If only ..

- We could make this operation atomic

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Make atomic

Context Switches

Hardware to the rescue....

Hardware Support (Test & Set)

- Write to a memory location, return its old value

LOCK 0 1

atomic

```
int test_and_set(int  *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

```
while(1){
    while(test_and_set(&lock) == 1);
    critical section
    lock = 0;  // unlock
    other code
}
```

Usage for locking

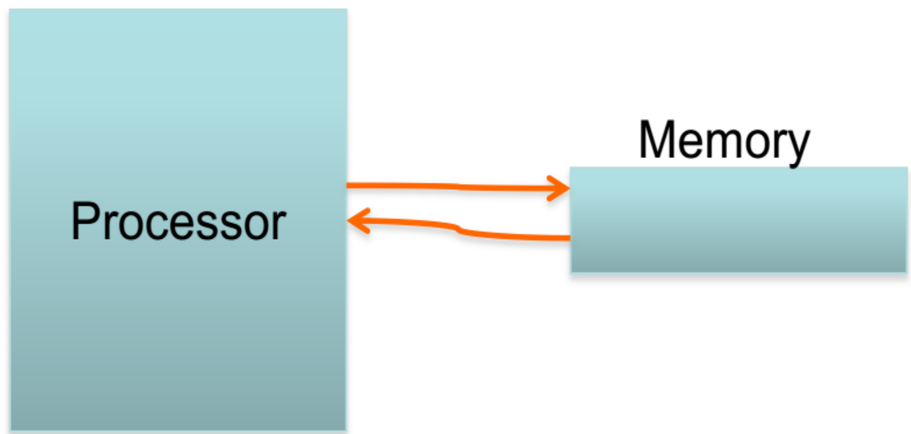
Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
So the first invocation of test_and_set will read a 0 and set lock to 1 and return. The second test_and_set invocation will then see lock as 1, and will loop continuously until lock becomes 0

Test & Set Instruction

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```



equivalent software representation
(the entire function is executed atomically)

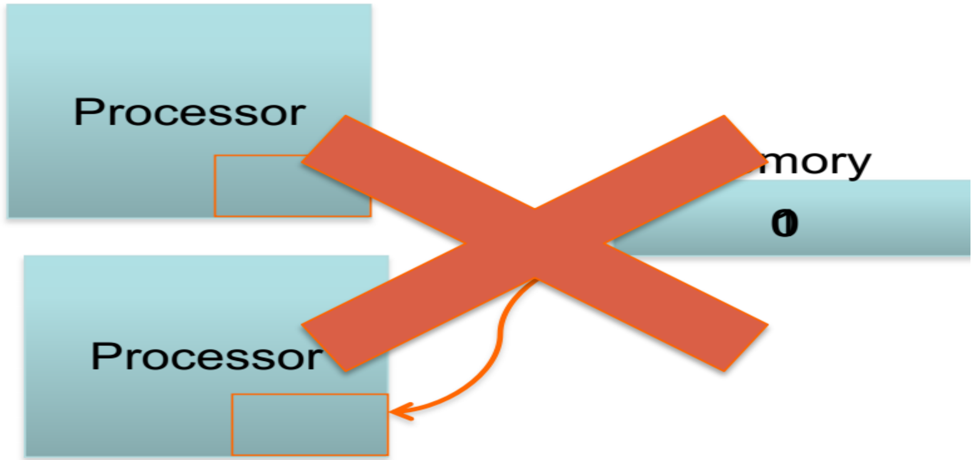
Test and Set instruction

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

equivalent software representation (the entire function is executed atomically)



Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

Intel Hardware Support (xchg instruction)

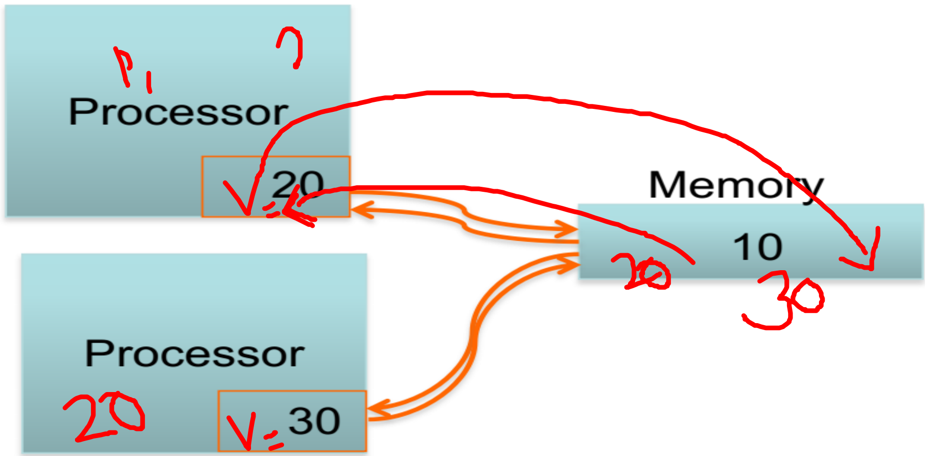
xCHG (%addr), Reg

- Write to a memory location, return its old value

atomic

