Atomic Instructions

Hakim Weatherspoon
CS 3410, Spring 2011
Computer Science
Cornell University

P&H Chapter 2.11

Announcements

PA4 due *next*, Friday, May 13th

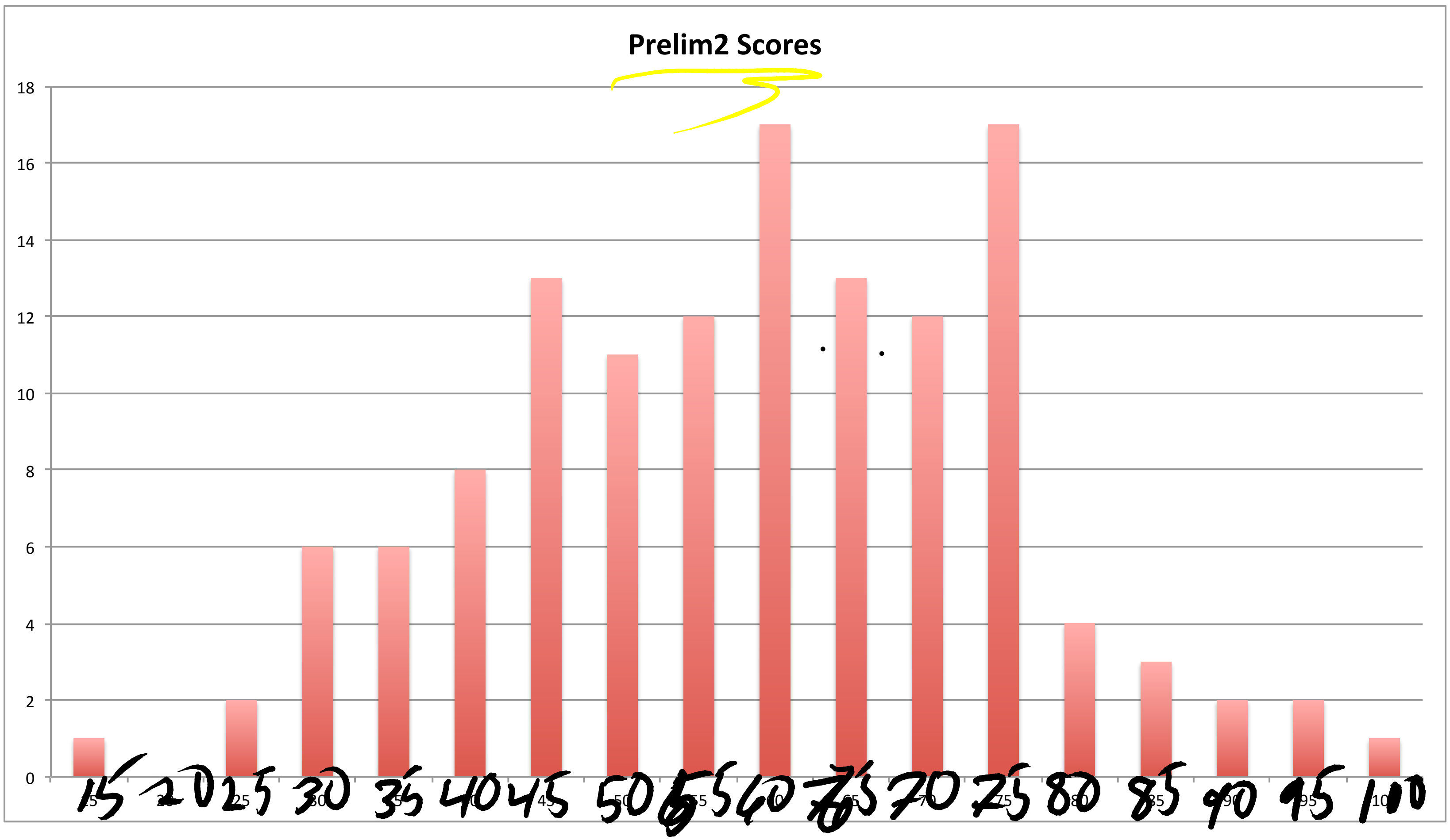
- Work in pairs
- ***Will not be able to use slip days***

- Need to schedule time for presentation May 16, 17, or 18
- Signup *today after class (in front)*

Announcements

Prelim2 results

- Mean 56.4 ± 16.3 (median 57.8), Max 95.5
- Pickup in Homework pass back room (Upson 360)



Goals for Today

Finish Synchronization

- Threads and processes
- Critical sections, race conditions, and mutexes
- Atomic Instructions
  - HW support for synchronization
  - Using sync primitives to build concurrency-safe data structures
  - Cache coherency causes problems
  - Locks + barriers
  - Language level synchronization

Mutexes

Q: How to implement critical section in code?

A: Lots of approaches....

Mutual Exclusion Lock (mutex)

lock(m): wait till it becomes free, then lock it

unlock(m): unlock it

```
safe_increment() {
    pthread_mutex_lock(m);
    hits = hits + 1;
    pthread_mutex_unlock(m)
}
```

lock(m)
lw $1, addr(hit)
Addi $1 $1 1
sw $1, addr(hit)
unlock(m)

Synchronization

Synchronization techniques

clever code

- must work despite adversarial scheduler/interrupts
- used by: hackers
- also: noobs

disable interrupts

- used by: exception handler, scheduler, device drivers, …

disable preemption

- dangerous for user code, but okay for some kernel code

mutual exclusion locks (mutex)

- general purpose, except for some interrupt-related cases

Hardware Support for Synchronization

Atomic Test and Set

Mutex implementation

- Suppose hardware has atomic test-and-set

Hardware atomic equivalent of…

```
int test_and_set(int *m) {
    old = *m;
    *m = 1;
    return old;
}
```

Using test-and-set for mutual exclusion

Use test-and-set to implement mutex / spinlock / crit. sec.

```
int m = 0;
...

while (test_and_set(&m)) { /* skip */ };



m = 0;
```

Spin waiting

Also called: spinlock, busy waiting, spin waiting, …

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

Possible heuristic:

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

Alternative Atomic Instructions

Other atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)

very expensive

rd-modify-wr cycle

Alternative Atomic Instructions

Other atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and exchange (x86, ARM deprecated)
- linked load / store conditional
  (MIPS, ARM, PowerPC, DEC Alpha, …)

mutex from LL and SC

Linked load / Store Conditional

```
mutex_lock(int *m) {
again:
  LL t0, 0(a0)
  BNE t0, zero, again
  ADDI t0, t0, 1
  SC t0, 0(a0)
  BEQ t0, zero, again
}
```

Using synchronization primitives to build concurrency-safe datastructures

Broken invariants

Access to shared data must be synchronized

- goal: enforce datastructure invariants

```
// invariant:
// data is in A[h … t-1]
char A[100];
int h = 0, t = 0;
```



```
// writer: add to list tail
void put(char c) {
  A[t] = c;
  t++;
}
```

```
// reader: take from list head
char get() {
  while (h == t) { };
  char c = A[h];
  h++;
  return c;
}
```

Protecting an invariant

```
// invariant: (protected by m)
// data is in A[h … t-1]
pthread_mutex_t *m = pthread_mutex_create();
char A[100];
int h = 0, t = 0;
```

issues? can't wait while holding lock

```
// writer: add to list tail
void put(char c) {
  pthread_mutex_lock(m);
  A[t] = c;
  t++;
  pthread_mutex_unlock(m);
}
```

t=(t+1)%n

```
// reader: take from list head
char get() {
  pthread_mutex_lock(m);
  char c = A[h];
  h++;
  pthread_mutex_unlock(m);
  return c;
}
```

h=(h+1)%n

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

Guidelines for successful mutexing

Insufficient locking can cause races

- Skimping on mutexes? Just say no!

Poorly designed locking can cause deadlock

```
P1: lock(m1);   P2: lock(m2);
    lock(m2);       lock(m1);
```

- know why you are using mutexes!
- acquire locks in a consistent order to avoid cycles
- use lock/unlock like braces (match them lexically)
  - lock(&m); …; unlock(&m)
  - watch out for return, goto, and function calls!
  - watch out for exception/error conditions!

Cache Coherency

causes yet more trouble

Remember: Cache Coherence

Recall: Cache coherence defined...

