Hardware Solutions to Critical Sections

- Recap: Critical Sections and Locks
- Disabling Interrupts
- Atomic Instructions (TAS, XCHG, FAA, CAS)
- Performance Analysis of TAS based locks
- Improving the basic algorithms

Recall: Critical Sections & Locks

- Execution of critical section by threads must be mutually exclusive.
- Need protocol to enforce mutual exclusion.
- Easy to do with locks.

```
class Lock {
  public void lock() {???}
  public void unlock() {???}
};
```

```
Lock lck;
while (TRUE) {
  lck.lock();
  critical section;
  lck.unlock();
  remainder section;
}
```

Simple Hardware Support: Disable Interrupts

By disallowing interrupts the current thread cannot be interrupted.

This "turns the entire executable into one critical section".

Pros:

- simplicity!

Cons:

- interrupt service latency
- contention for the one critical section
- what about multiprocessors?

Hardware Support: Atomic Operations

Atomic operations: Operations that check and modify memory areas in a single step (i.e. operation can not be interrupted)

- Test-And-Set
- Fetch-And-Add
- Exchange, Compare-And-Swap
- Load-Link/Store Conditional

In a **multiprocessor**, atomicity is achieved by locking the bus for the length of the operation.

In this way, the memory locations cannot be accessed by other threads on the current or on other processor.

Hardware Support: Test-and-Set (TAS)

```
bool TAS(bool * var) { /* Test-And-Set */
  bool temp;
  temp = *var;
  *var  = true;
  return temp;
}
```

atomic!

The function **bool TAS(bool * var)** **atomically** stores **true** in the given memory location and returns previous value.

Locks with Test-and-Set

```
class TASLock {
  private bool locked = false; /* locked?                */

  public void lock() {
    while (TAS(&locked));      /* loop until not locked */
  }
  public void unlock() {
    locked = false;            /* mark lock as free     */
  }
};
```

Recall: **bool TAS(bool * var)** stores **true** in memory location and returns previous value.

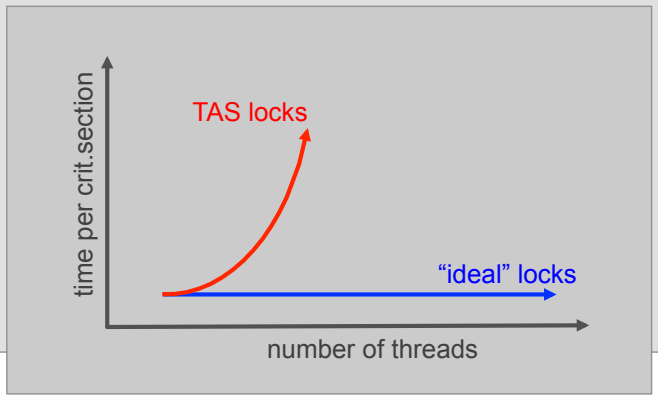
Locks with TAS: Assembly Implementation

```
lock_acquire:           ; function entry point
  tas reg, locked       ; Test-and-set; 'locked' is the
                        ; shared variable; is is copied
                        ; into register reg and 'locked'
                        ; then atomically set to 1.
  cmp reg, 0            ; Was 'locked zero' on entry?
  jnz lock_acquire      ; Jump to lock_acquire if reg is non-zero;
                        ; i.e, 'locked' was non-zero on entry.
  ret                   ; Exit; 'locked' was zero on entry.
                        ; If we get here, tas will have set it
                        ; to non-zero; thus, we have acquired the
                        ; lock associated with variable 'locked'.
```

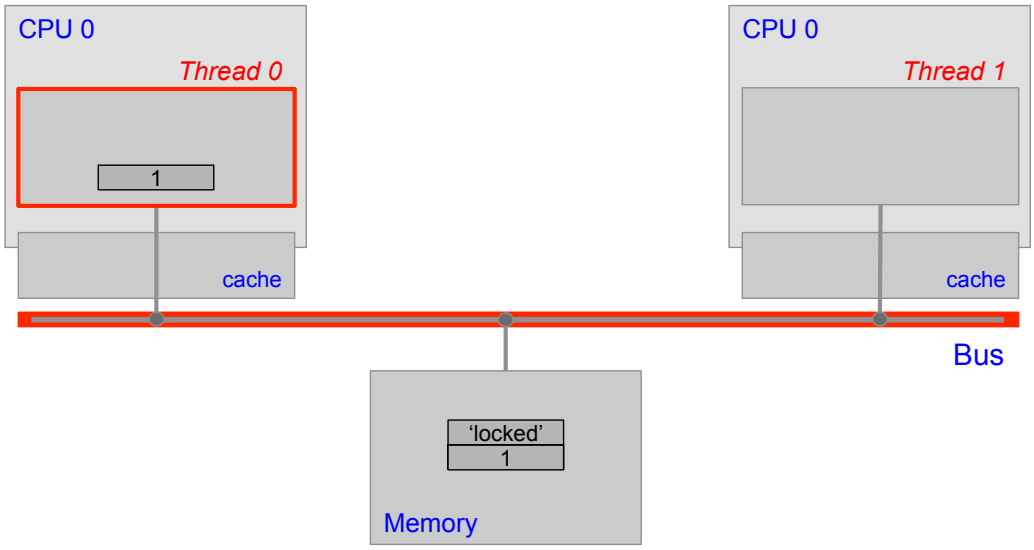
```
lock_release:           ; function entry point for unlock
  mov locked, 0         ; store 0 in 'locked'
  ret                   ; return to caller
```

Analysis of TAS Locks

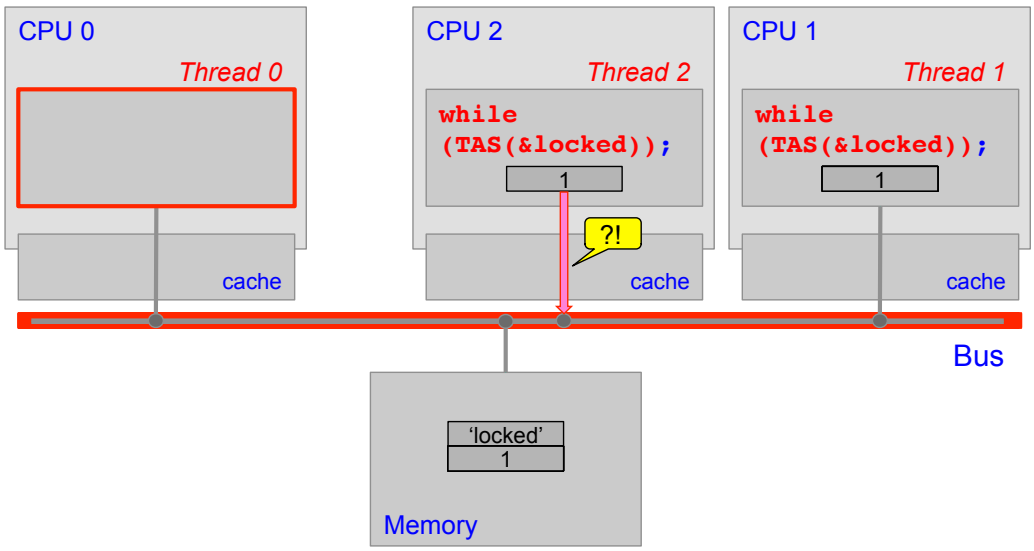
```
class TASLock {
  private bool locked = false; /* locked?                */

  public void lock() {
    while (TAS(&locked));
  }
  public void unlock() {
    locked = false;
  }
};
```



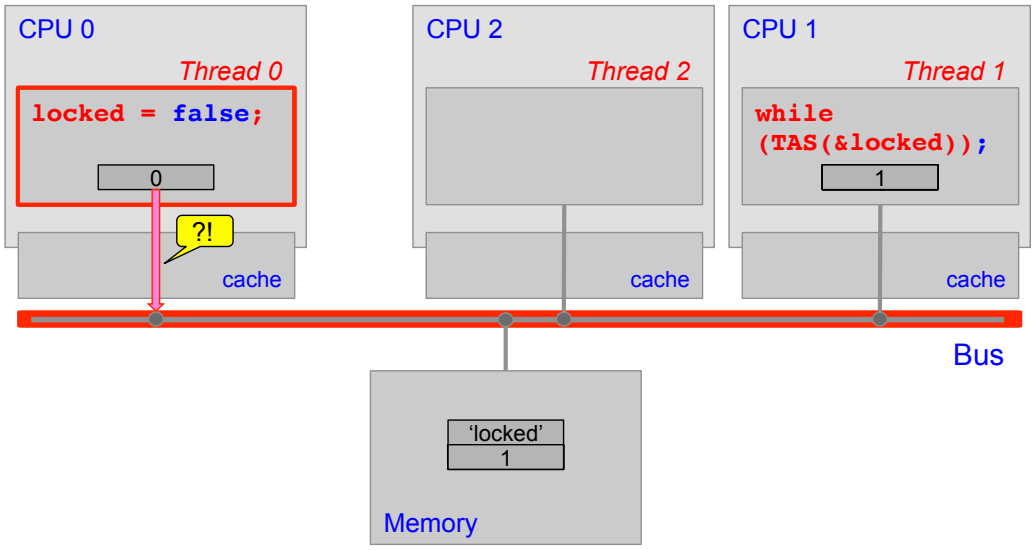
Analysis of TAS Locks



Analysis of TAS Locks



Analysis of TAS Locks



Improving Locks with Test-and-Set

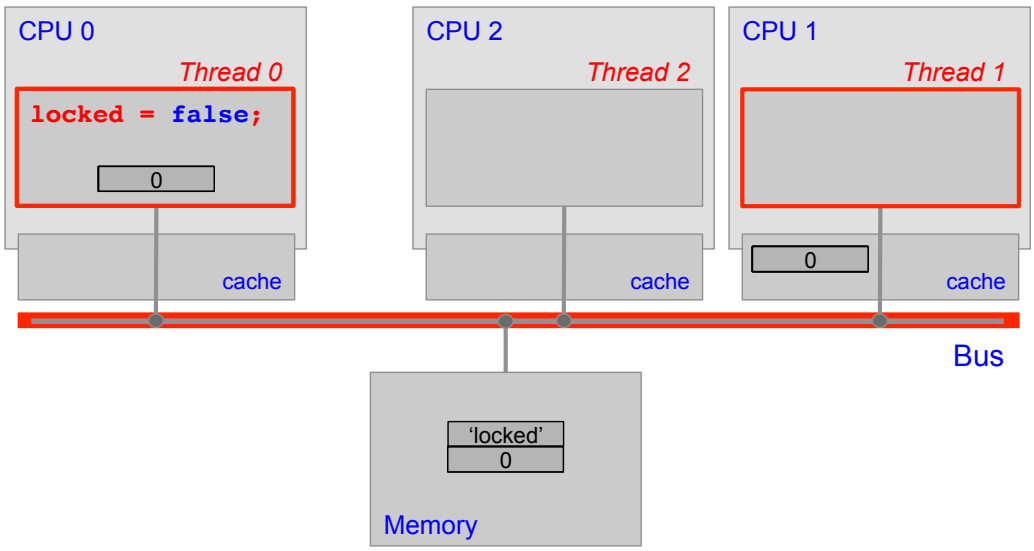
```
class TASLock {
  private bool locked = false; /* locked?                */

  public void lock() {
    while (TAS(&locked));      /* loop until not locked */
  }
  public void unlock() {
    locked = false;            /* mark lock as free     */
  }
};
```

Recall: **bool TAS(bool * var)** stores **true** in memory location and returns previous value.

Locks with **Test-and-**Test-and-Set

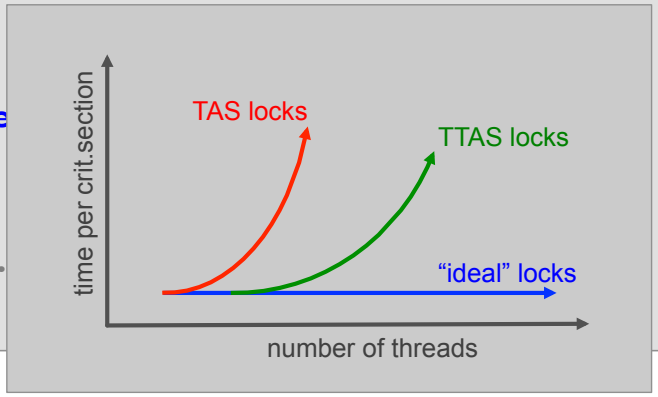
```
class TTASLock {
  private bool locked = false; /* locked?                */
  public void lock() {
    for(;;) {
     while (locked){};          /* certainly locked       */
     if (!TAS(&locked)) break; /* appears free; check!   */
    }
  }
  public void unlock() {...}
};
```

