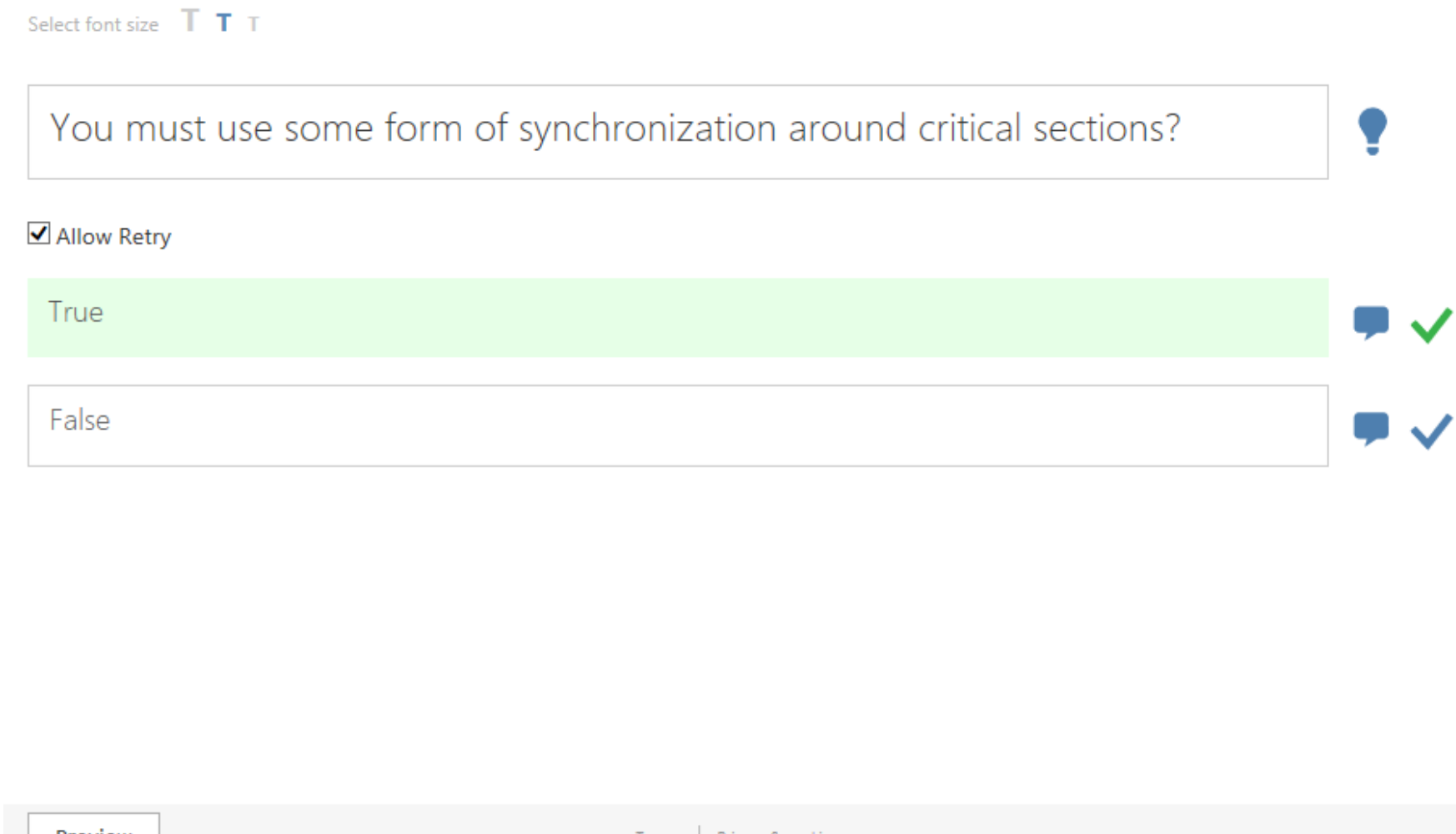
Implementing Synchronization

- Topics
  - Hardware support for synchronization
  - A walk-through of the OS161 spinlock code
- Learning Objectives:
  - Build synchronization primitives out of HW instructions.

