Locks and Semaphores

CS439: Principles of Computer Systems
February 8, 2016
Last Time

• Introduced Threads
  – Why we want them, what they are, how they differ from processes
  – Kernel vs. User
  – Independent vs. Cooperating

• Too Much Milk
  – Race conditions: different result based on scheduling
  – Critical Sections: a piece of code only one thread can execute at a time
  – Atomic Operations: uninterruptible operations
  – Mutual exclusion
  – Safety
  – Liveness
  – Bounded waiting
Threads and the Scheduler
(or, Why Multi-threaded Programming is Hard)

Given two threads, A and B, how might their executions be scheduled?

A → B
B → A
A → B → A
B → A → B
A → B → A → B
B → A → B → A
Concurrency is great...

```c
int a=1, b=2;
main() {
    createThread(fn1, 4);
    createThread(fn2, 5);
    thread_join(all);
}

fn1(int arg1){
    if(a) b++;
}

fn2(int arg1){
    a=arg1;
}
```

What are the values of a and b after execution?

A. a=1, b=2  
B. a=1, b=3  
C. a=5, b=2  
D. a=5, b=3
... but can be problematic

```c
int a=1, b=2;
main() {
    createThread(fn1, 4);
    createThread(fn2, 5);
    thread_join(all);
}

fn1(int arg1){
    if(a) b++;
}

fn2(int arg1){
    a=0;
}
```

What are the values of a and b after execution?

A. a=0, b=2
B. a=0, b=3
C. a=1, b=2
D. a=1, b=3
Today’s Agenda

• Synchronization Terminology
  – Atomic Operations: uninterruptible operations
  – Mutual exclusion
  – Safety
  – Liveness
  – Bounded waiting
• Hardware Support for Synchronization
  – disabling interrupts (what is an interrupt?)
  – read-write-modify instructions
• Synchronization in Software
  – Abstractions built on top of hardware support
  – Semaphores
Terminology

• **Atomic Operation**: an operation that is uninterruptible

• **Synchronization**: Using atomic operations to ensure cooperation between threads

• **Mutual Exclusion**: Exactly one thread (or process) is doing a particular activity at a time. Usually related to critical sections

• **Critical Section**: A piece of code that only one thread can execute at a time
Critical Sections and Correctness

Four properties are required for correctness:
1. *Safety*: only one thread in the critical section
2. *Liveness*: if no threads are executing a critical section, and a thread wishes to enter a critical section, that thread must be guaranteed to eventually enter the critical section
3. *Bounded waiting*: if a thread wishes to enter a critical section, then there exists a bound on the number of other threads that may enter the critical section before that thread does
4. *Failure atomicity*: it’s okay for a thread to die in the critical section
Safety and Liveness for Critical Sections

• Only one thread is concurrently in the critical section
  A. Safety
  B. Liveness
  C. Both

• A thread that wants to enter the critical section will eventually succeed
  A. Safety
  B. Liveness
  C. Both

• Bounded waiting: If a thread $i$ is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section (only 1 thread is allowed in at a time) before thread $i$’s request is granted.
  
  A. Safety  B. Liveness  C. Both
Safety and Liveness, More Generally

Properties defined over the execution of a program

• Safety: “nothing bad happens”
  – Holds in every finite execution prefix
    • Windows never crashes
    • No patient is ever given the wrong medication
    • A program never terminates with the wrong answer

• Liveness: “something good eventually happens”
  – No partial execution is irremediable
    • Windows always reboots
    • Medications are eventually distributed to patients
    • A program eventually terminates
Mutual Exclusion

• Exactly one thread (or process) is doing a particular activity at a time. Usually related to critical sections.
  – Active thread excludes its peers

• Some computer resources cannot be accessed by multiple threads at the same time
  – E.g., a printer can’t print two documents at once

• For shared memory architectures, data structures are often mutually exclusive
  – Two threads adding to a linked list can corrupt the list
When to Use Mutual Exclusion/Critical Sections

Anytime you access shared data
  – If a thread checks a value
    • Even if it is “just a quick” read
  – If a thread updates a piece of shared data
    • What data is shared?

