CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 9

How to work in a group / Synchronization (finished)

February 24th, 2014

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Tips for Programming in a Design Team
- Synchronization (continued)
  - Lock Free Synchronization
  - Monitors

Interactive is important!

Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: Synchronization

- **Atomic Operation**: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its hard to build anything useful with only reads and writes
- **Critical Section**: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

Recall: Atomic Instructions

- test&set (&address) { /* most architectures */
  result = M[address];
  M[address] = 1;
  return result;
  }
- swap (&address, register) { /* x86 */
  temp = M[address];
  M[address] = register;
  register = temp;
  }
- compare&swap (&address, reg1, reg2) { /* 68000 */
  if (reg1 == M[address]) {
  M[address] = reg2;
  return success;
  } else {
  return failure;
  }
  }
- load-linked&store conditional(&address) {
  /* R4000, alpha */
  loop:
  ll r1, M[address];
  movi r2, 1; /* Can do arbitrary comp */
  sc r2, M[address];
  beqz r2, loop;
  }

Recall: Portable Spinlock constructs in Linux

- Linux provides lots of synchronization constructs
  - We will highlight them throughout the term
- Example: Spin Lock support: Not recursive!
  - Only a lock on multiprocessors: Becomes simple preemption disable/enable on uniprocessors

```
#include <linux/spinlock.h>
DEFINE_SPINLOCK(my_lock);

spin_lock(&my_lock);
/* Critical section … */
spin_unlock(&my_lock);
```

- Disable interrupts and grab lock (while saving and restoring state in case interrupts already disabled):

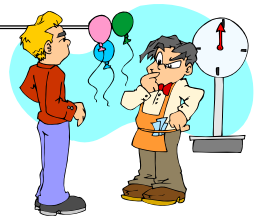
```
DEFINE_SPINLOCK(my_lock);
unsigned long flags;

spin_lock_irqsave(&my_lock, flags);
/* Critical section … */
spin_unlock_irqrestore(&my_lock);
```


Tips for Programming in a Project Team

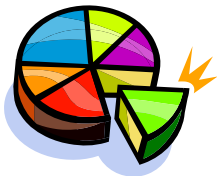
"You just have to get your synchronization right!"

- Big projects require more than one person (or long, long, long time)
  - Big OS: thousands of person-years!
- It's very hard to make software project teams work correctly
  - Doesn't seem to be as true of big construction projects
    - » Empire state building finished in one year: staging iron production thousands of miles away
    - » Or the Hoover dam: built towns to hold workers
  - Is it OK to miss deadlines?
    - » We make it free (slip days)
    - » Reality: they're very expensive as time-to-market is one of the most important things!

Big Projects



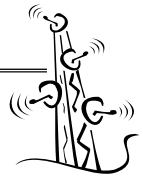
- What is a big project?
  - Time/work estimation is hard
  - Programmers are eternal optimistics (it will only take two days)!
    - » This is why we bug you about starting the project early
    - » Had a grad student who used to say he just needed "10 minutes" to fix something. Two hours later…
- Can a project be efficiently partitioned?
  - Partitionable task decreases in time as you add people
  - But, if you require communication:
    - » Time reaches a minimum bound
    - » With complex interactions, time increases!
  - Mythical person-month problem:
    - » You estimate how long a project will take
    - » Starts to fall behind, so you add more people
    - » Project takes even more time!



Techniques for Partitioning Tasks

- Functional
  - Person A implements threads, Person B implements semaphores, Person C implements locks…
  - Problem: Lots of communication across APIs
    - » If B changes the API, A may need to make changes
    - » Story: Large airline company spent $200 million on a new scheduling and booking system. Two teams "working together." After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another $200 million to fix.
- Task
  - Person A designs, Person B writes code, Person C tests
  - May be difficult to find right balance, but can focus on each person's strengths (Theory vs systems hacker)
  - Since Debugging is hard, Microsoft has *two* testers for *each* programmer
- Most Berkeley project teams are functional, but people have had success with task-based divisions

Communication



- More people mean more communication
  - Changes have to be propagated to more people
  - Think about person writing code for most fundamental component of system: everyone depends on them!
- Miscommunication is common
  - "Index starts at 0? I thought you said 1!"
- Who makes decisions?
  - Individual decisions are fast but trouble
  - Group decisions take time
  - Centralized decisions require a big picture view (someone who can be the "system architect")
- Often designating someone as the system architect can be a good thing
  - Better not be clueless
  - Better have good people skills
  - Better let other people do work

Coordination

- More people ⇒ no one can make all meetings!
  - They miss decisions and associated discussion
  - Example from earlier class: one person missed meetings and did something group had rejected
  - Why do we limit groups to 5 people?
    - » You would never be able to schedule meetings otherwise
  - Why do we require 4 people minimum?
    - » You need to experience groups to get ready for real world
- People have different work styles
  - Some people work in the morning, some at night
  - How do you decide when to meet or work together?
- What about project slippage?
  - It will happen, guaranteed!
  - Ex: phase 4, everyone busy but not talking. One person way behind. No one knew until very end – too late!
- Hard to add people to existing group
  - Members have already figured out how to work together

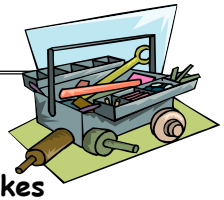
How to Make it Work?

- People are human. Get over it.
  - People will make mistakes, miss meetings, miss deadlines, etc. You need to live with it and adapt
  - It is better to anticipate problems than clean up afterwards.
- Document, document, document
  - Why Document?
    - » Expose decisions and communicate to others
    - » Easier to spot mistakes early
    - » Easier to estimate progress
  - What to document?
    - » Everything (but don't overwhelm people or no one will read)
  - Standardize!
    - » One programming format: variable naming conventions, tab indents,etc.
    - » Comments (Requires, effects, modifies)—javadoc?

Suggested Documents for You to Maintain

- Project objectives: goals, constraints, and priorities
- Specifications: the manual plus performance specs
  - This should be the first document generated and the last one finished
  - Consider your Cucumber specifications as one possibility
- Meeting notes
  - Document all decisions
  - You can often cut & paste for the design documents
- Schedule: What is your anticipated timing?
  - This document is critical!
- Organizational Chart
  - Who is responsible for what task?

Use Software Tools



- Source revision control software
  - Git – check in frequently. Tag working code
  - Easy to go back and see history/undo mistakes
  - Work on independent branches for each feature
    - » Merge working features into your master branch
    - » Consider using "rebase" as well
  - Communicates changes to everyone
- Redmine
  - Make use of Bug reporting and tracking features!
  - Use Wiki for communication between teams
  - Also, consider setting up a forum to leave information for one another about the current state of the design
- Use automated testing tools
  - Rebuild from sources frequently
  - Run Cucumber tests frequently
    - » Use tagging features to run subsets of tests!

Test Continuously

- Integration tests all the time, not at 11pm on due date!
  - Utilize Cucumber features to test frequently!
  - Write dummy stubs with simple functionality
    - » Let's people test continuously, but more work
  - Schedule periodic integration tests
    - » Get everyone in the same room, check out code, build, and test.
    - » Don't wait until it is too late!
    - » This is exactly what the autograder does!
- Testing types:
  - Integration tests: Use of Cucumber with BDD
  - Unit tests: check each module in isolation (CUnit)
  - Daemons: subject code to exceptional cases
  - Random testing: Subject code to random timing changes
- Test early, test later, test again
  - Tendency is to test once and forget; what if something changes in some other part of the code?

Administrivia

- Recall: Midterm I: 2½ weeks from today!
  - Wednesday 3/12
  - Intention is a 1.5 hour exam over 3 hours
  - No class on day of exam!
- Midterm Timing:
  - Probably 4:00-7:00PM in 320 Soda Hall
  - Does this work? (Still confirming room)
- Topics: everything up to the previous Monday
  - OS Structure, BDD, Process support, Synchronization, Memory Management, File systems

Computers in the News: Net Neutrality?

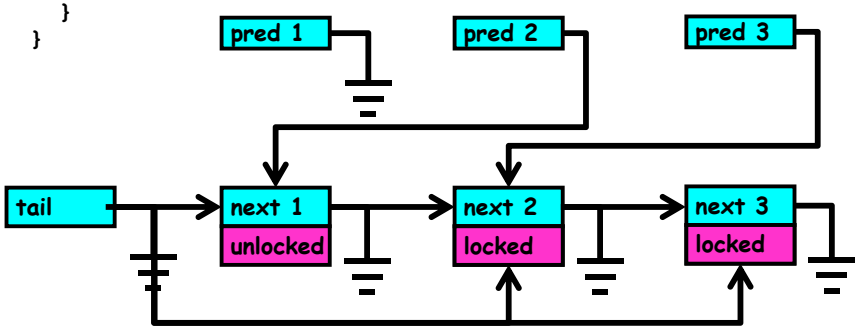
- Net Neutrality:
  - "The principle that Internet service providers and governments should treat all data on the Internet equally, not discriminating or charging differentially by user, content, site, platform, application, type of attached equipment, and modes of communication."

    From Wikipedia
- Do we have Net Neutrality?
  - FCC has issued "rules" or "policies", but these do not have a lot of teeth
  - Supporters want internet service providers to be declare as "common carriers"
    - » A common carrier holds itself out to provide service to the general public without discrimination for the "public convenience and necessity".
- In the news: Netflix
  - Comcast and/or others reported to have dropped Netflix Bandwidth intentionally
  - Just announced: new "agreements" with Comcast and Verizon to get more bandwidth (i.e. pay more for access!)

Take 2: Mellor-Crummey-Scott Lock

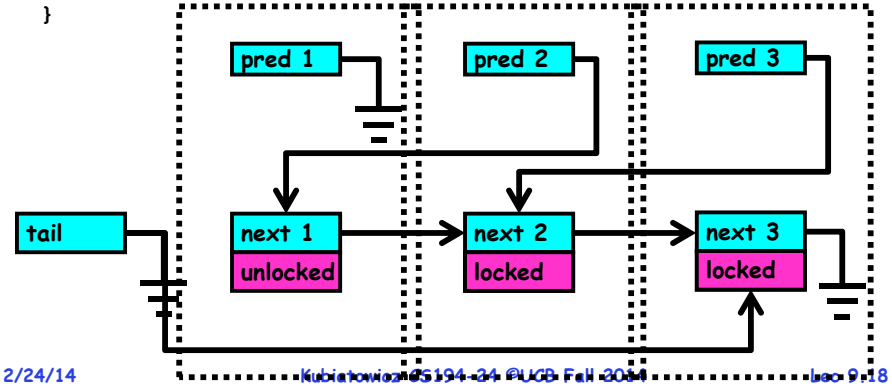
- Another lock-free option – Mellor-Crummey Scott (MCS):

```
public void lock() {
  QNode qnode = myNode.get();
  qnode.next = null;
  QNode pred = swap(tail,qnode);
  if (pred != null) {
    qnode.locked = true;
    pred.next = qnode;
    while (qnode.locked); // wait for predecessor
  }
}
```



Mellor-Crummey-Scott Lock (con't)

```
public void unlock() {
  QNode qnode = myNode.get();
  if (qnode.next == null) {
    if (compare&swap(tail,qnode,null))
      return;
    // wait until predecessor fills in my next field
    while (qnode.next == null);
  }
  qnode.next.locked = false;
}
```



Mellor-Crummey-Scott Lock (Con't)