Analysis of TAS Locks



Locks with **Test-and-**Test-and-Set

```
class TTASLock {
  private bool locked = false; /* locked?                */
  public void lock() {
    for(;;) {
     while (locked){};
     if (!TAS(&locked)) bre
    }
  }
  public void unlock() {...
};
```



Hardware Support: Exchange (XCHG)

```
void XCHG(bool * a, bool * b) { /* Exchange */
  bool temp;
  temp = *a;
  *a   = *b;
  *b   = temp;
}
```

atomic!

The function **`void XCHG(bool * a, bool * b)`** **atomically** exchanges the values stored in locations a and b.

Locks with Exchange

```
class XCHGLock {
  private bool locked = false; /* locked?                */
  public void lock() {
    bool dummy = true;
    do XCHG(&locked, &dummy);
    while (dummy)              /* loop until not locked */
  }
  public void unlock() {
    locked = false;            /* mark lock as free     */
  }
};
```

The function **`XCHG(bool * a, bool * b)`** **atomically** exchanges the values stored in locations a and b.

Hardware Support: Fetch-And-Add (FAA)

```
int FAA(int * addr) {
  int value = *addr;
  *location = *addr + 1;
  return value;
}
```

atomic!

The function **`int FAA(int * addr)`** **atomically** increments the value stored in location addr and returns value before increment.

Locks with Fetch-and-Add (Ticket-Lock Algorithm)

```
class FAALock {
  private int ticketnumber = 0; int turn = 0;

  public void lock() {
    int myturn = FAA(ticketnumber);
    while (turn != myturn); /* wait for our turn */
  }
  public void unlock() {
    FAA(turn);
  }
};
```

Recall: **int FAA(int * loc)** **atomically** increments the value stored in location loc and returns value before increment.

Hardware Support: Compare-and-Swap (CAS)

```
int CAS(Type old, Type new, Type * addr) {
  if (*addr == old) {
    *addr = new;
    return true;
  }
  else {
    return false;
  }
}
```

atomic!

The function **int CAS(T o, T n, T * a)** **atomically** compares the value stored in location addr with old. If they are equal, it assigns new to location addr, and returns true. Returns false otherwise.

Locks with Compare-and-Swap

```
class CASLock {
  private bool locked = false; /* locked?                    */

  public void lock() {
    while (!CAS(false, true, &locked));
       /* loop until not locked */
  }
  public void unlock() {
    locked = false;
  }
};
```

Recall: **bool CAS(T o, T n, T * a)** **atomically** compares the value stored in location addr with old. If they are equal, it assigns new to location addr, and returns true. Returns false otherwise.

Compare-and-Swap on the x86 (simplified!)

```
[lock] cmpxchg 'destination', 'source'

  if (accumulator == destination) {
       destination <- source
       zero flag   <- 1
  }
  else {
       accumulator <- destination
       zero flag   <- 0
  }
```

Prefix 'lock' is used to lock memory bus in multiprocessors.

'Accumulator' is the A, AX, EAX register, depending on size of operands.

Locks with CAS: Assembly Implementation

```
lock_acquire:              ; function entry point
  mov edx, 1               ; value to set 'locked' to.
lock_retry:
  mov eax, 0               ; value to check 'locked' against.
  ; if (locked == eax): locked    <- edx, zero flag <- 1
  ; else:               eax       <- edx, zero flag <- 0
  ; If 'locked' is zero, then it will have same value as eax and 'locked' will
  ; be set to edx which is one and we acquire the lock. If the lock was
  ; acquired before the cmpxchg then zero flag will be zero and we spin again.
  cmpxchg [locked], edx    ; CAS(0,1,&locked)
  jz lock_retry            ; Jump to lock_retry if CAS returned false
  ret
```

```
lock_release:
  mov [locked], 0          ; use xchg on m-procs
  ret                      ; return to caller
```

Hardware Solutions to Critical Sections

- Disabling Interrupts
- Atomic Instructions (TAS, XCHG, FAA, CAS)
- Performance Analysis of TAS based locks
- Improving the basic algorithms