Atomic Instructions

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

P&H Chapter 2.11

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

Q: How to implement critical section in code?

A: Lots of approaches....

Mutual Exclusion Lock (mutex)

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

release(m): unlock it

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

Hardware Support for Synchronization

Mutex implementation

- Suppose hardware has atomic test-and-set

Hardware equivalent of…

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

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

```
int lock = 0;
...

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









lock = 0;
```

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

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, …)

Linked load / Store Conditional

```
mutex_lock(int *L) {
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

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;
}
```

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

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

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

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

Insufficient locking can cause races

- Skimping on mutexes? Just say no!

Poorly designed locking can cause deadlock

```
P1: lock(L1);   P2: lock(L2);
    lock(L2);       lock(L1);
```

- 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

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

* MIPS supports this; Intel does not

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

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: ...
    acquire(L);
    A[t] = c;
    t++;
    release(L2);
```

```
P2: ...
    acquire(L);
    A[t] = c;
    t++;
    unlock(L2);
```

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

Are Locks + Barriers enough?

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

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

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

Language-level Synchronization

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 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;
}
```

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 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?

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

Hardware Primitives

… used to build …

Synchronization primitives (mutexes, locks, etc.)

… used to build …

Language constructs (monitors, etc.)