Learn it! Live it! Breathe it!
Formalizing “Too Much Milk”

• Shared variables
  – “Look in the fridge for milk” – check a variable
  – “Put milk away” – update a variable

• Safety property
  – At most one person buys milk

• Liveness
  – Someone buys milk when needed
Formalizing “Too Much Milk”

You (Thread A)
leave note A
while(note B)
do nothing;

if(noMilk)
buy milk;
remove note A

Your Roommate (Thread B)
leave note B
if(noNote A)

if(noMilk)
buy milk;

remove note B

Entry Section

Critical Section

Exit Section
Atomic Operations

• Operations that are uninterruptible---run to completion or not at all
  – What about \( x = x + 1 \)?
    • load x
    • add 1
    • store x
  – if(\( x == 1 \)) \( x=2 \)?
    • load x
    • compare
    • store x (maybe)
• What operations are uninterruptible?
Revisiting Too Much Milk: Solution #3 (Works!)

You (Thread A)
leave note A
while(note B)
do nothing;
if(noMilk)
buy milk;
remove note A

Your Roommate (Thread B)
leave note B
if(noNote A)
if(noMilk)
buy milk;
remove note B
Our Ideal Solution

• Satisfies correctness properties
  – Safety, liveness, bounded wait
  – Easy to convince ourselves it does so
• No busy waiting (spin locks)
  – Threads should block when waiting and then be awakened when it is their turn (a wait queue)
• Extendable to many threads (not just two!)
  – Symmetric
• Anything else?
Too Much Milk: Taking Turns

You (Thread A)
while (turn != A)
    do nothing;
if (noMilk)
    buy milk;
turn = B;

Your Roommate (Thread B)
while (turn != B)
    do nothing;
if (noMilk)
    buy milk;
turn = A;

Does this work?
Language Support for Synchronization

Some programming languages provide support for *atomic routines* for synchronization

- **Locks**: One thread holds a lock at a time, executes the critical section, releases the lock
- **Semaphores**: More general version of locks
- **Monitors**: Connects shared data to synchronization primitive

=> *All require some hardware support (and waiting!)*.
Locks, Generally

A *lock* prevents another process from doing something

– Lock before entering a critical section or before accessing shared data
– Unlock when leaving a critical section or when access to shared data is complete
– Wait if locked
Locks, More Formally

• *Locks* provide mutual exclusion to shared data with two atomic routines:
  – *Lock::*Acquire*: wait until lock is free, then grab it
  – *Lock::*Release*: unlock and wake up any thread waiting in Acquire

• Rules for using a lock:
  – Always acquire the lock before accessing shared data
  – Always release the lock after finishing with shared data
  – Lock is initially free
Locks and Too Much Milk

Our solution used notes as locks:

1. Leave a note (acquire a lock)
2. Remove a note (release the lock)
3. Do not buy any milk if there is a note (wait)

What would it look like with actual locks?
Too Much Milk: Lock Solution

Lock myLock;

**You (Thread A)**
myLock->Acquire();
if(noMilk)
    buy milk;
myLock->Release();

**Your Roommate (Thread B)**
myLock->Acquire();
if(noMilk)
    buy milk;
myLock->Release();
So... Implementing Locks
Key Observations

• Why do we need mutual exclusion?
  – The scheduler!

• On a uniprocessor, a operation is atomic if no context switch can occur in the middle of the operation
  – Mutual exclusion by preventing the context switch

• Context switches occur because of:
  – Internal events: systems calls and exceptions
  – External events: interrupts
Thwarting the Scheduler (or Keeping Control)

So... how can a thread keep control?

– Internal events: Easy! Don’t yield, don’t request I/O, don’t cause any exceptions
– External events: ????
Disabling Interrupts

- Tells the hardware to delay handling any external events until after the thread is finished modifying the critical section
- In some implementations, done by setting and unsetting the interrupt status bit
Disabling Interrupts: Simplest Solution

Lock::Acquire(int thread) {
    disable interrupts;
}

Lock::Release(int thread) {
    enable interrupts;
}

Does this work?

Is this a good idea?

No!

