CS 316: Synchronization-II

Andrew Myers

Fall 2007

Computer Science

Cornell University

Announcements

- Corewars due tonight
- Sections about PA 6: Attend!

- PA 6 will be shipped after exam
  - Demos: 2-6pm, Dec 13 (Graphics lab Rhodes 455)
  - Due: 10am, Dec 13

- Prelim 2: Thursday 29, 7:30-10:00
  - Location PH 219
  - Prelim review: Wed: 6-8, location: Upson 315

- Corewars Pizza Party: Friday night 5-9pm
  - Location: Hollister 110

Programming with threads

- Need it to exploit multiple processing units
  …to provide interactive applications
  …to write servers that handle many clients
- Problem: hard even for experienced programmers
  - Behavior can depend on subtle timing differences
  - Bugs may be impossible to reproduce
- Needed: synchronization of threads

Goals

- Concurrency poses challenges for:
- Correctness
  - Threads accessing shared memory should not interfere with each other
- Liveness
  - Threads should not get stuck, should make forward progress
- Efficiency
  - Program should make good use of available computing resources (e.g., processors).
- Fairness
  - Resources apportioned fairly between threads

Andrew Myers, Computer Science, Cornell University

Two threads, one counter

Web servers use concurrency:

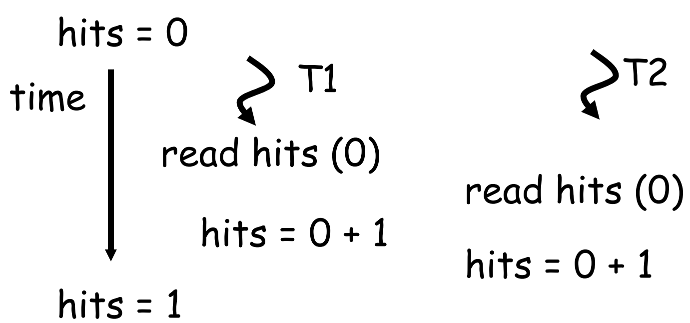
- Multiple threads handle client requests in parallel.
- Some shared state, e.g. hit counts:
  - each thread increments a shared counter to track number of hits

```
…
hits = hits + 1;
…
```

- What happens when two threads execute concurrently?

Shared counters

- Possible result: lost update!



- Timing-dependent failure ⇒ race condition
  - hard to reproduce ⇒ Difficult to debug

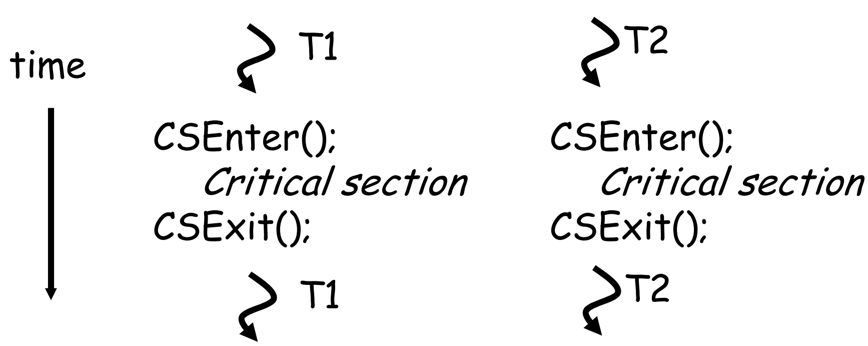
Race conditions

- Def: timing-dependent error involving access to shared state
  - Whether it happens depends on how threads scheduled: who wins “races” to instruction that updates state vs. instruction that accesses state
  - Races are intermittent, may occur rarely
    - Timing dependent = small changes can hide bug
  - A program is correct *only* if *all possible* schedules are safe
    - Number of possible schedule permutations is huge
    - Need to imagine an adversary who switches contexts at the worst possible time

Andrew Myers, Computer Science, Cornell University

Critical sections

- To eliminate races: use *critical sections* that only one thread can be in
  - Contending threads must wait to enter



Andrew Myers, Computer Science, Cornell University

Mutexes

- Critical sections typically associated with mutual exclusion locks (*mutexes*)
- Only one thread can hold a given mutex at a time
- Acquire (lock) mutex on entry to critical section
  - Or block if another thread already holds it
- Release (unlock) mutex on exit
  - Allow one waiting thread (if any) to acquire & proceed

```
pthread_mutex_init(m);
…
```

```
pthread_mutex_lock(m);       pthread_mutex_lock(m);
   hits = hits+1;               hits = hits+1;
pthread_mutex_unlock(m);     pthread_mutex_unlock(m);
```

