Pheeeeeeewwww...

but what if we have more than 2 threads?

Peterson's Reconsidered

- Mutual Exclusion can be implemented with atomic LOAD and STORE instructions
  - multiple STOREs and LOADs
- Peterson's can be generalized to more than 2 processes (as long as the number of processes is known) but it is a mess...
  - ...and even more STOREs and LOADs

Too inefficient in practice!

Peterson's even more Reconsidered!

- It assumes LOAD and STORE instructions are atomic, but that is not guaranteed on a real processor
  - Suppose x is a 64-bit integer, and you have a 32-bit CPU
  - Then $x = 0$ requires 2 STORES (and reading x two LOADs
    - because it occupies 2 words!
  - Same holds if x is a 32-bit integer, but it is not aligned on a word boundary

Concurrent Writing

- Say x is a 32 bit word @ 0x12340002 ← not word aligned!
- Consider two threads, T1 and T2
  - T1: $x = $ 0xFFFFFFFF (i.e., $x = -1$)
  - T2: $x = 0$
- After T1 and T2 are done, x may be any of
  - 0, 0xFFFFFFFF, 0xFFFF0000, or 0X0000FFFF
- The outcome of concurrent write operations to a variable is undefined

Concurrent R/W

- Say x is a 32 bit word @ 0x12340002, initially 0 (not word aligned!)
- Consider two threads, T1 and T2
  - T1: $x = $ 0xFFFFFFFF (i.e., $x = -1$)
  - T2: $y = x$ (i.e., T2 reads x)
- After T1 and T2 are done, y may be any of
  - 0, 0xFFFFFFFF, 0xFFFF0000, or 0X0000FFFF
- The outcome of concurrent read and write operations to a variable is undefined

Data Race

- When two threads access the same variable…
- …and at least one is a STORE…
- …then the semantics of the outcome is undefined

Harmony's "sequential" statement

- sequential $turn, flags$
- Ensures that LOADs and STOREs are atomic
  - concurrent HVM operations appear to be executed sequentially, in the order in which they appear on each thread
  - this is the definition of sequential consistency
- Say x's current value is 3; T1 STOREs 4 into x; T2 LOADs x
  - with atomic LOAD/STORE, T2 reads 3 or 4
  - with modern CPUs/compilers, what T2 reads is undefined - e.g., Intel, ARM do not guarantee SC!

Sequential Consistency

- Java has a similar notion to Harmony's sequential
  - volatile int x
- Loading/Storing sequentially consistent variables is more expensive than loading/storing ordinary variables
  - it restricts CPU or compiler optimizations

So, what do we do?

Interlock Instructions

- Machine instructions that do multiple shared memory accesses (read/write) atomically
- TestAndSet s
  - returns the old value of s (LOAD r0,s)
  - sets s to True (STORE s, 1)
- Entire operation is atomic
  - other machine instructions cannot interleave

Harmony Interlude: Pointers

- If x is a shared variable, $?x$ is the address of x
- If p is a shared variable, and $p == ?x$, then we say that p is a pointer to x
- Finally, $!p$ refers to the value of x

Test-and-Set in Harmony

```
def test_and_set(s):
    atomically:
        result = !s
        !s = True
```

takes the address of s as input

- For example:

```
lock1 = False
lock2 = True
r1 = test_and_set(?lock1)
r2 = test_and_set(?lock2)
assert lock1 and lock2
assert (not r1) and r2
```

Recall: bad lock implementation

```
lockTaken = False

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        await not lockTaken
        lockTaken = True

        # Critical section
        cs: assert countLabel(cs) == 1

        # Leave critical section
        lockTaken = False

spawn thread(0)
spawn thread(1)
```

Test..

..and set

Test and set not atomic!!

A good implementation ("Spinlock")

```
lockTaken = False

def test_and_set(s):
    atomically:
        result = !s
        !s = True

def thread(self):
    while choose ( {False, True} ):
        # enter critical section
        while test_and_set(?lockTaken):
            pass

        cs: countLabel(cs) == 1

        # exit critical section
        atomically lockTaken = False

spawn thread(0)
spawn thread(1)
```

Same idea as before, but now with an atomic test&set!

Lock is repeatedly "tried", checking on a condition in a tight loop ("spinning")

Locks

- Think of locks as "baton passing"
  - at most one thread can "hold" False

Specifying (a Lock)

```
def Lock():
    result = False

def acquire(lk):
    atomically when not !lk:
        !lk = True

def release(lk):
    assert !lk
    atomically !lk = False
```

> atomically when x: y
> tests atomically x;
> when x is true,
> it atomically executes y

A specification describes an object, and the behavior of the methods that are invoked on it

- uses atomically to specify the behavior of these methods when executed in isolation

Specification

```
def Lock() returns result:
    result = False

def acquire(lk):
    atomically when not !lk:
        !lk = True

def release(lk):
    atomically:
        assert !lk
        !lk = False
```

What an abstraction does

Implementation*

*just one way to do it!

```
Def Lock()
    result = False

def test_and_set(s):
    atomically:
        result = !s
        !s = True

def atomic_store(var, val):
    atomically !var = val

def acquire(lk):
    while test_and_set(lk):
        pass

def release(lk):
    atomic_store(lk, False)
```

How the abstraction does it

Using a lock for a critical section

```
import synch

const NTHREADS = 2

lock = synch.Lock()

def thread():
    while choose({ False, True }):
        synch.acquire(?lock)
        cs: assert countLabel(cs) == 1
        synch.release(?lock)

for i in {1..NTHREADS}:
    spawn thread()
```

Spinlocks and Time Sharing

- Spinlocks work well when threads on different cores need to synchronize
- But what if two threads are on the same core?
  - when there is no preemption?
    - all threads may get stuck while one is trying to obtain the spinlock — BAD!!!
  - when there is preemption?
    - still delays and a waste of CPU cycles while a thread consumes a quantum trying to obtain a spinlock

Beyond Spinlocks

- We would like to be able to suspend a thread that is trying to acquire a lock that is being held
  - until the lock is ready
- A context switch!

Support for context switching in Harmony

- Harmony allows contexts to be saved and restored (i.e., enables a context switch)
  - r = stop p
    - stops the current thread and stores context in !p (p must be a pointer). If go is later invoked on that thread, then stop returns the value of r specified by go
  - go (!p) r
    - adds a thread with the given context (i.e., the one pointed by p) to the bag of threads. Threads resumes from stop expression, returning r

Lock specification using stop and go

```
import list

def Lock():
    result = { .acquired: False, .suspended: [] }

def acquire(lk):
    atomically:
        if lk→acquired:
            stop ?lk→suspended[len lk→suspended]
            assert lk→acquired
        else:
            lk→acquired = True

def release(lk):
    atomically:
        assert lk→acquired
        if lk→suspended == []:
            lk→acquired = False
        else:
            go (list.head(lk→suspended)) ()
            lk→suspended = list.tail(lk→suspended)
```

. acquired: boolean
. suspended: queue of contexts

add stopped context at the end of queue associated with lock

restart thread at head of queue and remove it from queue

Lock specification using stop and go

```
import list

def Lock():
    result = { .acquired: False, .suspended: [] }

def acquire(lk):
    atomically:
        if lk→acquired:
            stop ?lk→suspended[len lk→suspended]
            assert lk→acquired
        else:
            lk→acquired = True

def release(lk):
    atomically:
        assert lk→acquired
        if lk→suspended == []:
            lk→acquired = False
        else:
            go (list.head(lk→suspended)) ()
            lk→suspended = list.tail(lk→suspended)
```

Similar to Linux "futex": with no contention (hopefully the common case) acquire() and release() are cheap. With contention, a context switch is required

Choosing Modules in Harmony

- "synch" is the (default) module that has the specification of a lock
- "synchS" is the module that has the stop/go version of the lock
- You can select which one you want"
  - harmony -m synch=synchS x.hny
- "synch" tends to be faster than "synchS"
  - smaller state graph