# Criticism

Use of lock for mutual exclusion satisfies all 5 criteria

However ...

- Depends on machine-specific instruction
- Looping (aka **busy-waiting** or **spinning**) by wait-er is wasteful of CPU time

  Better solution might "suspend" waiter (i.e., place it on a queue) and "resume" it once lock available
- Locking doesn't provide *coordination*—no way to tell waiting thread that it can now proceed because lock has been dropped

# Mechanisms for Mutual Exclusion

1. Disable interrupts
2. *Without* hardware atomic instructions other than single-word read or single-word write
3. Hardware instructions for locking

WE ARE HERE

4. Hardware instructions for atomic access/update of certain common data structures
5. Higher level solutions (built on lower level solutions)

# 4. Compare-and-Swap

Compare-and-swap: compare memory location versus value held in register, then swap value in *second* register with memory location if compared values are equal

Two-word compare-and-swap: performs two comparisons, updates two locations if both comparisons are equal

# CAS Semantics, I

Semantics of CAS(r1, r2, mem): atomically perform these actions ...

```
if (r1 == mem) {    /* compare */
    mem = r2        /* swap r2 & mem */
    r2 = mem
}
set Z CC bit based on (r1-mem)
```

# CAS Semantics, II

Sometimes defined as CAS(r1, r2, mem) ...

```
if (r1 == mem) {   /* compare */
    mem = r2    /* not a true swap */
else
    r2 = mem
}
set Z CC bit based on (r1-mem)
```

Influential Motorola 680x0 implemented first TAS & last CAS definitions

# Expressive Power

CAS "more powerful" than TAS because of possibility of conditional action provided by comparison & 2nd register

2-word CAS "more powerful" than CAS because can update 2 locations

More powerful $\Rightarrow$ can do more in a *single instruction*

(Can always get a lock then do what you want in several instructions)

# Use of CAS for Locking

Assume: *first* definition of CAS

```
        mov r4 <- 0
loop:   mov r5 <- 1

//
// if (lock is 0)
//   then set it to 1
//   else do nothing
//
        cas r4, r5, lock

//
// if (lock was 1) try again;
// jump back to "mov" to be
// cautious -- lock's value
// (now in r5) should have been 1
// if other threads are
// well behaved but it doesn't
// hurt to be cautious
//
        bnz loop
```

# Use of CAS for Increment

```
// read old value
loop:   mov r4 <- counter

// increment in r5
        mov r5 <- r4
        inc r5

//
// if (counter still equals value read)
//    then swap counter & new value
//
        cas r4, r5, counter

// if (counter != value read) try again
        bnz loop
```

# Use of CAS for Insert into Singly-linked List

Assume:

- Within list element, next pointer is at offset NEXT_OFF
- Insert at head

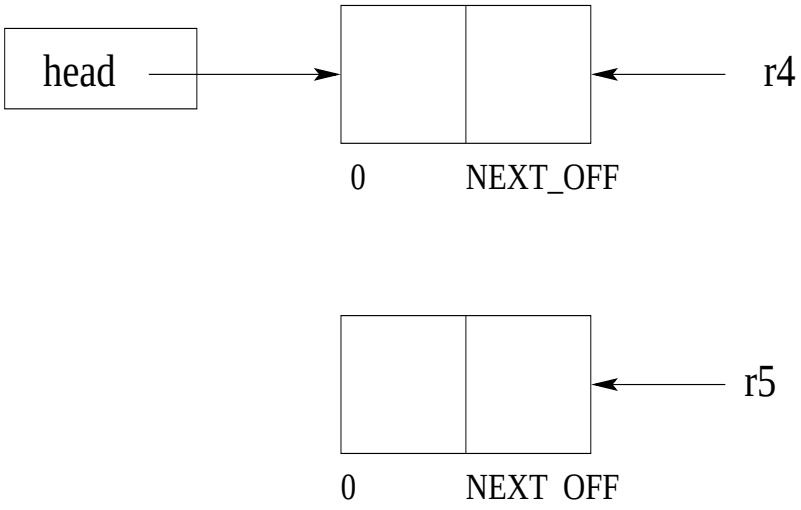
# CAS Insert

```
// r5 permanently points to new element
        mov r5 <- new

// r4 points to current head pointer
loop:   mov r4 <- head

// set new->next = current head
        mov (r5,NEXT_OFF) <- r4

//
// if (head is unchanged)
//   then swap to make head = new
//
        cas r4, r5, head

// if (head was changed), try again
        bnz loop
```