• Once interrupts are disabled, thread can’t be stopped
• Critical section can be very long---can’t wait too long to respond to interrupts
Disabling Interrupts: Simple Solution

```cpp
Lock::Acquire()
{
    disable interrupts;
    while(value == BUSY){
        enable interrupts;
        disable interrupts;
    }
    value = BUSY;
    enable interrupts;
}

Lock::Release(int thread){
    disable interrupts;
    value = FREE;
    enable interrupts;
}
```

So... Let’s shorten the length of the critical section. Instead of disabling interrupts for the entire critical section, let’s only use them to protect the lock’s data structure.
Disabling Interrupts: No Busy Wait

Lock::Acquire(int thread) {
    disable interrupts;
    if (value == BUSY) {
        add thread to wait queue
        thread->block()
    } else
    value = BUSY;
    enable interrupts;
}

Lock::Release(int thread) {
    disable interrupts;
    if (queue is not empty) {
        take thread1 off wait queue
        put thread1 on ready queue
    } else
    value = FREE;
    enable interrupts;
}
Re-enabling Interrupts

Lock::Acquire(int thread){
    disable interrupts;
    if(value==BUSY) {
        enable interrupts;
        add thread to wait queue;
        thread->block();
    }
    else
        value = BUSY;
    enable interrupts;
}

Lock::Release(int thread){
    disable interrupts;
    if queue is not empty{
        take thread1 off wait queue
        put thread1 on ready queue
    }
    else
        value = FREE;
    enable interrupts;
}
Lock::Acquire(int thread){
    disable interrupts;
    if(value==BUSY) {
        add thread to wait queue
        enable interrupts;
        thread->block();
    }
    else
    value = BUSY;
    enable interrupts;
}

Lock::Release(int thread){
    disable interrupts;
    if queue is not empty{
        take thread1 off wait queue
        put thread1 on ready queue
    }
    else
    value = FREE;
    enable interrupts;
}
Re-enabling Interrupts

Where else?

– The running thread itself: the first thing a thread does when it starts to execute is enable interrupts

– In the CPU scheduler: When the scheduler selects and starts the next running process, it can enable interrupts
  • Remember, the scheduler can get control when a thread gives it up voluntarily
Larger Question: Is this a good idea?

- Should user processes be able to disable interrupts?
  - No.

- What happens on multiprocessors?
  - Disabling interrupts affects only the CPU on which the thread is executing
    - Threads on other CPUs can enter the critical section!

- On a uniprocessor, the OS does use this technique when it is updating some data structures
  - Important for Pintos!
What are we trying to do?

• Ensure mutual exclusion, liveness, etc.
• But, practically?
  – See if another thread is executing the section 
    \textit{(read a variable)}
  – If it isn’t, grab the lock \textit{(modify and write a variable)}
  – If it is, wait
  – Atomically
• So we want a read-modify-write instruction
Atomic Read-Modify-Write Instructions

- Atomic read-modify-write instructions *atomically* read a value from memory into a register and write a new value.
  - read a memory location into a register AND
  - write a new value to the location
- Uniprocessor just needs a new instruction
- On multiprocessors, the processor issuing the instruction:
  - must invalidate the value other processes may have in their caches
  - must lock the memory bus to prevent other processors from accessing memory until it is finished
Example RMW Instructions

- **Test&Set:** most architectures
  - Reads a value from memory
  - Writes “1” back to the memory location

- **Compare&Swap (CAS):** 68000
  - Test the value against some constant
  - If the test is true, set value in memory to a different value
  - Report the result of the test in a flag

- **Load Linked/Store Conditional (LL/SC):** Alpha, PowerPC, ARM
  - LL returns value of memory location
  - A subsequent SC to that memory location succeeds only if that location has not been updated since LL

- **Exchange:** x86
  - Swaps value between register and memory
Implementing Locks with Test&Set

Lock::Acquire()
{
    while (test&set(value)==1)
    {
    }
}

Lock::Release()
{
    value = 0;
}

• If lock is free (value==0), test&set reads 0, sets value to 1, and returns 0. The Lock is not busy, test in the while fails, and Acquire is complete

• If lock is busy (value==1), test&set reads 1, sets value to 1, and returns 1. The while continues to loop until an Release executes
Problems!

