Locks and Semaphores

CS439: Principles of Computer Systems
September 18, 2019
Bringing It All Together

• Processes
  – Abstraction for protection
  – Define address space

• Threads
  – Every process has at least one kernel thread
  – More threads can be created using either system calls (kernel threads) or libraries (user threads)
  – Help us model the real world and exploit concurrency in the system
  – Share (and communicate) through global and static data, share the heap, each has its own stack and full use of the registers
  – Race conditions may be a problem!

• Dual-Mode Execution
  – Processes and threads can execute in either mode

• CPU Schedulers
  – Schedule both kernel threads and processes
  – Since each process has one kernel thread, sometimes processes are treated as threads
  – Helps with the illusion of infinite resources!
Too Much Milk: Solution #2

You (Thread A)
leave note A
if(noNote B)
  if(noMilk)
    buy milk;
remove note A

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

Does this work?  A. Yes    B. No
Too Much Milk: Solution #3

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

Does this work?  A. Yes  B. No
Your Roommate (Thread B)
leave note B
if(noNote A)
  if(noMilk)
    buy milk;
remove note B

At this if, either there is a note A or not.

If not, it is safe for B to check and buy milk, if needed. (Thread A has not started yet.)

If yes, then thread A is checking and buying milk as needed or is waiting for B to quit, so B quits by removing note B.
Why is it correct?

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

if(noMilk)
buy milk;

remove note A

At this while, either there is a note B or not.

If not, it is safe for A to buy since B has either not started yet or quit.

If yes, A waits until there is no longer a note B, and either finds milk that B bought or buys it if needed.
Why is it correct?

So Thread B buys milk (which Thread A finds) or not, but either way it removes note B. Since Thread A loops, it waits for B to buy milk or not, and then if B did not buy it, it buys the milk.
So it’s correct, but... is it good?

1. It is too complicated. It was hard to convince ourselves this solution worked.
2. It is asymmetrical---thread A and thread B are different. What would we need to do to add new threads?
3. A is busy waiting, or consuming CPU resources despite the fact it is not doing any useful work.
Too Much Milk: Taking Turns

turn=A

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

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

turn = B;

turn = A;
Too Much Milk: Lock Solution

```cpp
Lock lock;

All Threads
lock->Acquire();
if(noMilk)
    buy milk;
lock->Release();
```
Today’s Additions

• Eliminating race conditions! OR
  Forcing threads to behave properly

• Synchronization Terminology
  – Atomic Operations
  – Mutual exclusion, critical sections
    • Safety, Liveness, Bounded waiting

• Synchronization Primitives
  – Locks
  – Semaphores
Formalizing “Too Much Milk”

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

• We did a lot of work to prevent race conditions around these actions!
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
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
Formalizing “Too Much Milk”

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

• Safety property
  – One person buys milk

• Liveness
  – Someone buys milk when needed
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
Safety and Liveness for Critical Sections

• No more than 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
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!*
Concurreny Quiz

If two threads execute this program concurrently, how many different final values of the global variable X are there?

Initially, X == 0.

Thread 1

```c
void increment() {
    int tmp = X;
    tmp = tmp + 1;
    X = tmp;
}
```

Thread 2

```c
void increment() {
    int tmp = X;
    tmp = tmp + 1;
    X = tmp;
}
```

A. 0  B. 1  C. 2  D. More than 2
Schedules/Interleavings

• Model of concurrent execution
• Interleave statements from each thread into a single thread
• If any interleaving yields incorrect results, some synchronization is needed

If X==0 initially, X == 1 at the end. WRONG result!
Terminology

✓ **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

- **Atomic Operation**: an operation that is uninterruptible
- **Synchronization**: Using atomic operations to ensure cooperation between threads
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
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?
Synchronization Primitives
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?
Locks, Generally

A lock allows one thread to prevent another thread 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

- Locks have two states: Busy and Free
  - Lock is initially free

- Rules for using a lock:
  - Acquire the lock before accessing shared data
  - Release the lock after finishing with shared data
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 lock;

All Threads
lock->Acquire();
if(noMilk)
   buy milk;
lock->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()
{
    disable interrupts;
}

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

Lock::Release()
{
    disable interrupts;
    if queue is not empty{
        take thread1 off wait queue
        put thread1 on ready queue
    }
    value = FREE;
    enable interrupts;
}
Interrupts and Context Switches

- The interrupt state is part of the thread state
- When a thread blocks, its interrupt state is saved with the rest of its context
- When a process continues running, its interrupt state is restored with the rest of its context
Using this Technique

• Disabling interrupts can have unwanted side effects
  – Why do we have interrupts?
• 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!
Let’s Try Again... What is Our Goal?

- Ensure mutual exclusion, liveness, etc.
- But, practically?
  - See if another thread is executing the section (*read* a variable)
  - If it isn’t, grab the lock (*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
Test&Set with Cheaper Busy Waiting

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

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

Voluntary yield of the CPU
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;
}
```
Is mutual exclusion enough?

• 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?
  – What if there is more than one resource available?
Semaphores
Semaphores

• Semaphores offer elegant solutions to synchronization problems
  – Mutual exclusion and more general synchronization

• Semaphores are basically generalized locks
  – Support two atomic operations (Up & Down!)
  – Has a value, but the value has more options than busy/free
  – Supports a queue of threads that are waiting to access a resource

• 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 more general synchronization

Only implementation difference is the initial value!
Semaphores as Locks
(Binary Semaphores)
Using Binary Semaphores

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

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

• 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() \textit{(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() \textit{(Verhogen, or “release” in Dutch)}
  – Increments the value
  – Never blocks
  – If a thread is asleep on the wait queue, it will be awakened
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 sema = 1;

All Threads
sema->down();
if(noMilk)
  buy milk;
sema->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 threads 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> //decrement

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

• 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
  – down() and up() are called by different threads from different parts of the code base
Summary

• Critical sections identify pieces of code that cannot be executed in parallel by multiple threads
  – Typically code that accesses or modifies shared variables
• Locks define critical sections
  – 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 more general synchronization
  – Each semaphore supports a queue of processes that are waiting to access a critical section
  – No busy waiting! Threads sleep inside down() until they have the resource
Announcements

• Discussion sections Friday! Problem Set 3 posted.
• Project 0 due Friday, 5:59p for code, 11:59p for design doc
• Project 1 posted today
  – Discussion Section this week will provide an introduction to Pintos. Read the documentation first!
• Exam 1 in two weeks (Monday)
  – 7p-9p UTC 2.102A