Synchronization (2)

David A. Eckhardt
School of Computer Science
Carnegie Mellon University

de0u@andrew.cmu.edu

Status Rendezvous

Handin: rough summary

- /afs/andrew.cmu.edu/scs/cs/15-412/usr/$USER
- You will *need* to cross-cell authenticate: one of
  - /usr/local/bin/aklog cs.cmu.edu
  - /usr/local/bin/afslog cs.cmu.edu
- Watch academic.cs.15-412.announce for precise directions
  - (please follow them!)

Partner selection for Project 2

- de0u+partner@andrew
  - or de0u+partners@andrew (I am learning)
- By Tuesday 2002-03-04 23:59 EST
- Only 6 as of midnight
- At least one bboard post

Outline

Ways to get mutual exclusion

- Hardware, software

Mutexes & Condition variables

Mutual Exclusion: Reminder

Mutual Exclusion

- Want to protect an atomic instruction sequence
- Do “something” to guard against
  - ...CPU switching to another thread
  - ...thread running on another CPU

Assumptions

- Atomic instruction sequence will be “short”
- No other thread is “likely” to be competing

Desiderata

- Typical case (no competitor) should be fast
- Atypical case can be slow
  - Should not be “too wasteful”

Mutual Exclusion: XCHG instruction

Exchange (XCHG) instruction on 80386 et seq.

```
int32 xchg(int32 *lock, int32 val) {
  register int old;
  old = *lock; /* bus is locked */
  *lock = val; /* bus is locked */
  return (old);
}
```

Initialization

- int lock_available = 1;

Lock

- i_won = xchg(&lock_available, 0); /* try-lock */
- while (!xchg(&lock_available, 0)
  - /* spin-wait */ ;

Unlock

- xchg(&lock_available, 1); /* had *better* return 0! */

Does it work?

Mutual Exclusion

- Only one thread can see lock_available == 1

Progress

- Each time lock_available == 1 a waiting thread will snatch it

Bounded Waiting

- No (not always)
- Any *particular* thread could lose arbitrarily many times

Attaining Bounded Waiting

Textbook algorithm (paraphrased)

```
waiting[i] = true;
got_it = false;
while (waiting[i] && !got_it)
  got_it = xchg(&lock_available, false);
waiting[i] = false;

/* critical section */

j = (i + 1) % n;
while ((j != i) && !waiting[j])
  j = (i + 1) % n;

if (j == i)
  xchg(&lock_available, true); /* !text*/
else
  waiting[j] = false;
```

Evaluation

One awkward requirement

One unfortunate behavior

Evaluation

One awkward requirement

- Everybody knows size of thread population
  - Always & instantly!
  - Or uses an upper bound

One unfortunate behavior

- Recall: expect *zero* competitors
- Algorithm: O(n) in *maximum possible* competitors

Am I too demanding?

- Baker's Algorithm has these misfeatures...

Environmental Considerations

Uniprocessor

- Entry: what if xchg() didn't work the first time?
  - Some other process has the lock
  - That process isn't running (because you are)
  - xchg() is a poor way to yield the processor
- Exit: what about bounded waiting?
  - Next xchg() winner "chosen" by thread scheduler
  - How capricious are real thread schedulers?

Multiprocessor

- Entry
  - Spin-waiting probably justified
- Exit
  - Next xchg() winner "chosen" by memory hardware
  - How capricious are real memory controllers?

Other Hardware

Test&Set

```
boolean testandset(int32 *lock) {
  register boolean old;
  old = *lock;  /* bus is locked */
  *lock = true; /* bus is locked */
  return (old);
}
```

Load-linked, Store-conditional

- For multiprocessors - bus locking *considered harmful*
- Split XCHG into halves
- Load-linked fetches old value from memory
- Store-conditional stores new value *if nobody else did*
  - Your cache snoops the bus - better than locking it!

Other Hardware

Intel i860 magic lock bit

- Instruction sets processor in “lock” mode
  - Locks bus
  - Disables interrupts
- Isn’t that dangerous?
  - 32-cycle countdown timer triggers unlock
  - Exception triggers unlock
  - Memory write triggers unlock

Excessive for critical-section entry protocol?

- Yes, but not for ...

Mutual Exclusion: Software

Lamport's "Fast Mutual Exclusion" algorithm

- 5 writes, 2 reads (if no contention)
- Not bounded-waiting (in theory, i.e., if contention)
- http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-7.html

Passing the buck

- Q: Why not ask the OS to provide mutex_lock()?
  - Uniprocessor
    - Kernel *automatically* excludes other threads
    - Kernel can easily disable interrupts
  - Multiprocessor
    - Kernel can issue "remote interrupt" to other CPU
- A: *Too expensive*
  - Because... (you know this song!)

