?

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

Enable Position

• Before Putting thread on the wait queue?
Interrupt Re-enable in Going to Sleep

• What about re-enabling ints when going to sleep?

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

Enable Position

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread
Interrupt Re-enable in Going to Sleep

• What about re-enabling ints when going to sleep?

```c
int Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread

• After putting the thread on the wait queue
Interrupt Re-enable in Going to Sleep

• What about re-enabling ints when going to sleep?
  Acquire() {
    disable interrupts;
    if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
    } else {
      value = BUSY;
    }
    enable interrupts;
  }

Enable Position

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread
• After putting the thread on the wait queue
  – Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  – Misses wakeup and still holds lock (deadlock!)
Interrupt Re-enable in Going to Sleep

• What about re-enabling ints when going to sleep?
  
  ```
  Acquire() {
    disable interrupts;
    if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
    } else {
      value = BUSY;
    }
    enable interrupts;
  }
  ```

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread
• After putting the thread on the wait queue
  – Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  – Misses wakeup and still holds lock (deadlock!)
• Want to put it after `sleep()`. But – how?
How to Re-enable After Sleep()?

- In scheduler, since interrupts are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

```
Thread A                          Thread B
  .
  .
disable ints
  sleep
  context switch

  sleep return
  enable ints
  .
  .
disable int
  sleep
  context switch

  sleep return
  enable ints
  .
  .
```
INIT

```c
int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

lock.Acquire();

... critical section;
...
lock.Release();

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}

lock.Acquire();

... critical section;
...
lock.Release();
```
INIT
int value = 0;

Acquire() {
    disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
        go to sleep() //??
    } else {
        value = 1;
    }
    enable interrupts;
}

lock.Acquire();
 ...
critical section;
 ...
lock.Release();

lock.Acquire();
 ...
critical section;
 ...
lock.Release();

Release() {
    disable interrupts;
    if anyone on wait queue { take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}
In-Kernel Lock: Simulation

Thread A

int value = 0;

lock.Acquire();

... critical section;

lock.Release();

Acquire() {
    disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
        go to sleep() //??
    } else {
        value = 1;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}

Thread B

Value: 1
waiters
owner
READY

In-Kernel Lock: Simulation

Critical section: ...

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Thread A

Thread B

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY

Value: 1
waiters
owner
READY
In-Kernel Lock: Simulation

int value = 0;

Acquire() {
    disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
        go to sleep() ??
    } else {
        value = 1;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}
In-Kernel Lock: Simulation

INIT

Thread A

Thread B

lock.Acquire();

... critical section;

lock.Release();

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep(); ??
  } else {
    value = 1;
  }
  enable interrupts;
}

lock.Acquire();

... critical section;

lock.Release();

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
INIT

int value = 0;

Acquire() {
    disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
        go to sleep() //??
    } else {
        value = 1;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}

lock.Acquire();
...
critical section;
...
lock.Release();

lock.Acquire();
...
critical section;
...
lock.Release();

Thread A

Value: 1

waiters

owner

READY

Thread B

In-Kernel Lock: Simulation
Recall: Multithreaded Server

- **Bounded** pool of worker threads
  - Allocated in **advance**: no thread creation overhead
  - **Queue** of pending requests
• Given that the overhead of a critical section is $X$
  – User->Kernel Context Switch
  – Acquire Lock
  – Kernel->User Context Switch
  – <perform exclusive work>
  – User->Kernel Context Switch
  – Release Lock
  – Kernel->User Context Switch

• Even if everything else is infinitely fast, with any number of threads and cores

• What is the maximum rate of operations that involve this overhead?
Highly Contended Case – in a picture

\[ \text{Time} = p \times X \text{ sec} \]
\[ \text{Rate} = \frac{1}{X} \text{ ops/sec, regardless of # cores} \]

All try to grab lock
Back to system performance

More Practical Motivation

Back to Jeff Dean's "Numbers everyone should know"

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 ns</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000 ns</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000 ns</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000 ns</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000 ns</td>
</tr>
<tr>
<td>Send packet CA--&gt;Netherlands--&gt;CA</td>
<td>150,000,000 ns</td>
</tr>
</tbody>
</table>

- $X = 1\text{ms} \Rightarrow 1,000 \text{ ops/sec}$

Handle I/O in separate thread, avoid blocking other progress
What if sys overhead is \( Y \), even when the lock is free?

What if the OS can only handle one lock operation at a time?
Recall: Basic cost of a system call

- Min System call ~ 25x cost of function call
- Scheduling could be many times more
- Streamline system processing as much as possible
- Other optimizations seek to process as much of the call in user space as possible (eg, Linux vDSO)
Atomic Read-Modify-Write Instructions

- Problems with previous solution:
  - Can’t 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: atomic instruction sequences
  - These instructions read a value and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - on both uniprocessors (not too hard)
    - and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors
Examples of Read-Modify-Write

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

- **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]) { // If memory still == reg1,
    M[address] = reg2; // then put reg2 => memory
    return success;
  } else { // Otherwise do not change memory
    return failure;
  }

- **load-linked&store-conditional(&address) {** /* R4000, alpha, ARM, RISC-V */
  
  loop:
    ll r1, M[address];
    movi r2, 1; // Can do arbitrary computation
    sc r2, M[address];
    beqz r2, loop;
  }
Conclusion

• Important concept: Atomic Operations
  – An operation that runs to completion or not at all
  – These are the primitives on which to construct various synchronization primitives

• Talked about hardware atomicity primitives:
  – Disabling of Interrupts, test&set, swap, compare&swap, load-linked & store-conditional

• Showed several constructions of Locks
  – Must be very careful not to waste/tie up machine resources
    » Shouldn’t disable interrupts for long
    » Shouldn’t spin wait for long
  – Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable