# CSE 451: Operating Systems
# Spring 2010

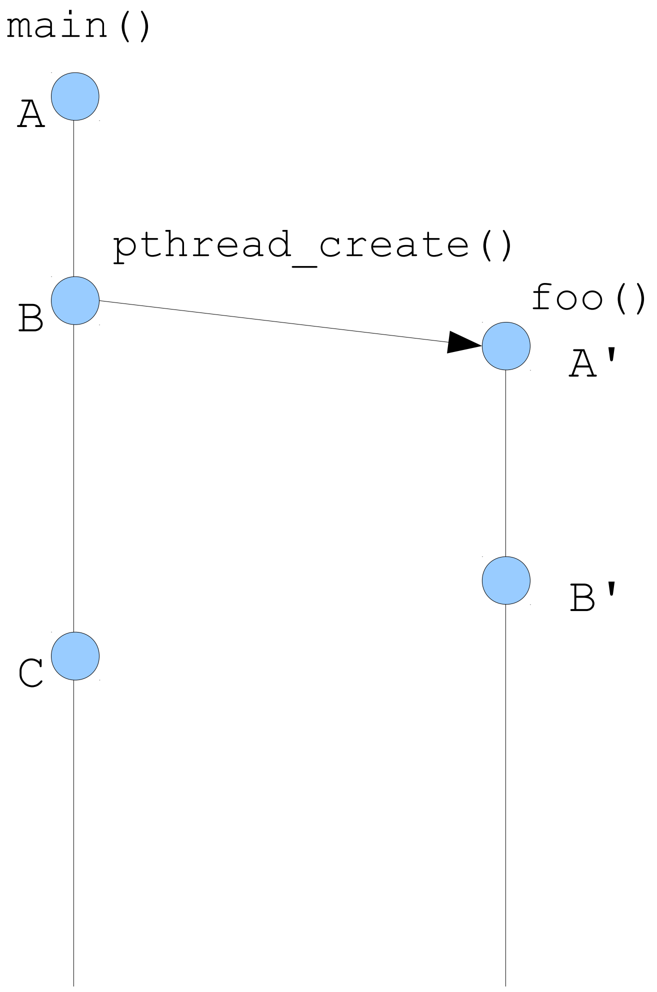
## Module 7
## Synchronization

John Zahorjan
zahorjan@cs.washington.edu
Allen Center 534

# Temporal Relations: Key Concept Review

- Instructions executed by a single thread are totally ordered
  - A < B< C < …
- Absent synchronization, instructions executed by distinct threads are simultaneous
  - (not A < A') and (not A' < A)
- A sequence of instructions is atomic if the effects of all of them appear to occur at once as viewed by any other (correctly operating) thread
- (Nearly all) single machine instructions are atomic
  - Write x
  - Read y

# Example: In the beginning...




*Y-axis is "time."*

*Could be one CPU, could be multiple CPUs (cores).*

- A < B < C
- A' < B'
- A < A'
- C == A'
- C == B'

# Critical Sections / Mutual Exclusion

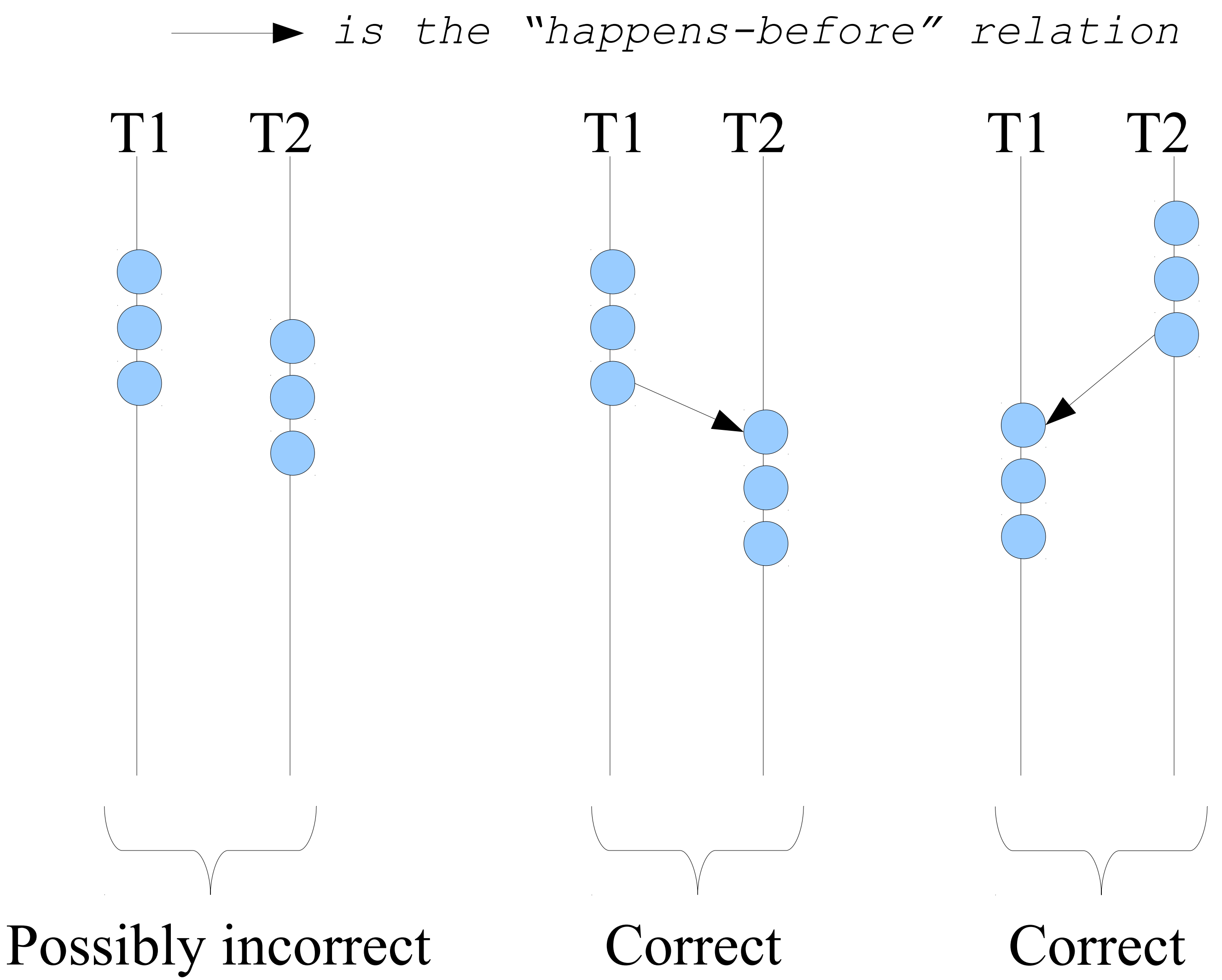
- Sequences of instructions that *may* get incorrect results if executed simultaneously are called critical sections

- Mutual exclusion means "not simultaneous"
  - (A < B) or (B < A)
  - We don't care which

- Forcing mutual exclusion between two critical section executions is sufficient to ensure correct execution
  - There are complicated code sequences that you'd think were critical sections, but aren't, but you don't want to be programming in units of complicated sections

- One way to guarantee mutually exclusive execution is using locks

# Critical sections



# When Do Critical Sections Arise?

- Well... the simple answer is "whenever simultaneous execution *could* result in incorrect answers," but that isn't very helpful
- One common pattern:
  - read-modify-write of
  - A shared value (variable)
  - In code that can be executed concurrently
    Note: There may be only one copy of the code (e.g., a procedure), but it can be executed by more than one thread at a time
- Shared variable:
  - Globals and heap allocated
  - NOT local variables
  - *Note: never give a reference to a stack allocated (local) variable to another thread (unless you're superhumanly careful...)*

# The classic example

- Suppose we have to implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {
  int balance = get_balance(account);     // read
  balance -= amount;                      // modify
  put_balance(account, balance);          // write
  return balance;
}
```

- Now suppose that you and your S.O. share a bank account with a balance of $100.00
  - what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?

- Assume the bank's application is multi-threaded
- A random thread is assigned a transaction when it is submitted

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

# Interleaved schedules

- The problem is that the execution of the two threads can be interleaved:

Time

```
balance = get_balance(account);
balance -= amount;
```

context switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
```

context switch

```
put_balance(account, balance);
```

- What’s the account balance after this sequence?
- How often is this sequence likely to occur?

# Aside: Other Execution Orders

- Which interleavings are ok? Which are not?

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

# How About Now?

```
int xfer(from, to, amt) {
  int bal = withdraw(from, amt);
  withdraw( to, -amt );
  return bal;
}
```

```
int xfer(from, to, amt) {
  int bal = withdraw(from, amt);
  withdraw( to, -amt );
  return bal;
}
```

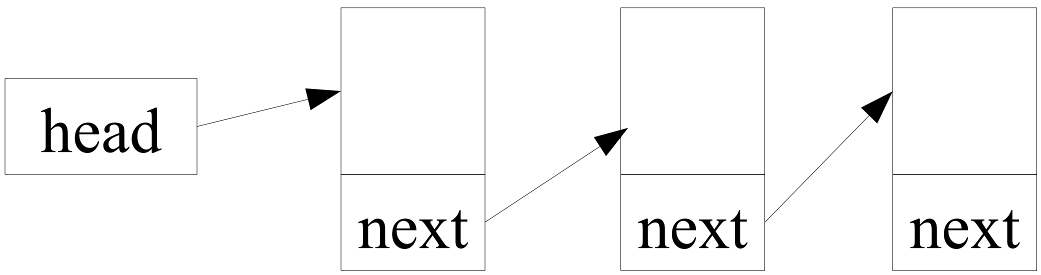
- Morals:
  - Interleavings are hard to reason about
    - We make a lot of mistakes
    - Control-flow analysis is hard for tools to get right
  - Identifying critical sections and ensuring mutually exclusive execution is... "easier"

# Another Classic Example

```
i++;
```

```
i++;
```

# Final Classic Example




```
for (p=head; p; p = p->next ) {
  <examine *p>
}
```

```
while (head) {
  oldHead = head;
  head = head->next;
  free(oldHead);
}
```

# "Critical section solution" requirements

- Critical sections have the following requirements
  - mutual exclusion
    - at most one thread is in the critical section
  - progress
    - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
  - bounded waiting (no starvation)
    - if thread T is waiting on the critical section, then T will eventually enter the critical section
      - assumes threads eventually leave critical sections
    - vs. fairness?
  - performance
    - the overhead of entering and exiting the critical section is small with respect to the work being done within it

# Mechanisms for building critical sections

- Locks (today)
  - very primitive, minimal semantics; used to build others
- Semaphores (tomorrow)
  - basic, easy to get the hang of, hard to program with
- Monitors (tomorrow)
  - high level, requires language support, implicit operations
  - easy to program with; Java "`synchronized()`" as an example
- Messages (day after tomorrow)
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems

# Locks, But First...

- A possible critical section solution is to arrange for all executions to occur on a single thread
  - E.g., use thread `n` where `n == account % #threads`
  - This turns a sharable variable into an un-shared variable
- Pros:
  - Simple
  - Fast
- Cons:
  - Load balancing among threads
  - What to do if the CS involves two accounts (e.g., xfer())?
  - Assigning tasks to threads probably involves a critical section (!)
- This idea is useful on multi-cores, and perhaps even more common in distributed systems

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

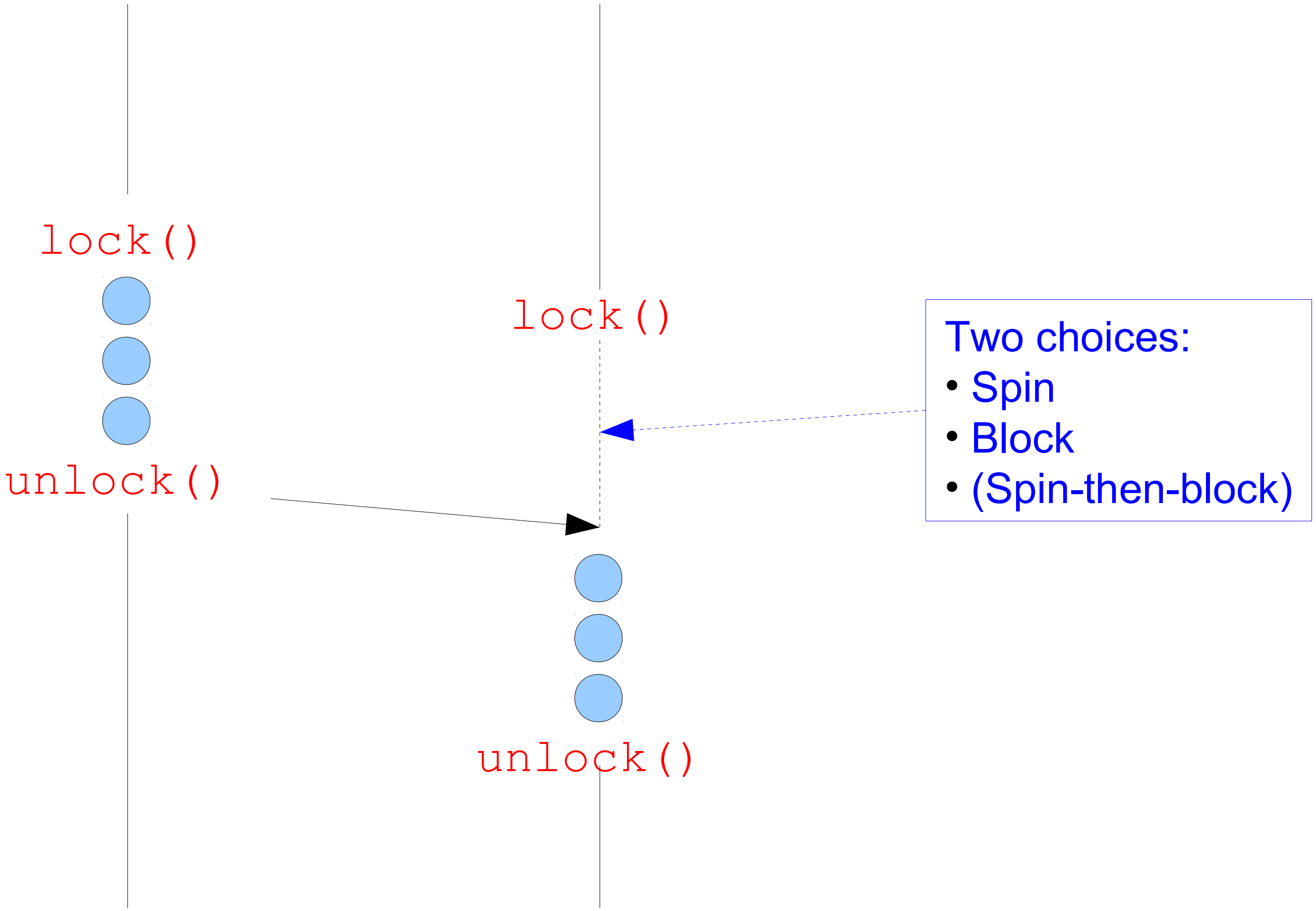
```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

# Locks

- Locks are memory objects with two operations
  - `acquire()`: obtain the right to enter the critical section
  - `release()`: give up the right to be in the critical section
- `acquire()` prevents progress of the thread until the lock can be acquired

*Note: terminology varies. In project 2, we use LOCK and UNLOCK for acquire/release, and "acquire" and "release" for memory allocation operations!*

# Locks: Example execution



# Using locks

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}
```

critical section

```
acquire(lock)
balance = get_balance(account);
balance -= amount;
```

```
acquire(lock)
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
```

- What happens when green tries to acquire the lock?
- Why is the “return” outside the critical section?
  - is this ok?

# Spinlocks

- How do we implement locks? Here's one attempt:

```
struct lock_t {
  int held = 0;
} lock;
void acquire(lock) {
   while (lock->held);
   lock->held = 1;
}
void release(lock) {
  lock->held = 0;
}
```

```
the caller "busy-waits",
or spins, for lock to be
released ⇒ hence spinlock
```

- Why doesn't this work?
  - where is the race condition?

# Implementing locks (cont.)

- Problem is that implementation of locks has critical sections, too!
  - the acquire/release must be **atomic**
    - atomic == executes as though it could not be interrupted
    - code that executes "all or nothing"
- Need help from the hardware
  - atomic instructions
    - test-and-set, compare-and-swap, …
  - disable/reenable interrupts
    - to prevent context switches

# Spinlocks redux: Hardware Test-and-Set

- CPU provides the following as one atomic instruction:

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

- Remember, this is a single instruction…

# Implementing Locks Using Test-and-Set

- So, to fix our broken spinlocks, do:

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while(test_and_set(&lock->held));
}
void release(lock) {
  lock->held = 0;
}
```

mutual exclusion?
progress?
bounded waiting?
performance?

# Reminder of use …

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}
```

critical section

```
acquire(lock)
balance = get_balance(account);
balance -= amount;
```

```
acquire(lock)
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
```

- How does a thread spinning in an “acquire” (that is, stuck in a test-and-set loop) yield the CPU?
  - calls yield( ) *(spin-then-block)*
  - there’s an involuntary context switch

# Problems with Locks

- Spinlocks work, but can be horribly wasteful!
  - if the thread holding the lock is not running, you'll spin for a scheduling quantum
    - Certainly the case on a single-core machine
  - `(pthread_spin_t)`

- Blocking locks work, but can be horribly wasteful!
  - If the lock is busy, there's a two context switch overhead cost to be paid to acquire it, minimum
    - The lock might be busy for only a few cycles, so it could have been cheaper to spin
  - `(pthread_mutex_t)`

- Spin-then-block locks
  - Spin for a little while (10's or 100's of cycles), then block
  - Why?
    - If you know the typical lock holding time is small, and it's been 100's of cycles, odds are the lock holder isn't currently running
      - This is an example of residual life that increases (steeply) after some short amount of time has elapsed

# Race Conditions

- Informally, we say a program has a race condition (aka "data race") if the result of an execution depends on timing
  - i.e., is non-deterministic
- Typical symptoms:
  - I run it on the same data, and sometimes it prints 0 and sometimes it prints 4
  - I run it on the same data, and sometimes it prints 0 and sometimes it crashes

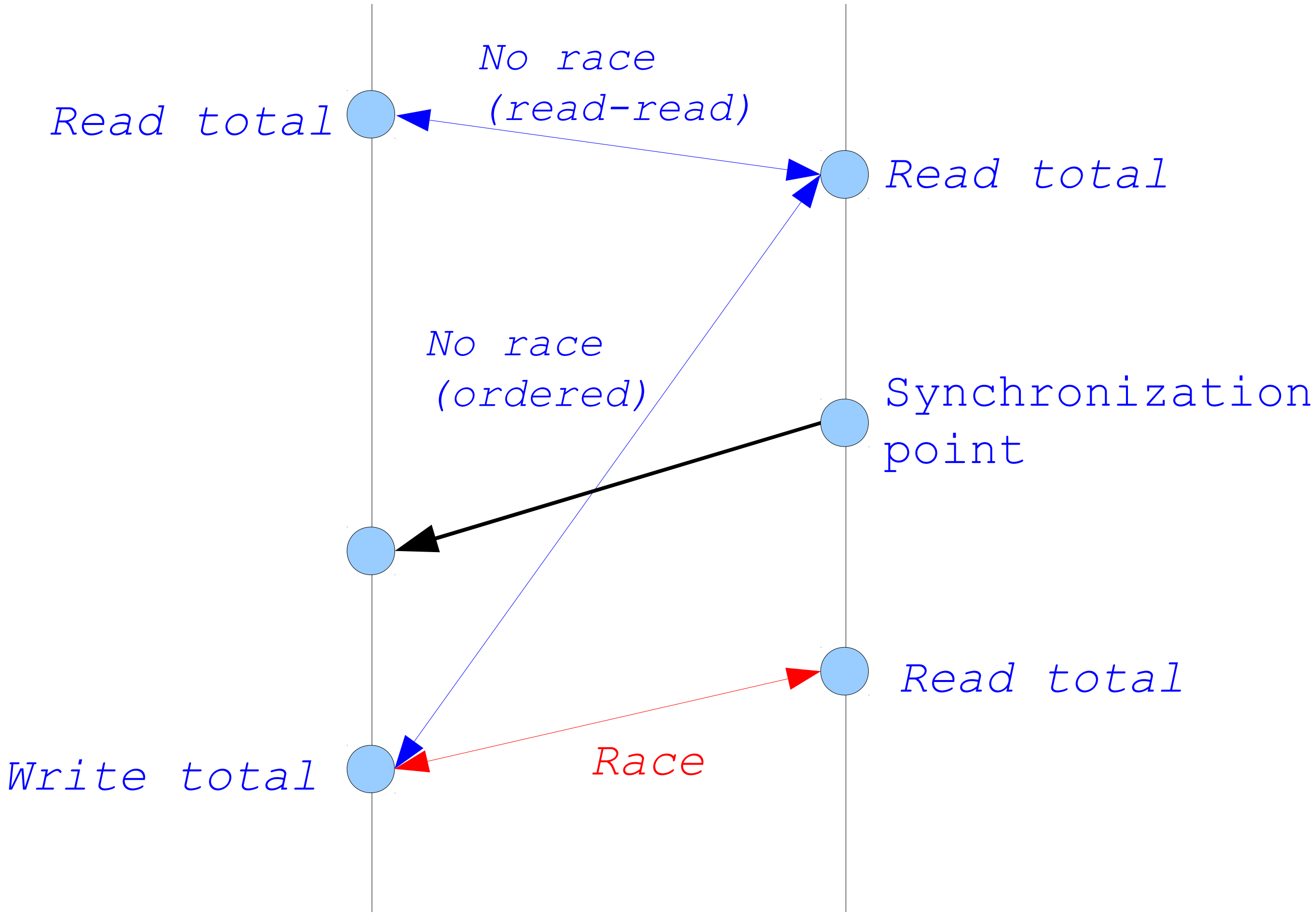
# Race Detectors

- There are tools that try to detect race conditions
  - We'll use one called `helgrind`
- They need a formal definition of what a race is
  - The definition varies, but the key is two accesses to a shared variable that are "simultaneous" (not ordered), at least one of which is a write
- Note: the formal definition can result in many false positives (detections of non-problems)
  - Example: two threads write 0 to shared variable `total`

# How They Work

- First of all, they're still kind of exotic / experimental / primitive
- Basically, they monitor thread executions to construct a "happens before" thread graph relating them
  - Happens-before arcs are introduced by things like locks, which they recognize as a call to pthread_mutex_lock()
- They then detect unsynchronized accesses by annotating each word/byte of memory with tags indicating where in the thread synchronization graph the operations arose
- They manage that by simulating the hardware instructions...
- They can be "a wee slow"

# Race Detection Example



# What's Next?

- Synchronization introduces temporal ordering
  - E.g., adds a "not simultaneous" edge
    - Critical sections
  - Or adds a "happens before" edge to the thread graph
    - Other kinds of synchronization
- Adding synchronization can eliminate races
  - That's handy!
- There are other synchronization primitives
  - For mutual exclusion
  - For "happens before"
- We'll have a look at some...