COS 318: Operating Systems

Mutex Implementation

Kai Li and Andy Bavier
Computer Science Department
Princeton University

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

- Mutex problem
- Interrupts for mutex
- Critical section with atomic reads and writes
- Mutex with atomic test-and-set instructions
- Spin locks
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
Simple Lock Variables

Acquire(lock) {
while (lock.value == 1)
    ;
lock.value = 1;
}

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

Thread 1:
Acquire(lock) {
while (lock.value == 1)
    ;
    {context switch}
lock.value = 1;
}

Thread 2:
    {context switch}
    Acquire(lock) {
        while (lock.value == 1)
            ;
    {context switch}
lock.value = 1;
Interrupt in A Simplified System

I/O devices can raise interrupts to CPU
Instruction Execution with Interrupt

**User Program**

- ld
- add
- st
- mul
- ld
- sub
- bne
- add
- jmp
- ...

**IP**

1. Fetch
2. Decode
3. Execute
4. Advance IP
5. Interrupt?

- no
- yes

**Interrupt handler**

```plaintext
{   ...
    iret
}
```

**Most instructions can be interrupted in the middle**
Exceptions: Similar to Interrupts

- Interrupts are asynchronous
  - From external sources
  - Examples: alarm clock, I/O bus signals from devices

- Exceptions are synchronous (more later)
  - Processor-detected exceptions
    - Faults — correctable; offending instruction is *retried*
    - Traps — often for debugging; instruction is *not* retried
    - Aborts — errors when executing instructions
  - Programmed exceptions
    - Requests for kernel intervention (intr/sysevents)
Use and Disable Interrupts

- **Use interrupts**
  - Process I/O requests (e.g. keyboard)
  - Implement preemptive CPU scheduling

- **Disable interrupts**
  - Introduce uninterruptible code regions
  - Think sequentially most of the time
  - **Delay** handling of external events
Disabling Interrupts for Critical Section?

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

Release() {
    enable interrupts;
}
```

### Issues

- Kernel cannot let users disable interrupts
- Critical sections can be arbitrarily long
- Used on uniprocessors, but does not work on multiprocessors
“Disable Interrupts” to Implement Mutex

Acquire(lock) {
    disable interrupts;
    while (lock.value != 0)
        ;
    lock.value = 1;
    enable interrupts;
}

Release(lock) {
    disable interrupts;
    lock.value = 0;
    enable interrupts;
}

◆ Issues
  ● May wait forever
  ● Not designed for user code to use
Fix “Wait Forever” problem?

```
Acquire(lock) {
    disable interrupts;
    while (lock.value != 0){
        enable interrupts;
        disable interrupts;
    }
    lock.value = 1;
    enable interrupts;
}
```

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

Issues

- Consume CPU cycles
- Won’t work with multiprocessors
Another Implementation

Acquire(lock) {
    disable interrupts;
    while (lock.value == 1) {
        Enqueue me for lock;
        Yield();
    }
    lock.value = 1;
    enable interrupts;
}

Release(lock) {
    disable interrupts;
    if (anyone in queue) {
        Dequeue a thread;
        make it ready;
    }
    lock.value = 0;
    enable interrupts;
}

Questions

• Would this work for multiprocessors?
Critical Section with Atomic Reads/Writes

- Peterson’s solution (see textbook pp. 123)
  ```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.value = 0;
}
```
Mutex with Less Waiting?

```c
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

```c
Acquire(lock);    /* system call */
count++;
Release(lock)     /* system call */
```

- **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
Always Block

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

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

- **Good**
  - Acquire won’t make a system call if TAS succeeds

- **Bad**
  - TAS instruction locks the memory bus
  - Block/Unblock still has substantial overhead
Always Spin

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

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

- Two spinning loops in `Acquire()`?

![Diagram showing the structure of Multicore and SMP systems with TAS and Memory components.]
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

```c
void 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 \( \text{Acquire}() \) blocked
    • Increment \( N \) by a unit if the last \( \text{Acquire}() \) didn’t block
  ● Recompute \( N \) each time for each \( \text{Acquire}() \) based on some lock-waiting distribution for each lock

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

  The competitive factor is about 1.58.
Empirical Results

<table>
<thead>
<tr>
<th></th>
<th>Block</th>
<th>Spin</th>
<th>Fixed C/2</th>
<th>Fixed C</th>
<th>Opt 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>1.559</td>
<td>1.078</td>
<td>1.225</td>
<td>1.093</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>
</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>
</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>
</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>
</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>
</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>1.693</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>
</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>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Max spins</th>
<th>Elapsed time (seconds)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always-block</td>
<td>N/A</td>
<td>10529.5</td>
<td>0.0%</td>
</tr>
<tr>
<td>Always-spin</td>
<td>N/A</td>
<td>8256.3</td>
<td>21.5%</td>
</tr>
<tr>
<td>Fixed-spin</td>
<td>100</td>
<td>9108.0</td>
<td>13.5%</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>8000.0</td>
<td>24.0%</td>
</tr>
<tr>
<td>Opt-known</td>
<td>1008</td>
<td>7881.4</td>
<td>25.1%</td>
</tr>
<tr>
<td>Opt-approx</td>
<td>1008</td>
<td>8171.2</td>
<td>22.3%</td>
</tr>
<tr>
<td>3-samples</td>
<td>1008</td>
<td>8011.6</td>
<td>23.9%</td>
</tr>
<tr>
<td>Random-walk</td>
<td>1008</td>
<td>7929.7</td>
<td>24.7%</td>
</tr>
</tbody>
</table>

Table 3: Elapsed times of Regsim using different spinning strategies.

# The Big Picture

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