More THREADS and Assignment 2!

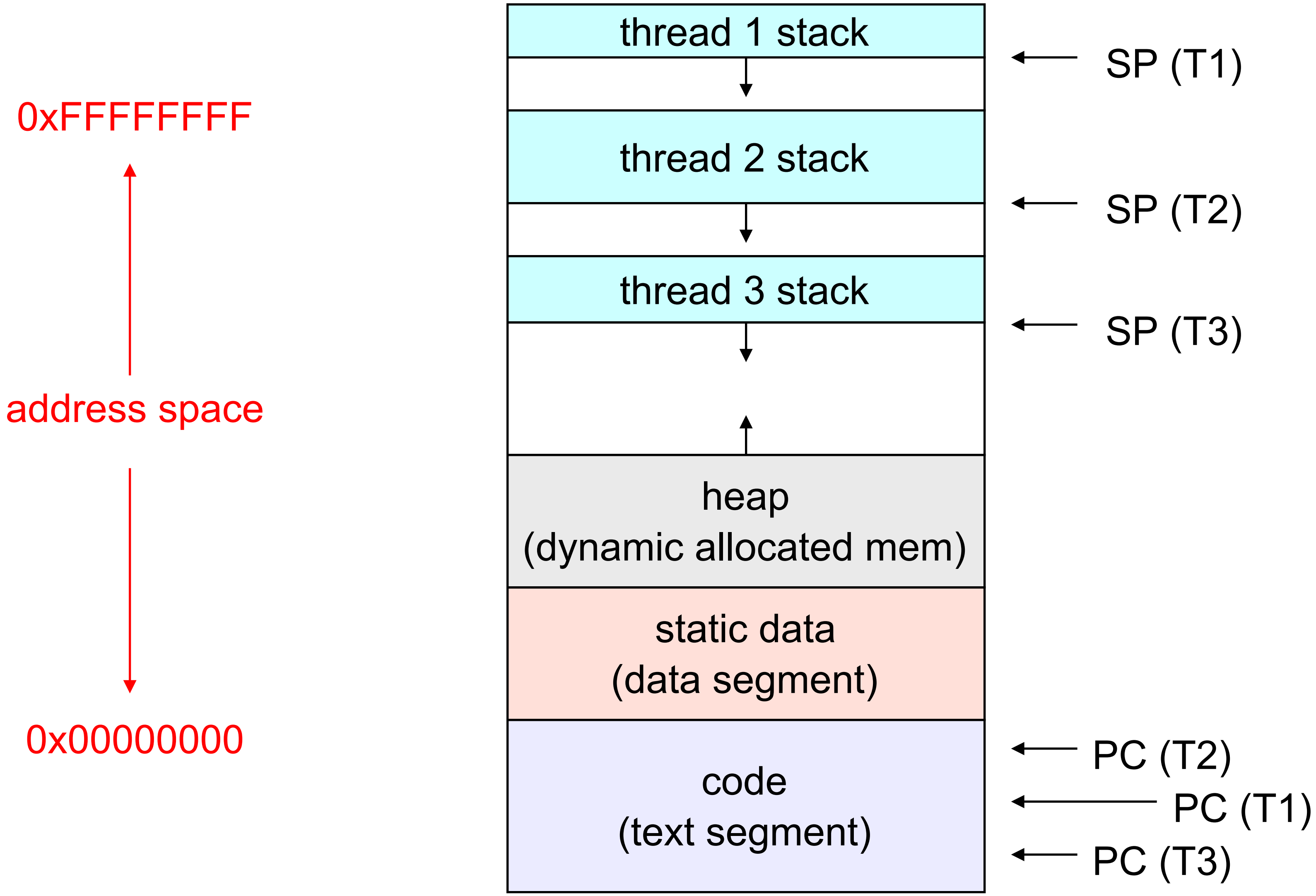
Concept overview

- Big picture: Achieving concurrency/parallelism
- Kernel threads
- User-level threads
- Race Conditions and **Critical Sections**
- Dangerous interleavings!
  - read-modify-write!
  - With threads this can become