T1 T2

Using atomic hardware primitives

- Mutex implementations usually rely on special hardware instructions that *atomically* do a read and a write.
- Requires special memory system support on multiprocessors

Mutex init: lock = false;

```
while (test_and_set(&lock));
```

Critical Section

```
lock = false;
```

test_and_set uses a special hardware instruction that sets the lock and returns the OLD value (true: locked; false: unlocked)
- Alternative instruction: compare & swap
- on multiprocessor/multicore: expensive, needs hardware support

Test-and-set

```
boolean test_and_set (boolean *lock) {
    boolean old = *lock;
   *lock = true;
   return old;
}
```

…but guaranteed to act as if no other thread is interleaved

Used to implement `pthread_mutex_lock()`

Andrew Myers, Computer Science, Cornell University

Using test-and-set for mutual exclusion

```
boolean lock = false;

while test_and_set(&lock) skip
//spin until lock is acquired.

... do critical section ...
//only one process can be in this section at a time

lock = false ;
// release lock when finished with the
// critical section
```

```
boolean test_and_set (boolean *lock) {
   boolean old = *lock;
   *lock = true;
   return old;
}
```

Andrew Myers, Computer Science, Cornell University

Spin waiting

- Example is a *spinlock*
  - Also: busy waiting or spin waiting

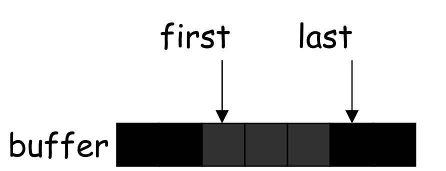
  - Efficient if wait is short
  - Wasteful if wait is long

- Heuristic:
  - spin for time proportional to expected wait time
  - If time runs out, context-switch to some other thread

Mutexes protect invariants

- Shared data must be guarded by synchronization to enforce any invariant. Example: shared queue

```
// invariant: data is in buffer[first..last-1]
char buffer[1000];
int first = 0, last = 0;
void put(char c) { // writer
   buffer[last] = c;
   last++;
}
char get() { // reader
   while (first == last);
   char c = buffer[first];
   first++;
}
```



invariant temporarily broken!

Protecting an invariant

```
// invariant: data is in buffer[first..last-1]. Protected by m.
pthread_mutex_t *m;
char buffer[1000];
int first = 0, last = 0;

void put(char c) {
    pthread_mutex_lock(m);
    buffer[last] = c;
    last++;
    pthread_mutex_unlock(m);
}
```

```
char get() {
   pthread_mutex_lock(m);
   char c = buffer[first];
   first++;         X what if first==last?
   pthread_mutex_unlock(m);
}
```

- Rule of thumb: all updates that can affect invariant become critical sections.

Guidelines for successful mutexing

- Adding mutexes in wrong place can cause *deadlock*

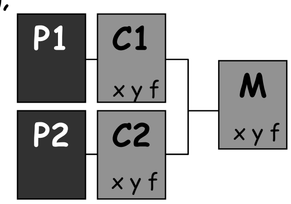
  T1: pthread_lock(m1); pthread_lock(m2);

  T2: pthread_lock(m2); pthread_lock(m1);

  - know why you are using mutexes!
  - acquire locks in a consistent order to avoid cycles
  - match lock/unlock *lexically* in program text to ensure locks/unlocks match up
    - lock(m); …; unlock(m)
    - watch out for exception/error conditions!
- Shared data should be protected by mutexes
  - Can we cheat on using mutexes? Just say no…

Relaxed consistency implications

- Nice mental model: sequential consistency
  - Memory operations happen in a way consistent with interleaved operations of each processor
  - Other processors' updates show up in program order
  - Generally thought to be expensive
- But wrong! modern multiprocessors may see inconsistent views of memory in their caches
  - P1: `x=1; y=2; f = true;`
  - P2: `while (!f) { }; print(x); print(y);`
  - Could print 12, 00, 10, 02 !



