COS 318: Operating Systems

Mutex Implementation

Kai Li
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)
Today’s Topics

- Disabling Interrupts for mutual exclusion
- Hardware support for mutual exclusion
- Competitive spinning
Revisit Mutual Exclusion (Mutex)

- **Critical section**
  ```
  Acquire(lock);
  if (noMilk)
      buy milk;
  Release(lock);
  ```

- **Conditions of a good solution**
  - Only one process/thread inside a critical section
  - No assumption about CPU speeds
  - A process/thread inside a critical section should not be blocked by any processes/threads outside the critical section
  - No one waits forever
  - Works for multiprocessors
  - Same code for all processes/threads
Use and Disable Interrupts

**Use interrupts**
- Implement preemptive CPU scheduling
- Internal events to relinquish the CPU
- External events to reschedule the CPU

**Disable interrupts**
- Introduce uninterruptible code regions
- Think sequentially most of the time
- **Delay** handling of external events

**Diagram:**
- CPU
- Memory
- Interrupt

**Code Snippet:**
```
DisableInt();
Delay;
EnableInt();
```
A Simple Way to Use Disabling Interrupts

```c
Acquire() {
    disable interrupts;
}

Release() {
    enable interrupts;
}

◆ Issues with this approach?
```
One More Try

```c
Acquire(lock) {
    disable interrupts;
    while (lock.value != FREE)
        ;
    lock.value = BUSY;
    enable interrupts;
}

Release(lock) {
    disable interrupts;
    lock.value = FREE;
    enable interrupts;
}
```

◆ Issues with this approach?
Another Try

Acquire(lock) {
    disable interrupts;
    while (lock.value != FREE){
        enable interrupts;
        disable interrupts;
    }
    lock.value = BUSY;
    enable interrupts;
}

Release(lock) {
    disable interrupts;
    lock.value = FREE;
    enable interrupts;
}

◆ Does this fix the “wait forever” problem?
Acquire(lock) {
    disable interrupts;
    while (lock.value == BUSY)
    {
        enqueue me for lock;
        Yield();
    }
    lock.value = BUSY;
    enable interrupts;
}

Release(lock) {
    disable interrupts;
    if (anyone in queue) {
        dequeue a thread;
        make it ready;
    }
    lock.value = FREE;
    enable interrupts;
}

- Any issues with this approach?
Atomic Memory Load and Store

- Assumed in textbook (e.g. Peterson’s solution)
  ```c
  int turn;
  int interested[N];

  void enter_region(int process)
  {
    int other;

    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while(turn == process && interested[other] == TRUE);
  }
  ```

  - 5 writes and 2 reads
Atomic Read-Modify-Write Instructions

- **LOCK prefix in x86**
  - Make a specific set instructions atomic
  - Together with BTS to implement Test&Set
- **Exchange (xchg, x86 architecture)**
  - Swap register and memory
  - Atomic (even without LOCK)
- **Fetch&Add or Fetch&Op**
  - Atomic instructions for large shared memory multiprocessor systems
- **Load link and conditional store**
  - Read value in one instruction (load link)
  - Do some operations;
  - When store, check if value has been modified. If not, ok; otherwise, jump back to start
A Simple Solution with Test&Set

- Define TAS(lock)
  - If successfully set, return 1;
  - Otherwise, return 0;
- Any issues with the following solution?

```c
Acquire(lock) {
    while (!TAS(lock.value))
        ;
}

Release(lock.value) {
    lock = 0;
}
```
What About This Solution?

Acquire(lock) {
    while (!TAS(lock.guard))
        ;
    if (lock.value) {
        enqueue the thread;
        block and lock.guard = 0;
    } else {
        lock.value = 1;
        lock.guard = 0;
    }
}

Release(lock) {
    while (!TAS(lock.guard))
        ;
    if (anyone in queue) {
        dequeue a thread;
        make it ready;
    } else
        lock.value = 0;
        lock.guard = 0;
}

◆ How long does the “busy wait” take?
Example: Protect a Shared Variable

```
Acquire(lock)
count++;
Release(lock)
```

- **Acquire(mutex) system call**
  - Pushing parameter, sys call # onto stack
  - Generating trap/interrupt to enter kernel
  - Jump to appropriate function in kernel
  - Verify process passed in valid pointer to mutex
  - Minimal spinning
  - Block and unblock process if needed
  - Get the lock

- Executing "count++;"

- Release(mutex) system call
Available Primitives and Operations

- **Test-and-set**
  - Works at either user or kernel

- **System calls for block/unblock**
  - *Block* takes some token and goes to sleep
  - *Unblock* “wakes up” a waiter on token
Block and Unblock System Calls

Block( lock )
- Spin on lock.guard
- Save the context to TCB
- Enqueue TCB to lock.q
- Clear lock.guard
- Call scheduler

Unblock( lock )
- Spin on lock.guard
- Dequeue a TCB from lock.q
- Put TCB in ready queue
- Clear lock.guard

Questions
- Do they work?
- Can we get rid of the spin lock?
Always Block

```c
Acquire(lock) {
    while (!TAS(lock.value))
        Block(lock);
}
```

```c
Release(lock) {
    lock.value = 0;
    Unblock(lock);
}
```

- What are the issues with this approach?
Always Spin

```c
Acquire(lock) {
    while (!TAS(lock.value))
        while (lock.value) ;
}

Release(lock) {
    lock.value = 0;
}
```

- Two spinning loops in `Acquire()`?