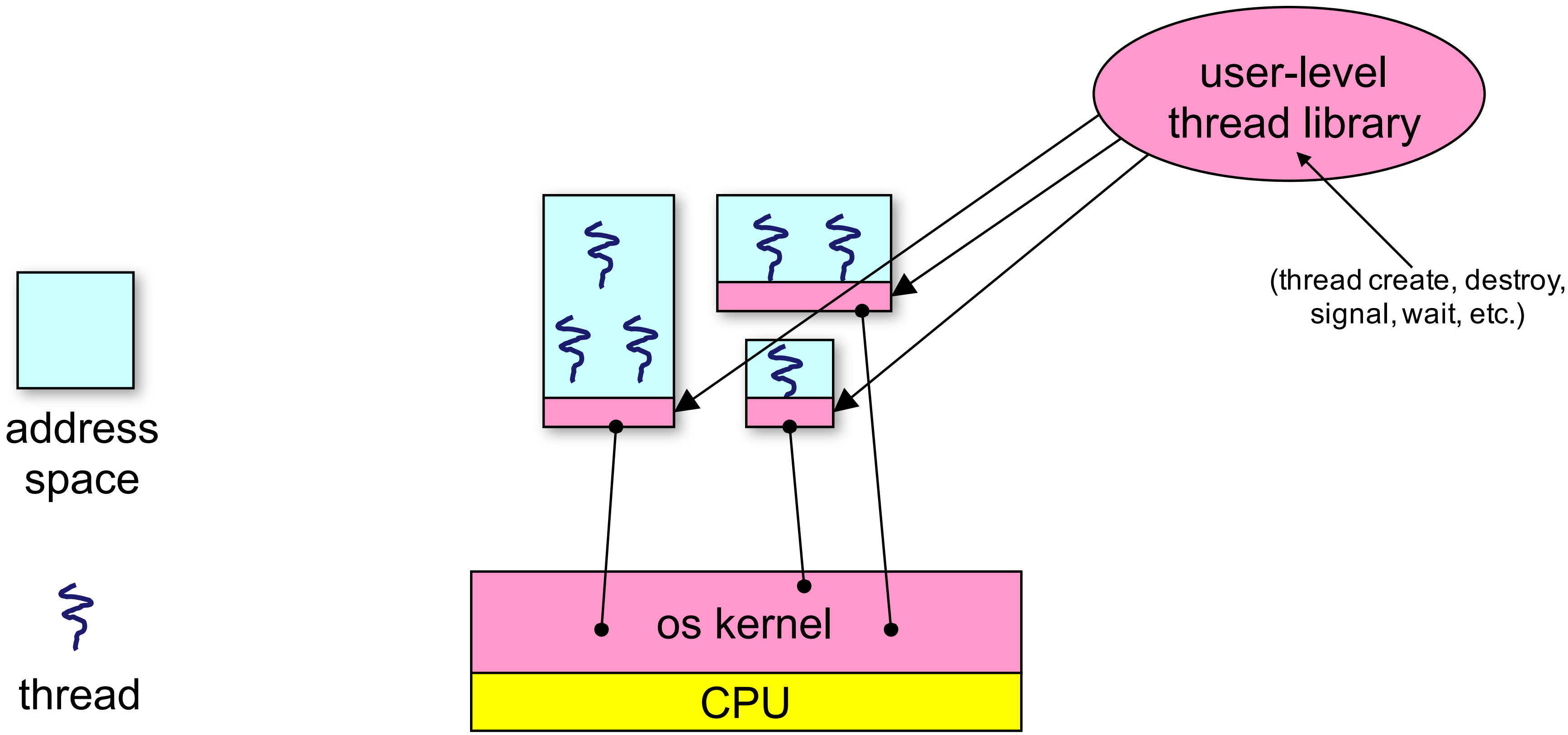
read modify write
read modify write…

- A2 due **Feb 29 (let's USE that leap day!)**
- **Victory Lap (term test) 2 on Threads Mar 2!**