Mutual Exclusion: *Tricky* Software

Fast Mutual Exclusion for Uniprocessors

- Bershad, Redell, Ellis: ASPLOS V (1992)

Want uninterruptable instruction sequences?

- Pretend!
- After all, they *usually* aren't interrupted...

When pretense fails?

- Kernel can simulate unfinished instructions (yuck)
- Special contract between user and OS
  - Certain sequences are *restartable* (idempotent)
    - Maybe a special memory area
    - Maybe sequences using only selected instructions
  - Thread-switch slides program counter back to start

Review

Atomic instruction sequence

- Nobody else may interleave same/“related” sequence
- *Short* sequence of instructions
  - Ok to force competitors to wait
- Probability of collision is “low”
  - Avoid expensive exclusion method

Voluntary de-scheduling

- Can’t proceed with this world state
- Wrong to hold world locked while others wait
  - It will be a while
  - We *want* others to run - they *enable* us
- CPU *de*-scheduling is an OS service!

Atomic instruction sequences

Mutex aka Lock aka Latch

- Use object to specify interfering code sequence/sequences
- Object methods encapsulate entry & exit protocols

Code example

```
mutex_lock(&store->lock);
cash = store->cash
cash += 50;
personal_cash -= 50;
store->cash = cash;
mutex_unlock(&store->lock);
```

What's inside?

- xchg() (or something else)
- spin-wait (on a multiprocessor; maybe limited)
- thread_yield() (especially on uniprocessor)

Voluntary de-scheduling

The Situation

- You hold lock on shared resource, not in “right mode”
- Action sequence
  - Unlock shared resource
  - Go to sleep until resource changes state

Very Wrong

```
while (!reckoning)
  mutex_lock(&scenario_lk);
  if ((date >= 1906-04-18) && (hour >= 5))
    reckoning = true;
  else
    mutex_unlock(&scenario_lk);

wreak_general_havoc();
mutex_unlock(&scenario_lk);
```

Voluntary de-scheduling

Arguably Less Wrong

```
while (!reckoning)
  mutex_lock(&scenario_lk);
  if ((date >= 1906-04-18) && (hour >= 5))
    reckoning = true;
  else {
    mutex_unlock(&scenario_lk);
    sleep(1);
}

wreak_general_havoc();
mutex_unlock(&scenario_lk);
```

Something is missing

- Mutex for shared state: good
- How can we sleep for the *right* duration?
  - Get an expert to tell us!

Condition Variable

Once more, with feeling!

```
mutex_lock(&scenario_lk);
while (!reckoning)
  if ((date >= 1906-04-18) && (hour >= 5))
    reckoning = true;
  else {
    condition_wait(&scenario_lk, &clock);
  }
wreak_general_havoc(); /* locked! */
mutex_unlock(&scenario_lk);
```

What wakes us up?

```
iterator = universe_iterator();
while (o = iterator->next())
  o->update();
/* done with all objects, time can pass */
condition_signal(&clock);
```

Condition Variable Design

Basic Requirements

- Keep track of threads asleep “for a while”
- Allow notifier thread to wake sleeping thread(s)
- Must be thread-safe

condition_wait(mutex, cvar) - why two params?

- Lock required to access/modify the shared state
- So whoever awakens you will need to hold that lock
    - ...you’d better give it up.
- When you wake up, you will need to re-lock to access state
- “Natural” for condition_wait() to handle un-lock/re-lock
    - ...but there’s something more subtle

Condition Variable Implementation

Under the hood

- mutex - multiple threads could condition_wait() at same time
- “queue” - of sleeping processes
  - May be FIFO or more exotic

condition_wait sequence

- lock(cvar->mutex);
- enqueue(cvar->queue, my_thread_id());
- unlock(mutex);
- *ATOMICALLY*
  - unlock(cvar->mutex);
  - pause_thread();

Condition Variable Atomic Sleep

What is this "atomic" stuff?

- ...and why can't we use a mutex?

Pathological execution sequence

condition_wait(mutex, cvar);	condition_signal(cvar);
enqueue(cvar->queue, my_thread_id());	
unlock(mutex);	
unlock(cvar->mutex);	
	lock(cvar->mutex);
	tid = dequeue(cvar->q);
	wake_thread(tid);
	unlock(cvar->mutex);
pause_thread(); /* asleep forever */	

Achieving condition_wait() Atomicity

Some choices

- Disable interrupts (if you are a kernel)
- Rely on OS to implement conditio variables (yuck?)
- Have a “better” sleep()/wait() interface

Summary

We did it!

- Two objects for two core operations
- Understanding of underlying techniques
- Understanding of environmental factors

What next?

- [Project 2 handout!]
- Semaphores, monitors, Java, deadlock