Implementing Mutual Exclusion

Arvind Krishnamurthy
Spring 2004

Disable Interrupts

- Uniprocessor only: an operation will be atomic as long as a context switch does not occur
- Two ways for dispatcher to get control:
  - Internal events – thread does something to relinquish the CPU
  - External events – interrupts cause dispatcher to take CPU away
- Need to prevent both internal and external events
  - Preventing internal events is easy
  - Preventing external events require disabling interrupts
    - Hardware delays handling of external events

Using interrupts (1)

- Flawed but simple:

```
Lock::Acquire()   { disable interrupts; }

Lock::Release()   { enable interrupts; }
```

- Problems:
  - the above may make some user thread never give back CPU
  - critical sections can be arbitrarily long --- it may take too long to respond to an interrupt --- real-time system won't be happy
  - this won't work for other higher-level primitives such as semaphores and condition variables

Using interrupts (2)

Key idea: maintain a lock variable and impose mutual exclusion only on the operations of testing and setting that variable

```
class Lock { int value = FREE; }

Lock::Acquire()  {
   Disable interrupts;
   while (value != FREE) {
      Enable interrupts;
      Disable interrupts;
   }
   value = BUSY;
   Enable interrupts;
}
```

```
Lock::Release()  {
   Disable interrupts;
   value = FREE;
   Enable interrupts;
}
```

Using interrupts (3)

Key idea: Use a queue to maintain a list of threads waiting for the lock. Avoid busy-waiting:

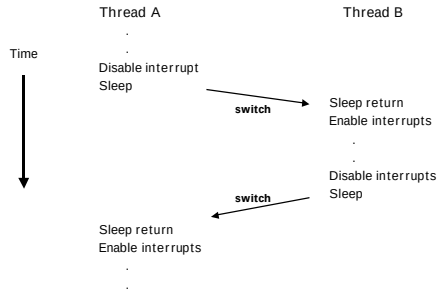
```
class Lock { int value = FREE; }

Lock::Acquire()  {
   Disable interrupts;
   if (value == BUSY) {
      Put on queue of threads waiting for lock;
      Go to sleep;
            // Enable interrupts ? No!
   } else {
      value = BUSY;
   }
   Enable interrupts
}
```

```
Lock::Release()  {
   Disable interrupts;
   if anyone on wait queue {
      Take a waiting thread off wait
      queue and put it at the front
      of the ready queue;
   } else {
      value = FREE;
   }
   Enable interrupts
}
```

When to re-enable interrupts?

- Before putting the thread on the wait queue ?
  - Then Release can check the queue, and not wake the thread up
- After putting the thread on the wait queue but before going to sleep
  - Then Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep!
  - It will go to sleep and miss the wakeup from Release
- In Nachos, interrupts are disabled when you call Thread:Sleep; it is the responsibility of the next thread-to-run to re-enable interrupts.

Interrupt disable/enable pattern



Atomic read-modify-write

- On a multiprocessor, interrupt disable does not provide atomicity
  - other CPUs could still enter the critical section
  - disabling interrupts on all CPUs would be expensive
- Solution: HW provides some special instructions
  - test&set (most arch) --- read value, write 1 back to memory
  - exchange (x86) --- swaps value between register and memory
  - compare&swap (68000) --- read value; if value matches register, do exchange
  - load linked and conditional store (MIPS R4000, Alpha)
    - read value in one instruction, do some operations, when store occurs, check if value has been modified in the meantime. If not, ok; otherwise, abort, and jump back to start.