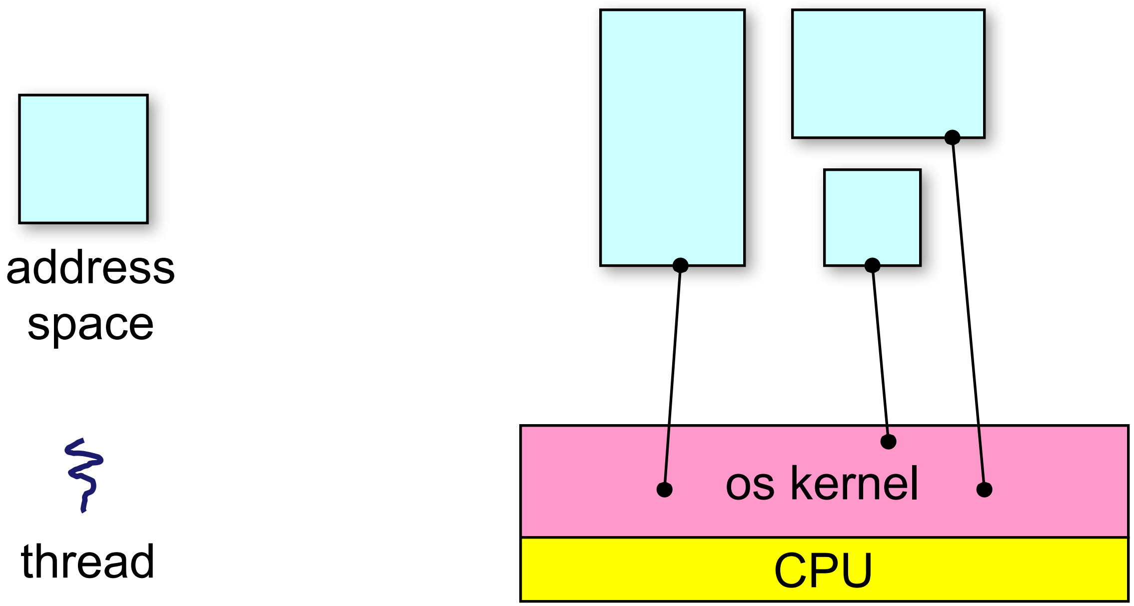
(new) Address space with threads



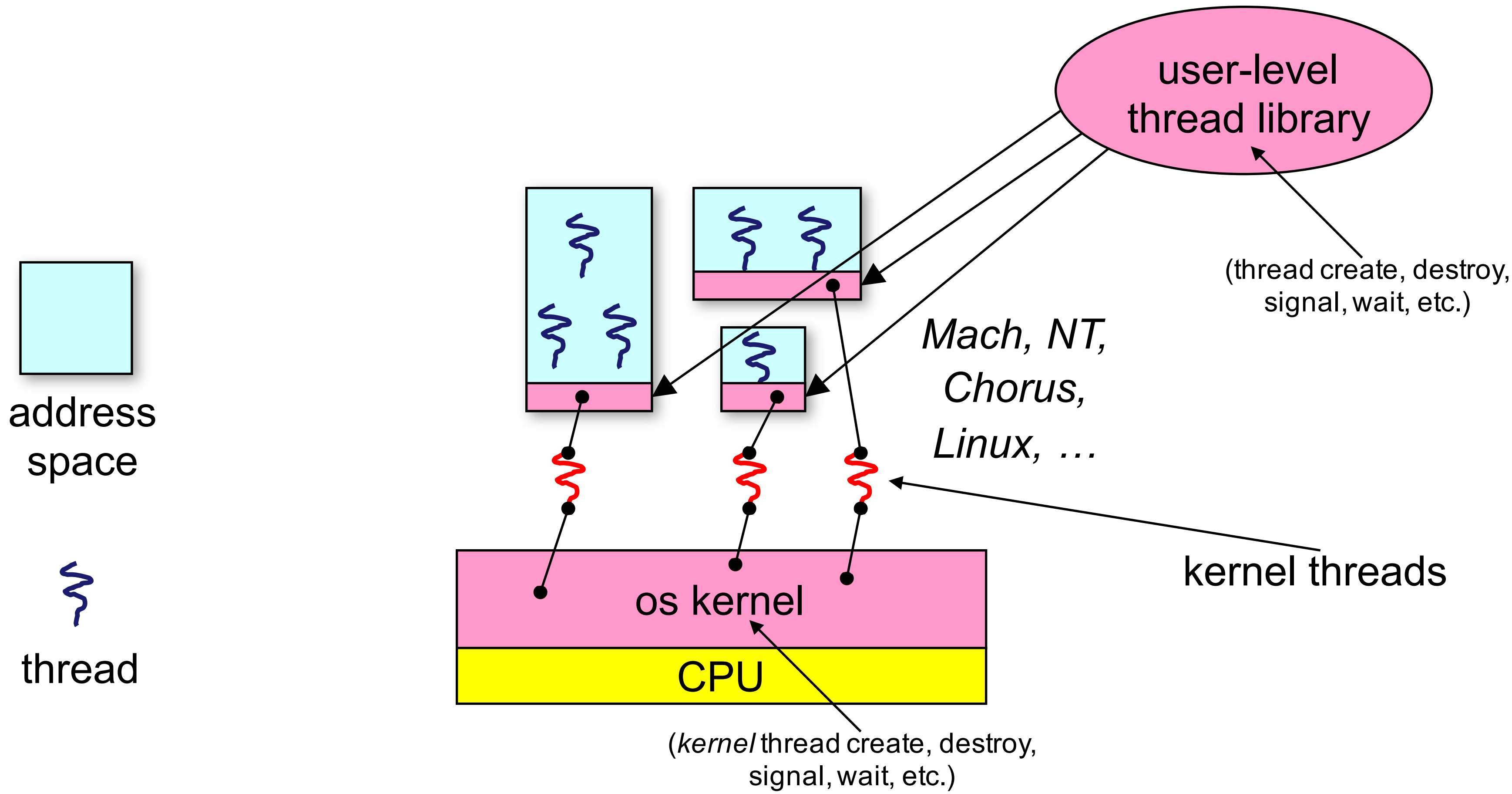
User-level threads



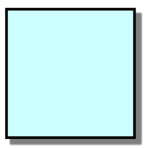
User-level threads: what the kernel sees



User-level threads: the full story

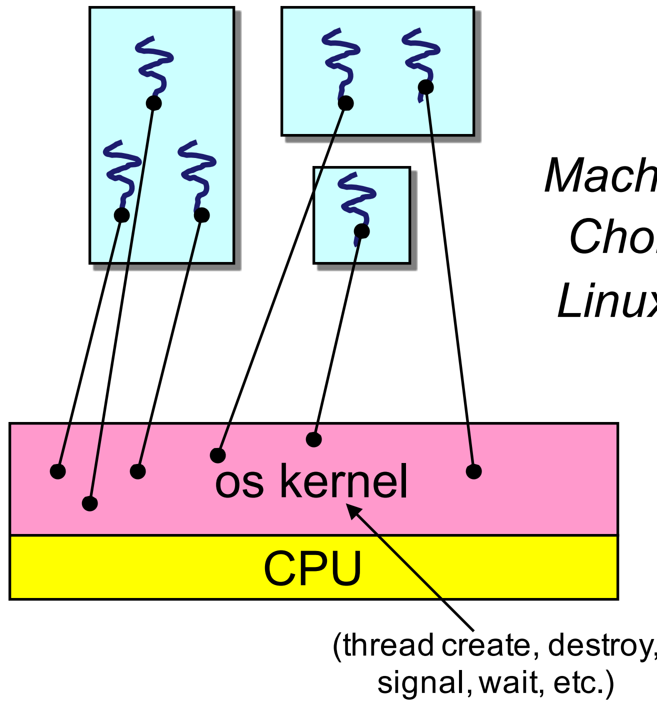


Kernel threads



address space

thread



Mach, NT, Chorus, Linux, …

Kernel threads

- OS now manages threads *and* processes / address spaces
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - possible to overlap I/O and computation inside a process
- Kernel threads are *cheaper* than processes
  - less state to allocate and initialize
- But, they're still pretty expensive for fine-grained use
  - orders of magnitude more expensive than a procedure call
  - thread operations are all system calls
    - context switch
    - argument checks

"Where do threads come from?" (Assignment 2 Part 1!)

- There is an alternative to kernel threads
- Threads can also be managed at the user level (that is, entirely from within the process)
  - a library linked into the program manages the threads
    - because threads share the same address space, the thread manager doesn't need to manipulate address spaces (which only the kernel can do)
    - threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
    - the thread package multiplexes user-level threads on top of kernel thread(s)
    - each kernel thread is treated as a "virtual processor"
  - we call these user-level threads

User-level threads

- User-level threads are small and fast
  - managed entirely by user-level library
    - E.g., pthreads (`libpthreads.a`) can be the interface!
  - each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  - creating a thread, switching between threads, and synchronizing threads are done via procedure calls
    - no kernel involvement is necessary!
  - user-level thread operations can be 10-100x faster than kernel threads as a result
  - *BUT WHAT IS THE TRADEOFF??!*

Performance example

- On relatively old hardware/OS (only the relative numbers matter)…
  - Processes
    - `fork/exit`: 251 μs
  - Kernel threads
    - `pthread_create()/pthread_join()`: 94 μs **(2.5x faster)**
  - User-level threads
    - `pthread_create()/pthread_join`: 4.5 μs **(another 20x faster)**

Why?

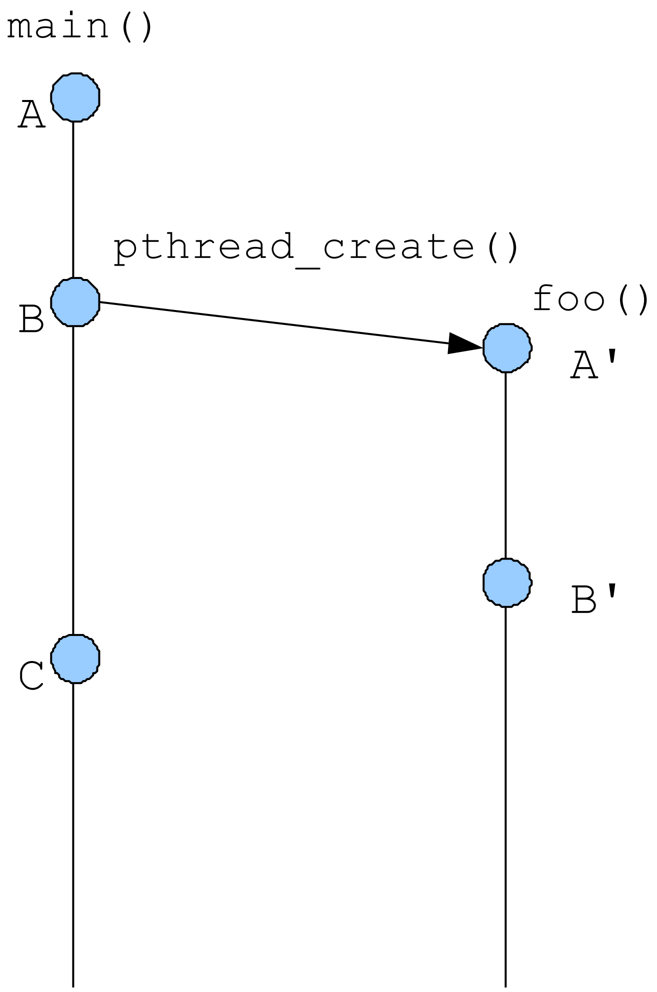
Why?