- Occupies CPU by performing busy waiting, or *spinning*
  - Could be okay as long as critical section is much shorter than the scheduling quantum
- What happens if threads have different priorities?
  - If the thread waiting for the lock has higher priority than the thread using the lock?
  - This is called the *priority inversion* problem
  - Possible whenever there is a busy wait
- BUT there is low latency to acquire the lock
  - If it becomes free, waiting thread gets it as soon as it is scheduled again
Voluntary yield of the CPU: Test&Set with Cheaper Busy Waiting

Lock::Acquire()
{
    while(1) {
        if(test&set(value) == 0)
            break;
        else
            sleep(1);
    }
}

Lock::Release()
{
    value = 0;
}

What is the tradeoff?

A. CPU usage
B. Memory usage
C. Lock::Acquire() latency
D. Memory bus usage
E. Messes up interrupt handling
Test&Set and Busy Waiting

• Can we implement locks with test&set without
  – busy waiting OR
  – disabling interrupts?
• No.
• BUT we can busy wait on the lock rather than the critical section...
  – Add a variable that tracks whether the lock is in use (for us, guard)
Test&Set with Minimal Busy Waiting

```cpp
int value; /*critical section indicator*/
int guard; /*lock indicator*/

Lock::Acquire(int thread){
    while(test&set(guard)==1) ;
    if(value != FREE){
        put thread on wait queue;
        thread->block()&set guard=0;
    } else {
        value=BUSY;
        guard = 0;
    }
}

Lock::Release(int thread){
    while(test&set(guard)==1) ;
    if wait queue is not empty{
        take thread off wait queue;
        put thread on ready queue;
    } else {
        value=FREE;
    }
    guard = 0;
}
```
Beyond Mutual Exclusion

• Locks provide mutual exclusion
  – Protect critical sections
  – Implementing them may require a critical section
    • Use atomic RMW operations to break the cycle
• But... we need more
  – What if we need to wait for another thread to take action?
    • Coke machine! (Bounded queue, producer/consumer)
| #43083A71 |
| #8FB58BB1 |
| #9AEBD8A9 |
| #A08E7F51 |
Semaphores
Semaphores

• Semaphores are basically generalized locks
  – Support two atomic operations (Up & Down!)
  – Offer elegant solutions to synchronization problems

• Used for all types of synchronization
  – including mutual exclusion

• Each semaphore has a value associated with it

• Each semaphore supports a queue of threads that are waiting to access a critical section (e.g., to buy milk)

• Invented by Dijkstra in 1965
Two Types of Semaphores

• Binary semaphore
  – Same as a lock
  – Guarantees mutually exclusive access to a resource
  – Has two values: 0 or 1 (busy or free)
  – Initial value is always free (1)

• Counted semaphore
  – Represents a resource with many units available
  – Initial count is typically the number of resources
    • always a non-negative integer
  – Allows a thread to continue as long as more instances are available
  – Used for synchronization

• Only difference is the initial value...
Semaphores as Locks
(Binary Semaphores)
Using Binary Semaphores

S->Down() //wait until semaphore S
    //is available (value==1), then
<critical section> //set as busy (value==0)

S->Up () //signal to other processes
    //that semaphore S is free
    //set value = 1

• If a process executes S->Down() and semaphore S is free, it continues executing. Otherwise, the OS puts the process on the wait queue for semaphore S.
• S->Up() unblocks one process on semaphore S’s wait queue
Semaphores: Atomic Operations

- **Down()**
  - Actually P() (*Proberen*, or “pass” in Dutch)
  - Decrements the value
  - When down() returns, the thread has the resource
  - Can block: if resource not available (as indicated by count), the thread will be placed on a wait queue and put to sleep

- **Up()**
  - Actually V() (*Verhogen*, or “release” in Dutch)
  - Increments the value
  - Never blocks
  - If a thread is asleep on the wait queue, it will be awakened
Semaphore Atomic Operations

Semaphore::Down()
    *if value <= 0 , block
    *on up(), wake up
    *decrement semaphore value

When it returns, it has the lock/resource.

