Thread Parallelism

Stephen Beard

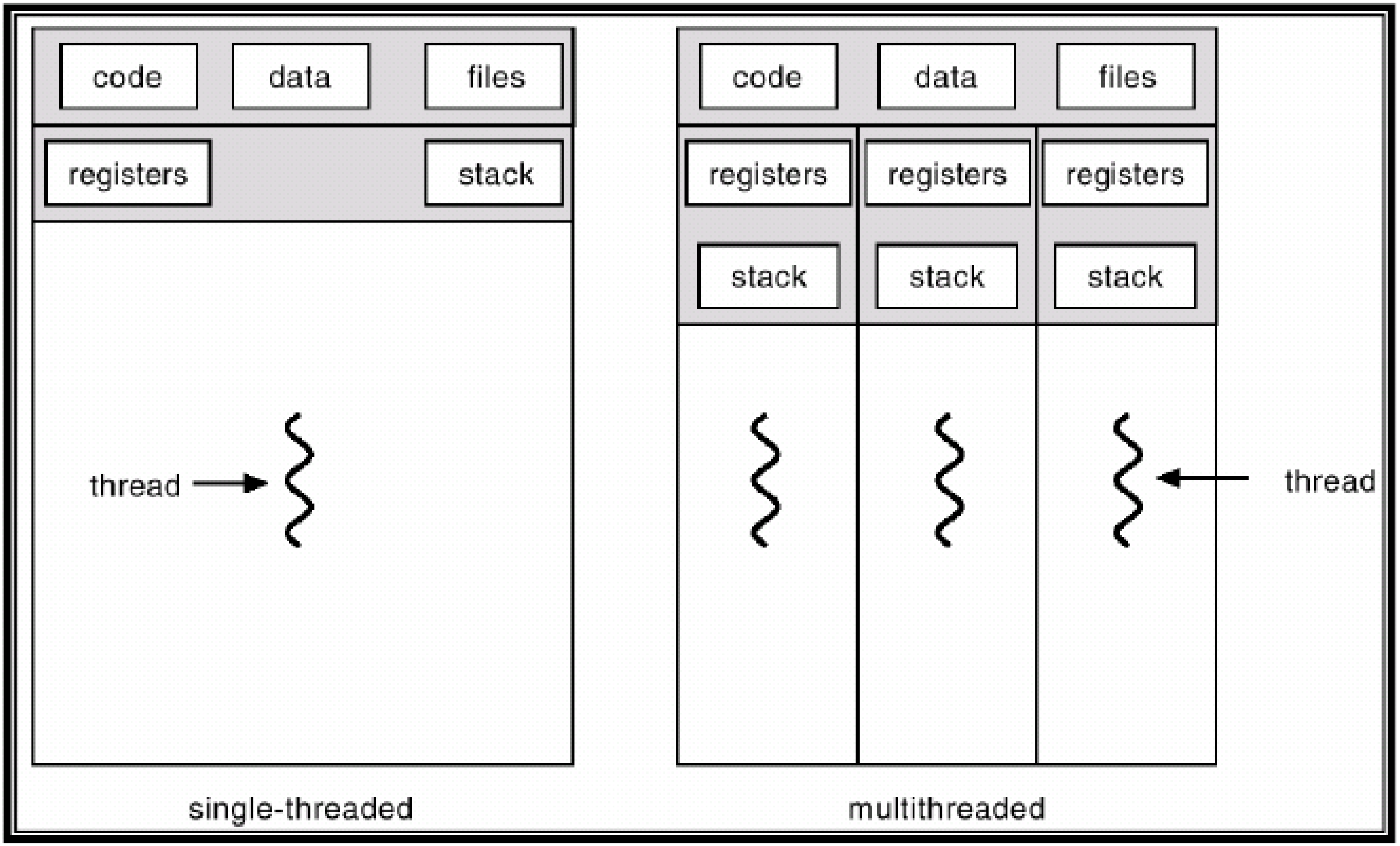
Lecture Outline

- Introduction to Threads
- Correctness
- Performance

Introduction to Threads

What is a thread?

What is a thread?



Threads vs. Processes

(Generalities)

Process

- "Heavyweight"
- Slower context switches
- Expensive IPC
- Independent
- Secure
  - Protected memory space

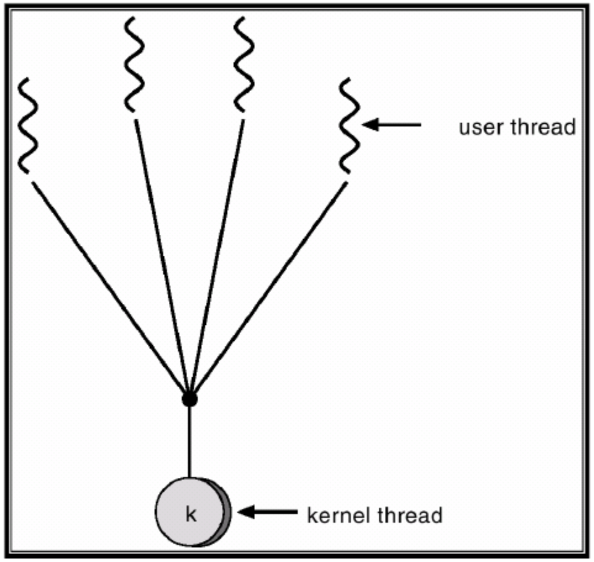
Thread

- "Lightweight"
- Faster context switches
- Direct communication
- Share state and resources
- Insecure
  - Shared memory space

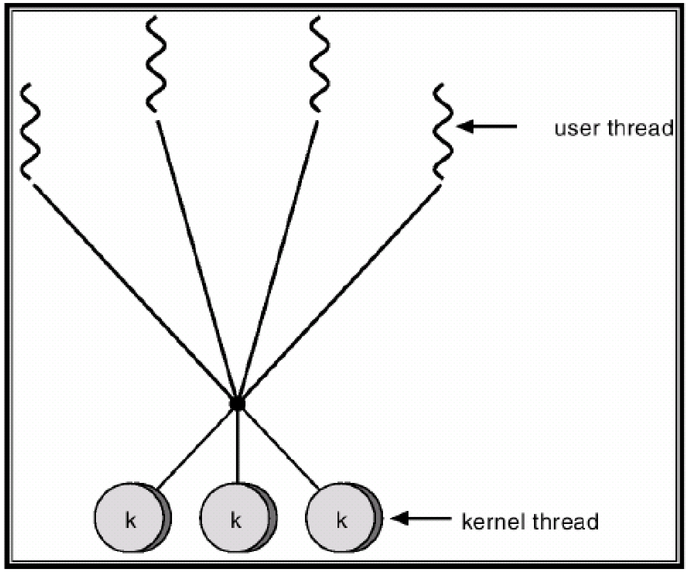
User Threads and Kernel Threads

- User Thread
  - Implemented in software library
  - Transparent to the OS
  - Will block other threads
  - Library typically uses non-blocking calls then manages threads
  - Fast to create and manage
  - Do not benefit from multithreading or multiprocessing
- Kernel Thread
  - Managed by OS
  - Will not block other threads
  - Slower to swap than user threads

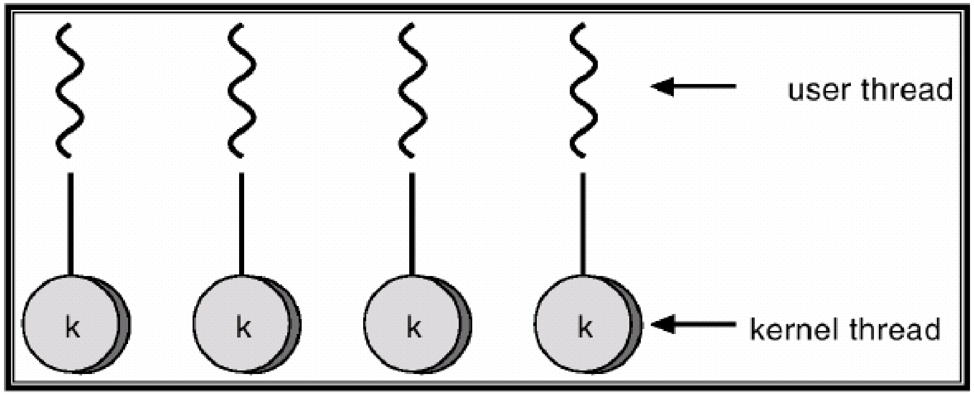
THREAD IMPLEMENTATIONS



Many to One



Many to Many

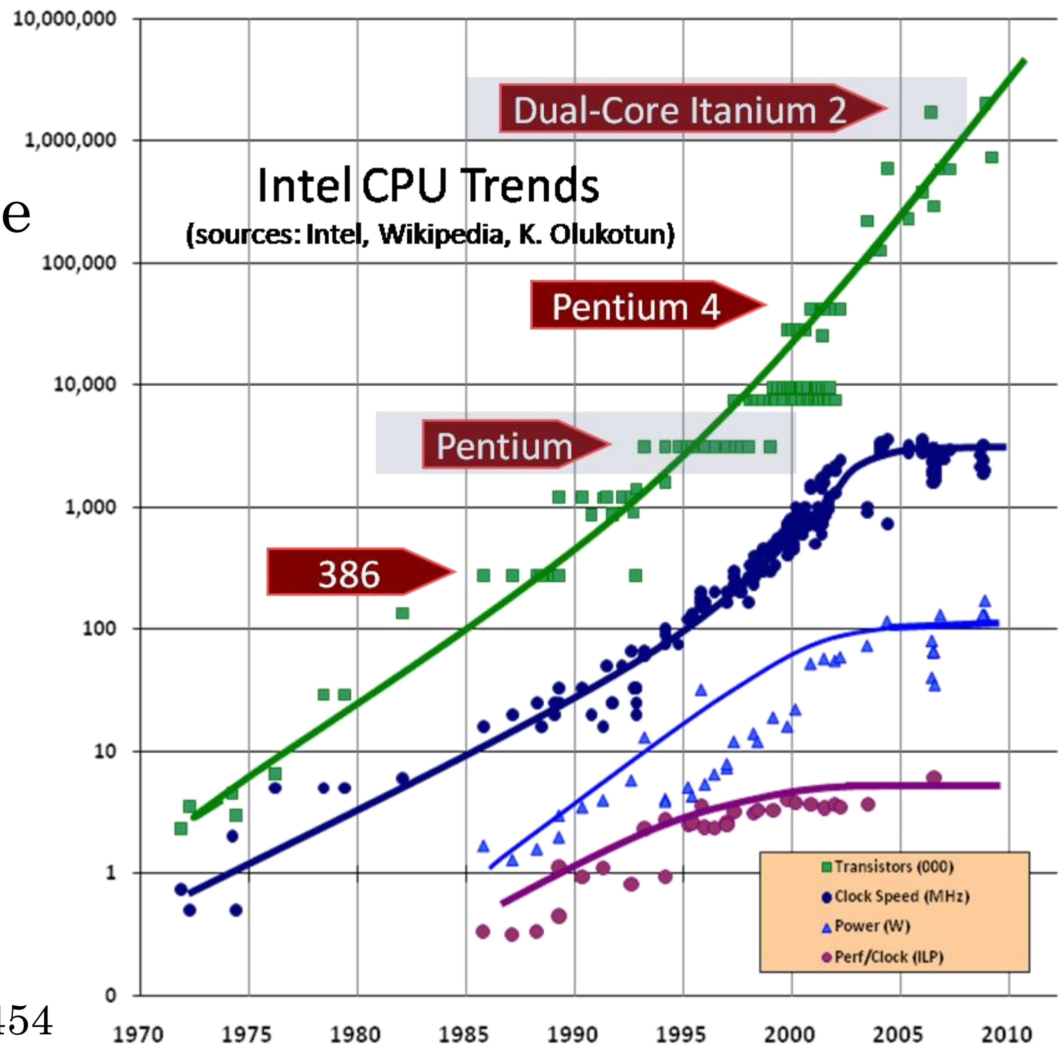


One to One

Why use threads?

WHY USE THREADS?

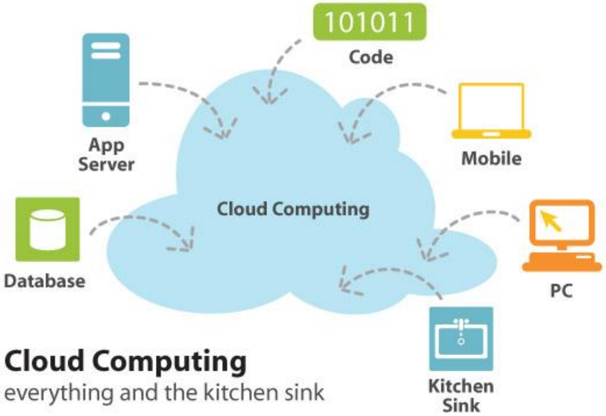
- Interactive Programs – Avoid blocking!
- Modern Hardware is designed for thread level parallelism (TLP)



Source: Tom Ball - PPCP-54454

Hardware for TLP

- Chip Multi-Processors
- GPUs
- Clusters
- Cloud Computing
- Multithreading

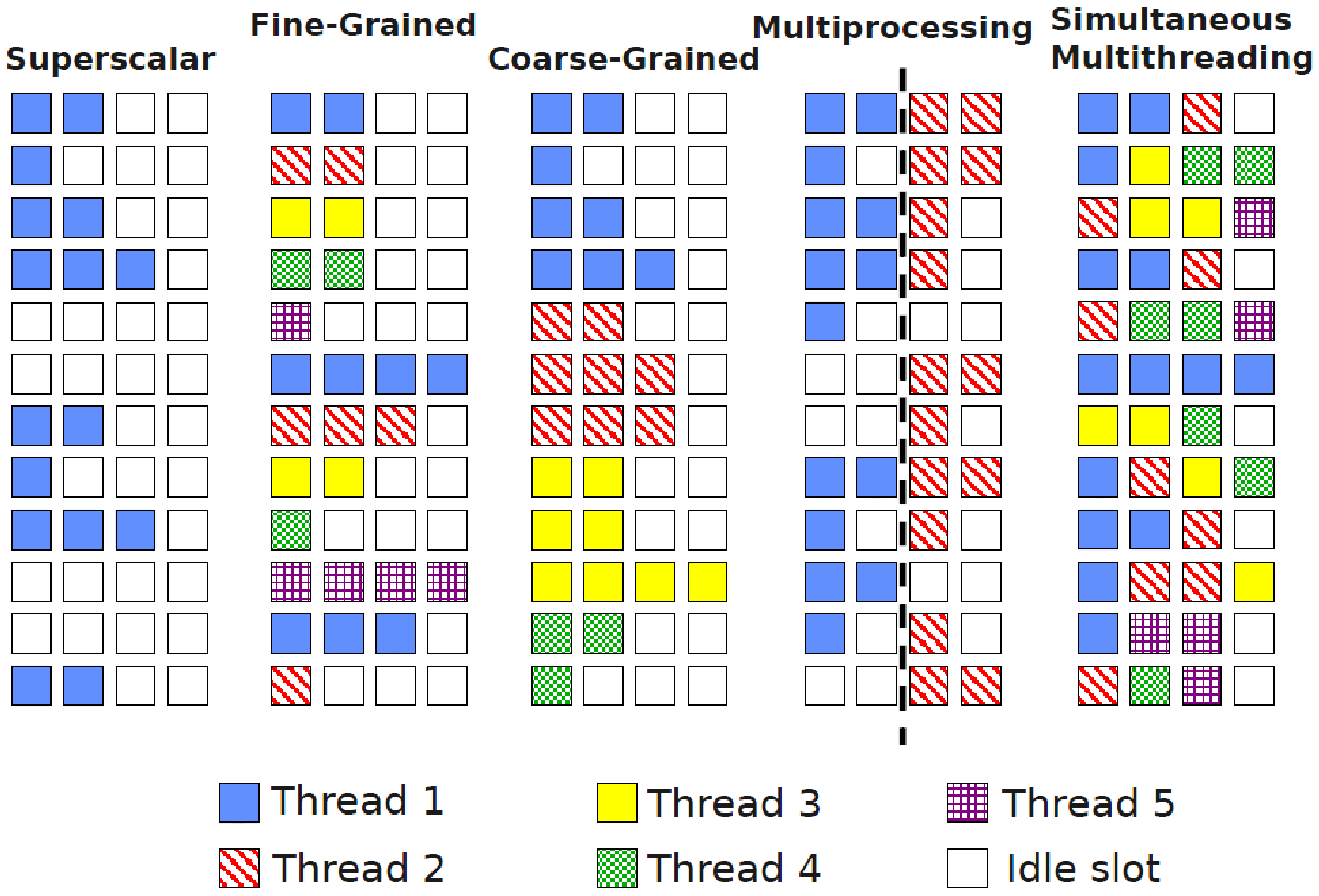


Multi-Threading Terms

- Superscalar – ILP mechanism for performing multiple instructions concurrently (One CPU with multiple functional units)
- Fine-Grained – Switch between threads on each cycle
- Coarse-Grained – Switch between threads on 'costly' stalls (such as L2 cache miss)
- Multiprocessing – Multi-core
- Simultaneous – Multiple threads running concurrently on single processor

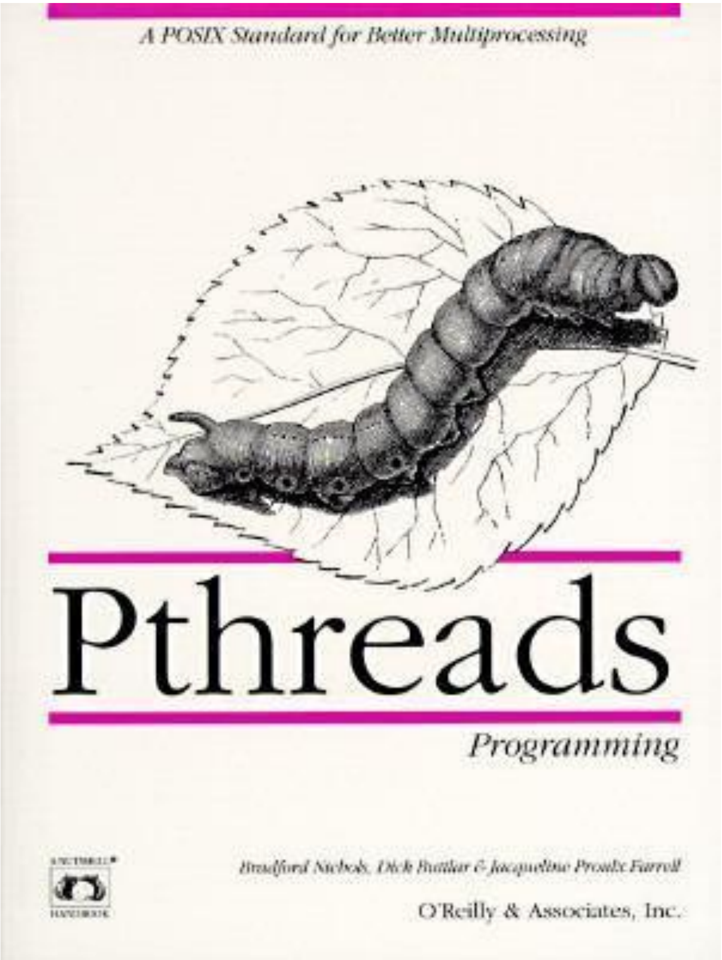
Multithreading

Ex: Intel Pentium 4 | Sun UltraSPARC | Intel Itanium 2 | Intel Core 2 Duo | Intel Hyper-Threading



Source: Dr. Chris Lupo – CPE520 Advanced Computer Architecture Winter 2010

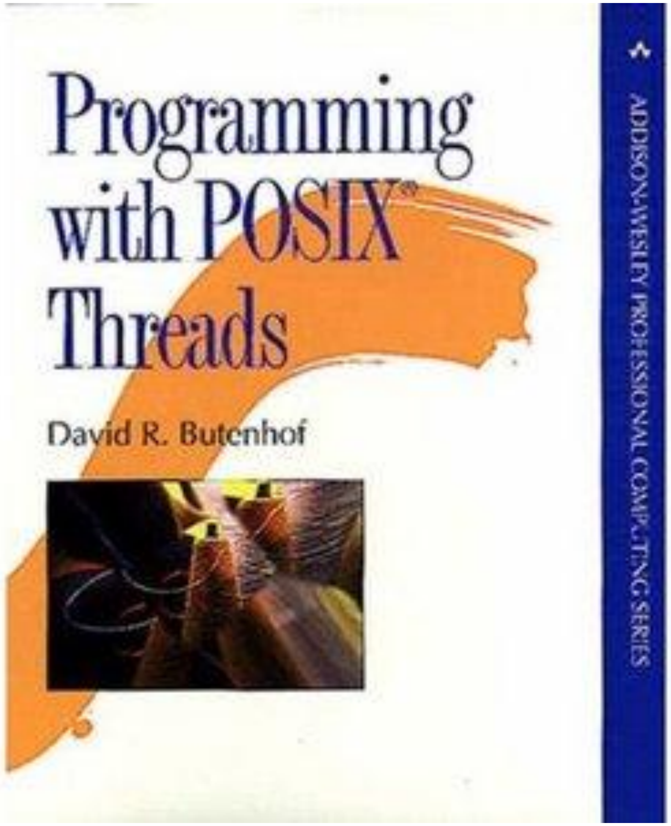
Pthreads


A POSIX Standard for Better Multiprocessing
Pthreads
Programming
O'Reilly & Associates, Inc.

PTHREADS (POSIX THREADS)

- C library that provides
  - Thread management
  - Shared Memory
  - Locks

- In Linux
  - One to One
  - Created using 'clone'

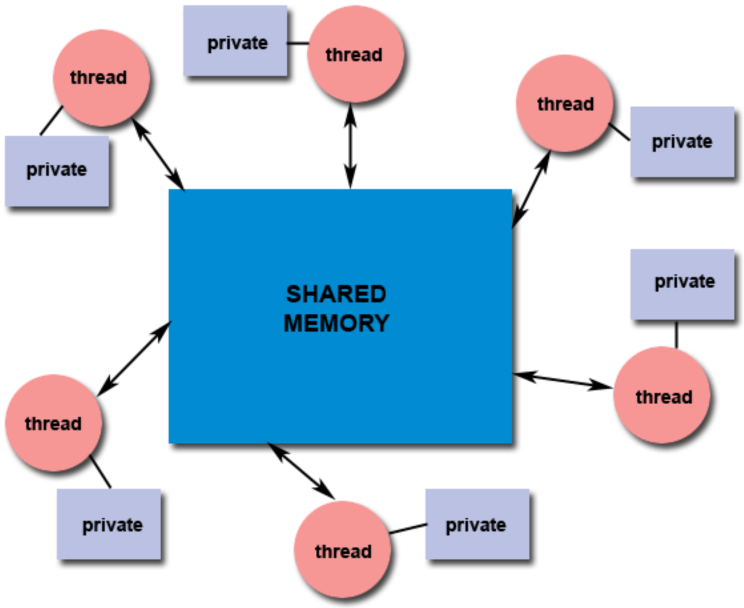


Simple Pthread Example

Methods of Thread Communication

Shared Memory -Memory that may be simultaneously accessed by multiple threads

```
int gInt;
spawn t1, t2;
```




```
t1:          t2:
…            …
gInt = 5     …
…            int lInt = gInt
             print lInt -> 5
             …
```

Methods of Thread Communication

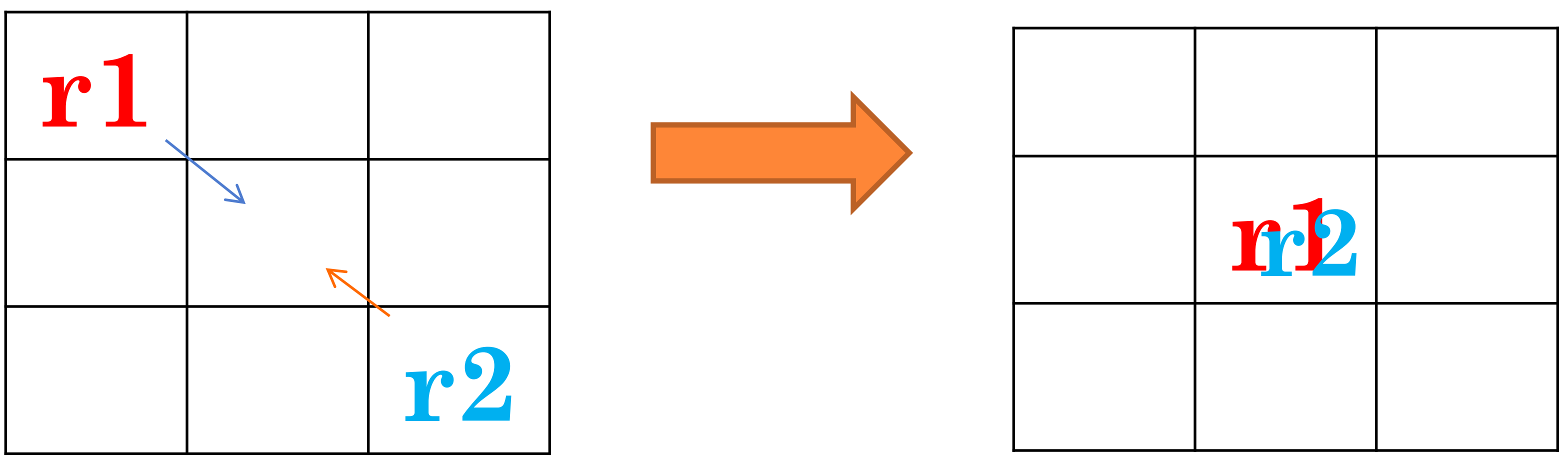
Message Passing - Threads pass messages for data transfer and synchronization

```
t1:                              t2:
send 5  ──────────────┐          …
                      │          …
                      └────────> recv lInt
                                 Print lInt -> 5
```

Thread Correctness

Race Conditions

- Unsynchronized access to shared state from multiple threads whose outcome depends upon the order of access
- r1.check, r2.check, r1.move, r2.move, **CRASH**



Source: Tom Ball - PPCP-54454

Race Condition Program

Synchronization

- Want to be able to control access to shared memory

- Several methods exist:
  - Mutex
  - Semaphore
  - Monitors
  - Barriers

Naively Fixing our Robots

Robot 1	Robot 2
lock()	lock()
r1.check()	…
unlock()	…
…	r2.check()
…	unlock()
lock()	lock()
r1.move()	…
unlock()	…
	r2.move()
	unlock()

CRASH

Atomicity

- A statement sequence S is atomic if S's effects appear to other threads as if S executed without interruption

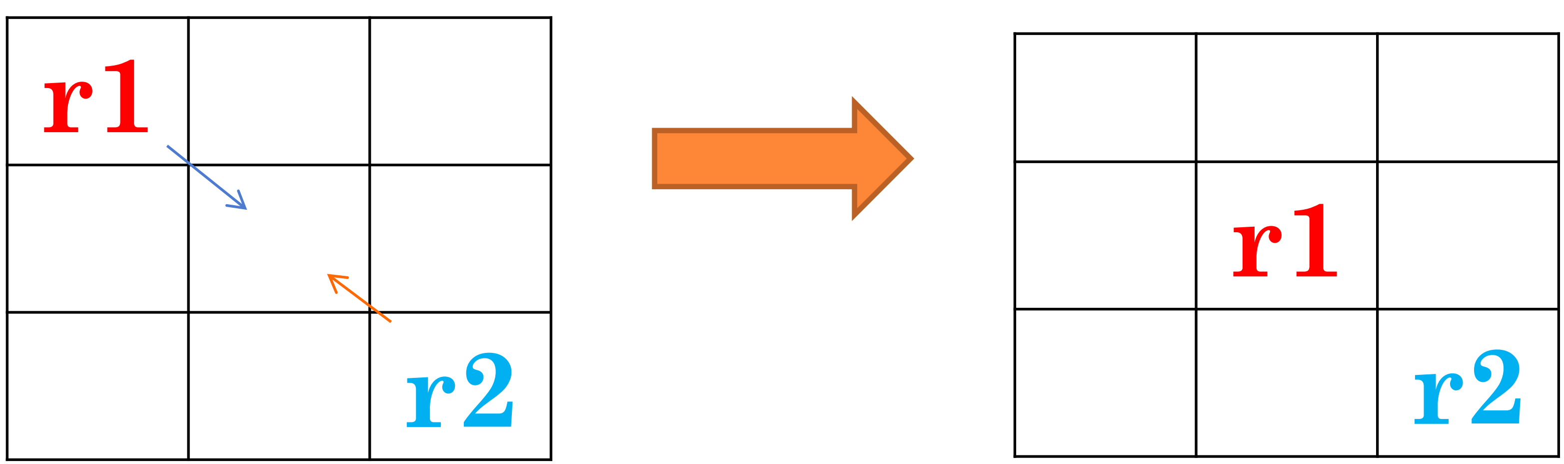
Fixing our Robots

Robot 1

lock()
r1.check()
r1.move()
unlock()

Robot 2

lock()
…
…
…
r2.check()
unlock()



Mutex Example

Mutex Implementation - Hardware

- Using XCHG on x86 to implement a mutex
  - XCHG exchanges two operands. If a memory operand is involved, BUS LOCK is asserted for the duration of the exchange.

```
LOCK:        ; mutex pointer is in EBX; clobbers EAX
    XOR EAX, EAX     ; Set EAX to 0
    XCHG EAX, [EBX]
    AND EAX, EAX     ; Test for 1
    JZ LOCK          ; if we got a zero, spin-wait
    RET

UNLOCK:      ; mutex pointer is in EBX
    MOV [EBX], 1
    RET
```

MUTEX IMPLEMENTATION - SOFTWARE

- Peterson's Algorithm
  - Works for two processes, but can generalize
  - Does not work with out-of-order execution

```
flag[0]   = 0;
flag[1]   = 0;

P0: flag[0] = 1;
    turn = 1;
    while (flag[1] == 1 && turn == 1)
    {
           // busy wait
    }
    // critical section
       ...
    // end of critical section
    flag[0] = 0;
```

```
P1: flag[1] = 1;
    turn = 0;
    while (flag[0] == 1 && turn == 0)
    {
        // busy wait
    }
    // critical section
       ...
    // end of critical section
    flag[1] = 0;
```

Mutex Implementation

- Exact locking mechanism is hardware dependent
- If a thread fails to acquire lock
  - Waits for lock
  - Spin vs Yield
- How to handle multiple threads waiting on single lock
  - Queue
  - Scheduler
- Reentrant Locks
  - Allowed to acquire same lock multiple times
  - Must be released same number of times

Other Issues with Locks

- Dead-lock – Circular waiting on locks
- Live-lock – Locks state changing with no progress
- Lock contention – Many threads require access to single lock
- Lock overhead – Locking mechanisms are slow
- Priority Inversion – Low priority thread holds lock, prevents progress of high priority
- Convoying – Lock contention with slowest threads acquiring the lock first

Performance

Thread Granularity

Thread Granularity

- Better to have lots of threads doing a little work or a few threads doing lots of work?

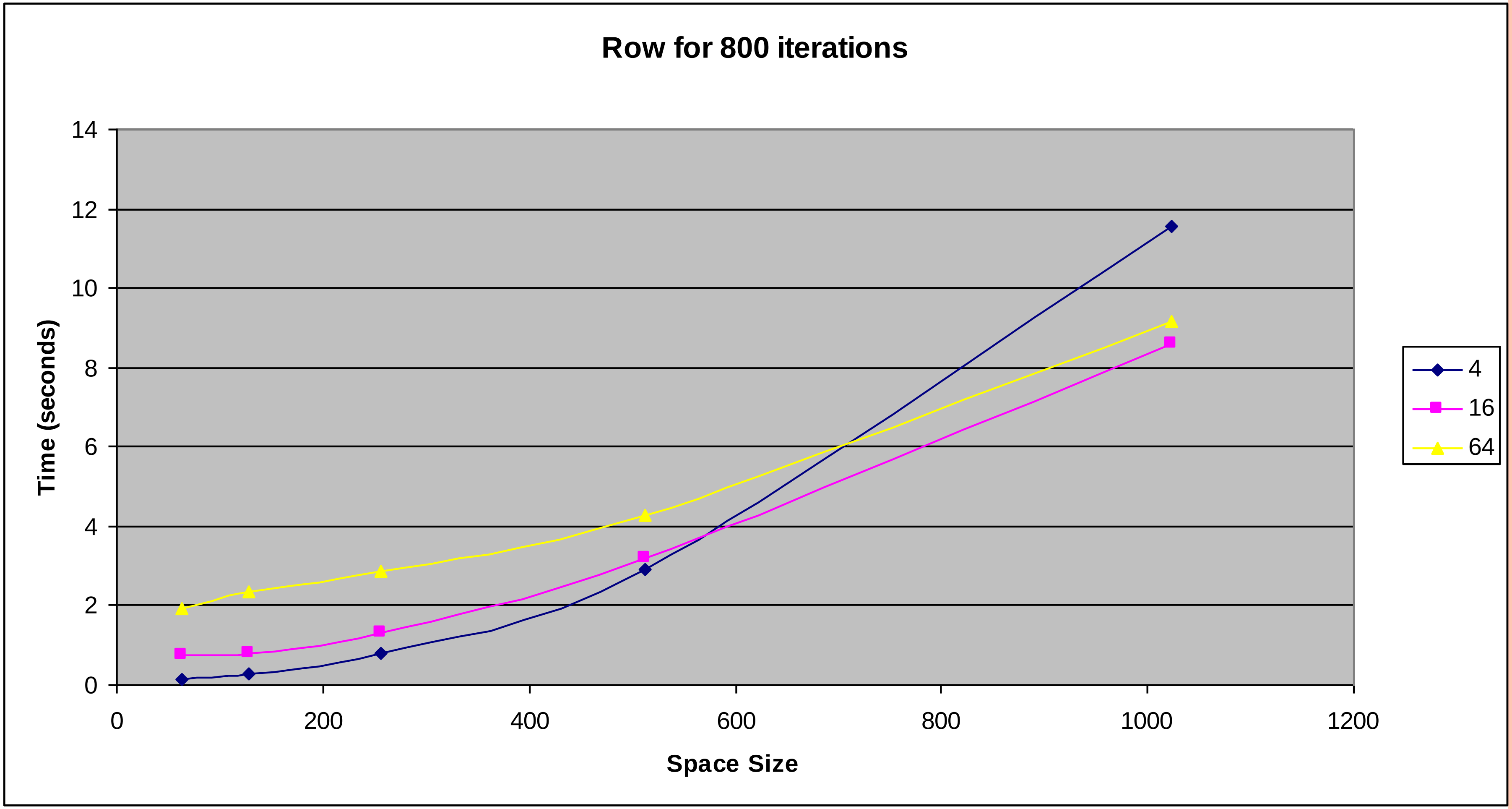
  - Depends on:
    - How much communication overhead will result?
    - Implementation of threads
    - Hardware

Jacobi Iterations

- For a matrix, on each iteration element's new value = average of neighbors old values

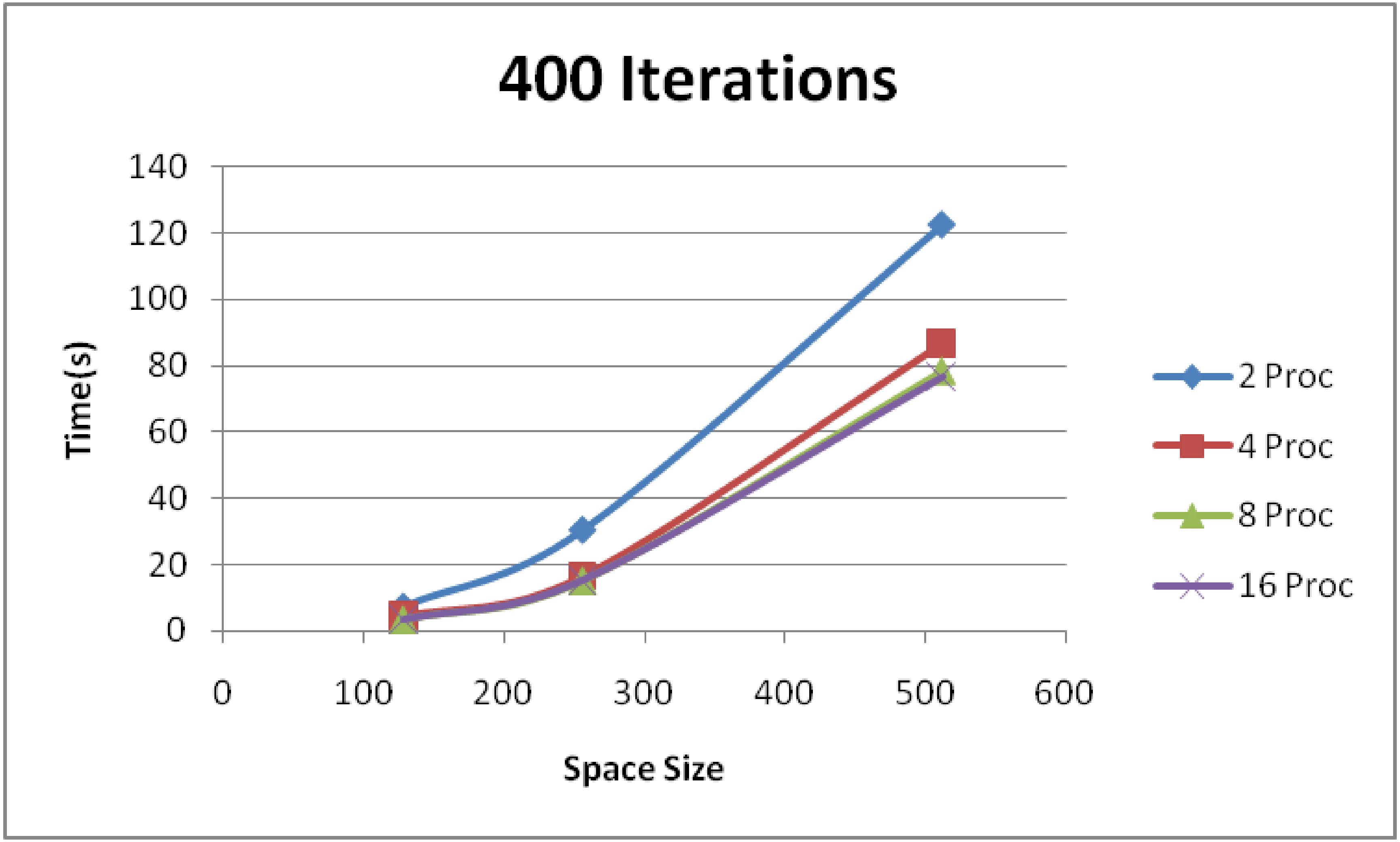
- How many threads?

Jacobi in C using MPI



Erlang MPI

Jacobi in Erlang



Locking Granularity

Locking Granularity

- Better to lock the entire structure, or parts?
  - Lock entire list when performing an operation
    - Only alter one lock per access to list
    - One thread in list blocks all others from accessing list
  - Lock each element of the list, hand-over-hand
    - Threads can work on different parts of the list concurrently
    - Lock per element, or group of elements
    - Threads in front of list prevent access to rest of list

Lock Free Data Structures

Lock-Free Algorithms

- Can be more efficient and scalable than locking
- Not the same as wait-free
  - Lock-free guarantees system progress
  - Wait-free guarantees thread progress
    - Operation must have bound on number of steps till completion
    - Very rare as their performance is generally low
- Good for many reads, few writes
  - Most attempt operation then retry if changed occurred during operation

Compare-and-Swap CMPXCHG on x86

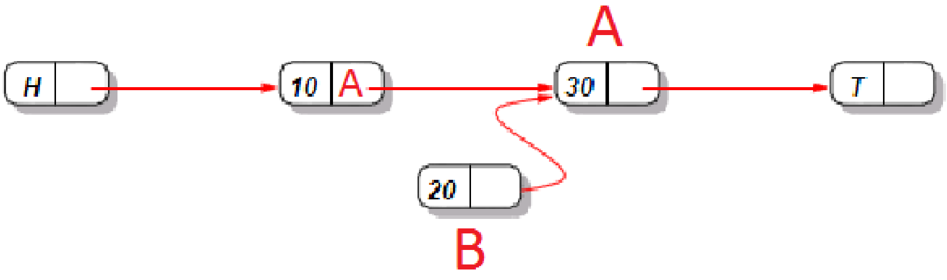
- Atomically compares contents of memory location to a given value, if they match it updates value

```
int compare_and_swap ( int* register, int oldval, int newval)
{
  int old_reg_val = *register;
  if (old_reg_val == oldval)
     *register = newval;
  return old_reg_val;
}
```

- Hardware support handles this operation atomically
- Integral in lock free structures

Lock-Free Linked List – Insertion

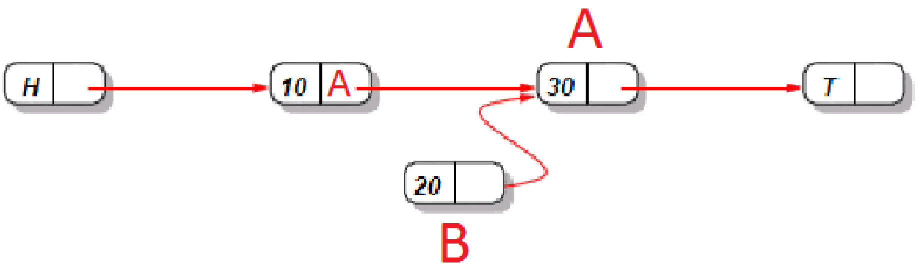
- Create new node
- do
  - Find insertion location, note left and right nodes



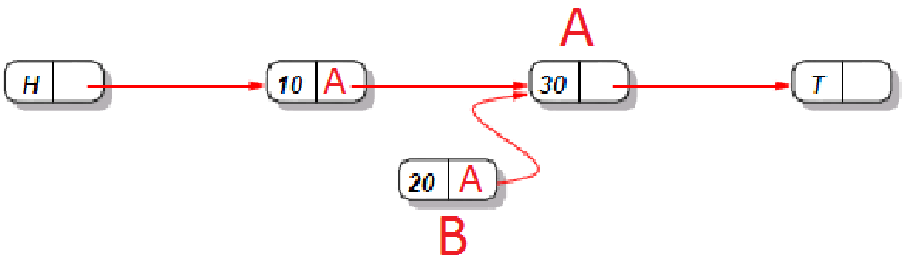
Harris, "A pragmatic implementation of non-blocking linked-lists", 2001 (15th International Symposium on Distributed Computing)

Lock-Free Linked List – Insertion

- Create new node
- do
  - Find insertion location, note left and right nodes



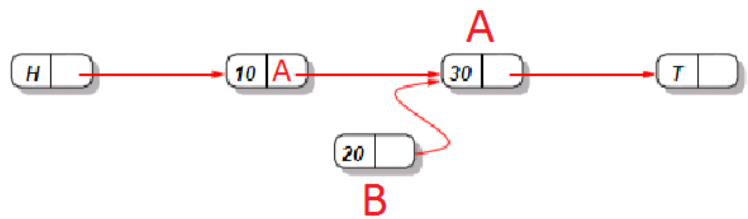
  - Set new.next = right



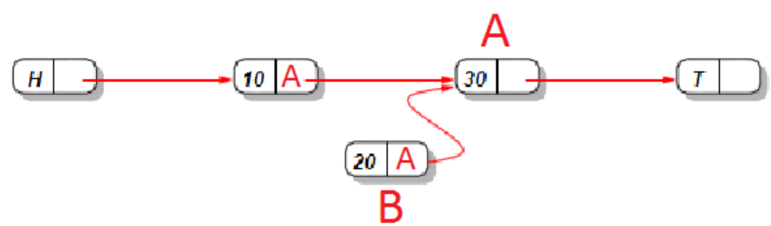
Harris, "A pragmatic implementation of non-blocking linked-lists", 2001 (15th International Symposium on Distributed Computing)

Lock-Free Linked List – Insertion

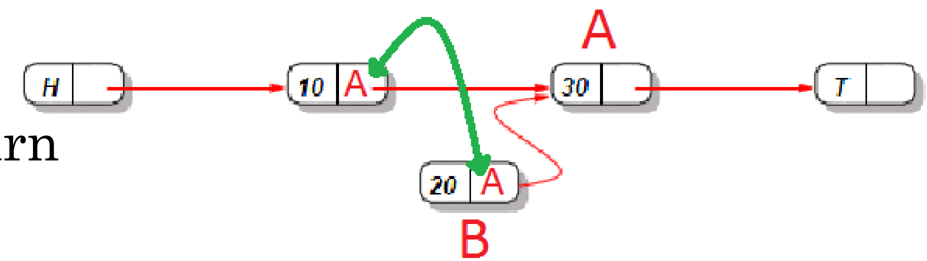
- Create new node
- do
  - Find insertion location, note left and right nodes



  - Set new.next = right



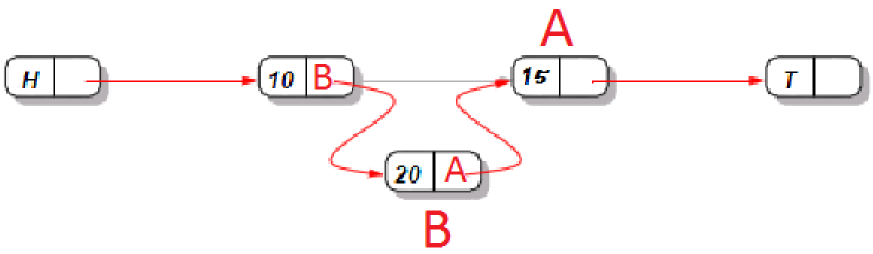
  - If(CAS &left.next, right, new) then return



Harris, "A pragmatic implementation of non-blocking linked-lists", 2001 (15th International Symposium on Distributed Computing)

Lock-Free Linked List – Insertion

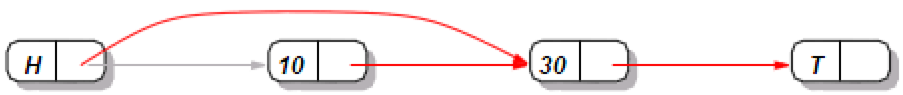
- Create new node
- do
  - Find insertion location, note left and right nodes
  - Set new.next = right
  - If(CAS &left.next, right, new) then return
- while(true)



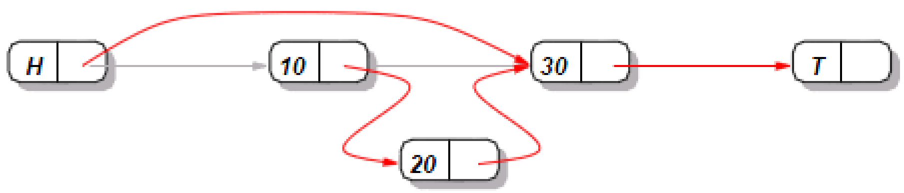
Harris, "A pragmatic implementation of non-blocking linked-lists", 2001 (15th International Symposium on Distributed Computing)

Lock-Free Linked List

- Delete creates problems
  - Naive Delete



  - Fails for concurrent insert

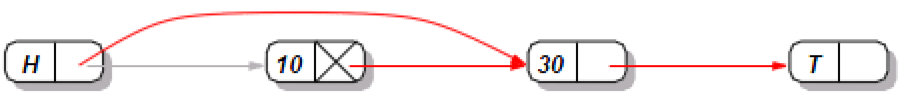


Harris, “A pragmatic implementation of non-blocking linked-lists”, 2001 (15th International Symposium on Distributed Computing)

Lock-Free Linked List

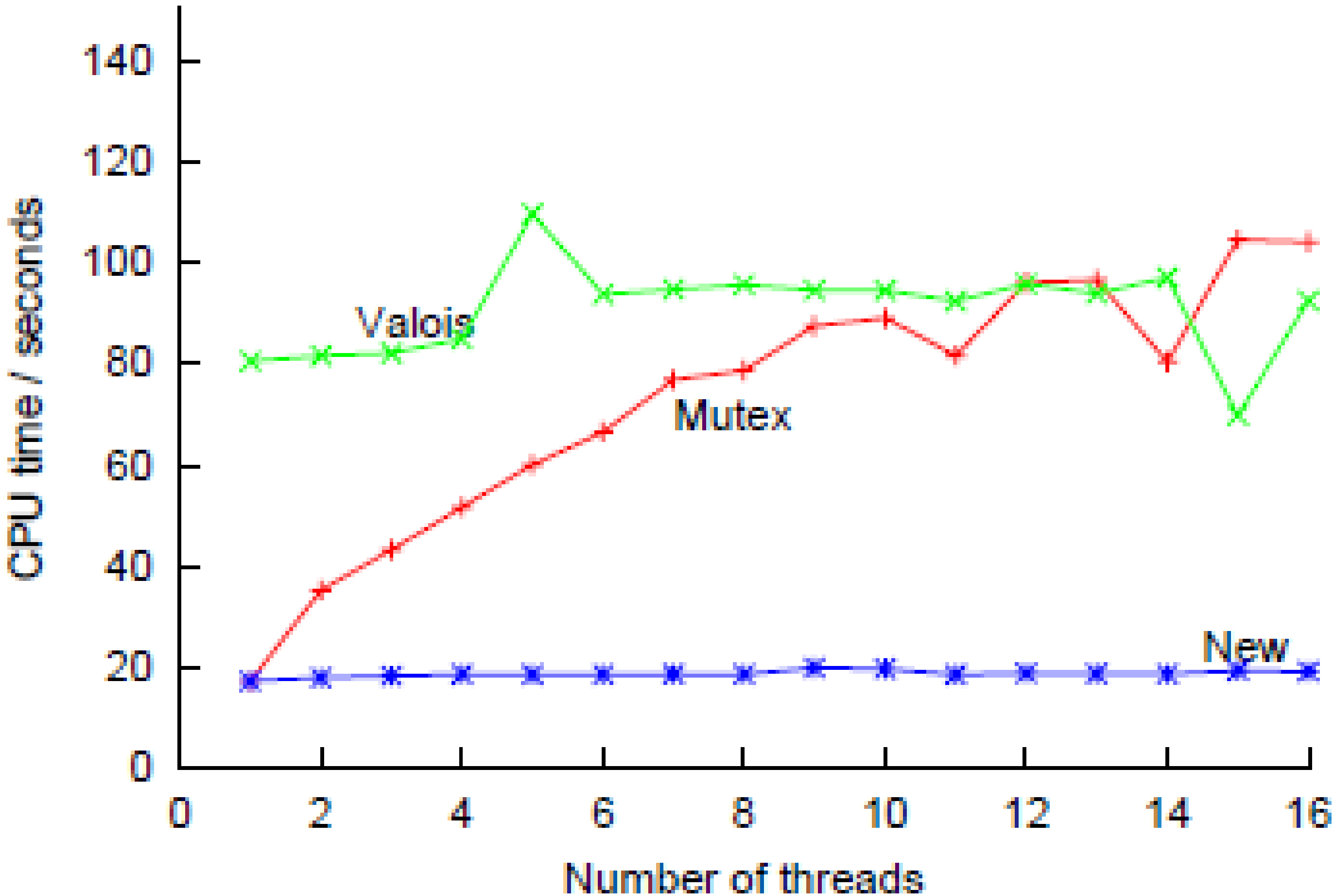
- Correct delete requires two compares
  - First mark deleted node as 'logically deleted'

  - Then 'physically delete' the node



Harris, "A pragmatic implementation of non-blocking linked-lists", 2001 (15th International Symposium on Distributed Computing)

Performance of Lock-Free Linked List



1 million random insertion, deletions on keys 0 - 8191

Harris, "A pragmatic implementation of non-blocking linked-lists", 2001 (15th International Symposium on Distributed Computing)

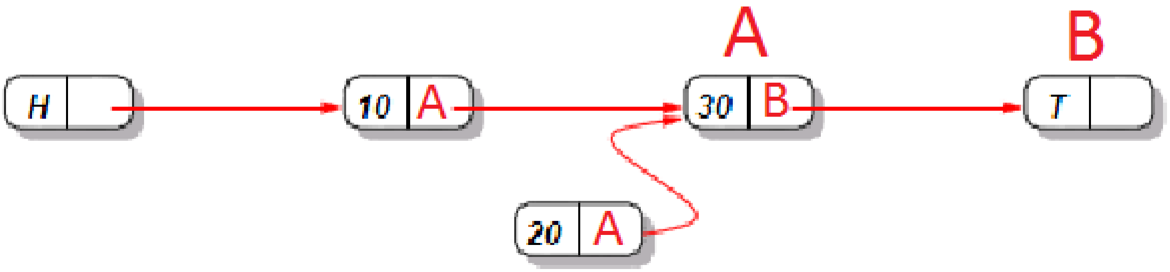
Lock-Free ABA Problem – 1

Thread 1:

`Insert 20 #interupted`

Thread 2:

`...`



ABA Problem – 2

Thread 1:

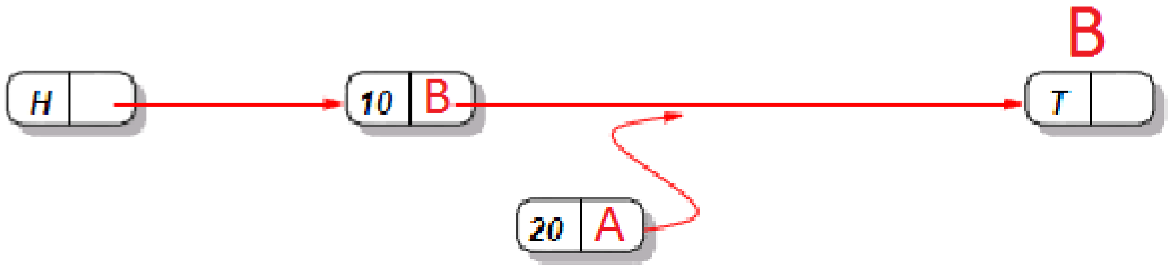
`Insert 20` `#partial completion`

`...`

Thread 2:

`...`

`delete 30` `address A`



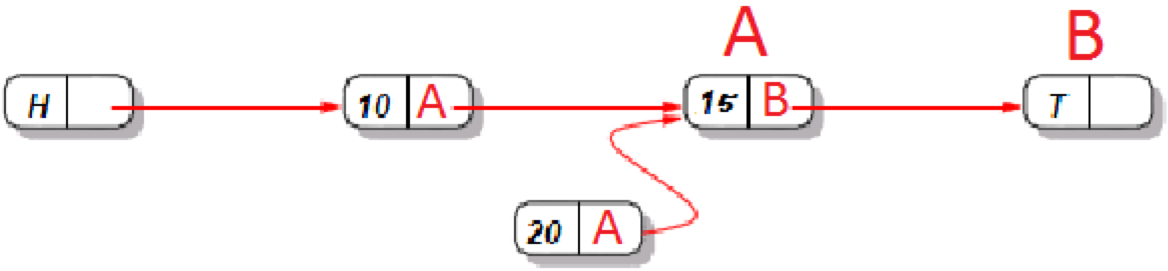
ABA Problem – 3

Thread 1:

```
Insert 20 #partial completion
...
...
```

Thread 2:

```
...
delete 30 #address A
insert 15 #address A
```



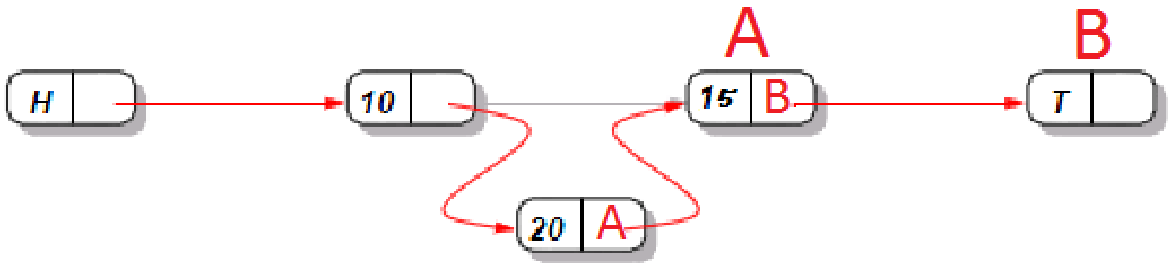
ABA Problem – 4

Thread 1:

```
Insert 20 #partial completion
...
...
Insert 20 #finishes and
          #improperly succeeds
```

Thread 2:

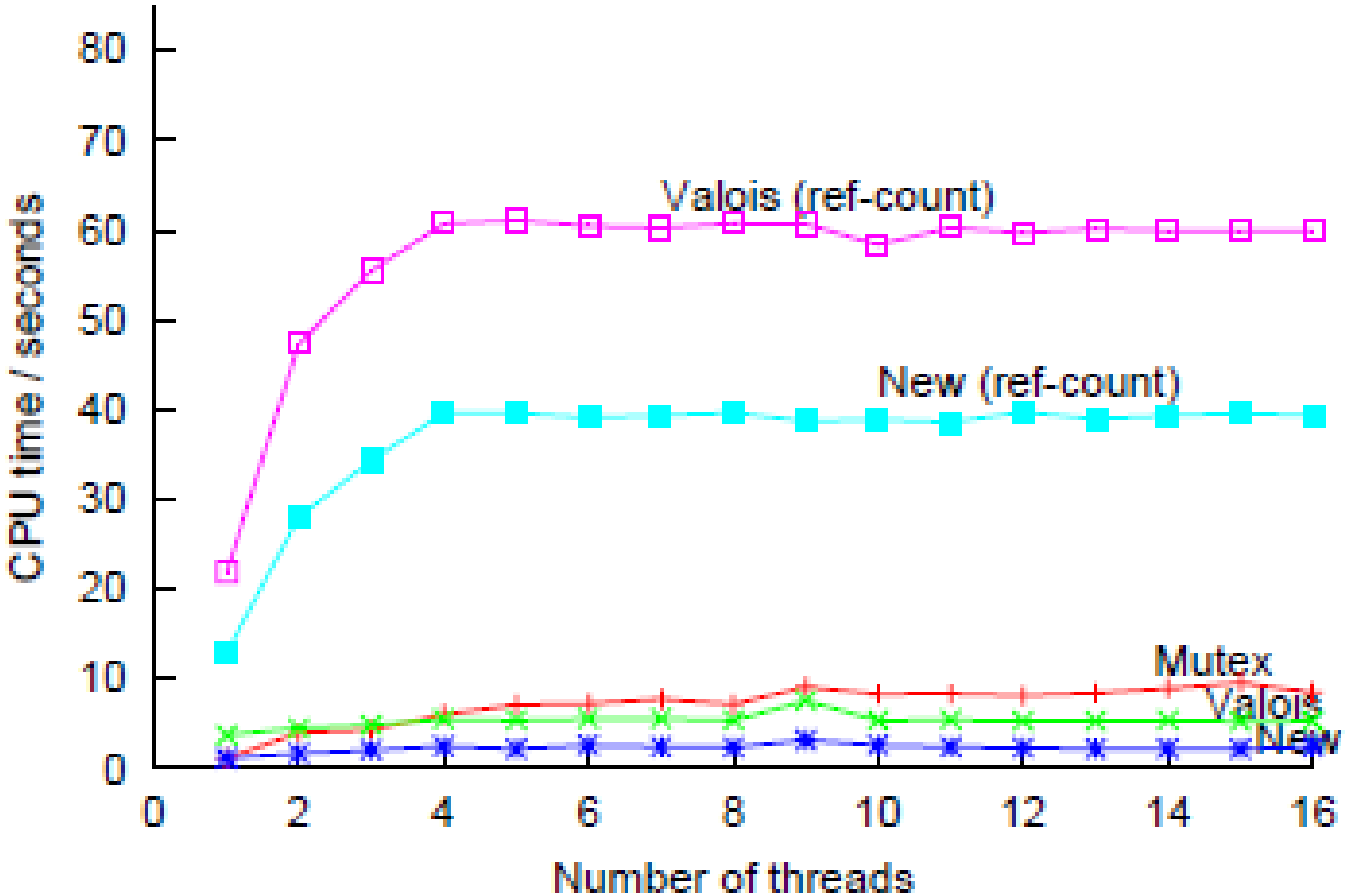
```
...
delete 30 #address A
insert 15 #address A
```



Solutions to ABA

- Keep "tag" bits on each pointer – ABA'
  - Requires double-word CAS
- Use reference counts on cells (Valois)
  - Only reuse cell when reference count = 0
- Use 'Load Linked' and 'Store Conditional'
  - LL returns value of memory location
  - SC stores only if no updates occurred since LL

Performance not always great



1 million random insertion, deletions on keys 0 - 255

Harris, "A pragmatic implementation of non-blocking linked-lists", 2001 (15th International Symposium on Distributed Computing)

Next time…

Multi-process synchronization problems

- **Producer Consumer!**
- **Reader-Writer!**
- **DOALL!**

Appendix

More interesting topics

Avoiding Errors with Pthreads

- Create data structures that handle most of the synchronization for you
  - Code the locks once correctly, then don't worry about them anymore

- For example:
  - Create a synchronized list
  - Perform locks inside add/remove/search functions
  - Synchronization now transparent to rest of program

Smart Programming with Pthreads

- Locks serialize the program, want to use as little as possible
- Only place lock around critical area
  - Less time spent holding lock, less lock contention
- Locks have high overhead
  - Constant locking and unlocking can result in poor performance

What is a Data Race?

- Two concurrent accesses to a memory location at least one of which is a write.
- Example: Data race between a read and a write

```
int x = 1;
Parallel.Invoke(
                () => { x = 2; },
                () => { System.Console.WriteLine(x); }
              );
```

writes x

reads x

- Outcome nondeterministic or worse
  - may print 1 or 2, or arbitrarily bad things on a relaxed memory model

Data Races and Happens-Before

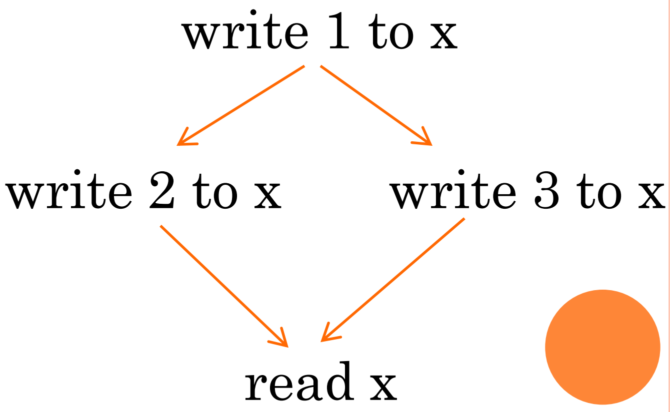
- Example of a data race with two writes:

```
int x = 1;
Parallel.Invoke( () => { x = 2; },
                 () => { x = 3;  }  );
System.Console.WriteLine(x);
```

- We visualize the ordering of memory accesses with a happens-before graph:

There is no path between
(write 2 to x) and (write 3 to x),
thus they are concurrent,
thus they create a data race

(note: the read is not in a data race)



Quiz: Where are the data races?

```
Parallel.For(1,2,
i => {
    x = a[i];
});
```

```
Parallel.For(1,2,
i => {
    a[i] = x;
});
```

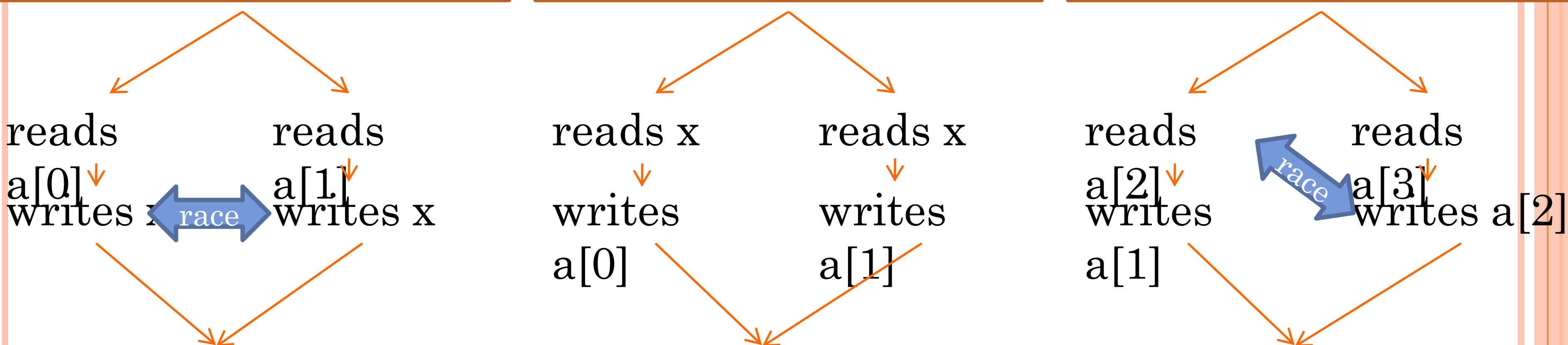
```
Parallel.For(1,2,
i => {
    a[i] = a[i+1];
});
```

QUIZ: WHERE ARE THE DATA RACES?

```
Parallel.For(1,2,
i => {
    x = a[i];
});
```

```
Parallel.For(1,2,
i => {
    a[i] = x;
});
```

```
Parallel.For(1,2,
i => {
    a[i] = a[i+1];
});
```



Race between two writes.

No Race between two reads.

Race between a read and a write.

Spotting Reads & Writes

- Sometimes a single statement performs multiple memory accesses

When you execute

```
x += y
```

there are actually two reads and one write:

reads x
reads y
writes x

When you execute

```
a[i] = x
```

there are actually three reads and one write:

reads x
reads a
reads i
writes a[i]

DATA RACES CAN BE HARD TO SPOT.

```
Parallel.For(0, 10000,
    i => {a[i] = new Foo();})
```

- Code looks fine... at first.

Data Races can be hard to spot.

```
Parallel.For(0, 10000,
    i => {a[i] = new Foo();})
```

- Problem: we have to follow calls… even if they look harmless at first (like a constructor).

```
class Foo {
    private static int counter;
    private int unique_id;
    public Foo()
        {
        unique_id = counter++;
        }
}
```

Data Race on static field !