Concurrency and Temporal relations

- Instructions executed by a single thread are totally ordered
  - A < B < C < …

- Absent synchronization, instructions executed by distinct threads must be considered unordered / simultaneous
  - Not X < X’, and not X’ < X

Example



Y-axis is "time."

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

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

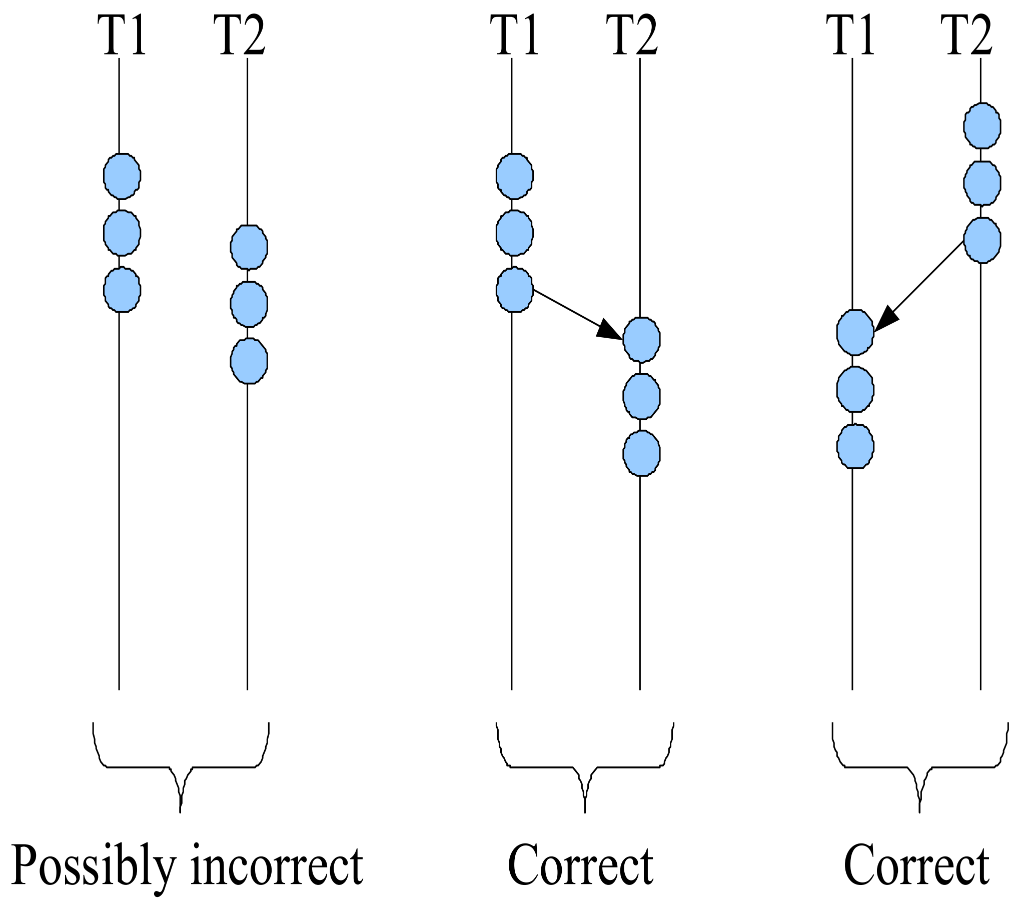
B < A' ?

Critical Sections / Mutual Exclusion

- Sequences of instructions that may get incorrect results if executed simultaneously are called critical sections
- (We also use the term race condition to refer to a situation in which the results depend on timing)
- 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 – guarantees ordering
- One way to guarantee mutually exclusive execution is using locks

Critical sections

→ *is the "happens-before" relation*



When do critical sections arise?

