# COS 318: Operating Systems

# Synchronization: Mutual Exclusion

## “Too Many Cookies” Problem

- Roommates Sam and Jianan want a bag of cookies in the room at all times, but don’t want to buy too many cookies

- They buy cookies independently, using the following sequence
  - Look in cabinet: Out of cookies
  - Leave for Wawa to buy cookies
  - Arrive at Wawa
  - Buy a bag of cookies
  - Arrive home and put cookies in cabinet

# “Too Many Cookies” Problem

| | Sam | Jianan |
|---|---|---|
| 15:00 | Look in cabinet: out of cookies | |
| 15:05 | Leave for Wawa | |
| 15:10 | Arrive at Wawa | Look in cabinet: out of cookies |
| 15:15 | Buy a bag of cookies | Leave for Wawa |
| 15:20 | Arrive home; put cookies away | Arrive at Wawa |
| 15:25 | | Buy a bag of cookies |
| | | Arrive home; put cookies away |

- Oh No! Too many cookies.

## Using A Note?

**Sam and Jianan's Cookie Optimization Algorithm:**

```
if (noCookies) {  // check if roommate left a note
   if (noNote) {
      leave note;   // let them know you went to Wawa
      buy cookies;
      remove note;
   }
}
```

- Q: Any issue with this approach of using a note?

## Using A Note?

**Sam**

```
if (noCookies) {
   if (noNote) {
      leave note;
      buy cookies;
      remove note;
   }
}
```

**Jianan**

```
if (noCookies) {
   if (noNote) {
      leave note;
      buy cookies;
      remove note;
   }
}
```

- Any issue with this approach?

# Why Solution #1 Does Not Work

| | **Sam** | **Jianan** |
|---|---|---|
| 3:00 | | if (noCookies) { |
| 3:05 | | if (noNote) { |
| 3:10 | if (noCookies) { | |
| 3:15 | if (noNote) { | |
| 3:20 | leave Note; | leave Note; |
| 3:25 | buy cookies; | buy cookies; |
| 3:30 | remove Note } } | remove Note} } |

Threads can get context-switched at any time

Too many cookies!

# Possible Solution #2: Leave Note First

**Sam**

```
leave noteA
if (noNoteB) {
 if (noCookies) {
     buy cookies
 }
}
remove noteA
```

**Jianan**

```
leave noteB
if (noNoteA) {
 if(noCookies){
     buy cookies
 }
}
remove noteB
```

Didn't buy cookies

Didn't buy cookies

- Does this method work?

## Possible Solution #3: One Spin-waits

- Problem was that threads checked once and moved on
  - So have one of them spin-wait on the note

**Sam**

```
leave noteA
while (noteB)
  do nothing;
if (noCookies)
  buy cookies;
remove noteA
```

**Jianan**

```
leave noteB
if (noNoteA) {
  if (noCookies) {
    buy cookies
  }
}
remove noteB
```

- Would this fix the problem?
- Yes, but complicated, different code for different threads, busy waiting wasteful, and not fair

# Threads Example: Shared Counter

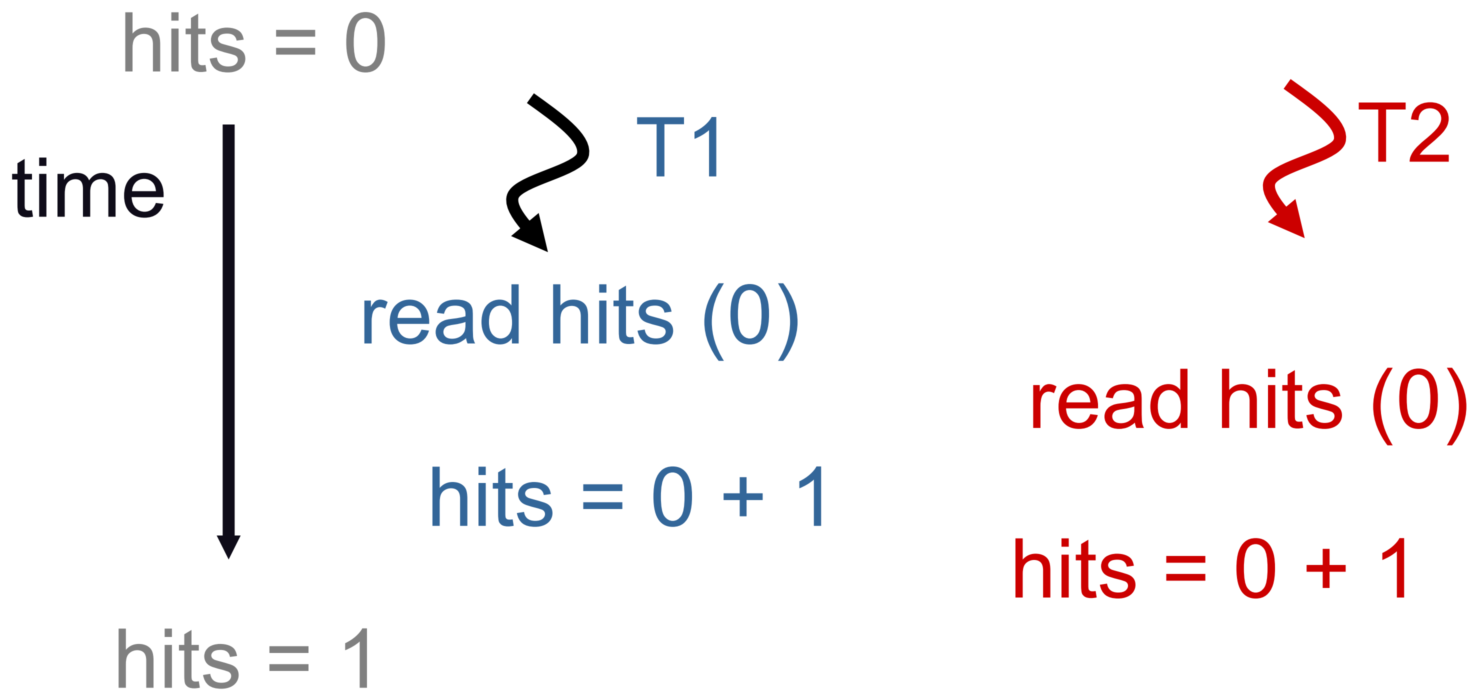
- Google gets millions of hits a day. Uses multiple threads (on multiple processors) to speed things up.
- Simple shared state error: each thread increments a shared counter to track the number of hits today:

```
…
hits = hits + 1;
…
```

- What happens when two threads execute this code concurrently?

# Problem with Shared Counters

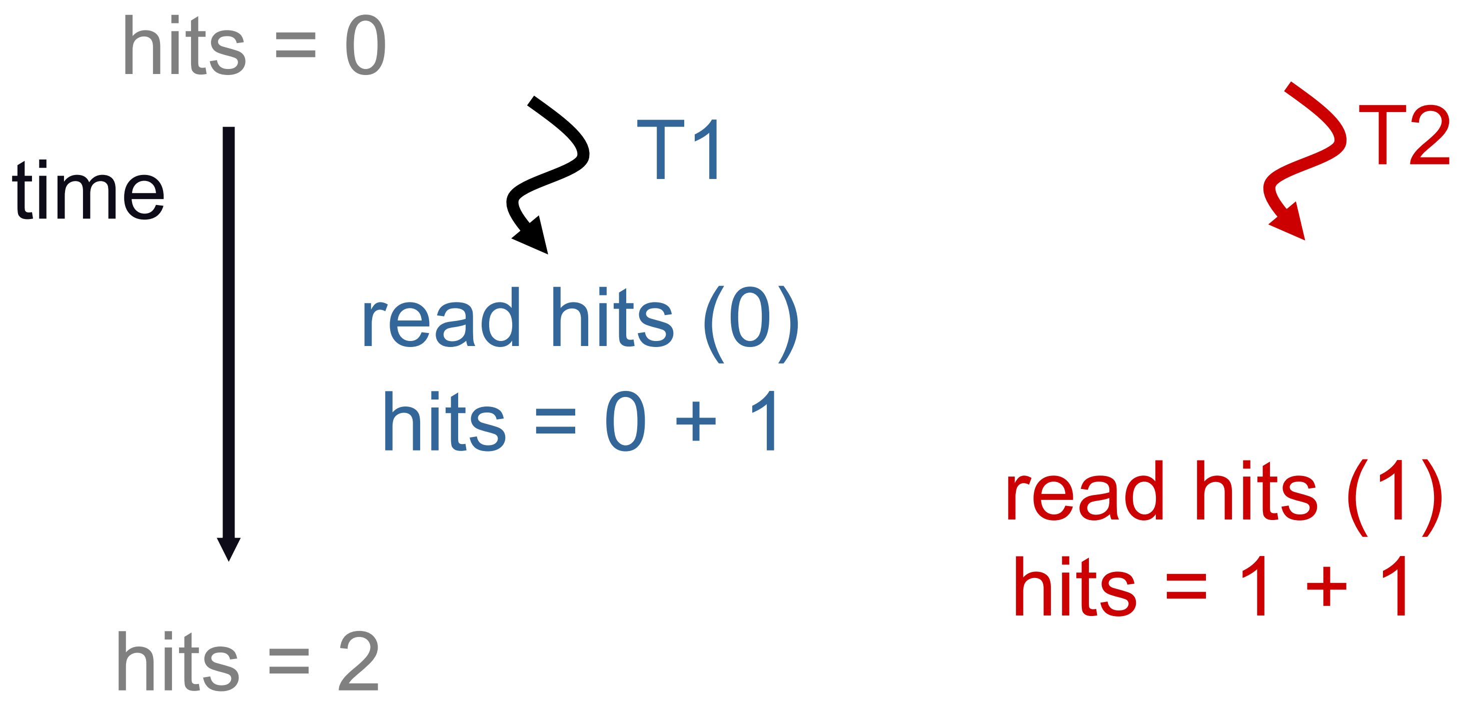
- One possible result: lost update!



- Q: What’s another possible result?

# Problem with Shared Counters

- Another possible result: everything works!



- **Another possible result: everything works**
- This is called a “race condition”

# Race Conditions

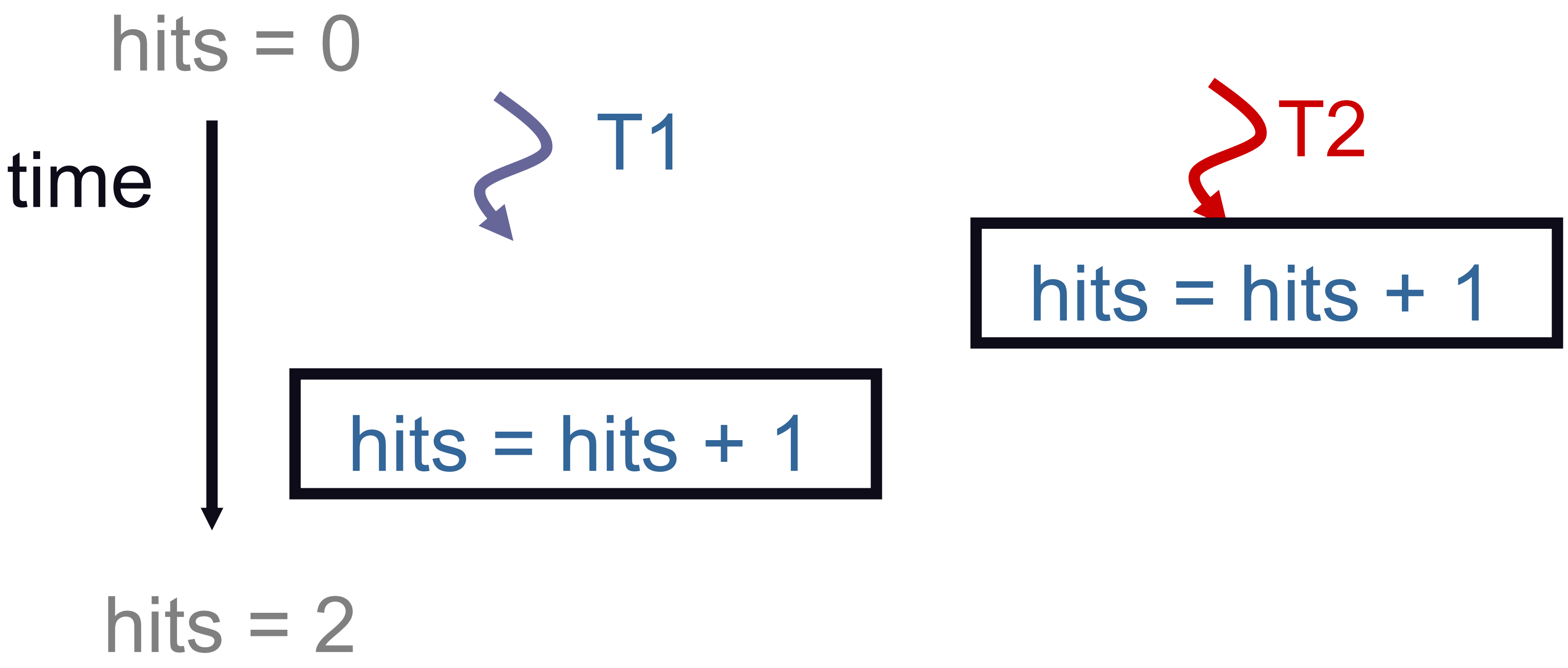
- Race condition: accesses to shared state that can lead to a timing dependent error
  - Whether it happens depends on how threads are scheduled
- Difficult to avoid because:
  - ***Must make sure all possible schedules are safe***.
  - Number of possible schedule permutations is huge.
  - One or more of them may be “bad”
  - They are intermittent
  - Timing dependent => small changes can hide or reveal bug
    - Adding a print statement
    - Running on a different machine

# It’s Actually Even Worse

- Compilers reorder instruction issue within a thread
  - To optimize register usage and hence run code faster
- Hardware reorders instruction execution/completion
  - E.g. write buffers, etc
- All done to optimize execution speed
- But they don’t know about multiple threads and issues across them

## Preventing Race Conditions: Atomicity

- Atomic unit = instruction sequence guaranteed to execute indivisibly (also called a “critical section”).
  - If two threads execute the same atomic unit at the same time, one thread will execute the whole sequence before the other begins



- How to make multiple instrs seem like an atomic one?

## Providing Atomicity

- Have hardware provide better primitives than atomic load and store.
- Build higher-level programming abstractions on this new hardware support.
- Example: locks

**Acquire** --- wait until lock is free, then grab it

**Release** --- unlock/release the lock, waking up a waiter if any

These must be atomic operations --- if two threads are waiting for the lock, and both see it is free, only one grabs it

## Preventing Race Conditions: Atomicity

- Counter problem

$$\left.\begin{array}{l}\texttt{Acquire(lock);}\\ \texttt{hits = hits + 1;}\\ \texttt{Release(lock);}\end{array}\right\}\ \textbf{Critical section}$$

# Preventing Race Conditions: Atomicity

- Cookies problem

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

**Critical section**

Desirable Properties:

1. At most one holder, or thread in critical section, at a time (safety)
2. If no one is holding the lock, an acquire gets the lock (progress)
3. If all lock holders finish and there are no higher priority waiters, waiter eventually gets the lock (progress)

## Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
- Always release after finishing with shared data
  - Only the lock holder can release
- Don’t access shared data without lock

## Some Definitions

- **Synchronization:**
  - Ensuring proper cooperation among threads
  - Mutual exclusion, event synchronization
- **Mutual exclusion:**
  - Ensuring that only one thread does a particular thing at a time. One thread doing it excludes another from doing it at the same time.
- **Event synchronization:**
  - Making sure an event in one thread does not happen before/after an event in another thread

## Some Definitions

- **Critical section:**
  - Piece of code that only one thread can “be in” at a given time. Only one thread at a time will be allowed to get into the section of code.

- **Lock:** prevents someone from doing something
  - Lock before entering critical section, before accessing shared data
  - Unlock when leaving, after done accessing shared data
  - Wait if locked

## Implementing Mutual Exclusion (Locks)

What makes a good solution?

- Only one process/thread inside a critical section at a time
- No assumptions need to be made about CPU speeds
- A process/thread inside a critical section should not be blocked by any process outside the critical section
- No one waits forever

- Should work for multiprocessors
- Should allow same code for all processes/threads

- Interrupts, atomic operations, spinning/blocking, competitive algorithms

# 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;
                                         }
```

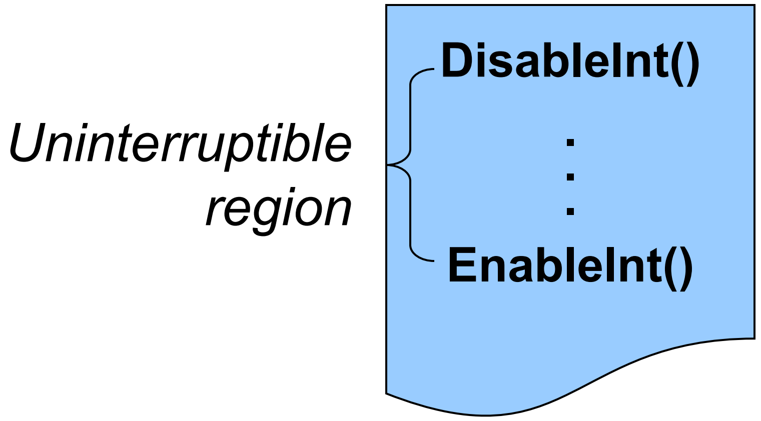
# Prevent Context Switches in Critical Section

- 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

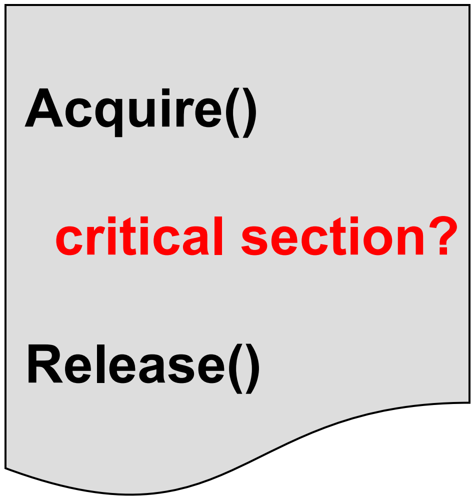
- Interrupts are important
  - Process I/O requests (e.g. keyboard)
  - Implement preemptive CPU scheduling
- Disabling interrupts can be helpful
  - Introduce uninterruptible code regions
  - Think sequentially most of the time
  - **Delay** handling of external events



# Disabling Interrupts for Critical Section?

**`Acquire():`** disable interrupts

**`Release():`** enable interrupts



```
Acquire()

  critical section?

Release()
```

Q: State 2 of the 3 main problems with this approach

## “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;
}
```

- Don’t let acquire be interrupted before sets lock.value to 1
  - This was the problem when interrupts weren’t disabled
- Don’t disable interrupts for entire critical section
- Issues:
  - May disable interrupts forever (while loop)

## 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;
}
```

- Enable interrupts during spin loop
- Disable interrupts only when accessing lock.value
  - Cannot be interrupted after loop and before setting value

Issues:

  - Consume a lot of CPU cycles doing enable and disable

## 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

- Interrupt based approaches don’t work for multiprocessors
- Cannot allow user code to disable interrupts

# Atomic Operations

- A thread executing an atomic instruction can't be preempted or interrupted while it's executing 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

# Peterson's Algorithm

- See textbook

```
int turn;
int interested[N];

void enter_region(int process)
{
    int other;

    other = 1 - process;
    interested[process] = TRUE; /* express interest */
    turn = other;     /* give turn to other process */
    while(turn == process && interested[other] == TRUE);
    /* wait till other loses interest or gives me turn */
}
```

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

# Atomic Read-Modify-Write Instructions

- LOCK prefix in x86
  - Make a specific of set instructions atomic
  - Can be used 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 multiprocessors
- 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

## Always Block

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

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

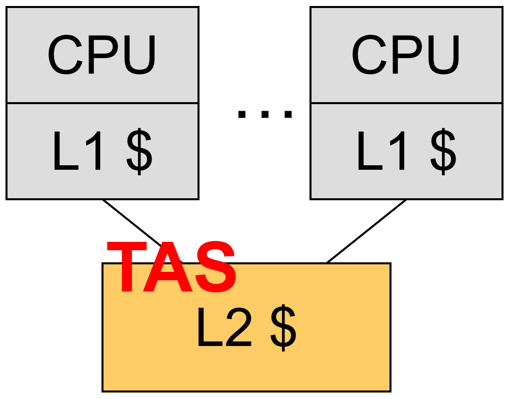
- Good
  - Acquire won’t make a system call if TAS succeeds
- Bad
  - TAS instruction locks the memory bus
  - Block/Unblock still has substantial overhead

# Always Spin

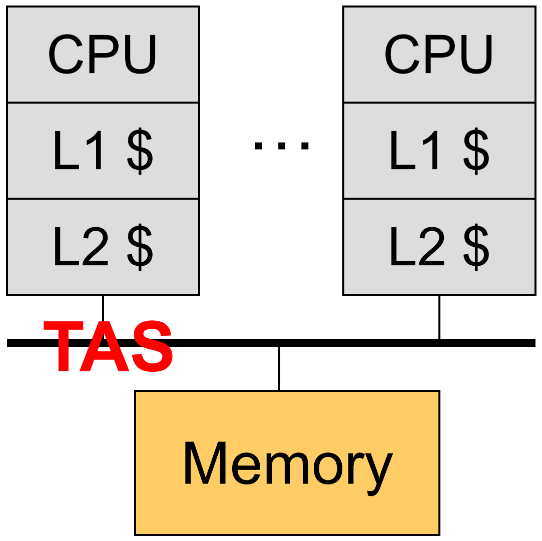
```
Acquire(lock) {
   while (!TAS(lock.value))
      while (lock.value)
      ;
}
```

```
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 $\sigma$

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

  - 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

- Q: 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

Concurrent applications/software

| | | | | |
|---|---|---|---|---|
| Shared Objects | Barrier | Bounded Buffer | | |
| High-Level Atomic API (portable) | Mutex | Semaphores | Monitors/Condition Variables | Send/Recv |
| Low-Level Atomic Ops (specific) | Load/store | Interrupt disable/enable | Test&Set | Other atomic instructions |
| | Interrupts (I/O, timer) | Multiple processors | | |

## 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