# Illustration

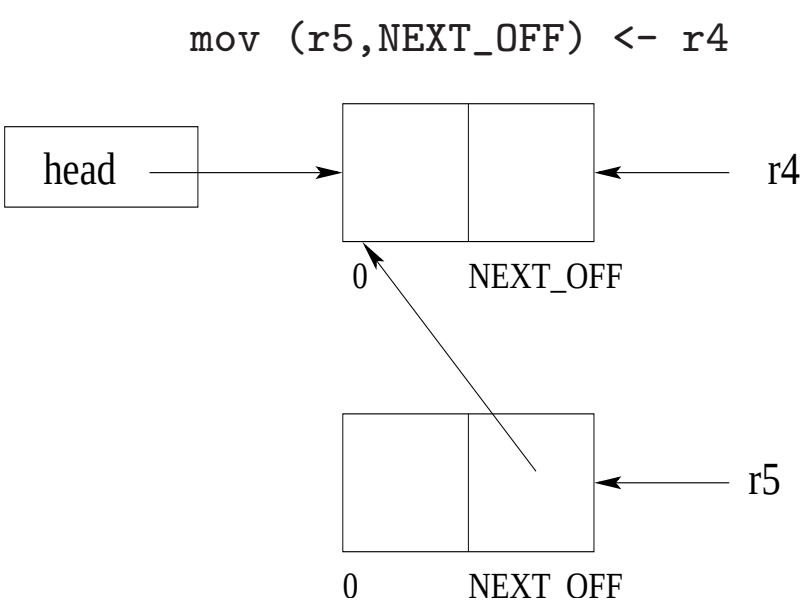
After

```
        mov r5 <- new
loop:   mov r4 <- head
```



# Illustration

After

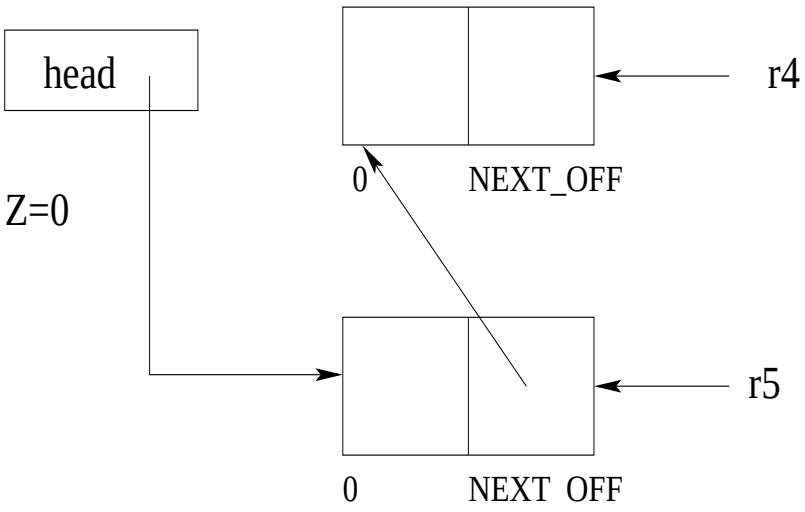


# Illustration

After

```
cas r4, r5, head
```

assuming `r4 = head`

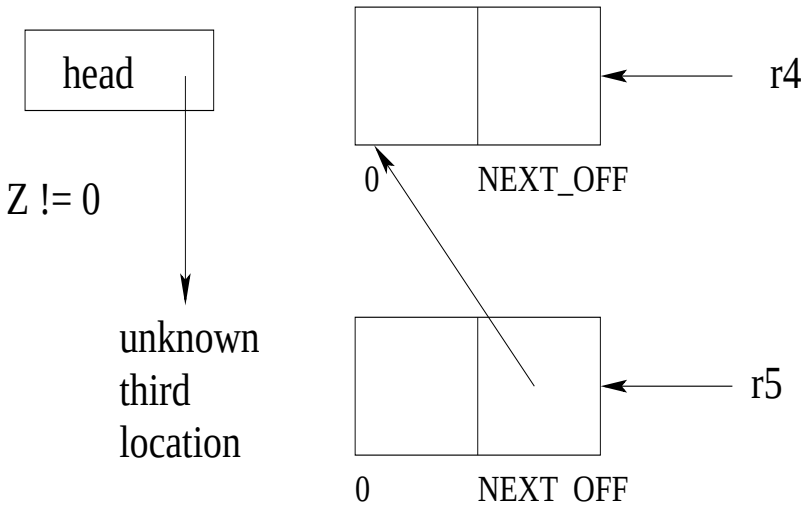


# Illustration

After

```
cas r4, r5, head
```

assuming `r4 != head`



# Summary of the Preceding Slides

TAS instruction was designed with only one use in mind – to get a lock atomically

CAS instruction was designed to get a lock atomically AND to do certain other common actions atomically

# 4. Load-Linked and Store-Conditional

TAS and CAS lock the processor-memory bus for their duration

- Bus is the bottleneck resource in many uniprocessor systems
- Even more so in many shared-memory multiprocessors

LL and SC designed with multiprocessing in mind

Assumes memory word can be "marked" with identity of marking CPU

Pioneered by MIPS architecture

# LL and SC Semantics

Load Linked (LL) atomically does:

1. load memory location into register
2. mark location with CPU's ID

(Note: memory location can be marked by >1 CPUs)

Store Conditional (SC) atomically does:

1. if memory location is marked *by this CPU*, store new value and remove ALL marks; else do nothing
2. return indication of which case occurred

# Use of LL+SC for Locking

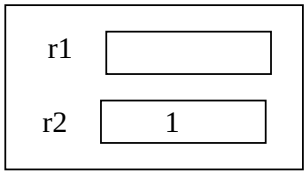
To get lock:

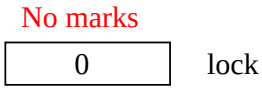
```
r2 = 1;
while (1) {
    LL r1, lock
    if (r1 == 0)
        if (SC r2, lock)
            break;
}
```

SC returns 1 iff `lock` was STILL marked by this CPU; i.e., no other CPU executed SC before this one did

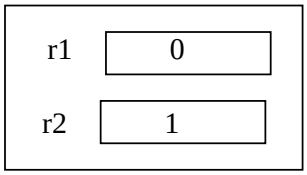
To drop lock: `STORE lock, 0`

# Example: No Contention



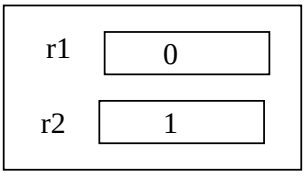



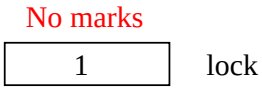

LL r1, lock



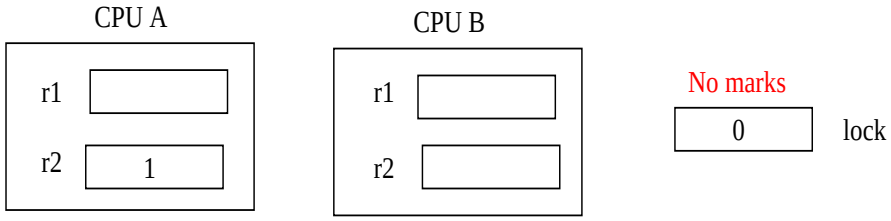


SC r2, lock

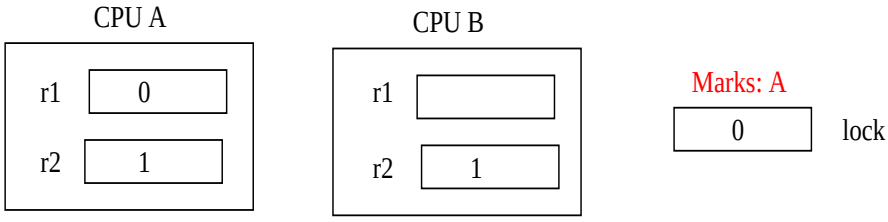


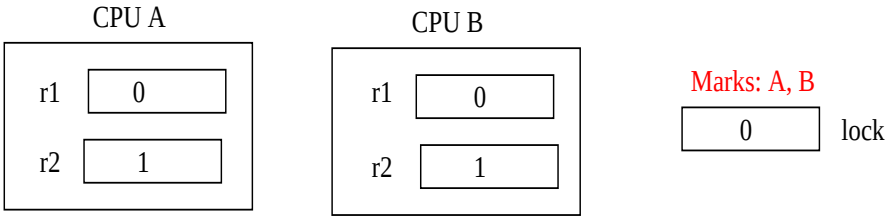
# Example: Contention



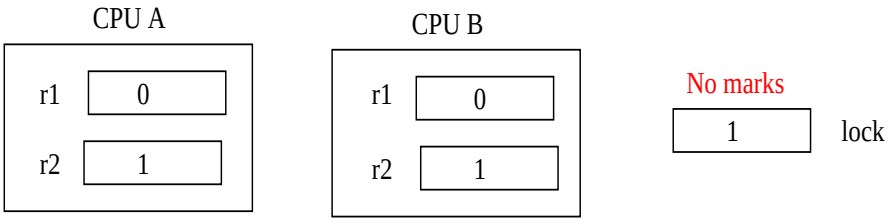

CPU A: LL r1, lock




CPU B: LL r1, lock




CPU A: SC r2, lock




CPU B: SC r2, lock ... do nothing because there is no B mark;
loop and perform LL and SC instructions again

CPU B: LL r1, lock ... r1 is 1 so keep looping

# Use of LL+SC for Increment

```
while (1) {
    LL r1, counter
    r2 = r1 + 1;
    if (SC r2, counter)
        break;
}
```

# Summary

Special atomic instructions can be used to implement locks AND ALSO to perform other common operations

Which other operations depends on the instruction

But: these "fancy uses" are rarely employed

# Mechanisms for Mutual Exclusion

1. Disable interrupts

2. *Without* hardware atomic instructions other than single-word read or single-word write

3. Hardware instructions for locking

4. Hardware instructions for atomic access/update of certain common data structures

WE ARE HERE

5. Higher level solutions (built on lower level solutions)

# Pthreads Locking API

`pthread_mutex_lock` — waits if lock already held

`pthread_mutex_trylock` — immediately returns indication whether lock is already held

`pthread_mutex_unlock` — drops lock

# Mutex Concept

Q: What is a pthread "mutex" lock?

A: A simple lock – either it's held or it's not, nothing fancier

Commonly implemented with:

- special instructions mentioned above
- semaphore initialized to 1 (and semaphore is implemented using special instructions)