```
int xchg(int  *L, int v){
   int prev = *L;
   *L = v;
   return prev;
}
```

equivalent software representation (the entire function is executed atomically)



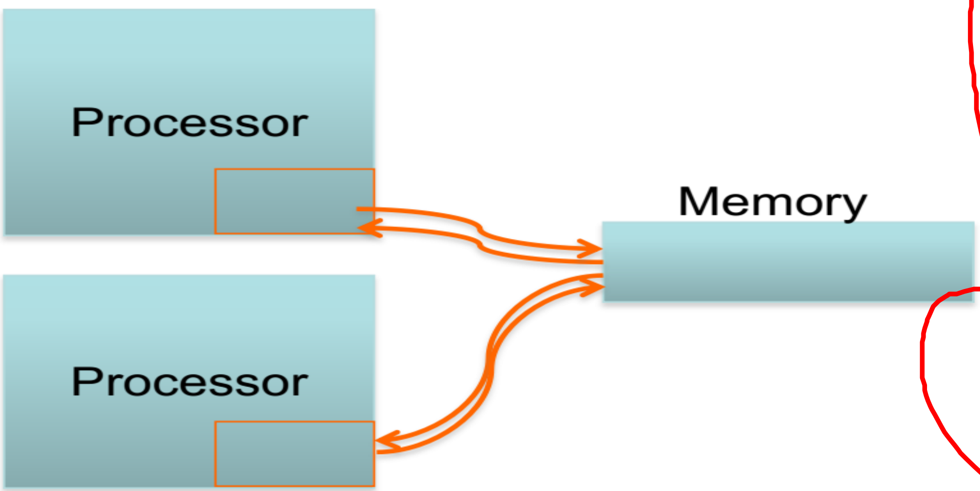
Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.

xchg instruction

Note. %eax is returned

typical usage :

xchg reg, mem



-sync

temp ← M[addr]
M[addr] ← EAX
EAX ← temp

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

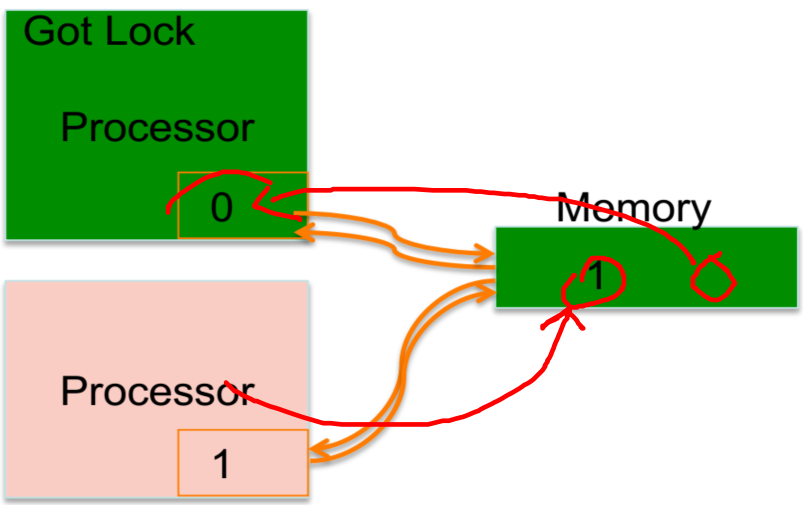
locked

CS

Note. %eax is returned

typical usage :

xchg reg, mem

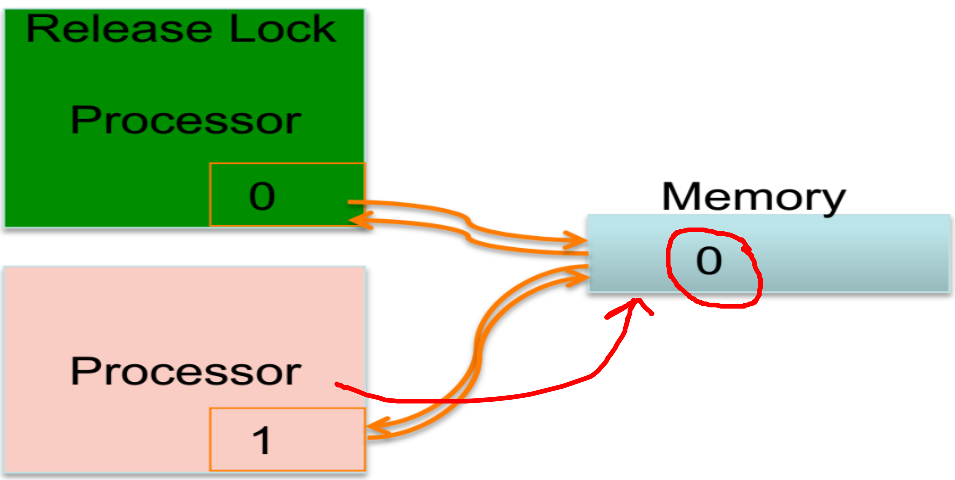



```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Note. %eax is returned

typical usage :

xchg reg, mem

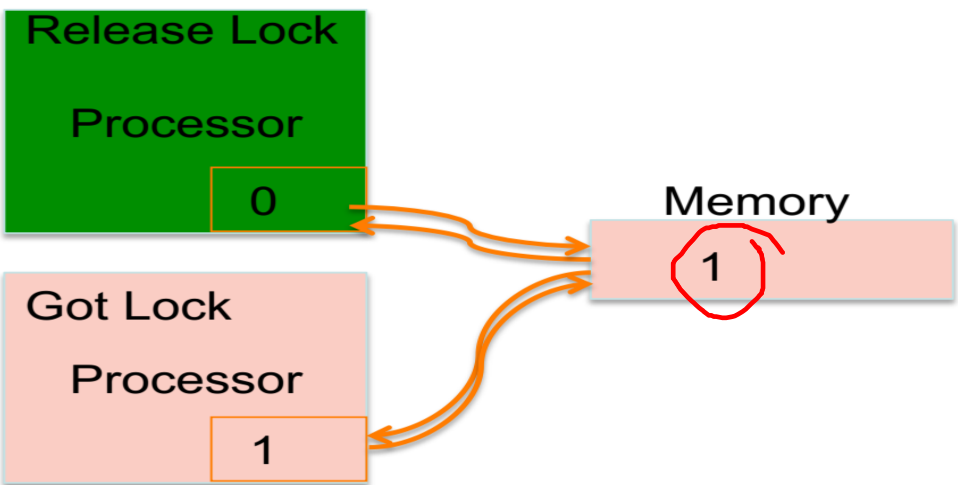



```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Note. %eax is returned

typical usage :

xchg reg, mem

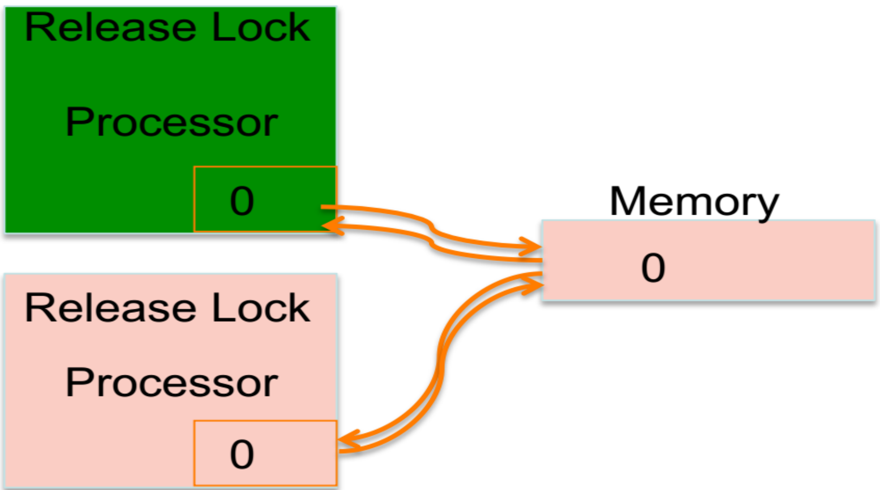



```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Note. %eax is returned

typical usage :

xchg reg, mem




```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Spinlocks

Process 1

```
acquire(&locked)
critical section
release(&locked)
```

Process 2

```
acquire(&locked)
critical section
release(&locked)
```

- One process will acquire the lock
- The other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process releases it

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Issues with Spinlock

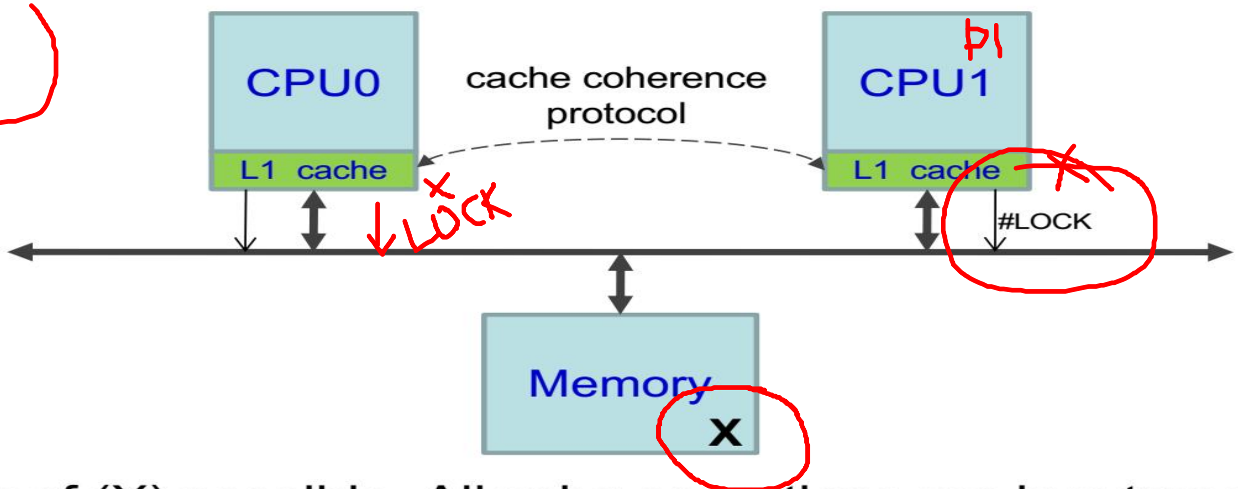
xchg %eax, X M

Reg

- No compiler optimizations should be allowed
  - Should not make X a register variable
    - Write the loop in assembly or use volatile
- Should not reorder **memory** loads and stores
    - Use serialized instructions (which forces instructions not to be reordered)
    - Luckly xchg is already implements serialization

Issues with Spinlock

xchg %eax, X



- No caching of (X) possible. All xchg operations are bus transactions.
  - CPU asserts the LOCK, to inform that there is a ‘locked ‘ memory access
- acquire function in spinlock invokes xchg in a loop…each operation is a bus transaction …. huge performance hits

When to use Spinlocks?

- Characteristic : busy waiting
  - Useful for short critical sections, where much CPU time is not wasted waiting
    - eg. To increment a counter, access an array element, etc.
  - Not useful, when the period of wait is unpredictable or will take a long time
    - eg. Not good to read page from disk.
    - Use mutex instead (…mutex)

Pthread example

```c
#include <pthread.h>
#include <stdio.h>

int global_counter;
pthread_spinlock_t splk;

void *thread_fn(void *arg){
  long id = (long) arg;
  while(1){
    pthread_spin_lock(&splk);
    if (id == 1) global_counter++;
    else global_counter--;
    pthread_spin_unlock(&splk);
    printf("%d(%d)\n", id, global_counter);
    sleep(1);
  }

  return NULL;
}

int main(){
  pthread_t t1, t2;

  pthread_spin_init(&splk, PTHREAD_PROCESS_PRIVATE);
  pthread_create(&t1, NULL, thread_fn, (void *)1);
  pthread_create(&t2, NULL, thread_fn, (void *)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_spin_destroy(&splk);
  printf("Exiting main\n");
  return 0;
}
```

lock

CS

unlock

create spinlock

destroy spinlock

Alternative to Spinning

- Alternative to spinlock: a (sleeping) mutex
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later
  – yield() moves thread from running to ready state

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}
```

Blocked

```
void unlock() {
    flag = 0;
}
```

wake up ( )

Mutexes

- Can we do better than busy waiting?
  - If critical section is locked then yield CPU
    - Go to a SLEEP state
  - While unlocking, wake up sleeping process

1/2

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

N=100

Thundering Herd Problem

- A large number of processes wake up (almost simultaneously) when the event occurs.
  - All waiting processes wake up
  - Leading to several context switches
  - All processes go back to sleep except for one, which gets the critical section
    - Large number of context switches
    - Could lead to starvation

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

- The Solution
  - When entering critical section, push into a queue before blocking
  - When exiting critical section, wake up only the first process in the queue

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else{
      // add this process to Queue
      sleep();
    }
  }
}

void unlock(int *locked){
  locked = 0;
  // remove process P from queue
  wakeup(P)
}
```

pthread Mutex

- pthread_mutex_lock ( )
- pthread_mutex_unlock ( )

Spinlock vs Sleeping Mutex

- Most userspace lock implementations are of the sleeping mutex kind
  - –CPU wasted by spinning contending threads
  - –More so if a thread holds spinlock and blocks for long
- Locks inside the OS are always spinlocks – Why? Who will the OS yield to?
- When OS acquires a spinlock:

OS code

  - –It must disable interrupts (on that processor core) while the lock is held. Why? An interrupt handler could request the same lock, and spin for it forever.
  - –It must not perform any blocking operation–never go to sleep with a locked spinlock!
- In general, use spinlocks with care, and release as soon as possible

Locks and Priorities

- What happens when a high priority task requests a lock, while a low priority task is in the critical section
  - Priority Inversion
  - Possible solution
    - Priority Inheritance

up graded

How locks should be used?

A lock should be acquired before accessing any variable or data structure that is shared between multiple threads of a process–“Thread-safe” data structures

•All shared kernel data structures must also be accessed only after locking

•Coarse-grained vs. fine-grained locking: one biglock for all shared data vs. separate locks

–Fine-grained allows more parallelism

–Multiple fine-grained locks may be harder to manage

•OS only provides locks, correct locking discipline is left to the user

Homework

The program introduces two employees competing for the "employee of the day" title, and the glory that comes with it.

To simulate that in a rapid pace, the program employs 3 threads: one that promotes Danny to "employee of the day", one that promotes Moshe to that situation, and a third thread that makes sure that the employee of the day's contents is consistent (i.e. contains exactly the data of one employee).

Two copies of the program are supplied. One that uses a mutex, and one that does not.

Try them both, to see the differences, and be convinced that mutexes are essential in a multi-threaded environment.