# COS 318: Operating Systems

# Mutex Implementation

Jaswinder Pal Singh
Computer Science Department
Princeton University

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

# Revisit Mutual Exclusion (Mutex)

- Critical section

```
Acquire(lock);
if (noCookies)
   buy cookies;
Release(lock);
```

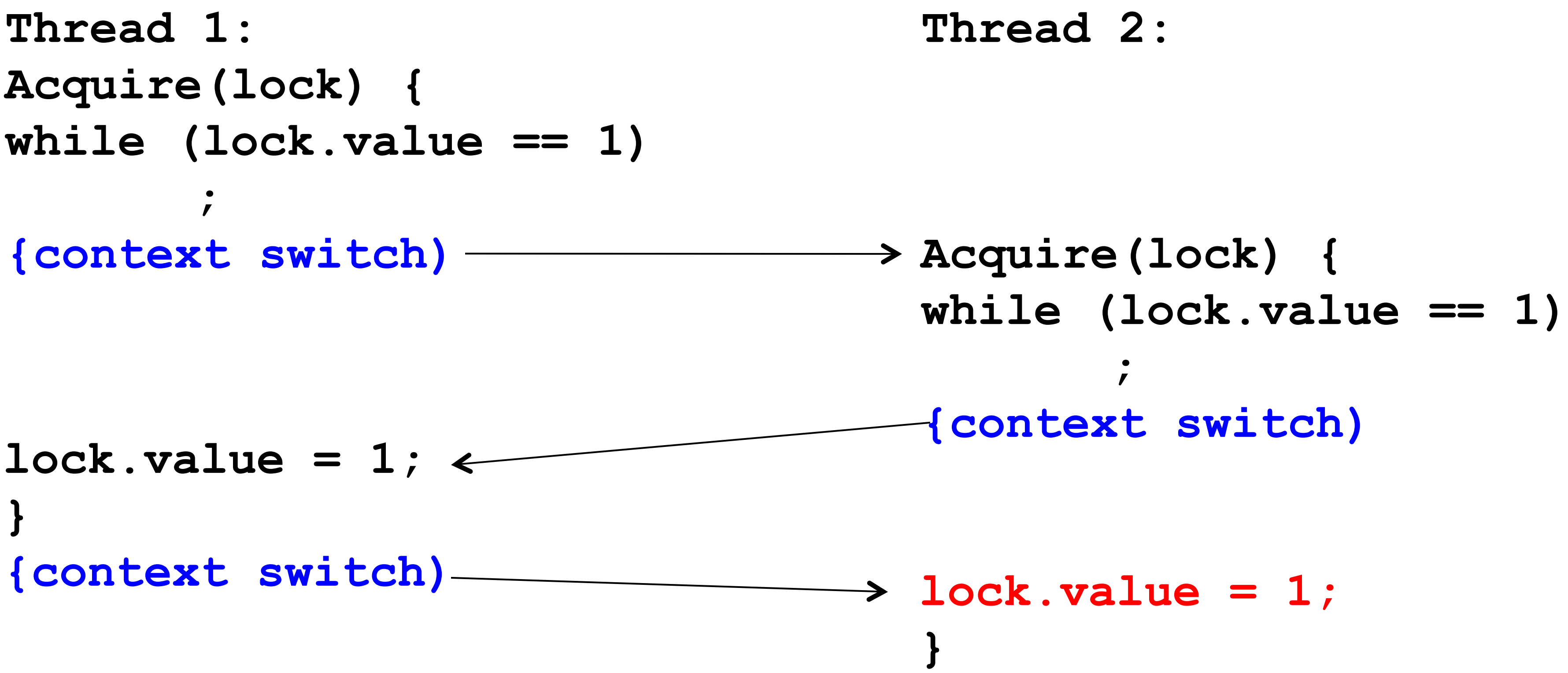
Critical section

- Requirements
  - 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) {                       Release(lock) {
while (lock.value == 1)                  lock.value = 0;
    ;                                 }
lock.value = 1;
}
```



```
Thread 1:                             Thread 2:
Acquire(lock) {
while (lock.value == 1)
       ;
{context switch)                      Acquire(lock) {
                                      while (lock.value == 1)
                                             ;
                                      {context switch)
lock.value = 1;
}
{context switch)                      lock.value = 1;
                                      }
```

# Prevent Context Switches

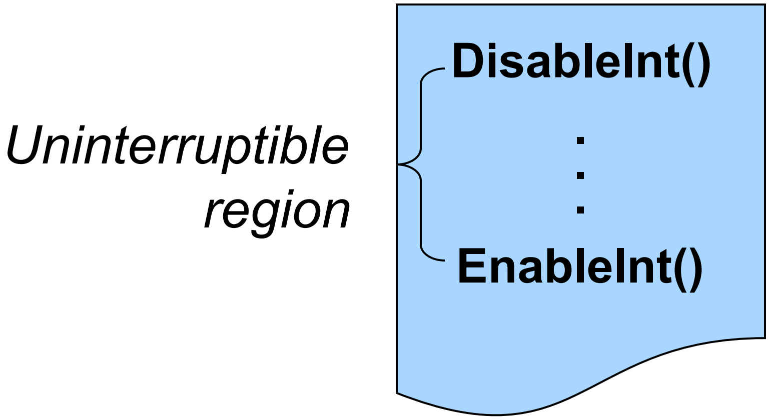
- On a uniprocessor, operations are atomic as long as a context switch doesn’t occur
- Context switches are caused either by actions the thread takes (e.g. traps etc) or by external interrupts
- The former can be controlled

- Disable interrupts during certain portions of code?
  - Delay the handling of external events

# Why Enable or Disable Interrupts

- Interrupts are important
  - Process I/O requests (e.g. keyboard)
  - Implement preemptive CPU scheduling

- Disabling interrupts can be helpful
  - Introduce uninterruptible code regions
  - Think sequentially most of the time
  - **Delay** handling of external events



*Uninterruptible region*

**DisableInt()**

:

**EnableInt()**

# Disabling Interrupts for Critical Section?

`Acquire():` disable interrupts

`Release():` enable interrupts

**Acquire()**

**critical section?**

**Release()**

Issues:

- Kernel cannot let users disable interrupts
- Critical sections can be arbitrarily long
- Works 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 disable interrupts forever
  - Not designed for user code to use

# Fix “Disable 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;
}
```

Disable interrupts only when accessing lock.value variable

Issues:

- Consume CPU cycles
- Won’t work with multiprocessors (like all attempts above)

# Another Implementation

```
Acquire(lock) {
  disable interrupts;
  if (lock.value != 0)
  {
    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;
}
```

Avoid busy-waiting

Issues

- Working for multiprocessors

# Peterson’s Algorithm

- See textbook

```
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);
}
```

- *L. Lamport, “A Fast Mutual Exclusion Algorithm,” ACM Trans. on Computer Systems, 5(1):1-11, Feb 1987.*
  - 5 writes and 2 reads

# Atomic Operations

- A thread executing an atomic instruction can’t be preempted or interrupted while it’s doing it
- Atomic operations on same memory value are serialized
  - ***Even on multiprocessors!***
  - Result is consistent with some sequential ordering of operations
  - Without atomic ops, simultaneous writes by different threads may produce a garbage value, or read that happens simultaneously with a write may read garbage value
- Don’t usually require special privileges, can be user level

# 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 linked and store conditional (LL-SC)
  - Two separate instructions (LL, SC) that are used together
  - Read value in one instruction (load linked)
    Do some operations;
  - When time to 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 (wasn't already set when tested but this operation set it), return 1;
  - Otherwise, return 0;
- Any issues with the following solution?

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

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

# Mutex with Less Waiting?

```
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;
}
```

- Separate access to lock variable from value of it

# Example: Protect a Shared Variable

```
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
- Execute “**count++;**”
- Release(mutex) system call

# 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

# 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 );
}
```

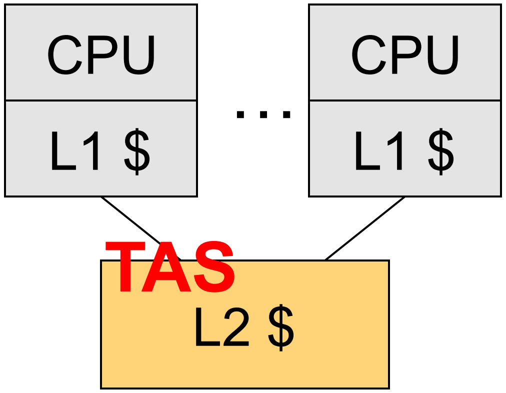
- Good
  - Acquire won’t make a system call if TAS succeeds
- Bad
  - TAS instruction locks the memory bus
  - Block/Unblock still has substantial overhead

# Always Spin

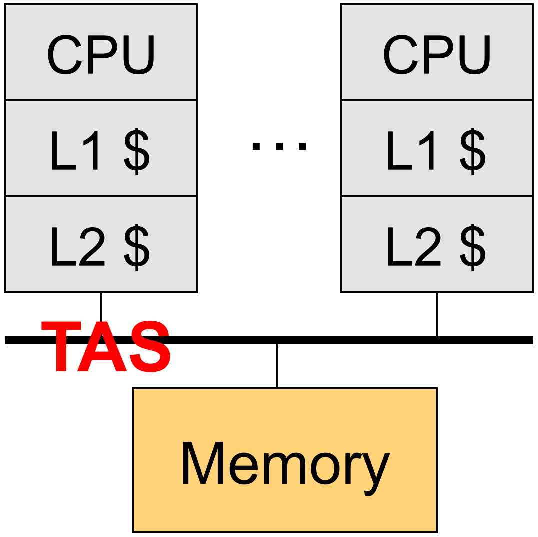
```
Acquire(lock) {
  while (!TAS(lock.value))
    while (lock.value)
    ;
}
```

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

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



Multicore



SMP

# 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

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

The competitive factor is about 1.58.

# 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 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