Semaphore::Up{
    *increment semaphore value
    *if any threads sleeping on semaphore, wake one of them up
    *return
}

When it returns, it has released the lock/resource.
Implementing Down() and Up()

int value = val; //initial value depends on the problem and
//indicates number of resources available

Semaphore::Down()
{
    if(value == 0)
    {
        add t to wait queue;
        t->block()
    }
    value = value – 1;
}

Semaphore::Up()
{
    value = value + 1;
    if(t on wait queue)
    {
        remove t from wait queue;
        wakeup(t);
    }
}
Too Much Milk: Semaphore Solution

Semaphore milkSema = 1;

You (Thread A)
milkSema->Down();
if(noMilk)
    buy milk;
milkSema->Up();

Your Roommate (Thread B)
milkSema->Down();
if(noMilk)
    buy milk;
milkSema->Up();
iClicker Question

If you have a binary semaphore, how many potential values does it have?

A. 0
B. 1
C. 2
D. 3
E. 4
Getting New Functionality
(Counted Semaphores)
Counted Semaphores

• Represent a resource with many units available
• Initial count is the number of resources
• Lets processes continue as long as more instances are available
Using Counted Semaphores

S->Down() //wait until semaphore S is available (value==1), then
<access the resource> //set as busy (value==0)

S->Up() //signal to other processes
//that semaphore S is free
//set value = 1

• If a process executes S->Down() and semaphore S is free, it continues executing. Otherwise, the OS puts the process on the wait queue for semaphore S.
• S->Up() unblocks one process on semaphore S’s wait queue
When to Use Semaphores

• Mutual Exclusion
  – Use to protect the critical section (see Too Much Milk Example)

• Control Access to a Pool of Resources
  – Counted semaphore

• General Synchronization
  – Use to enforce general scheduling constraints where the threads must wait for some circumstance
  – Value is typically 0 to start
Semaphore Example: Producers/Consumers

Semaphore mutex = 1   // access to buffer
Semaphore empty = N   // count of empty slots
Semaphore full = 0    // count of full slots
int buffer[N]

BoundedBuffer::Producer(){
  <produce item>
  empty->Down() // get empty spot
  mutex->Down() // get access to buffer

  <add item to buffer>

  mutex->Up() // release buffer
  full->Up() // another item in buffer
}

BoundedBuffer::Consumer(){
  full->Down() // get item
  mutex->Down() // get access to buffer

  <remove item from buffer>

  mutex->Up() // release buffer
  empty->Up() // another empty slot
  <use item>
}
Semaphore Summary

• Semaphores can be used for three purposes:
  – to ensure mutually exclusive execution of a critical section (like locks)
  – to control access to a shared pool of resources (using a counting semaphore)
  – to cause one thread to wait for a specific event

• AND
  – No busy wait

• So... They’re perfect! Right?
Um, No.
(Problems with Semaphores)

• Huge step up from what we had, but...
• Essentially shared global variables
• Too many purposes
  – Waiting for a condition is independent of mutual exclusion
• No control or guarantee of proper usage
• Difficult to read (and develop) code
• Often studied for history
  – Not typically used in new application code
  – (Where are they used?)
• So...
What NOT to do

Semaphore mutex = 1 //access to buffer
Semaphore empty = N //count of empty slots
Semaphore full = 0 //count of full slots
int buffer[N]

BoundedBuffer::Producer(){
    <produce item>
    empty->Down() //get empty spot
    mutex->Down() //get access to buffer

    <add item to buffer>

    mutex->Up() //release buffer
    full->Up() //another item in buffer
}

BoundedBuffer::Consumer(){
    mutex->Down() //get access to buffer
    full->Down() //get item

    <remove item from buffer>

    mutex->Up() //release buffer
    empty->Up() //another empty slot
    <use item>
}
Summary

• Locks are a higher-level programming abstraction
  – Mutual exclusion can be implemented using locks
  – Lock implementation generally requires hardware support
  – Locks can busy-wait, and busy-waiting cheaply is important

• Semaphores are basically generalized locks
  – Used for mutual exclusion and synchronization
  – Each semaphore supports a queue of processes that are waiting to access a critical section
  – No busy waiting! Threads sleep inside wait() until they have the resource
Announcements

• Homework 3 due Friday in section
• Project 0 due Friday, 11:59p
• Project 1 posted today
• Exam 1 in TWO weeks (Wednesday)
  – 7p-9p in WEL 2.224