Informal: Reads return most recently written value

Formal: For concurrent processes P_1 and P_2

- P writes X before P reads X (with no intervening writes)
  $\Rightarrow$ read returns written value
- P_1 writes X before P_2 reads X
  $\Rightarrow$ read returns written value
- P_1 writes X and P_2 writes X
  $\Rightarrow$ all processors see writes in the same order
  - all see the same final value for X

Relaxed consistency implications

Ideal case: sequential consistency

- Globally: writes appear in interleaved order
- Locally: other core's writes show up in program order

In practice: not so much…

- write-back caches → sequential consistency is tricky
- writes appear in semi-random order
- locks alone don't help

* MIPS has sequential consistency; Intel does not

Acquire/release

Memory Barriers and Release Consistency

- Less strict than sequential consistency; easier to build

One protocol:

- Acquire: lock, and force subsequent accesses after
- Release: unlock, and force previous accesses before

```
P1: ...                    P2: ...
    acquire(m);                acquire(m);
    A[t] = c;                  A[t] = c;
    t++;                       t++;
    release(m);                unlock(m);
```

Moral: can't rely on sequential consistency
(so use synchronization libraries)

Are Locks + Barriers enough?

Beyond mutexes

Writers must check for full buffer
  & Readers must check if for empty buffer

- ideal: don’t busy wait… go to sleep instead

```
char get() {
  acquire(L);
  char c = A[h];
  h++;
  release(L);
  return c;
}
```

while (empty) wait;

Beyond mutexes

Writers must check for full buffer
& Readers must check if for empty buffer

- ideal: don’t busy wait… go to sleep instead

```
char get() {
  while (h == t) { };
  acquire(L);
  char c = A[h];
  h++;
  release(L);
  return c;
}
```

double check

can't wait while holding lock (empty)

Have to check while holding lock, but cannot wait while holding lock

Beyond mutexes

Writers must check for full buffer

& Readers must check if for empty buffer

- ideal: don’t busy wait… go to sleep instead

```
char get() {
  acquire(L);
  while (h == t) { };
  char c = A[h];
  h++;
  release(L);
  return c;
}
```

Beyond mutexes

Writers must check for full buffer

& Readers must check if for empty buffer

- ideal

```
char get() {
  do {
      acquire(L);
      empty = (h == t);
      if (!empty) {
           c = A[h];
           h++;
      }
      release(L);
  } while (empty);
  return c;
}
```

Language-level Synchronization

Condition variables

Use [Hoare] a condition variable to wait for a condition to become true (without holding lock!)

wait(m, c) :

- atomically release m and sleep, waiting for condition c
- wake up holding m sometime after c was signaled

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 until c, wake up holding m

signal(c) : wake up one thread waiting on c

```
cond_t *not_full = ...;
cond_t *not_empty = ...;
mutex_t *m = ...;

void put(char c) {
  lock(m);
  while ((t-h) % n == 1)
    wait(m, not_full);
  A[t] = c;
  t = (t+1) % n;
  unlock(m);
  signal(not_empty);
}
```

```
char get() {
  lock(m);
  while (t == h)
    wait(m, not_empty);
  char c = A[h];
  h = (h+1) % n;
  unlock(m);
  signal(not_full);
  return c;
}
```

Using a condition variable

wait(m, c) : release m, sleep until c, wake up holding m

signal(c) : wake up one thread waiting on c

```
cond_t *not_full = ...;
cond_t *not_empty = ...;
mutex_t *m = ...;

void put(char c) {
  lock(m);
  while ((t-h) % n == 1)
    wait(m, not_full);
  A[t] = c;
  t = (t+1) % n;
  unlock(m);
  signal(not_empty);
}
```

```
char get() {
  lock(m);
  while (t == h)
    wait(m, not_empty);
  char c = A[h];
  h = (h+1) % n;
  unlock(m);
  signal(not_full);
  return c;
}
```

Monitors

A Monitor is a concurrency-safe datastructure, with…

- one mutex
- some condition variables
- some operations

All operations on monitor acquire/release mutex

- one thread in the monitor at a time

Ring buffer was a monitor

Java, C#, etc., have built-in support for monitors

Java concurrency

Java objects can be monitors

- "synchronized" keyword locks/releases the mutex
- Has one (!) builtin condition variable
  - o.wait() = wait(o, o)
  - o.notify() = signal(o)
  - o.notifyAll() = broadcast(o)

- Java wait() can be called even when mutex is not held. Mutex not held when awoken by signal(). Useful?

dangerous

More synchronization mechanisms

Lots of synchronization variations…
(can implement with mutex and condition vars.)

Reader/writer locks

- Any number of threads can hold a read lock
- Only one thread can hold the writer lock

Semaphores

- N threads can hold lock at the same time

Message-passing, sockets, queues, ring buffers, …

- transfer data and synchronize

Summary

Hardware Primitives: test-and-set, LL/SC, barrier, ...

… used to build …

Synchronization primitives: mutex, semaphore, ...

… used to build …

Language Constructs: monitors, signals, ...