- Nice properties of MCS Lock
  - Never more than 2 processors spinning on one address
  - Completely fair – once on queue, are guaranteed to get your turn in FIFO order
    - » Alternate release procedure doesn't use compare&swap but doesn't guarantee FIFO order
- Bad properties of MCS Lock
  - Takes longer (more instructions) than T&T&S if no contention
  - Releaser may be forced to spin in rare circumstances
- Hardware support?
  - Some proposed hardware queueing primitives such as QOLB (Queue on Lock Bit)
  - Not broadly available

Recall: Portable Semaphores in Linux

- Initialize general semaphore:

```
struct semapore my_sem;
sema_init(&my_sem, count);    /* Initialize semaphore */
```

- Initialize mutex (semaphore with count = 1)

```
static DECLARE_MUTEX(my_mutex);
```

- Acquire semaphore:

```
down_interruptible(&my_sem);  // Acquire sem, sleeps with
                              // TASK_INTERRUPTIBLE state
down(&my_sem);                // Acquire sem, sleeps with
                              // TASK_UNINTERRUPTIBLE state
down_trylock(&my_sem);        // Return ≠ 0 if lock busy
```

- Release semaphore:

```
up(&my_sem);                  // Release lock
```

Recall: Portable (native) mutexes in Linux

- Interface optimized for mutual exclusion

```
DEFINE_MUTEX(my_mutex);   /* Static definition */

struct mutex my_mutex;
mutex_init(&my_mutex);            /* Dynamic mutex init */
```

- Simple use pattern:

```
mutex_lock(&my_mutex);
/* critical section … */
mutex_unlock(&my_mutex);
```

- Constraints
  - Same thread that grabs lock must release it
  - pProcess cannot exit while holding mutex
  - Recursive acquisitions not allowed
  - Cannot use in interrupt handlers (might sleep!)
- Advantages
  - Simpler interface
  - Debugging support for checking usage
- Prefer mutexes over semaphores unless mutex really doesn't fit your pattern!

Recall: Completion Patterns

- One use pattern that does not fit mutex pattern:
  - Start operation in another thread/hardware container
  - Sleep until woken by completion of event
- Can be implemented with semaphores
  - Start semaphore with count of 0
  - Immediate down() – puts parent to sleep
  - Woken with up()
- More efficient: use "completions":

```
DEFINED_COMPLETION();         /* Static definition */

struct completion my_comp;
init_completion(&my_comp);    /* Dynamic comp init */
```

- One or more threads to sleep on event:

```
wait_for_completion(&my_comp);  /* put thead to sleep */
```

- Wake up threads (can be in interrupt handler!)

```
complete(&my_comp);
```


Definition of Monitor

- Semaphores are confusing because dual purpose:
  - Both mutual exclusion and scheduling constraints
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
- Lock: provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

Programming with Monitors

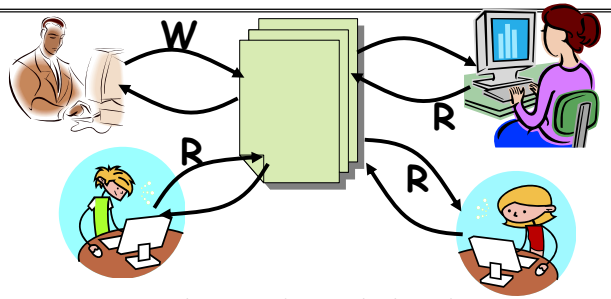
- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
   condvar.wait();
}
unlock

do something so no need to wait

lock

condvar.signal();

unlock
```

Check and/or update state variables
Wait if necessary

Check and/or update state variables

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - Reader()
    ```
    Wait until no writers
    Access data base
    Check out - wake up a waiting writer
    ```
  - Writer()
    ```
    Wait until no active readers or writers
    Access database
    Check out - wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called "lock"):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Conditioin okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) {  // Is it safe to read?
    WR++;                  // No. Writers exist
    okToRead.wait(&lock);  // Sleep on cond var
    WR--;                  // No longer waiting
  }
  AR++;                    // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                    // No longer active
  if (AR == 0 && WW > 0)   // No other active readers
    okToWrite.signal();    // Wake up one writer
  lock.Release();
}
```


Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) {  // Is it safe to write?
    WW++;                  // No. Active users exist
    okToWrite.wait(&lock); // Sleep on cond var
    WW--;                  // No longer waiting
  }
  AW++;                    // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                    // No longer active
  if (WW > 0){             // Give priority to writers
    okToWrite.signal();    // Wake up one writer
  } else if (WR > 0) {     // Otherwise, wake reader
    okToRead.broadcast();  // Wake all readers
  }
  lock.Release();
}
```

pThreads Monitors (Mutex + Condition Vars)

- To create a mutex:

```
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To create condition variables:

```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
pthread_cond_init(mycondvar, NULL);
```

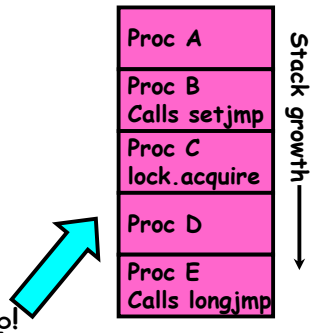
- To use them:

```
pthread_cond_wait(mycondvar,amutex);
pthread_cond_signal(mycondvar);
pthread_cond_broadcast(mycondvar);
```


C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know *all* the code paths out of a critical section

```
int Rtn() {
  lock.acquire();
  …
  if (exception) {
    lock.release();
    return errReturnCode;
  }
  …
  lock.release();
  return OK;
}
```



  - Watch out for `setjmp/longjmp`!
    - » Can cause a non-local jump out of procedure
    - » In example, procedure E calls longjmp, poping stack back to procedure B
    - » If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
  lock.acquire();
  try {
    …
    DoFoo();
    …
  } catch (…) {      // catch exception
    lock.release(); // release lock
    throw;          // re-throw the exception
  }
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  - Even Better: auto_ptr<T> facility. See C++ Spec.
    - » Can deallocate/free lock regardless of exit method

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor
- Negatives
  - This is very inefficient because the busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- Priority Inversion problem with original Martian rover
- For semaphores and monitors, waiting thread may wait for an arbitrary length of time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Exam solutions should not have busy-waiting!

Busy-wait vs Blocking

- Busy-wait: I.e. spin lock
  - Keep trying to acquire lock until read
  - Very low latency/processor overhead!
  - Very high system overhead!
    - » Causing stress on network while spinning
    - » Processor is not doing anything else useful
- Blocking:
  - If can't acquire lock, deschedule process (I.e. unload state)
  - Higher latency/processor overhead (1000s of cycles?)
    - » Takes time to unload/restart task
    - » Notification mechanism needed
  - Low system overheadd
    - » No stress on network
    - » Processor does something useful
- Hybrid:
  - Spin for a while, then block
  - 2-competitive: spin until have waited blocking time

What about barriers?

- Barrier – global (/coordinated) synchronization
  - simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier();
read_neighboring_values();
barrier();
```

  - barriers are not provided in all thread libraries
- How to implement barrier?
  - Global counter representing number of threads still waiting to arrive and parity representing phase
    - » Initialize counter to zero, set parity variable to "even"
    - » Each thread that enters saves parity variable and
      - Atomically increments counter if even
      - Atomically decrements counter if odd
    - » If counter not at extreme value spin until parity changes
      - i.e. Num threads if "even" or zero if "odd"
    - » Else, flip parity, exit barrier
  - Better for large numbers of processors – implement atomic counter via combining tree

Summary

- 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
- Semaphores: Like integers with restricted interface
  - Two operations:
    - » `P()`: Wait if zero; decrement when becomes non-zero
    - » `V()`: Increment and wake a sleeping task (if exists)
    - » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- Readers/Writers
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time