Assumed Knowledge

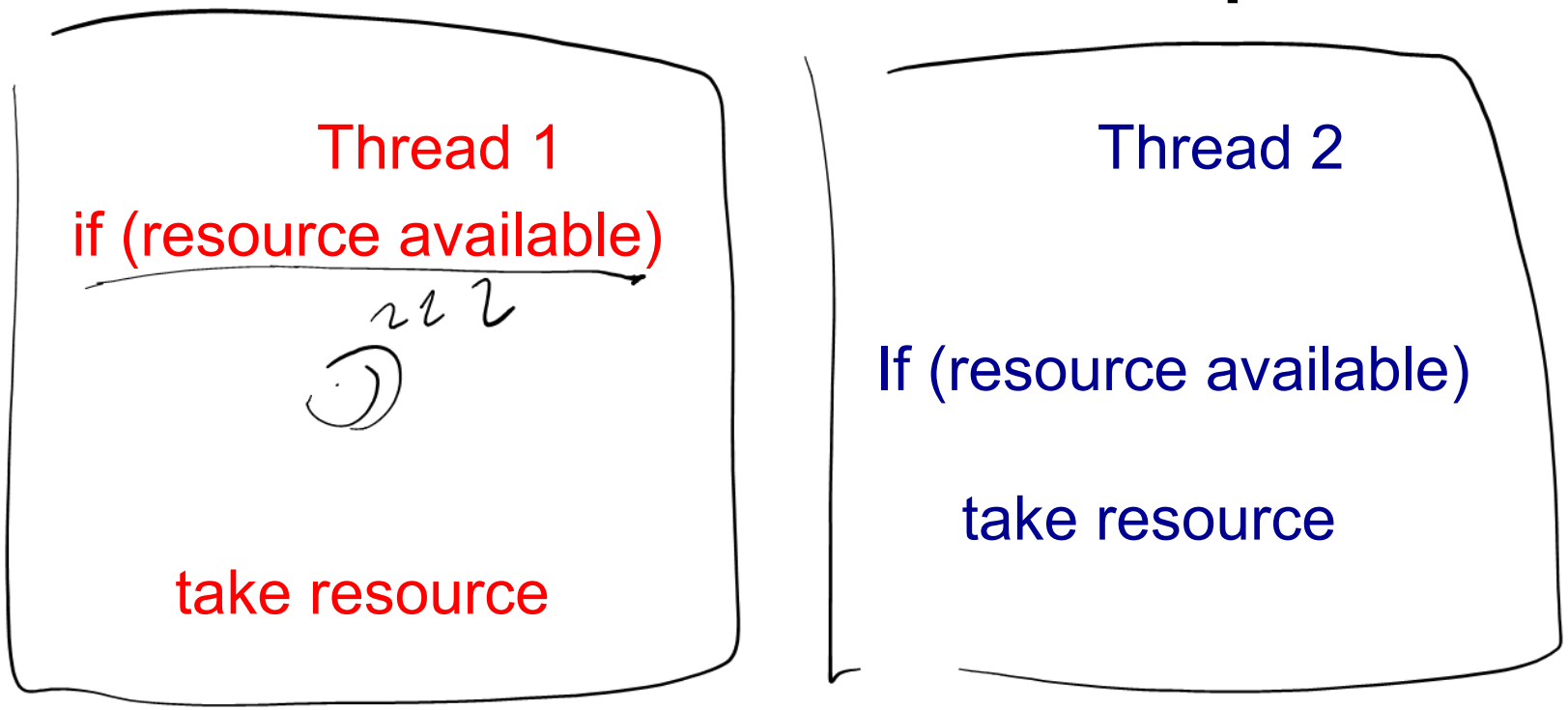
- We are assuming that you have some experience with synchronization
- You should know the meaning of:
  - Critical section
  - Mutual exclusion
  - Race condition
  - Deadlock
- You should know how to use:
  - Semaphores (both binary and counting)
  - Locks
  - Condition Variables
- If you are not familiar with these terms and/or primitives, we **strongly** encourage you to review these materials:
  - Overview: https://mix.office.com/watch/1f1z48ng5pl4z
  - Primitives: https://mix.office.com/watch/yffr5uz9opng


Select font size T T T
You must use some form of synchronization around critical sections?
Allow Retry
True
False
Preview
Terms | Privacy & cookies

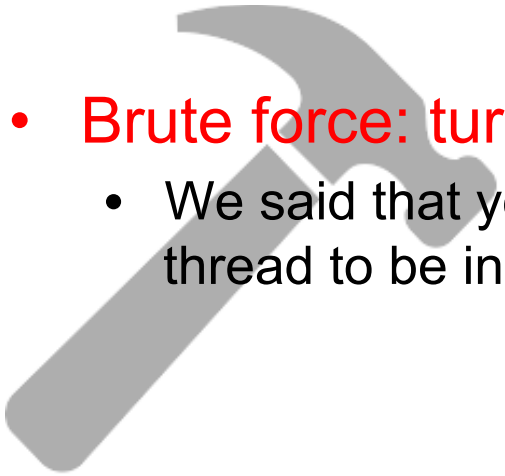
Providing Synchronization Primitives

- Deep deep in the heart of any synchronization primitive, we find ourselves wanting to do two things atomically:
  1. Check some condition
  2. Take some action depending on that condition
- For example, gaining exclusive access to a resource consists of:
  1. Checking that no one is using the resource
  2. Granting access to the resource
- If it is possible for another thread or process to run between steps 1 and 2, you have a problem.

Problematic example

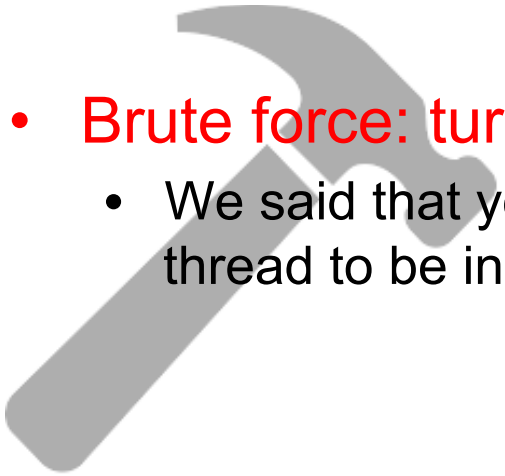


Hardware to the Rescue

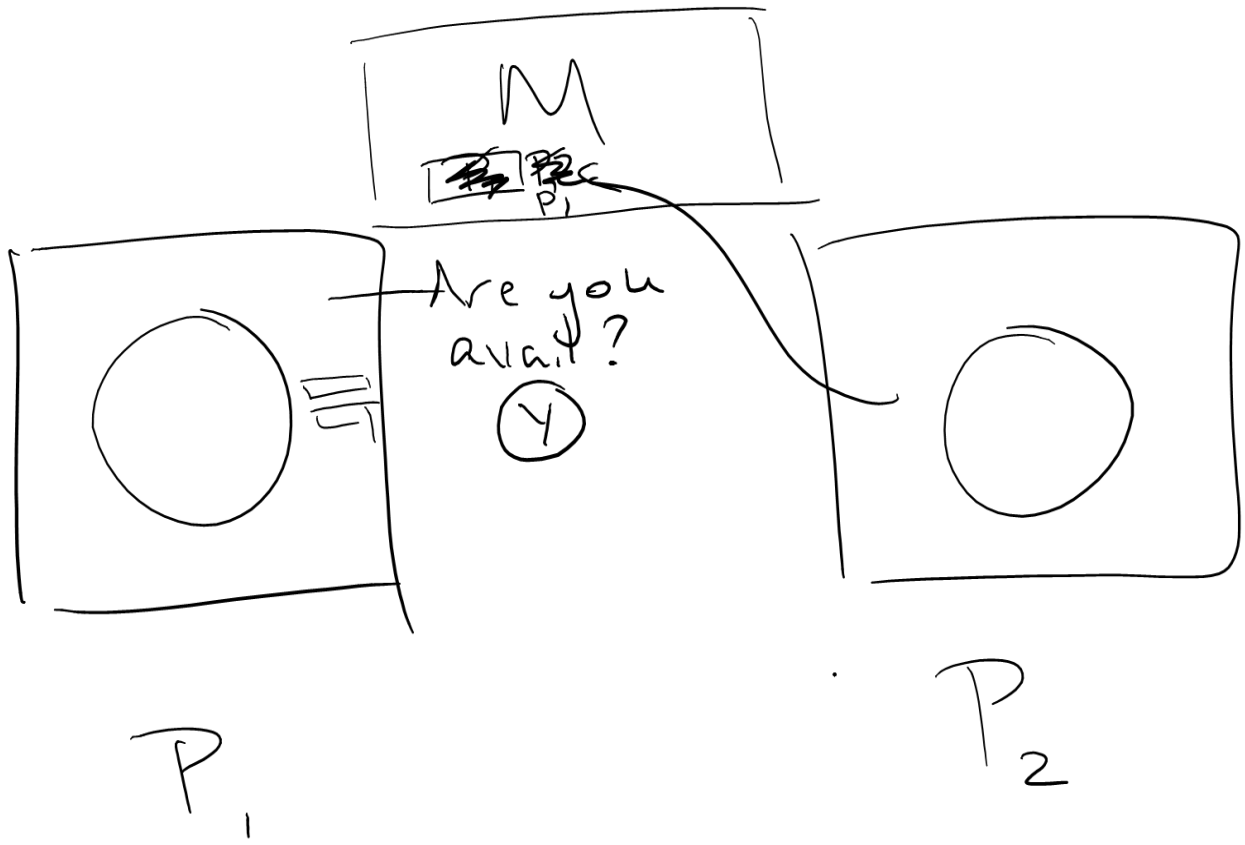


- Brute force: turn off interrupts
  - We said that you had a problem only if it was possible for the thread to be interrupted at any time…
  - Do you see a problem with this approach?

Hardware to the Rescue



- Brute force: turn off interrupts
  - We said that you had a problem only if it was possible for the thread to be interrupted at any time…
  - Do you see a problem with this approach?
    - What if you have multiple processors?


M
Are you
avail?
Y
P1
P2

Hardware to the Rescue Part 2

- Special hardware instructions
  - The hardware will provide you at least one instruction that lets you combine checking a condition and doing something (simple).
  - From that single instruction, it is possible to implement whatever synchronization primitives you want!

Spinlocks

- A spinlock is a memory location that can be in one of two states:
  - Zero when unlocked (no one holds the spinlock)
  - Non-zero when locked (someone holds the spinlock)
- Proposed (incorrect) implementation to acquire the spinlock:

```
1. while (lock_var != 0);
2. lock_var = 1;
```

Select font size T T T

What is wrong with the implementation on the previous slide?

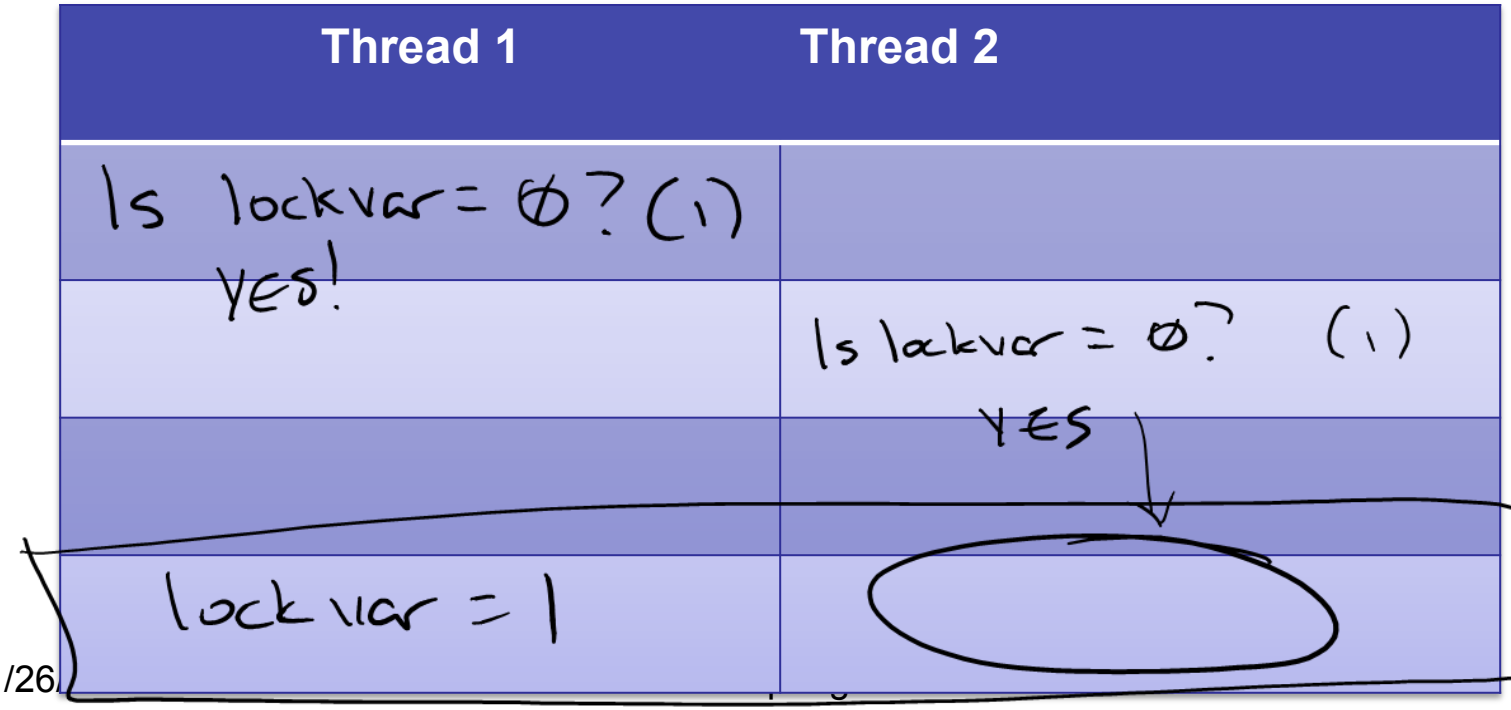
Preview

Terms | Privacy & cookies

Spinlocks: Race Condition!

- Proposed implementation:

```
1. while (lock_var != 0);
2.       lock_var = 1;
```



Hardware Primitive: TAS

- Test-and-set (TAS)
  - An atomic instruction, *RET* = TAS(*VAR*) equivalent to:

```
1. RET = VAR;
2. VAR = 1;
```

  - Usage
    - VAR is a spinlock.
    - If VAR = 0, the spinlock is available (unlocked)
    - If VAR != 0, then someone owns the spinlock
  - To acquire the spinlock:
    - RET = TAS(VAR)
    - If (RET == 0) I have the spinlock!

Hardware Primitive: CAS

- Compare and Swap (CAS)
  - Compares the contents of a memory location with a value and if they are the same, then modifies the memory location to a new value.
- CAS on Intel:

```
cmpxchg loc, val
```

- Compare value stored at memory location `loc` to contents of the *Compare Value Application Register*.
  - If they are the same, then set loc to val.
  - After, ZF flag is set if the compare was true, else ZF is 0

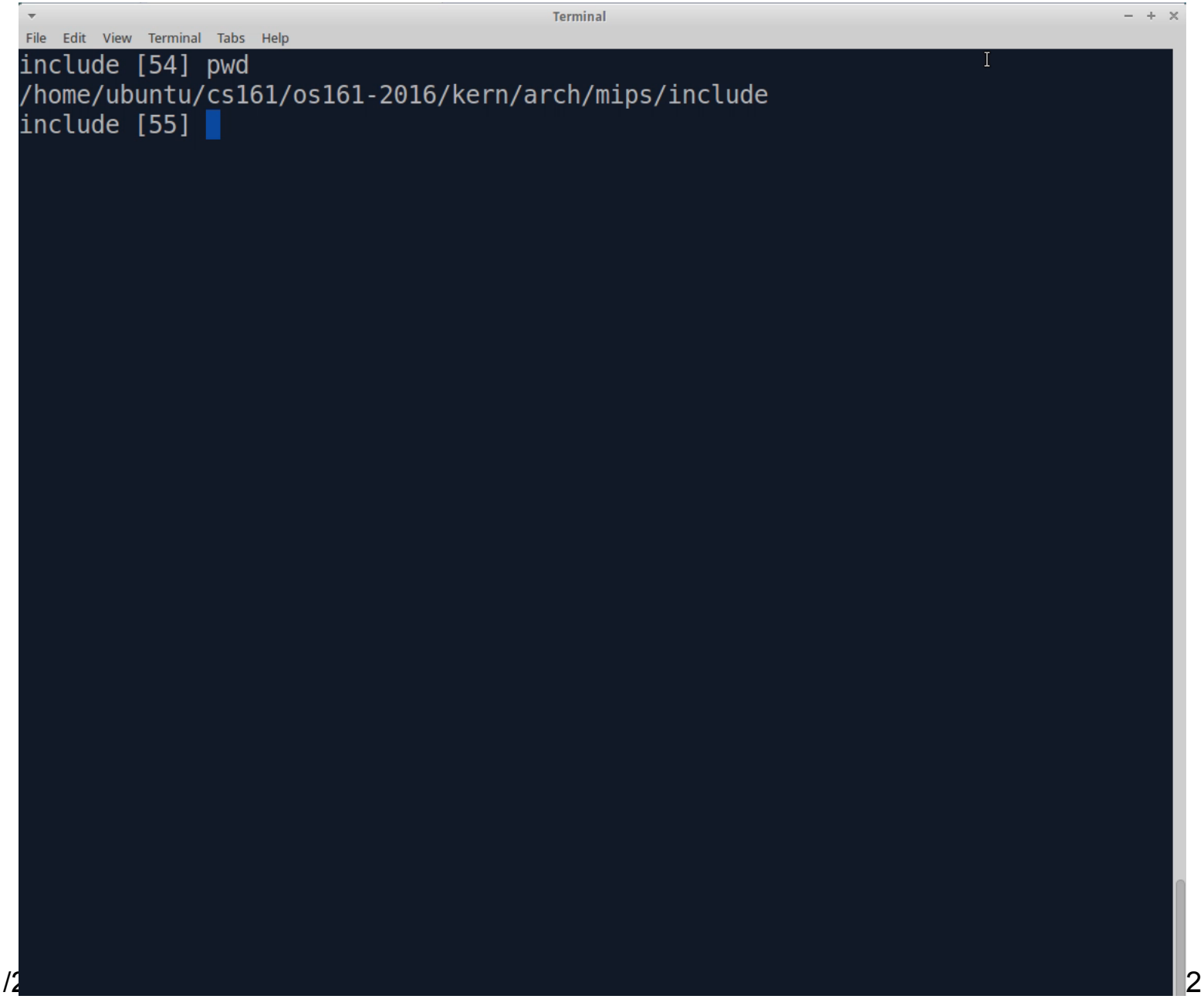
Using CAS

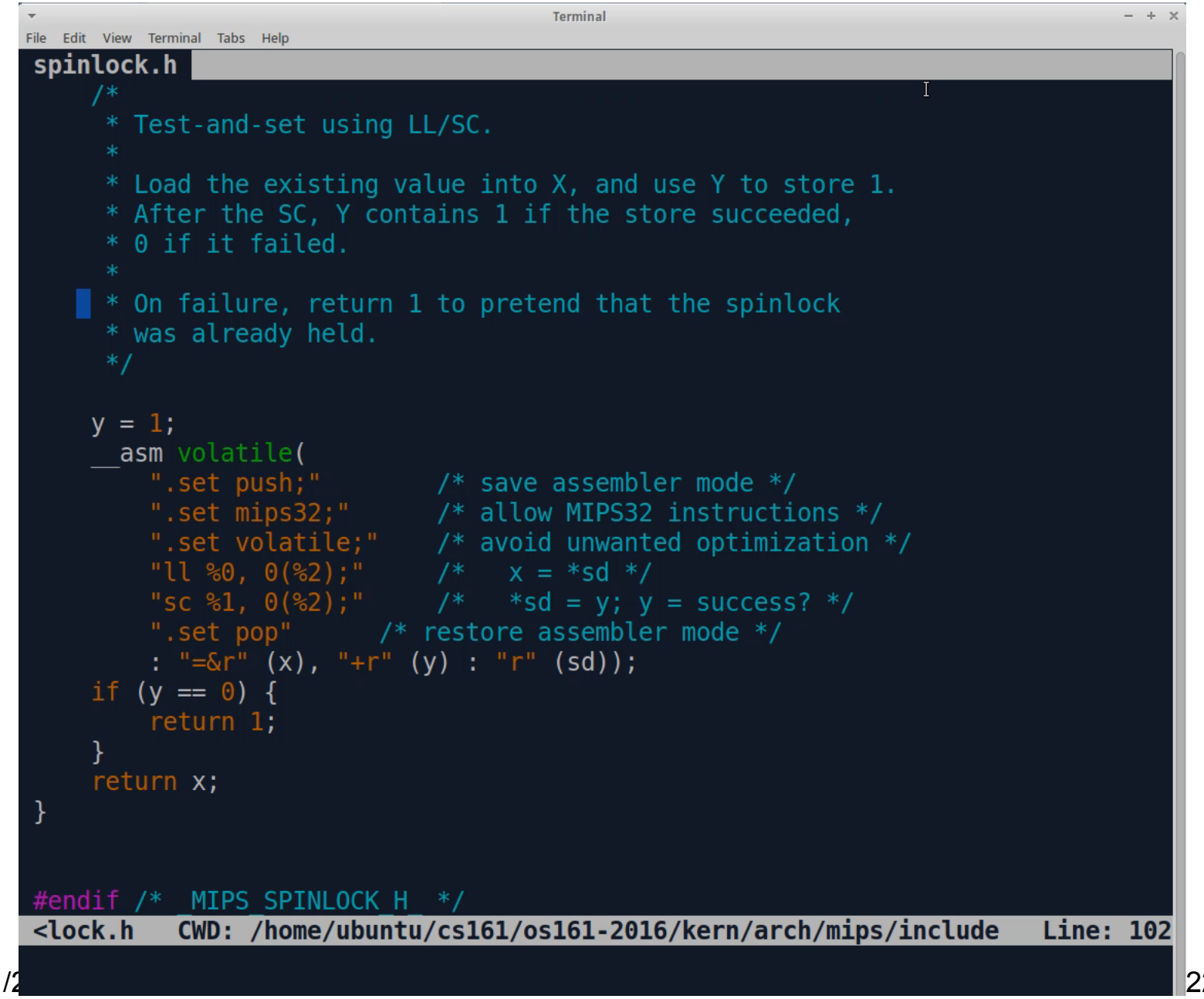
- Acquire a lock (*loc* is the spinlock).
  - Set *Compare Value Application Register* to 0