![Diagram showing CPU, L1, L2, TAS, Memory, and SMP in multicore and SMP architectures.](image)

17
Optimal Algorithms

- What is the optimal solution to spin vs. block?
  - Know the future
  - Exactly when to spin and when to block

- But, we don’t know the future
  - There is **no** online optimal algorithm

- Offline optimal algorithm
  - Afterwards, derive exactly when to block or spin (“what if”)
  - Useful to compare against online algorithms
Competitive Algorithms

- An algorithm is $c$-competitive if for every input sequence $\sigma$

\[ C_A(\sigma) \leq c \times C_{opt}(\sigma) + k \]

- $c$ is a constant
- $C_A(\sigma)$ is the cost incurred by algorithm A in processing $\sigma$
- $C_{opt}(\sigma)$ is the cost incurred by the optimal algorithm in processing $\sigma$

- What we want is to have $c$ as small as possible
  - Deterministic
  - Randomized
Constant Competitive Algorithms

Acquire(lock, N) {
    int i;

    while (!TAS(lock.value)) {
        i = N;
        while (!lock.value && i)
            i--;

        if (!i)
            Block(lock);
    }
}

- Spin up to N times if the lock is held by another thread
- If the lock is still held after spinning N times, block

- If spinning N times is equal to the context-switch time, what is the competitive factor of the algorithm?
Approximate Optimal Online Algorithms

◆ Main idea
  ● Use past to predict future

◆ Approach
  ● Random walk
    • Decrement N by a unit if the last Acquire() blocked
    • Increment N by a unit if the last Acquire() didn’t block
  ● Recompute N each time for each Acquire() based on some lock-waiting distribution for each lock

◆ Theoretical results
  \[ E \ C_A(\sigma(P)) \leq \frac{e}{(e-1)} \times E \ C_{opt}(\sigma(P)) \]

The competitive factor is about 1.58.
Empirical Results


### Table 1: Synchronization costs for each program relative to the optimal off-line algorithm

<table>
<thead>
<tr>
<th></th>
<th>Block</th>
<th>Spin</th>
<th>Fixed C/2</th>
<th>Fixed C</th>
<th>Opt</th>
<th>Online</th>
<th>3-samples</th>
<th>R-walk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nub (2h)</td>
<td>1.943</td>
<td>2.962</td>
<td>1.503</td>
<td>1.550</td>
<td>1.078</td>
<td>1.225</td>
<td>1.093</td>
<td></td>
</tr>
<tr>
<td>Taos (24h)</td>
<td>1.715</td>
<td>3.366</td>
<td>1.492</td>
<td>1.757</td>
<td>1.141</td>
<td>1.212</td>
<td>1.213</td>
<td></td>
</tr>
<tr>
<td>Taos (M2+)</td>
<td>1.776</td>
<td>3.535</td>
<td>1.483</td>
<td>1.750</td>
<td>1.106</td>
<td>1.177</td>
<td>1.160</td>
<td></td>
</tr>
<tr>
<td>Taos (Regsim)</td>
<td>1.578</td>
<td>3.293</td>
<td>1.499</td>
<td>1.748</td>
<td>1.161</td>
<td>1.260</td>
<td>1.268</td>
<td></td>
</tr>
<tr>
<td>Ivy (100m)</td>
<td>5.171</td>
<td>2.298</td>
<td>1.341</td>
<td>1.438</td>
<td>1.133</td>
<td>1.212</td>
<td>1.167</td>
<td></td>
</tr>
<tr>
<td>Ivy (18h)</td>
<td>7.243</td>
<td>1.562</td>
<td>1.274</td>
<td>1.233</td>
<td>1.109</td>
<td>1.233</td>
<td>1.141</td>
<td></td>
</tr>
<tr>
<td>Galaxy</td>
<td>2.897</td>
<td>2.667</td>
<td>1.419</td>
<td>1.740</td>
<td>1.237</td>
<td>1.390</td>
<td>1.633</td>
<td></td>
</tr>
<tr>
<td>Hanoi</td>
<td>2.997</td>
<td>2.976</td>
<td>1.418</td>
<td>1.726</td>
<td>1.200</td>
<td>1.366</td>
<td>1.642</td>
<td></td>
</tr>
<tr>
<td>Regsim</td>
<td>4.675</td>
<td>1.302</td>
<td>1.423</td>
<td>1.374</td>
<td>1.183</td>
<td>1.393</td>
<td>1.366</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Elapsed times of Regsim using different spinning strategies.
# The Big Picture

<table>
<thead>
<tr>
<th></th>
<th>OS codes and concurrent applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-Level Atomic API</td>
<td>Mutex, Semaphores, Monitors, Send/Recv</td>
</tr>
<tr>
<td>Low-Level Atomic Ops</td>
<td>Load/store, Interrupt disable/enable, Test&amp;Set, Other atomic instructions</td>
</tr>
<tr>
<td></td>
<td>Interrupts (I/O, timer), Multiprocessors, CPU scheduling</td>
</tr>
</tbody>
</table>
Summary

- **Disabling interrupts for mutex**
  - There are many issues
  - When making it work, it works for only uniprocessors

- **Atomic instruction support for mutex**
  - Atomic load and stores are not good enough
  - Test&set and other instructions are the way to go

- **Competitive spinning**
  - Spin at the user level most of the time
  - Make no system calls in the absence of contention
  - Have more threads than processors