Acquire/release

- Modern synchronization libraries ensure memory updates are seen by using hardware support:
  - Acquire: forces subsequent accesses after
  - Release: forces previous accesses before
  - P1: ... ; release; ...
  - P2: ... ; acquire; ...

release consistency,

not sequential consistency

See all ... effects here

See no ... effects here

- Moral: use synchronization, don't rely on sequential consistency

Beyond mutexes

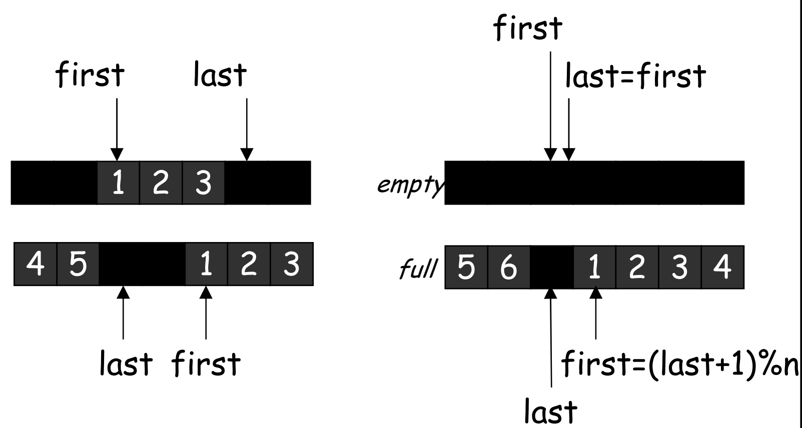
- Sometimes need to share resources in non-exclusive way
- Example: shared queue (multiple writers, multiple writers)
- How to let a reader wait for data without blocking a mutex?

```
char get() {
    while (first == last);
    char c = buffer[first];
    first++;
}
```

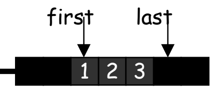
X

Example: ring buffer

- A useful data structure for IPC
- Invariant: active cells start at `first`, end at `last-1`, `last` never incremented up to `first`



A first broken cut




```
// invariant: data is in buffer[first..last-1].
mutex_t *m;
char buffer[n];
int first = 0, last = 0;

void put(char c) {
   lock(m);
   buffer[last] = c;
   last = (last+1)%n;
   unlock(m);
}
```

Same issues here for full queue

```
char get() {
  while (first == last);
  char c = buffer[first];
  first = (first+1)%n;
}
```

Oops! Reader spins on empty queue

```
char get() {
  lock(m);
  while (first == last);
  char c = buffer[first];
  first = (first+1)%n;
  unlock(m);
}
```

Oops! Blocks all writers if empty

```
char get() {
  bool done=false;
  while (!done) {
    lock(m);
    if (first != last) {
      char c = buffer[first];
      first = (first+1)%n;
      done = true;
    }
    unlock(m);
  }
}
```

Oops! Two readers can still race

Condition variables

- To let thread wait (not holding the mutex!) until a condition is true, use a *condition variable* [Hoare]
- wait(m, c) : atomically release m and go to sleep waiting for condition c, wake up holding m
  - Must be atomic to avoid *wake-up-waiting race*
- signal(c) : wake up one thread waiting on c
- broadcast(c) : wake up all threads waiting on c
- POSIX (e.g., Linux): pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast

Using a condition variable

- wait(m, c) : release m, sleep waiting for c, wake up holding m
- signal(c) : wake up one thread waiting on c

```
mutex_t *m;
cond_t *not_empty, *not_full;
char get() {
  lock(m);
  while (first == last)
     wait(m, not_empty);
  char c = buffer[first];
  first = (first+1)%n;
  unlock(m);
  signal(not_full);
}
```

```
char put(char c) {
  lock(m);
  while ((first-last)%n == 1)
     wait(m, not_full);
  buffer[last] = c;
  last = (last+1)%n;
  unlock(m);
  signal(not_empty);
}
```


Monitors

- A *monitor* is a shared concurrency-safe data structure
- Has one mutex
- Has some number of condition variables
- Operations acquire mutex so only one thread can be in the monitor at a time

- Our ring buffer implementation is a monitor
- Some languages (e.g. Java, C#) provide explicit support for monitors

Java concurrency

- Java object is a simple monitor
  - Acts as a mutex via synchronized { S } statement and synchronized methods
  - Has one (!) builtin condition variable tied to the mutex
    - o.wait() = wait(o, o)
    - o.notify() = signal(o)
    - o.notifyAll() = broadcast(o)
    - synchronized(o) {S} = lock(o); S; unlock(o)
  - Java wait() can be called even when mutex is not held. Mutex not held when awoken by signal(). Useful?

More synchronization mechanisms

Implementable with mutexes and condition variables:

- Reader/writer locks
  - Any number of threads can hold a read lock
  - Only one thread can hold the writer lock
- Semaphores
  - Some number n of threads are allowed to hold the lock
  - n=1 => semaphore = mutex
- Message-passing, sockets
  - send()/recv() transfer data and synchronize