Synchronization I

CS439: Principles of Computer Systems

February 9, 2015

Last Time

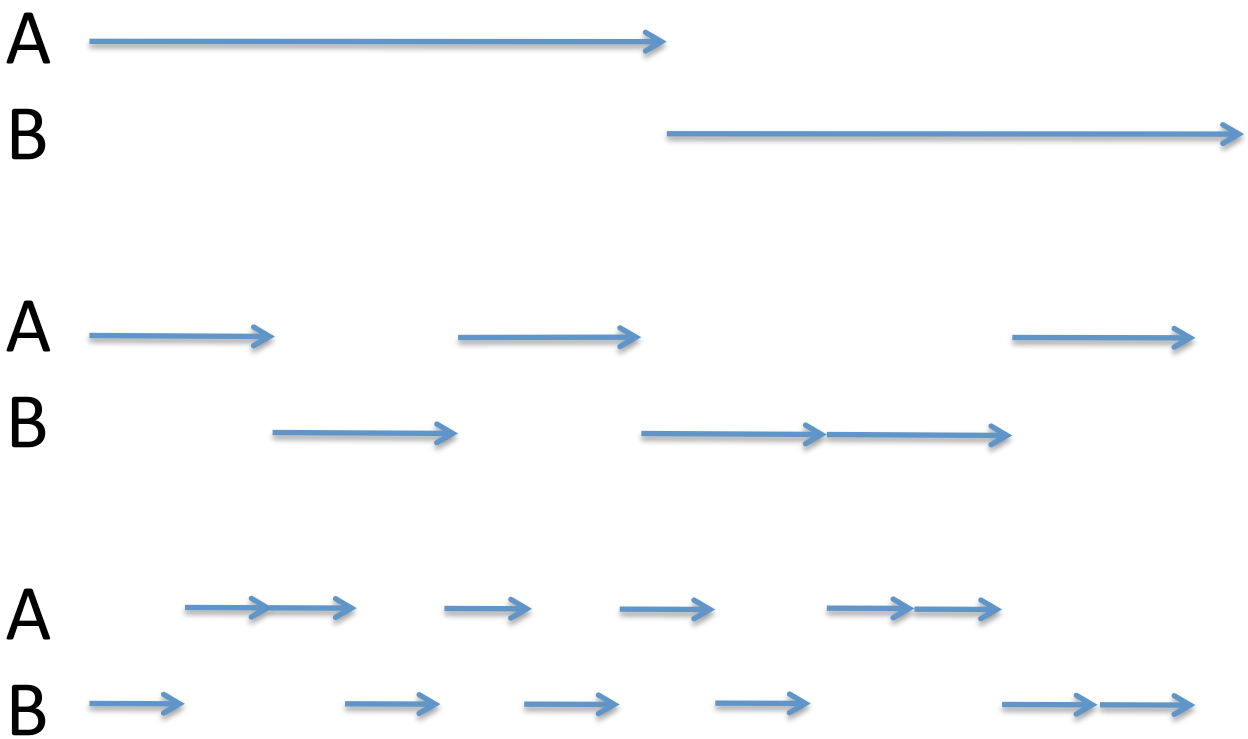
- Introduced Threads
  - Why we want them, what they are, how they differ from processes
  - Kernel vs. User
  - Independent vs. Cooperating
- Too Much Milk
  - Race conditions: different result based on scheduling
  - Critical Sections: a piece of code only one thread can execute at a time
  - Atomic Operations: uninterruptible operations
  - Mutual exclusion
  - Safety
  - Liveness
  - Bounded waiting

Terminology, Revisited

- **Safety**: At most one thread is executing in the critical section (mutual exclusion)
- **Liveness**: If no threads are executing in the critical section and some thread attempts to enter the critical section, then eventually the thread succeeds
- **Bounded Waiting**: If thread T attempts to enter the critical section, then there exists a bound on the number of times other threads succeed in entering the critical section before T does
  - If the bound is left unspecified, it is a **liveness** property, because we can always extend the execution to show that the bound exists
  - As soon as a specific bound is offered, it becomes a **safety** property, since it must hold in every prefix of the execution

Threads and the Scheduler
(or, Why Multi-threaded Programming is Hard)

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



Concurrency Quiz

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

Initially, X == 0.

Thread 1

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

Thread 2

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

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

Thread 1

```
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;
```

```
tmp1 = X;
tmp2 = X;
tmp2 = tmp2 + 1;
tmp1 = tmp1 + 1;
X = tmp1;
X = tmp2;
```

Thread 2

```
tmp2 = X;
tmp2 = tmp2 + 1;
X = tmp2;
```

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

Today's Agenda

- 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

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
- No busy waiting (spin locks)
  - Threads should go to sleep 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 process 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

Too Much Milk: Lock Solution

You (Thread A)

```
Lock->Acquire();
if(noMilk)
   buy milk;
Lock->Release();
```

Your Roommate (Thread B)

```
Lock->Acquire();
if(noMilk)
   buy milk;
Lock->Release();
```

So… Implementing Locks

Locks: API

- Create through declaration:

  Lock myLock;

  Lock yourLock;
- Two states
  - Busy
  - Free
- Two methods
  - Lock::acquire()
    - waits until lock is Free and then atomically makes lock Busy
  - Lock::release()
    - makes lock Free. If there are pending acquire()-s, causes one to proceed

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

```
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->sleep()
  }
  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->sleep()
  }
  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) {
     add thread to wait queue
     enable interrupts;
     thread->sleep()
  }
  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!
  - These are becoming more and more common
- The OS does use this technique when it is updating some data structures

What are we trying to do?

- 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 shorted 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

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

```
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->sleep()&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)

9a Unregistered iClickers

#07EB2BC7
#3D26667D
#891763FD
#894EA562
#92CCA3FD

12p Unregistered iClickers

#08AFB81F
#09557529
#835D4997
#83862520
#929F8D80

Semaphores

Semaphores

- Semaphores are basically generalized locks
  - Support two atomic operations (Up & Down!)
  - Offer elegant solutions to synchronization problems
- Used for mutual exclusion *and* synchronization
- 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.

Too Much Milk: Semaphore Solution

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()                 //Decrement value
                          //If value == 0, wait on queue
<critical section>

S->Up()                   //Increment value
                          //If there is a waiter on the
                          //queue, wake it up
```

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);
  }
}
```

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, 8:45a
- Project 0 due Friday, 11:59p
- Project 1 posted today
- Exam 1 in TWO weeks (Wednesday)
  - 7p-9p in UTC 2.112A