COS 318: Operating Systems

Mutex Implementation

(http://www.cs.princeton.edu/courses/cos318/)

Today's Topics

- Disabling Interrupts for mutual exclusion
- Hardware support for mutual exclusion
- Competitive spinning

The Big Picture

	OS codes and concurrent applications			
High-Level Atomic API	Mutex	Semaphores	Monitors	Send/Recv
Low-Level Atomic Ops	Load/store	Interrupt disable/enable	Test&Set	Other atomic instructions
	Interrupts (I/O, timer)	Multiprocessors	CPU scheduling	

Revisit Mutual Exclusion (Mutex)

- Critical section

```
Acquire(lock);
if (noMilk)
  buy milk;
Release(lock);
```

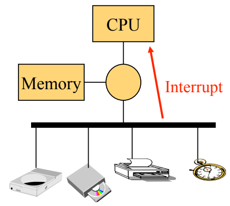
} **Critical section**

- 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

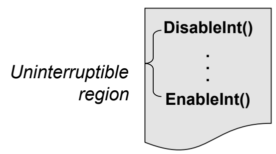
  - Works for multiprocessors
  - Same code for all processes/threads

Mutual Exclusion via Disabling Interrupts

- Use interrupts
  - Implement preemptive CPU scheduling
  - Provide mutual exclusion by preventing context switch between acquire and release
  - Two types of events can cause switches:
    - Internal events to relinquish the CPU
    - External events to reschedule the CPU



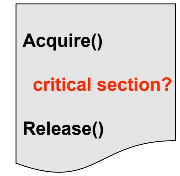
- Disable interrupts to prevent external events
  - Introduce uninterruptible code regions
  - Think sequentially most of the time
  - **Delay** handling of external events



A Simple Way to Use Disabling Interrupts

```
Acquire() {
    disable interrupts;
}

Release() {
    enable interrupts;
}
```



- Issues with this approach?

Another Try

- Use a lock variable

```
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?
- Why disable interrupts at all?

Another Try

- Use a lock variable, and impose mutual exclusion via interrupts only on testing and setting that variable

```
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?

Once more, with queuing …

```
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;
}
```

- What's going on here?
- When should acquirer re-enable interrupts?
  - Just before enqueue?
  - Just after enqueue and before Yield()?

Atomic Read-Modify-Write Instructions

- Test&Set
  - Read location, set its value to 1, return value read
- 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 linked and conditional store
  - Read value in one instruction (load linked)
  - Do some operations;
  - When store, check if value has been modified since load linked. 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?

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

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


What About This Solution?

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

```
Release(lock) {
  while (TAS(lock.guard)==1)
    ;
  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

13

Available Primitives and Operations

- Test-and-set
  - Works at either user or kernel level
- System calls for block/unblock
  - **Block** takes some token and goes to sleep
  - **Unblock** "wakes up" a waiter on token

14

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

15

Always Block

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

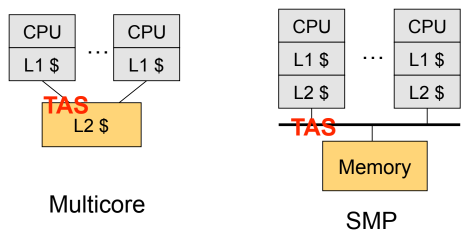
```
Release(lock) {
  lock.value = 0;
  Unblock( lock );
}
```

- What are the issues with this approach?

Always Spin

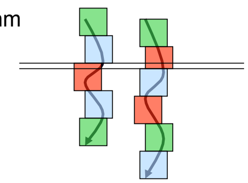
```
Acquire(lock) {                       Release(lock) {
  while (TAS(lock.value)==1)            lock.value = 0;
    while (lock.value)                }
    ;
}
```

- Two spinning loops in **`Acquire()`**?



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 σ

$$C_A(\sigma) \le c \times C_{opt}(\sigma) + k$$

  - c is a constant
  - $C_A(\sigma)$ is the cost incurred by algorithm A in processing σ
  - $C_{opt}(\sigma)$ is the cost incurred by the optimal algorithm in processing σ
- We want have c to be as small as possible
- Deterministic and randomized competitive algorithms

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

  $E\ C_A(\sigma(P)) \le (e/(e-1)) \times E\ C_{opt}(\sigma(P))$

  The competitive factor is about 1.58.

Empirical Results

	Block	Spin	Fixed C/2	Fixed C	Opt Online	3-samples	R-walk
Nub (2h)	1.943	2.962	1.503	1.559	1.078	1.225	1.093
Taos (24h)	1.715	3.366	1.492	1.757	1.141	1.212	1.213
Taos (M2+)	1.776	3.535	1.483	1.750	1.106	1.177	1.160
Taos (Regsim)	1.578	3.293	1.499	1.748	1.161	1.260	1.268
Ivy (100m)	5.171	2.298	1.341	1.438	1.133	1.212	1.167
Ivy (18h)	7.243	1.562	1.274	1.233	1.109	1.233	1.141
Galaxy	2.897	2.667	1.419	1.740	1.237	1.390	1.693
Hanoi	2.997	2.976	1.418	1.726	1.200	1.366	1.642
Regsim	4.675	1.302	1.423	1.374	1.183	1.393	1.366

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

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

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

A. Karlin, K. Li, M. Manasse, and S. Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," Proceedings of the 13th ACM Symposium on Operating Systems Principle, 1991.

The Big Picture

	OS codes and concurrent applications			
High-Level Atomic API	Mutex	Semaphores	Monitors	Send/Recv
Low-Level Atomic Ops	Load/store	Interrupt disable/enable	Test&Set	Other atomic instructions
	Interrupts (I/O, timer)	Multiprocessors	CPU scheduling	

Summary

- Disabling interrupts for mutex
  - There are many issues
  - When made to work, 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