Locks using test&set (1)

- Flawed but simple:

```
lock value = 0;

Lock::Acquire()   { while (test&set(value) == 1);  }

Lock::Release()   { value = 0;}
```

- Problems:
  - busy-waiting --- thread consumes CPU while it is waiting
    - also known as “Spin” lock
  - could cause problems if threads have different priorities

Locks using test&set (2)

Key idea: only busy-wait to atomically check lock value --- if lock is busy, give up CPU. Use a guard on the lock itself.

```
Lock::Acquire()  {
  while (test&set(guard))  // short wait time
      ;

  if (value == BUSY) {
     Put on queue of threads waiting for lock;
     Go to sleep and set guard to 0
  } else {
     value = BUSY;
     guard = 0;
  }
}
```

```
Lock::Release()  {
   while (test&set(guard))
         ;
   if  anyone on wait queue {
      Take a waiting thread off wait
      queue and put it at the front
      of the ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
}
```

Test-and-Set on Multiprocessors

- Each processor repeatedly executes a test_and_set
- In hardware, it is implemented as:
  - Fetch the old value
  - Write a “1” blindly
- Write in a cached system results in invalidations to other caches
- Simple algorithm results in a lot of bus traffic
- Wrap an extra test (test-and-test-and-set)

```
lock:  if (!location)
           if (!test-and-set(location))
               return;
       goto lock;
```

Ticket Lock for Multiprocessors

- Hardware support: fetch-and-increment
- Obtain a ticket number and wait for your turn

```
Lock:
    next_ticket = fetch_and_increment(next_ticket)
    while (next_ticket != now_serving);

Unlock:
    now_serving++;
```

- Ensures fairness
- Still could result in a lot of bus transactions
- Can be used to build concurrent queues

Array Locks

- Problem with ticket locks: everyone is polling the same location
- Distribute the shared value, and do directed "unlocks"

```
Initialization:
    int slots[numProcs] = {has_lock, must_wait,
                           must_wait, ... };
    next_slot = 0;

Lock:
    my_slot = fetch_and_increment(next_slot);
    my_slot = my_slot % numProcs;
    while (slots[my_slot] == must_wait);
    slots[my_slot] = must_wait;

Unlock:
    slots[(my_slot + 1) % numProcs] = has_lock;
```

Locks Summary

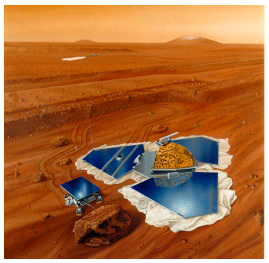
- Supports two operations: acquire and release
  - Is easy to write programs with mutual exclusion
  - Make accesses to shared data be guarded by a lock
- Implementation of locks:
  - Uniprocessor: can use interrupt-enable/disable
    - Solution should not require busy-waiting (or spin-locks)
    - Typically, requires a queue of threads waiting for each lock
  - Multiprocessors:
    - Requires special hardware instructions such as test-and-set
    - Also need to avoid busy-waiting
    - Interesting variants exist that try to minimize bus traffic generated by locks

Busy-waiting

- Busy-waiting wastes processor cycles
  - Might prevent the thread that has the lock from running
  - Scenario:
    - Thread A has lock but gets context-switched out
    - Thread B starts running, decides to acquire lock
    - Spins trying to acquire the lock; prevents thread A from making forward progress (and thus delaying the lock release)
  - If threads package does not use preemption (or if thread B has higher priority), could result in deadlocks
- Non-busy-waiting solutions are better:
  - Thread B just goes to sleep if lock is not available
  - Thread A executes, makes progress, releases lock, and thread B is woken up

Priorities, Locks, Scheduling

- Even without spin-locks, there are subtle interactions between priorities and scheduling and holding locks
- Mars Pathfinder:
  - Success story for the first few days
  - Landed with fancy airbags, released a "rover", shot some spectacular photos of the Mars landscape
  - A few days later after it started collecting meteorological data, system started resetting itself periodically



Priority Inversion

- "Information bus" shared between:
  - **Bus manager** (high priority)
  - Meteorological **data gatherer** (low priority)
  - Reset if **bus manager** hasn't run for a while
  - Protected by a lock
  - If **bus manager** is scheduled (by context-switching out the data gatherer), it will sleep for a bit, let the **data gatherer** run, release the lock
- Another thread: communications task
  - medium priority, long-running task
  - Sometimes the **communications task** would get scheduled instead of the **data gatherer** → neither the lower priority data gatherer nor the higher priority **bus manager** would run
- Works in pairs, but not all three together; resulted in periodic resets