    ```
    cmpxchg loc, 1
    ```
- Check ZF flag:
  - If ZF is 1, then the compare was true, loc now contains 1, and you have the lock
  - If ZF is 0, then the spinlock is already held so your attempt to acquire it failed and should retry.

Hardware Primitive: LL/SC

- Load Link/Store Conditional (LL/SC)
  - LL: load link (sticky load) returns value from memory
  - SC: store conditional: stores a value to the memory location ONLY if that location hasn't changed since the last load-link.
- If update has occurred, store-conditional will fail.
- Usage: Let's look at the actual spinlock implementation in OS161.


Terminal
File Edit View Terminal Tabs Help
include [54] pwd
/home/ubuntu/cs161/os161-2016/kern/arch/mips/include
include [55]


Terminal
File Edit View Terminal Tabs Help
spinlock.h
/*
* Test-and-set using LL/SC.
*
* Load the existing value into X, and use Y to store 1.
* After the SC, Y contains 1 if the store succeeded,
* 0 if it failed.
*
* On failure, return 1 to pretend that the spinlock
* was already held.
*/
y = 1;
__asm volatile(
".set push;" /* save assembler mode */
".set mips32;" /* allow MIPS32 instructions */
".set volatile;" /* avoid unwanted optimization */
"ll %0, 0(%2);" /* x = *sd */
"sc %1, 0(%2);" /* *sd = y; y = success? */
".set pop" /* restore assembler mode */
: "=&r" (x), "+r" (y) : "r" (sd));
if (y == 0) {
return 1;
}
return x;
}
#endif /* _MIPS_SPINLOCK_H_ */
<lock.h CWD: /home/ubuntu/cs161/os161-2016/kern/arch/mips/include Line: 102

Fancier Hardware Support: Transactional memory

- Introduced by Herlihy and Moss in 1993.
- Finally starting to get some traction in the past few years.
- Idea:
  - Implement an entire critical section exploiting hardware to make it atomic.
  - Code up the set of operations you want and then "try" to apply them all at once atomically -- that will either succeed or fail.
- Specify a set of "transactional operations"
  - load-transactional (LT): read memory into a register
  - load-transactional-exclusive (LTX): read memory into a register and hint that you'll be updating it (optimization)
  - store-transactional (ST): write value into a memory location
- Specify a set of transaction control instructions
  - begin: start a sequence of atomic instructions
  - commit: try to apply all the updates from the transaction. If possible, apply them and the transaction succeeds. If not possible, apply none and transaction fails.
  - abort: throw away all the current transactional changes.
  - validate: check if this transaction has aborted.

Implementing Transactional Memory

- Maintain a *read-set*: set of all memory locations read during a transaction (all locations accessed by LT).
- Maintain a *write-set*: set of all memory locations written during a transaction (all locations accessed by LTX and ST).
- *Data-set* is the union of read-set and write-set.
- Commit check that:
  - no other transaction has modified any item in this transaction's data set.
  - no other transaction has read anything in this transaction's write set.
- If commit check fails, restore everything to its initial state.