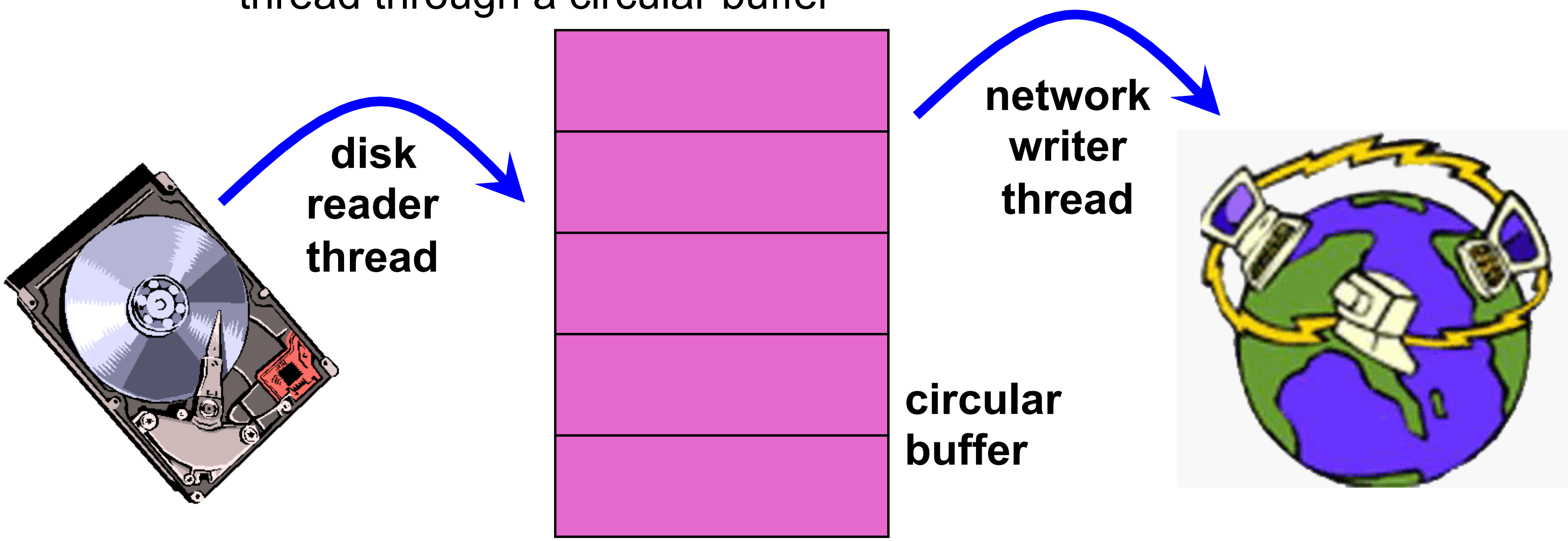
- One common pattern:
  - read-modify-write of
  - a shared value (variable)
  - in code that can be executed concurrently

    (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 variables
  - NOT local variables (which are on the stack)

    (Note: Never give a reference to a stack-allocated (local) variable to another thread, unless you're superhumanly careful …)

Example: buffer management

- Threads cooperate in multithreaded programs
  - to share resources, access shared data structures
    - e.g., threads accessing a memory cache in a web server
  - also, to coordinate their execution
    - e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer



Example: shared bank account

- 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
  spit out cash;
}
```

- Now suppose that you and your significant other 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 that transaction is submitted

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

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

Interleaved schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

Execution sequence as seen by CPU

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

context switch

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

context switch

```
put_balance(account, balance);
spit out cash;
```

- What's the account balance after this sequence?
  - who's happy, the bank or you?
- How often is this sequence likely to occur?

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);
  spit out cash;
}
```

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

How About Now?

```
int xfer(from, to, amt) {
  withdraw( from, amt );
  deposit( to, amt );
}
```

```
int xfer(from, to, amt) {
  withdraw( from, amt );
  deposit( to, amt );
}
```

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

Another example

```
i++;
```

```
i++;
```

Correct critical section requirements

- Correct 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
  - 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

- Spinlocks
  - primitive, minimal semantics; used to build others
- Semaphores (and non-spinning locks)
  - basic, easy to get the hang of, somewhat hard to program with
- Monitors
  - higher level, requires language support, implicit operations
  - easier to program with; Java "`synchronized()`" as an example
- Messages
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems