

Threads and Synchronization

Kevin Webb
Swarthmore College
February 13, 2020



Today's Goals

- Extend processes to allow for multiple execution contexts (threads)
- Benefits and challenges of concurrency
- Race conditions and atomicity
- Synchronization: hardware, OS, and userspace

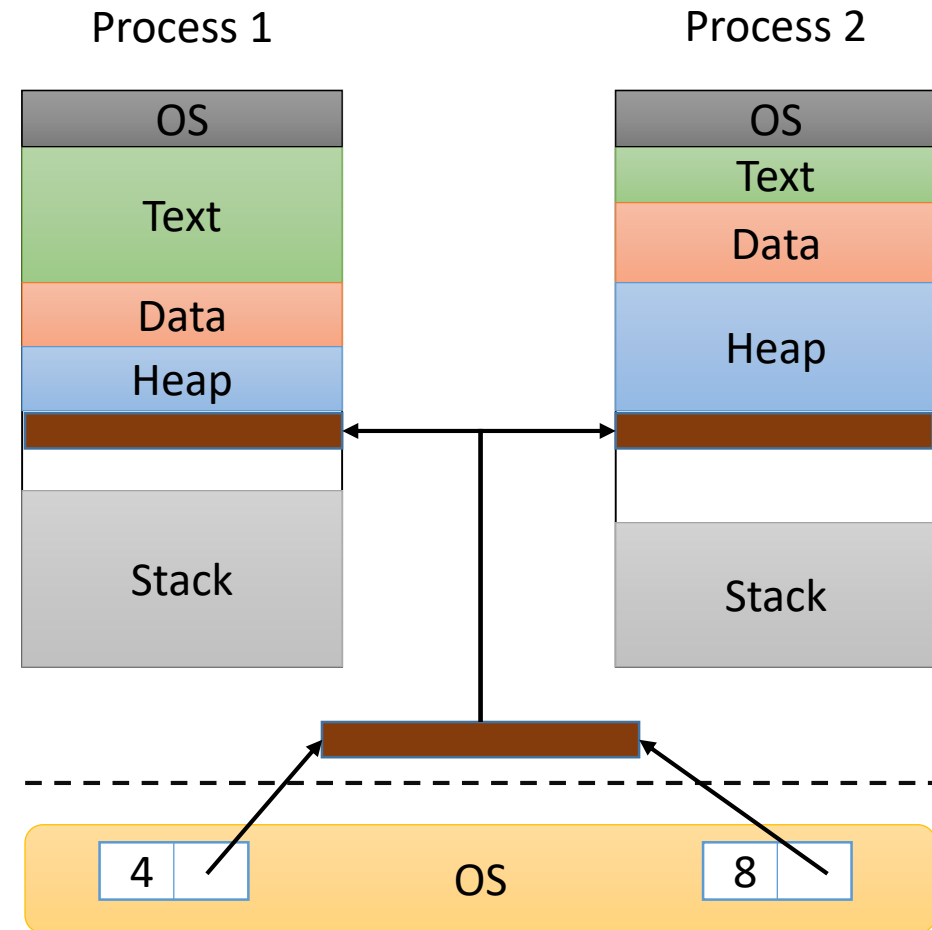
Recall: POSIX Shared Memory

- Explicitly request a chunk of memory to be shared.

```
int fd = shm_open(name, ...);  
ftruncate(fd, 8192);  
void *ptr = mmap(..., fd, ...);
```

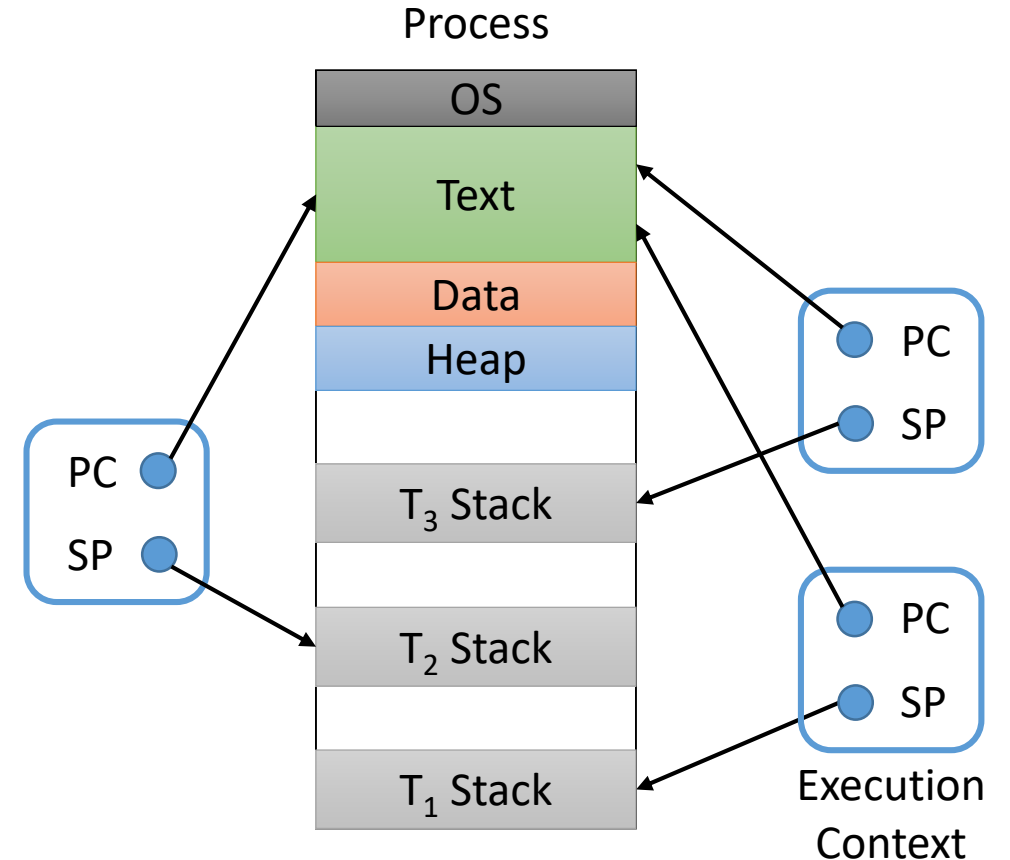
- Only works on shared hardware.

Apps that would have used this have largely switched to using threads.



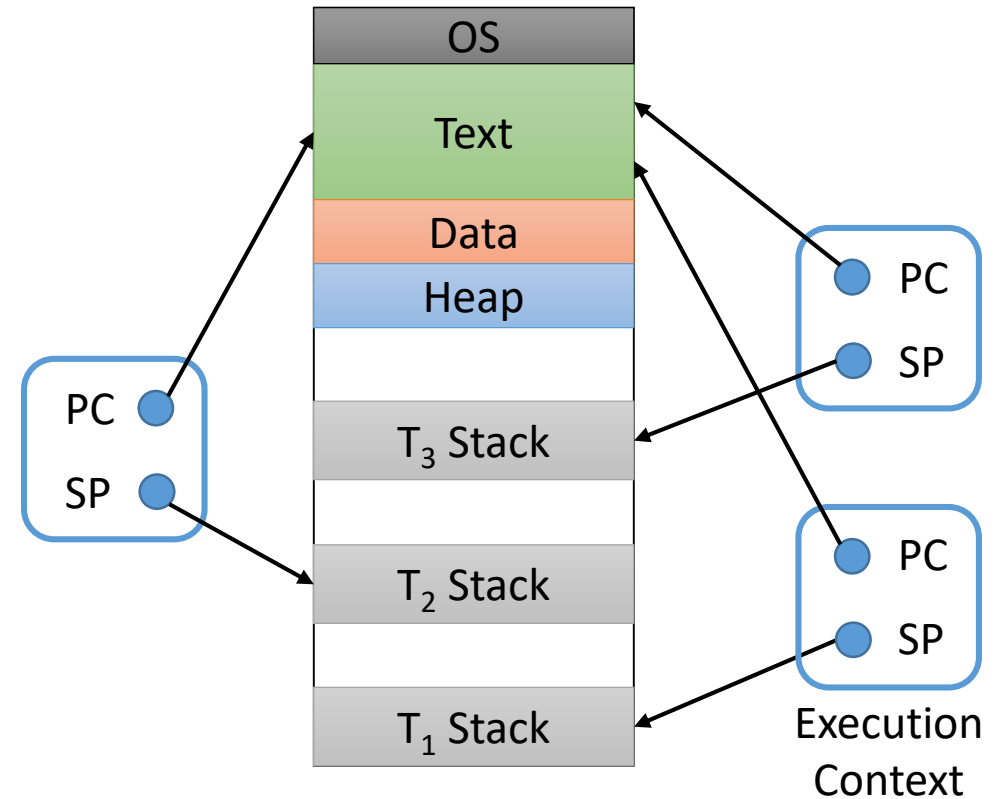
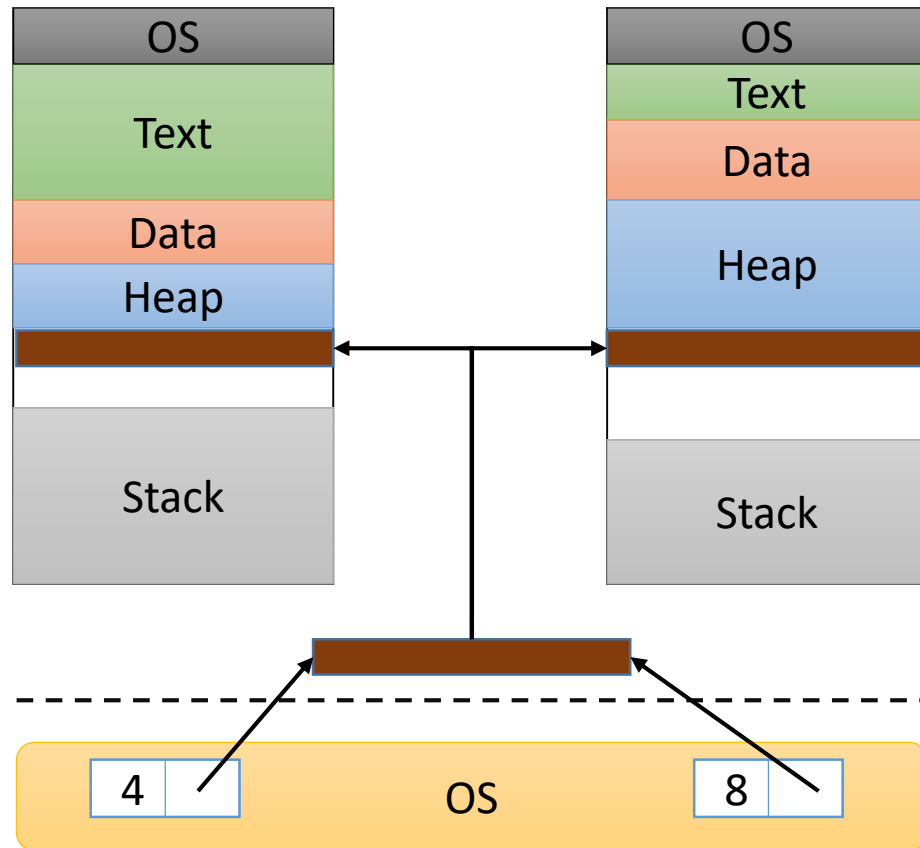
Thread Model

- Single process with multiple copies of execution resources.
- ONE shared virtual address space!
 - All process memory shared by every thread.
 - Threads coordinate by sharing variables (typically on heap)



Shared Memory is... Shared Memory

- These models are equally powerful (for modern thread libraries).



If inter-process shared memory and threads serve the same roles, why do we prefer threads?

- A. Threads are easier to use. (Why?)
- B. Threads provide higher performance. (Why?)
- C. Users have more control over thread execution / synchronization. (How?)
- D. Some other reason(s).

Threads vs. Inter-Process Shared Memory

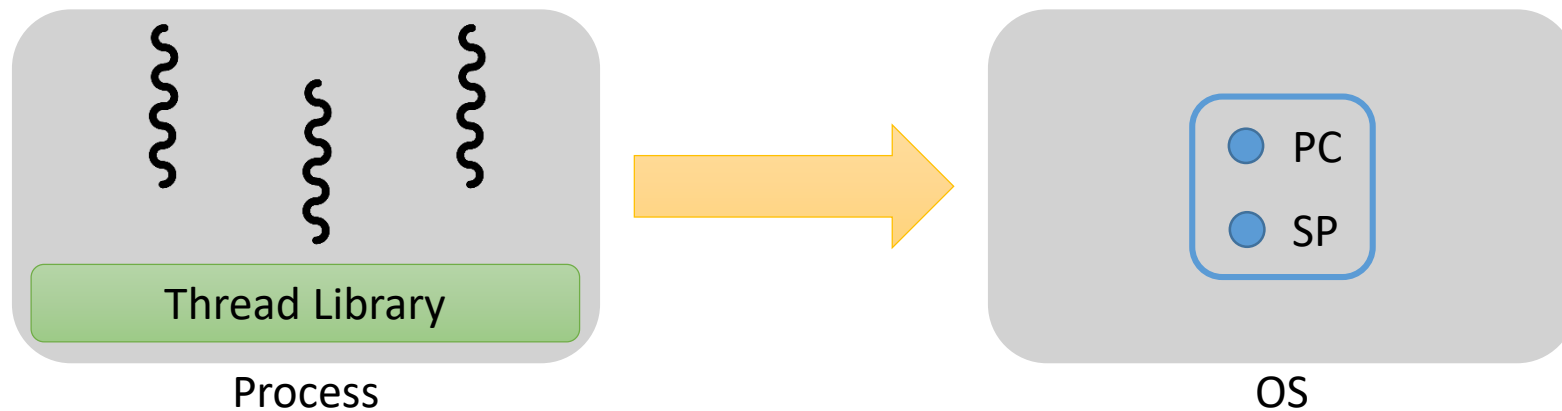
- Threads: shared virtual address space -> LOW context switch overhead
- Threads: implicit sharing, no extra calls necessary (opening FDs)
- Multiple processes: more protection
 - ONLY explicitly shared memory is accessible to multiple processes

Thread Abstraction vs. Implementation

- Abstraction: multiple execution contexts in a shared VAS
- Implementation decisions:
 - How much should the OS know about threads?
 - How much should the userspace process manage about threads?

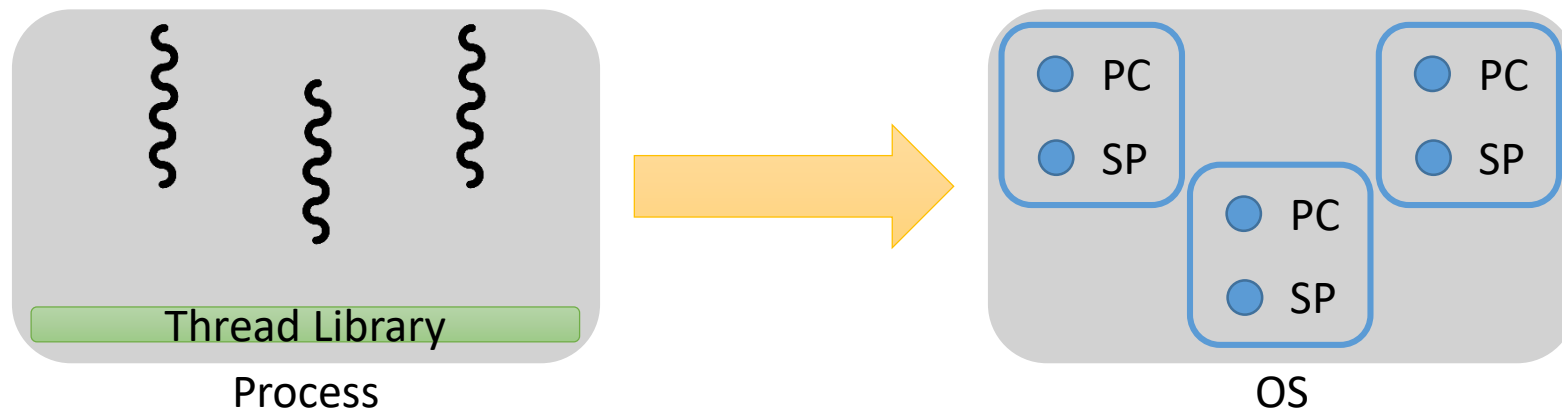
Implementation Option 1 (N:1)

- OS knows *nothing* about threads. All execution context stored in process memory.
- Threading implemented as a userspace library. Userspace code decides which thread to execute at any given time.



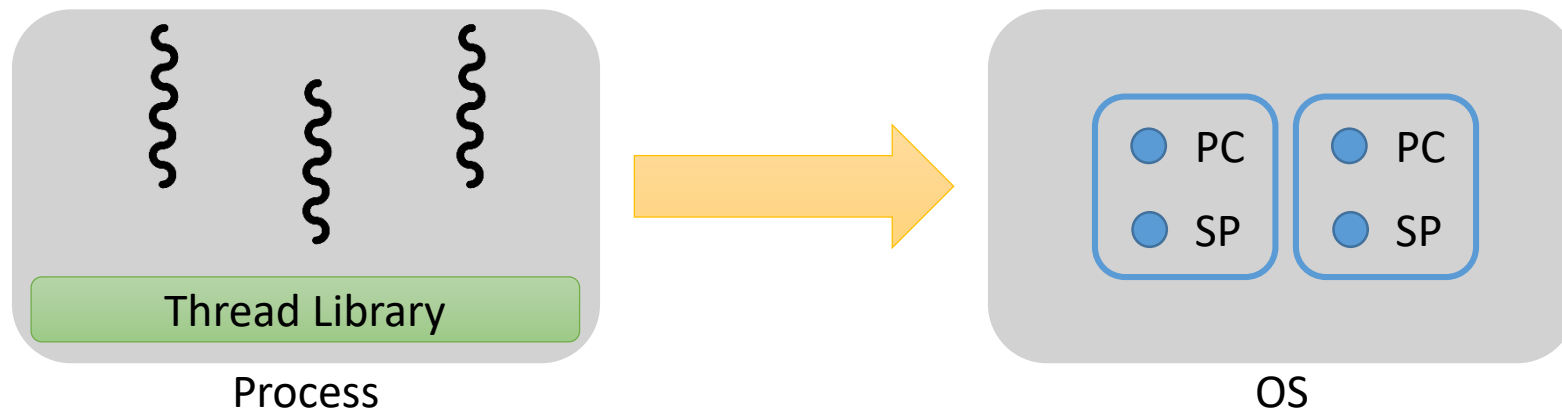
Implementation Option 2 (1:1)

- OS fully aware of threads. All execution context stored by kernel.
- OS creates a *kernel thread* for every new thread and schedules all threads.



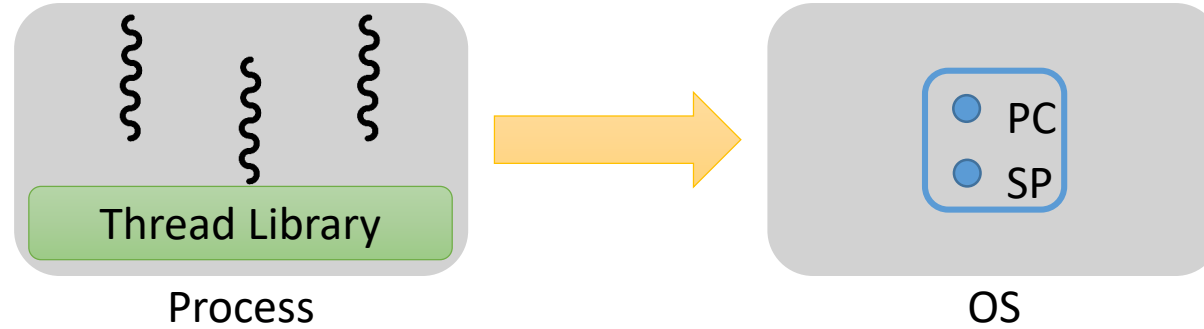
Implementation Option 3 (N:M)

- OS supports multiple execution contexts, but a process may have more threads than kernel execution contexts.
- Userspace code tracks threads and manages mapping them to kernel execution context.

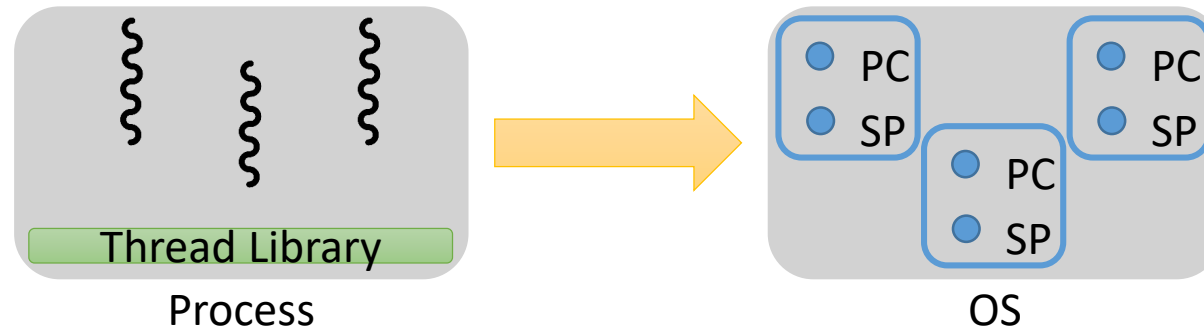


Which threading model would you use in your OS? Why?

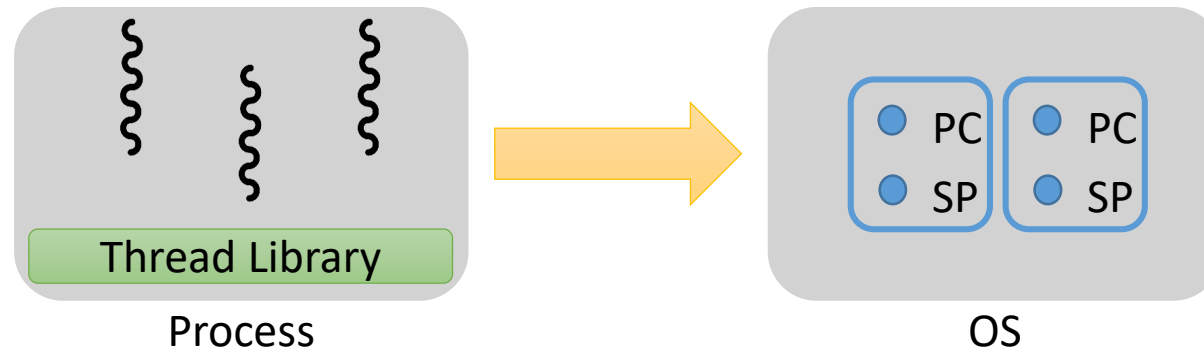
A: (N:1)



B: (1:1)



C: (N:M)



Extending OS Process Model

- Unless we say otherwise, assume 1:1 “kernel thread” model.
 - OS is aware of every thread, and schedules them all independently
 - Thread == “execution context”
- Before: storage for (one) execution context in PCB
 - Registers, PC, SP, kernel stack, etc.
- Now, with threads: PCB contains a collection of threads
 - Each thread represents an execution context

All of this is still true
for inter-process
shared memory!

Concurrency: Double-Edged Sword

- Benefit: OS will schedule threads concurrently on multiple CPUs
- Benefit: If a process blocks in one thread, it can continue in others
- Problem: OS is now in full control over when threads execute, and it doesn't know what your process is trying to do...

As the programmer, you're at the mercy of the scheduler. If execution order matters, it's up to you to ensure that good orderings happen!

Race Conditions

- Textbook: “A situation where ... the outcome of the execution depends on the particular order in which the [memory accesses take place].”
- Wikipedia: “A race condition is the behavior of [a system] where the output is dependent on the sequence or timing of other uncontrollable events.”

Eliminating Race Conditions

1. Identify orderings that must / must not happen.

2. Apply synchronization constructs to avoid the bad orderings identified in (1).

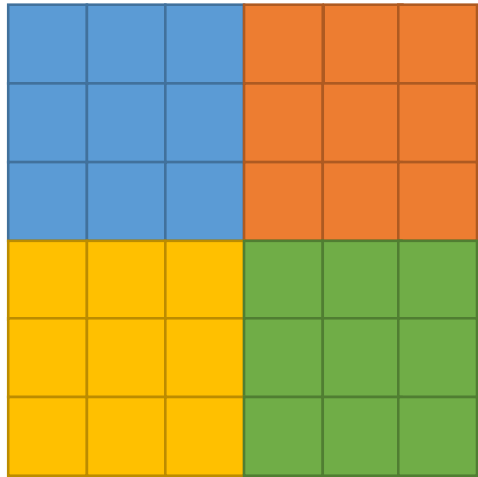
For non-trivial problems, (1) is often very difficult because humans are bad at thinking about concurrency.

Recall Synchronization:

Arranging events to happen at the same time (or ensuring that they don't)

1. Identify the potentially problematic ordering(s).
2. Propose a general solution (don't need to name specific constructs).

Example 1: Parallel Game of Life



Grid divided into one region per thread. Simulation in discrete rounds.

Each thread decides, for each cell, whether the cell is alive or dead in the *next* round.

Decision for a cell depends on state of neighbor cells.

1. Identify the potentially problematic ordering(s).
2. Propose a general solution (don't need to name specific constructs).

Example 2: Student Enrollment System

Thread 0:

```
add_course(Alice, CS45) {  
    roster = get_course(CS45);  
    roster.add(Alice);  
    roster.save_to_DB();  
}
```

Thread 1:

```
drop_course(Bob, CS45) {  
    roster = get_course(CS45);  
    roster.remove(Bob);  
    roster.save_to_DB();  
}
```

1. Identify the potentially problematic ordering(s).
2. Propose a general solution (don't need to name specific constructs).

Example 3: Bounded Producer / Consumer

Shared Memory

```
int buf[N], in = 0, out = 0, count = 0;
```

Producer Thread

```
while (TRUE) {  
    buf[in] = Produce ();  
    in = (in + 1) % N;  
    count++;  
}
```

Consumer Thread

```
while (TRUE) {  
    Consume (buf[out]);  
    out = (out + 1) % N;  
    count--;  
}
```

1. Identify the potentially problematic ordering(s).
2. Propose a general solution (don't need to name specific constructs).

Synchronization in Practice

- Voluntarily suspend a thread to ensure that events happen (or DON'T happen) at the same time – eliminate race conditions.
- Our job as the OS:
 - Provide abstract interface for applications to request voluntary blocking
 - Provide an implementation given the hardware reality
- OS must be involved if we want to block threads (or processes)!

Key Property: Atomicity

- An operation is **atomic** if the effects of its execution appear to be uninterruptable.
- Intuition: an atomic operation's effects are all-or-nothing.
- In our assumed 1:1 thread model, OS will need to provide atomicity to applications.
 - Otherwise, the scheduler's nondeterminism is chaos to the application.

Suppose `count` is a global variable, multiple threads increment it: `count++`;
Is there a race condition here?

- A. Yes, there's a race condition (`count++` is not atomic). Why?
- B. No, there's no race condition (`count++` is atomic). Why not?
- C. Cannot be determined. What's missing?

Suppose `count` is a global variable, multiple threads increment it: `count++`;
Is there a race condition here?

- A. Yes, there's a race condition (`count++` is not atomic).
- B. No, there's no race condition (`count++` is atomic).
- C. Cannot be determined

How about if compiler implements it as:

```
movl (%edx), %eax    // read count value
addl $1, %eax         // modify value
movl %eax, (%edx)     // write count
```

NOT atomic. Might
context switch between
any of these instructions.

How about if compiler implements it as:

```
incl (%edx)          // increment value
```

Depends on the HW
implementation of
instruction. (x86: NO)

Critical Sections

- A section of code, shared by multiple threads, that must be executed atomically.
- Common pattern:
 - read a value
 - modify the value
 - write the value
- Typically solved by enforcing mutual exclusion to achieve atomicity.

Four Rules for Mutual Exclusion

1. No two threads can execute the critical section at the same time.
2. No thread outside the critical section may prevent others from entering the critical sections.
3. No thread should have to wait forever to enter the critical section. (starvation)
4. No assumptions can be made about speeds or number of CPUs.

Sources of Atomicity

- Recall: In our assumed 1:1 thread model, OS will need to provide atomicity to applications.
 - Otherwise, the scheduler's nondeterminism is chaos to the application.
- **What can an OS use to enforce atomicity?**

Atomicity Option 1: Disable CPU Interrupts

- Idea: when starting an atomic operation, disable interrupts.
 - Can't be context switched!
 - Re-enable interrupts when done.

```
atomic_increment(x) {  
    disable_interrupts();  
    movl (%edx), %eax  
    addl $1, %eax  
    movl %eax, (%edx)  
    enable_interrupts();  
}
```

Atomicity Option 1: Disable CPU Interrupts

- Idea: when starting an atomic operation, disable interrupts.
 - Can't be context switched!
 - Re-enable interrupts when done.
- Problem: Only works with a single CPU (one core)
 - Interrupts are per-CPU
 - Code on other CPUs might be modifying shared state

```
atomic_lock(l) {  
    while (1) {  
        disable_interrupts();  
        if (l == LOCKED) {  
            enable_interrupts();  
            // block process  
            continue;  
        }  
        l = LOCKED;  
        enable_interrupts();  
    }  
}
```

Atomicity Option 2: Peterson's Solution

Shared Memory

```
int turn;  
boolean flag[2] = {FALSE, FALSE};
```

Thread₀

```
flag[T0] = TRUE;  
turn = T1;  
while (flag[T1] && turn==T1);  
< critical section >  
flag[T0] = FALSE;
```

Thread₁

```
flag[T1] = TRUE;  
turn = T0;  
while (flag[T0] && turn==T0);  
< critical section >  
flag[T1] = FALSE;
```

1. No two threads can execute the critical section at the same time.
2. No thread outside the critical section may prevent others from entering the critical sections.
3. No thread should have to wait forever to enter the critical section. (starvation)
4. No assumptions can be made about speeds or number of CPU's.

```
int turn;  
boolean flag[2] = {FALSE, FALSE};
```

Do nothing in while!

Thread₀

```
flag[T0] = TRUE;  
turn = T1;  
while (flag[T1] && turn==T1);  
< critical section >  
flag[T0] = FALSE;
```



Thread₁

```
flag[T1] = TRUE;  
turn = T0;  
while (flag[T0] && turn==T0);  
< critical section >  
flag[T1] = FALSE;
```



Are there problems with this? Would you use it?

Shared Memory

```
int turn;  
boolean flag[2] = {FALSE, FALSE};
```

Thread₀

```
flag[T0] = TRUE;  
turn = T1;  
while (flag[T1] && turn==T1);  
< critical section >  
flag[T0] = FALSE;
```

Thread₁

```
flag[T1] = TRUE;  
turn = T0;  
while (flag[T0] && turn==T0);  
< critical section >  
flag[T1] = FALSE;
```


Atomicity Option 3: Hardware

- Hardware can provide *atomic instructions* with guaranteed atomicity
- Indivisible instruction that performs multiple tasks
- Examples:
 - “Test and Set”
 - “Compare and Swap”
- Varies by CPU Instruction Set Architecture


x86: CMPXCHG
Compare and Exchange

Atomic “Test and Set” Instruction

- Retrieve a value from memory and set the value at that location to 1

- C Pseudocode:

```
int test_and_set(int *addr) {  
    int result = *addr;  
    *addr = 1;  
    return result;  
}
```



All of this happens atomically as part of one CPU instruction. ISA defines the specifics.

Using Test and Set

```
int mutex;    // 1 -> locked, 0 -> unlocked  
lock_mutex(&mutex);
```

```
lock_mutex(int *m) {  
    int result = test_and_set(m);  
    if (result == 0)  
        /* We now hold the lock. */  
    else  
        /* Lock held by someone else. */  
}
```

Atomic “Compare and Swap” Instruction

- Check the value at a memory location. If it matches the provided value, change the value at that memory location.

- C Pseudocode:

```
int compare_and_swap(int *addr, int check, int new) {  
    if (*addr == check) {  
        *addr = new;  
        return 1; // True - we performed the swap.  
    } else {  
        return 0; // False - we did not swap.  
    }  
}
```

Using Compare and Swap

```
int mutex;    // 1 -> locked, 0 -> unlocked  
lock_mutex(&mutex);
```

```
lock_mutex(int *m) {  
    int result = compare_and_swap(m, 0, 1);  
    if (result)  
        /* We now hold the lock. */  
    else  
        /* Lock held by someone else. */  
}
```

Implementing Atomic Instructions

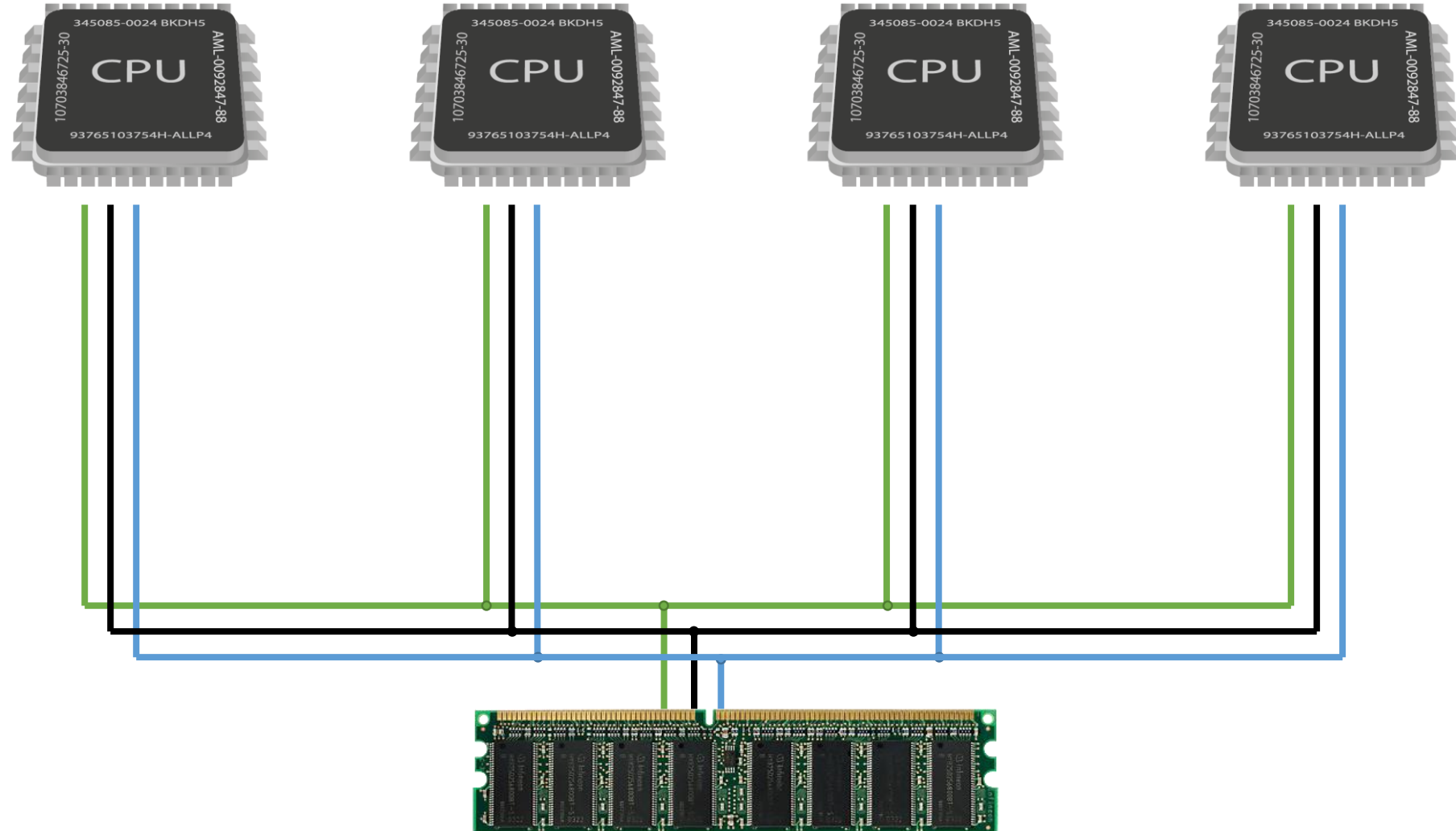
Bus: a collection of wires carrying one logical value

Connecting CPU(s) to memory:
three busses

- Address
- Data
- Control

Busses are shared by all the components they connect

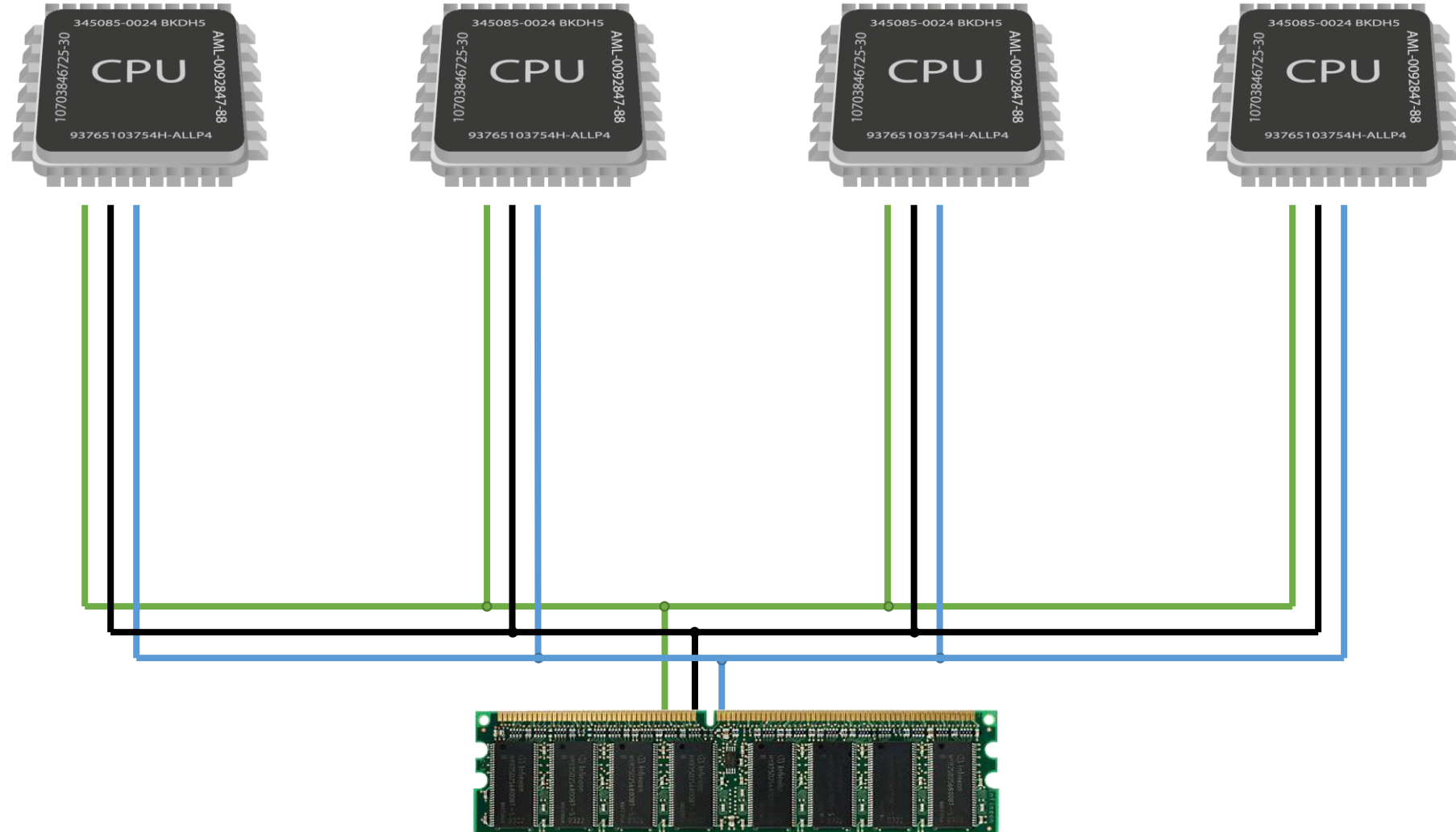
Only one component can write to the bus at a time, so it must be lockable in hardware!



Implementing Atomic Instructions

Normal instruction
(e.g., memory write):

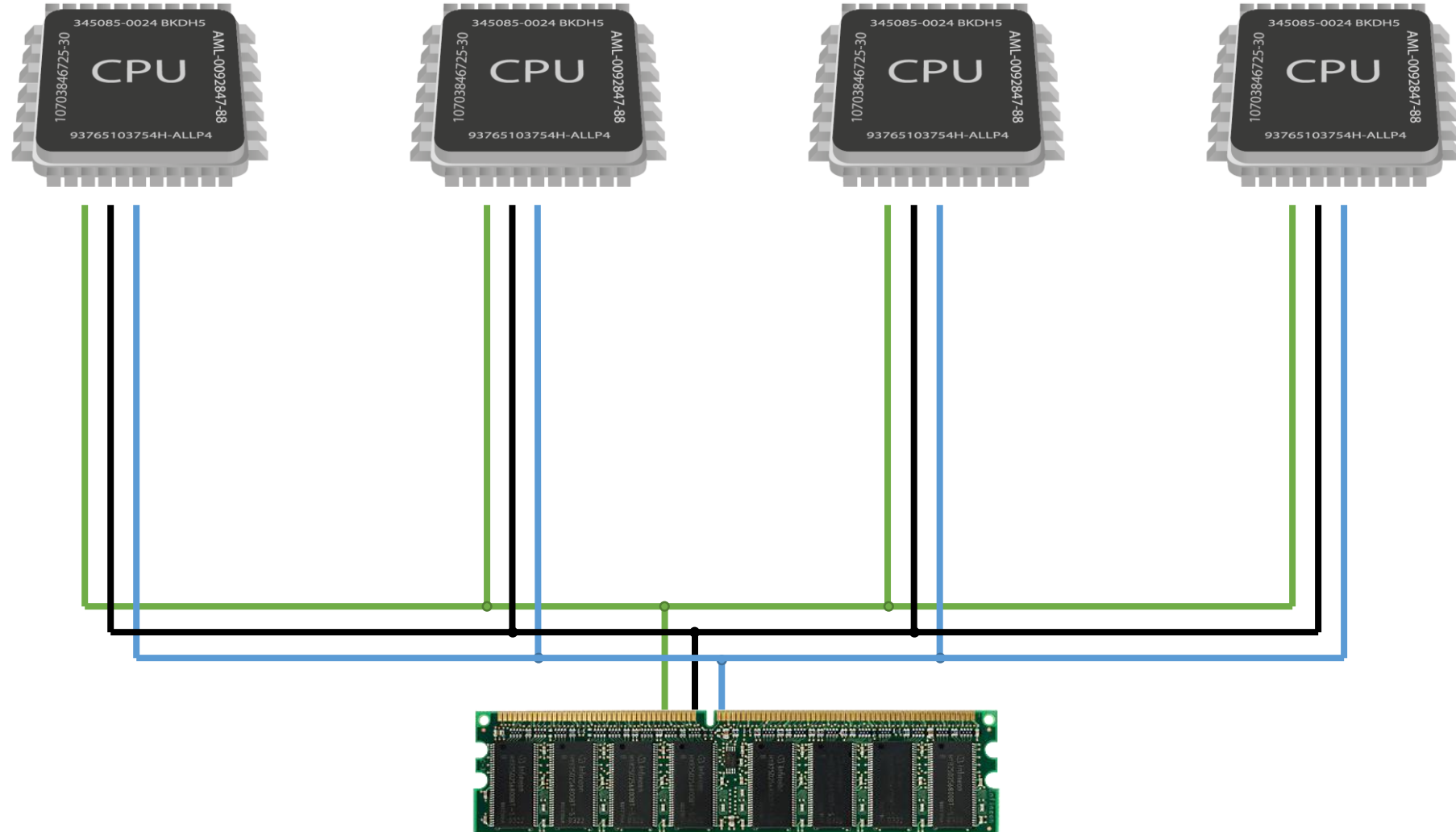
lock buses
place dest addr on address bus
place value on data bus
place write signal on control bus
unlock buses



Implementing Atomic Instructions

Atomic instruction
(e.g., compare and swap):

lock buses
perform normal read instruction
perform compare instruction
perform normal write (if needed)
unlock buses



Synchronization Constructs

- You've mostly seen/used the pthreads library:
 - mutex locks, condition variables, barriers, (maybe) readers/writer locks
- OS synchronization primitives:
 - semaphores, mutex locks, spin locks

Building Locks

Semantics: user calls lock_mutex.
Upon return, user has acquired the lock.

If the lock can't be acquired, calling process must wait.

```
int mutex; // 1 -> locked, 0 -> unlocked
lock_mutex(&mutex);

lock_mutex(int *m) {
while (1) {
    int result = compare_and_swap(m, 0, 1);
    if (result) {
        break;
    }
    ...
}
return; /* Lock is acquired. */
}
```

Building Locks

```
int mutex; // 1 -> locked, 0 -> unlocked
lock_mutex(&mutex);

lock_mutex(int *m) {
while (1) {
    int result = compare_and_swap(m, 0, 1);
    if (result) {
        break;
    }
    /* What should go here? */
}
return; /* Lock is acquired. */
}
```

- A. Nothing
- B. Wait a short time.
- C. Wait on a queue.
- D. Something else.

Lock Types

Spin Lock

- Keep trying to acquire lock without ever waiting
- Great when you are certain that critical section is short
- NO context switch overhead

Mutex Lock

- Try to acquire lock. If it's already held, block process, add to queue
- Used when waiting is potentially long (or unknown duration)

Building Locks

```
int mutex; // 1 -> locked, 0 -> unlocked
lock_mutex(&mutex);

lock_mutex(int *m) {
while (1) {
    int result = compare_and_swap(m, 0, 1);
    if (result) {
        break;
    }
    /* What should go here? */
}
return; /* Lock is acquired. */
}
```

Spin Lock

A. Nothing

B. Wait a short time.

C. Wait on a queue.

D. Something else.

Mutex Lock

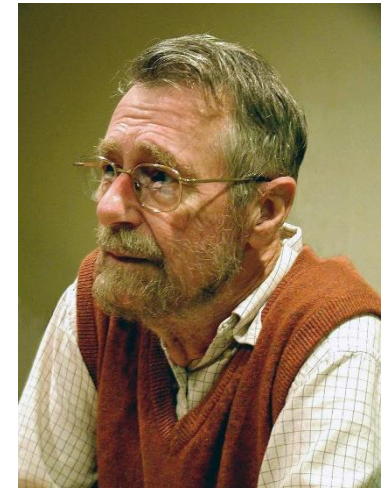
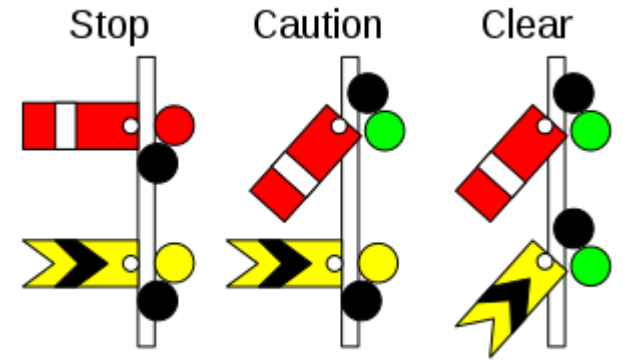
Semaphores

- Named after railway signals by Edsger Dijkstra
- Synchronization variable that keeps a count of how many users can acquire it.
- Example: suppose you have five study rooms:

Semaphore Count:
(# available rooms)

Study rooms:

--	--	--	--	--



Semaphores

- Observation: if a semaphore's count never goes above one -> mutex

Semaphore Count:

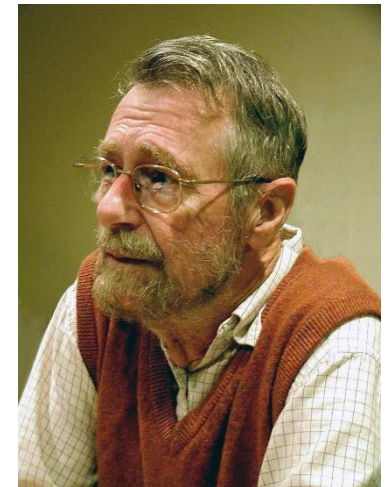
Study rooms:

- Example: suppose you have five study rooms:

Semaphore Count:

(# available rooms)

Study rooms:



Semaphore Interface

```
sem s = init_sem(n);    // declare & initialize

wait (sem s)            // Executes atomically
    decrement s;
    if s < 0, block thread (and add it to a queue);

signal (sem s)           // Executes atomically
    increment s;
    if blocked threads, unblock (any) one of them;
```


Semaphore Implementation (One Option)

```
sem s {
    int count;
    spin_lock l;
    queue waiters;
}

wait (sem *s) {
    lock(&s->l);
    s->count -= 1;
    if (s->count < 0)
        /* Add current process to waiters, unlock, and block process. */
        unlock(&s->l);
}

signal (sem *s) {
    lock(&s->l);
    s->count += 1;
    if ( /* Process(es) are waiting*/ )
        /* Unblock one queued waiter. */
        unlock(&s->l);
}
```

Should a process be able to check beforehand whether wait() will cause it to block?

```
sem s {
    int count;
    spin_lock l;
    queue waiters;
}

wait (sem *s) {
    lock(&s->l);
    s->count -= 1;
    if (s->count < 0)
        /* Add current process to waiters, unlock, and block process. */
        unlock(&s->l);
}

signal (sem *s) {
    lock(&s->l);
    s->count += 1;
    if ( /* Process(es) are waiting*/ )
        /* Unblock one queued waiter. */
        unlock(&s->l);
}
```

- A. Yes – how?
- B. No – why not?
- C. It depends – on what?

Higher-Level Constructs

- If the OS provides atomicity with locks and semaphores, others can build on top
- Example: pthread library – barrier (N process must all arrive before any proceed)

```
barrier {  
    sem = init_sem(0);  
    lock = init_lock();  
    int count = 0;  
    int target = N;  
}
```

Note: this barrier implementation can only be used once!

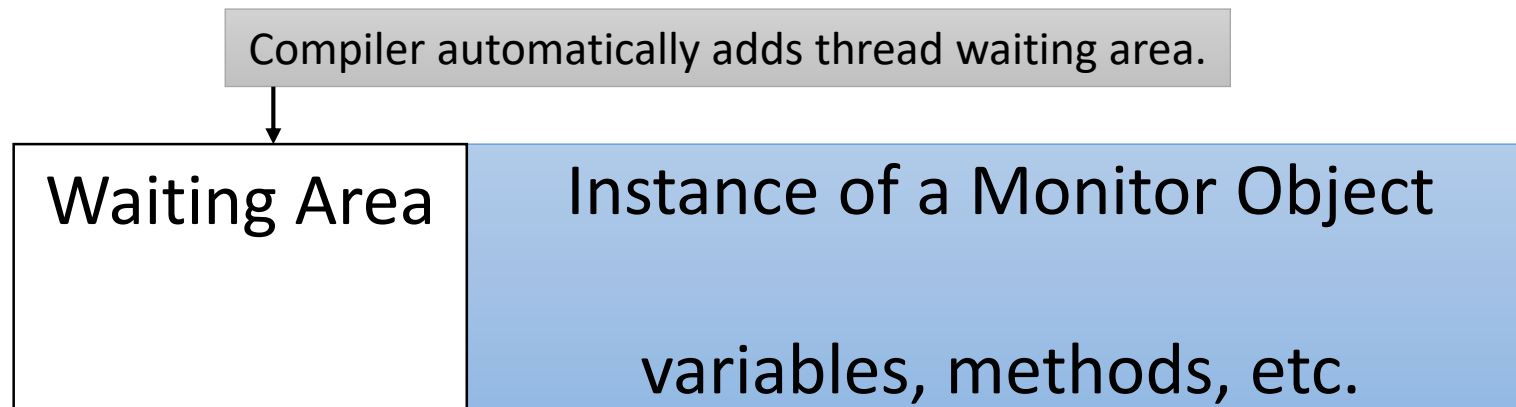
```
barrier_wait(barrier *b) {  
    lock(&b->lock);  
    b->count += 1;  
    unlock(&b->lock);  
  
    if (b->count == b->target) {  
        signal(b->sem); /* wake one */  
    }  
  
    wait(b->sem);  
    signal(b->sem); /* pass it on */  
}
```

Beyond CS 31

- In 31, you saw the pthreads library.
- So far, we've seen the OS support it by providing atomic primitives.
- Up next:
 1. Monitors: built-in language support for automatic synchronization
 2. Optimistic concurrency: concurrency without locking
 3. Mixing signals and threads

Monitors

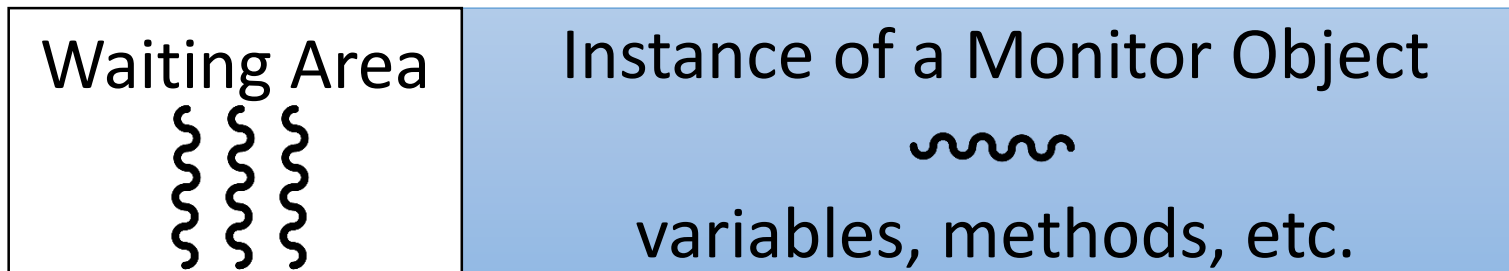
- Typically provided in object-oriented languages (Java, C#)
- Define an object as a monitor:
 - object's methods are automatically treated as critical sections
 - compiler adds implicit lock/unlock to beginning/end of every method
 - only one thread at a time is allowed to execute “in the monitor”



Monitors

- Typically provided in object-oriented languages (Java, C#)
- Define an object as a monitor:
 - object's methods are automatically treated as critical sections
 - compiler adds implicit lock/unlock to beginning/end of every method
 - only one thread at a time is allowed to execute “in the monitor”

Threads arrive to use the object. Only one is allowed to use it at a time.



Monitors

- Typically provided in object-oriented languages (Java, C#)
- Define an object as a monitor:
 - object's methods are automatically treated as critical sections
 - compiler adds implicit lock/unlock to beginning/end of every method
 - only one thread at a time is allowed to execute “in the monitor”
- For *some* tasks, adding a keyword is all you need!

Easy Example: linked lists

- Suppose you're implementing a linked list class in Java, and it's allowed to grow unbounded.

```
public class LinkedList {  
    public synchronized void add(int value) {  
        /* Code to add new LL node. */  
    }  
  
    public synchronized void remove(int value) {  
        /* Code to find and remove LL node. */  
    }  
}
```

`synchronized`:
automatically associate a
mutex with every instance
of this class.

Acquire the mutex before
executing any synchronized
method.

Add one keyword, compiler adds all the locking for you.
No worries about multiple threads modifying the list simultaneously!

Monitors

- Some problems still require more explicit synchronization.
- Solution: give users condition variables. They work like the pthread variety you (maybe) saw in CS 31:

- `wait(cond) :`
- `signal(cond) :`



The interface looks the same as semaphores. DON'T BE FOOLED!

Monitors

- Some problems still require more explicit synchronization.
- Solution: give users condition variables. They work like the pthread variety you (maybe) saw in CS 31:
 - `wait(cond)` : release mutex (implicit for monitors) and block calling thread until another thread signals the condition variable.
 - `signal(cond)` :



The interface looks the same as semaphores. DON'T BE FOOLED!

Monitors

- Some problems still require more explicit synchronization.
- Solution: give users condition variables. They work like the pthread variety you (maybe) saw in CS 31:
 - `wait(cond)` : release mutex (implicit for monitors) and block calling thread until another thread signals the condition variable.
 - `signal(cond)` : (must be holding mutex). wake up one blocked thread and then release the mutex and exit the monitor.



The interface looks the same as semaphores. DON'T BE FOOLED!

Unlike the semaphore routines of the same names, condition variables DO NOT count the number of times they've been called. If nobody is waiting, signal does nothing!

How far will these code blocks execute?

Semaphore

```
semaphore s (count = 0)
signal(s)
wait(s)
wait(s)
(other code)
```

Condition variable

```
cond_var c
signal(c)
wait(c)
wait(c)
(other code)
```

Answer choice	Semaphore blocks on...	Condition variable blocks on...
A.	First wait()	First wait()
B.	Second wait()	First wait()
C.	Second wait()	Second wait()
D.	(other code)	Second wait()

Bounded Producer / Consumer

Monitor Object: Instance Variables & Methods

```
int buf[N], in = 0, out = 0, count = 0;  
cond spot_avail, item_avail;
```

```
synchronized PutItem (int item) {  
    if (count == N)  
        wait (spot_avail);  
    buf[in] = item;  
    in = (in + 1)%N;  
    count++;  
    signal (item_avail);  
}
```

```
synchronized GetItem () {  
    int item;  
    if (count == 0)  
        wait (item_avail);  
    item = buf[out];  
    out = (out + 1)%N;  
    count--;  
    signal (spot_avail);  
    return (item);  
}
```

Producer Thread

```
while (TRUE) {  
    PutItem(Produce());  
}
```

Using the object is still easy:
synchronization hidden.

Consumer Thread

```
while (TRUE) {  
    Consume (GetItem());  
}
```

Beyond CS 31

- In 31, you saw the pthreads library.
- So far, we've seen the OS support it by providing atomic primitives.
- Up next:
 1. Monitors: built-in language support for automatic synchronization
 2. Optimistic concurrency: concurrency without locking
 3. Mixing signals and threads

Optimistic Concurrency

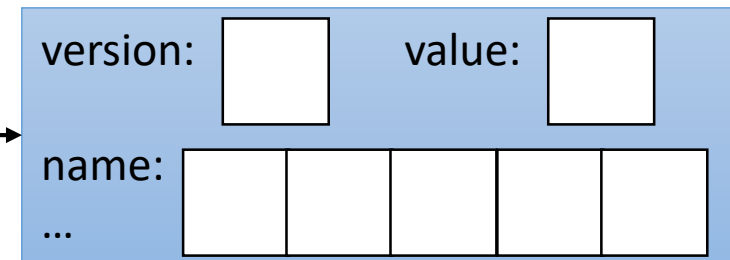
- Traditional synchronization is pessimistic
 - assume the worst case will happen (context switch at inopportune time)
 - prevent it from happening by locking in advance to protect critical section
- Observation: if multiple threads aren't trying to write, the locking is only getting in the way.
- Example: Wikipedia
 - Lots of pages, anyone can edit them. Most are NOT being edited right now.

Optimistic Concurrency

- Instead of locking, make copies of data and swap the pointer to write.

```
struct data {  
    int version; /* for writes */  
    int value;  
    char name[100];  
    ...  
}
```

```
/* Shared by multiple threads. */  
struct data *d
```



Any thread that wants to read the shared data can do so.

Optimistic Concurrency

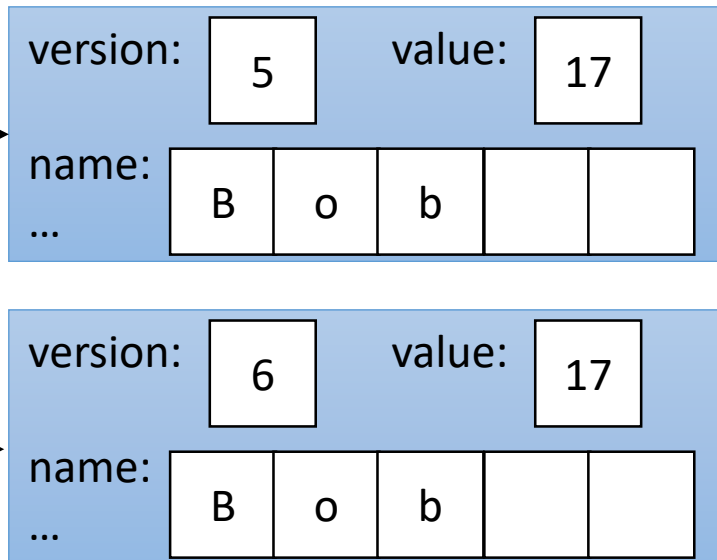
- Instead of locking, make copies of data and swap the pointer to write.

```
/* Shared by multiple threads. */
```

```
struct data *d
```

```
/* Private copy for writer. */
```

```
struct data *d_copy
```



If a thread wants to write, it first makes a copy of the whole data & increments version.

Optimistic Concurrency

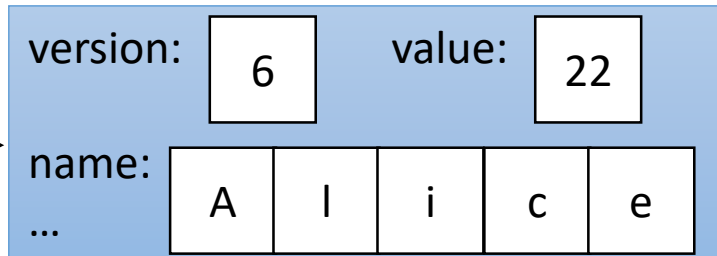
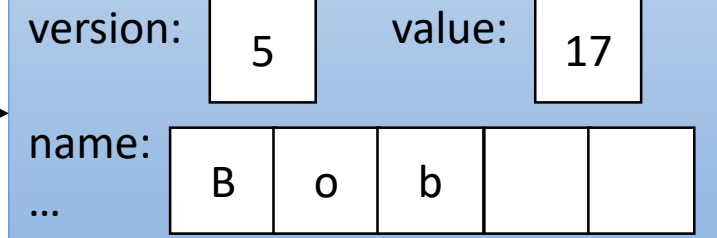
- Instead of locking, make copies of data and swap the pointer to write.

```
/* Shared by multiple threads. */
```

```
struct data *d
```

```
/* Private copy for writer. */
```

```
struct data *d_copy
```



Writer thread can update whatever it wants in private copy.

This is where compare and swap-like functionality shines! In one atomic operation: compare the versions, and if no other thread has modified the structure (changed the version before you), atomically swap the pointers.

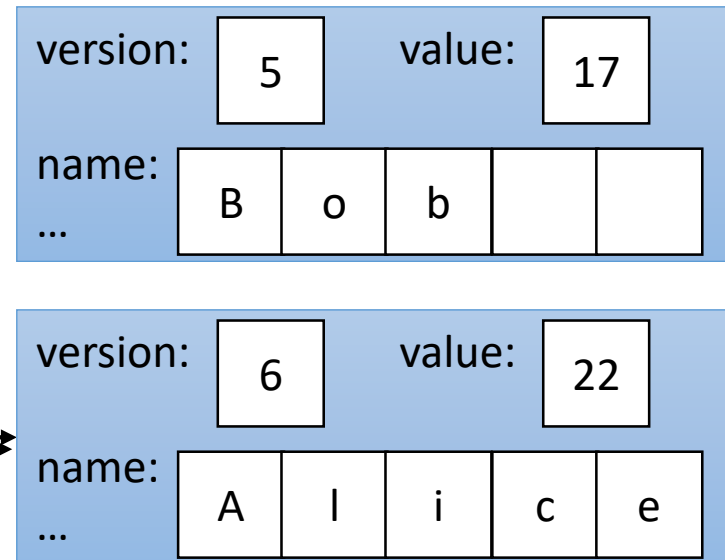
- Instead of locking, make copies of data and swap the pointer to write.

```
/* Shared by multiple threads. */
```

```
struct data *d
```

```
/* Private copy for writer. */
```

```
struct data *d_copy
```



When it's done, atomically swap the original pointer, IF version is expected value.

Works best when writes are scarce.
Commonly used in database transactions.

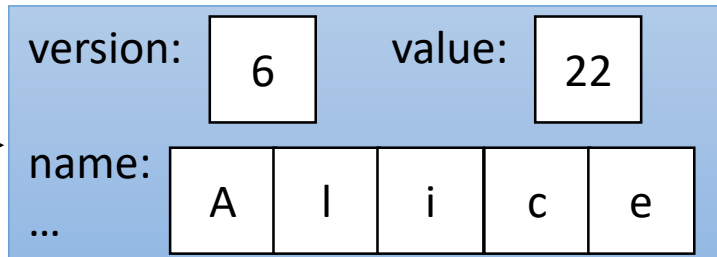
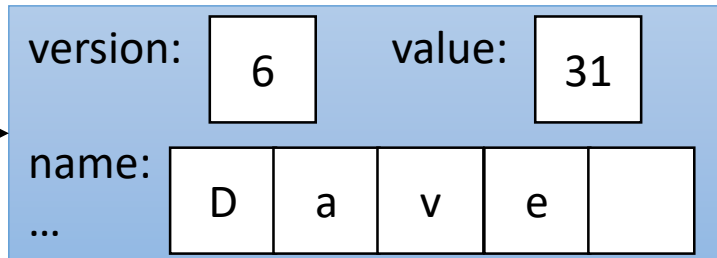
- Instead of locking, make copies of data and swap the pointer to write.

```
/* Shared by multiple threads. */
```

```
struct data *d
```

```
/* Private copy for writer. */
```

```
struct data *d_copy
```



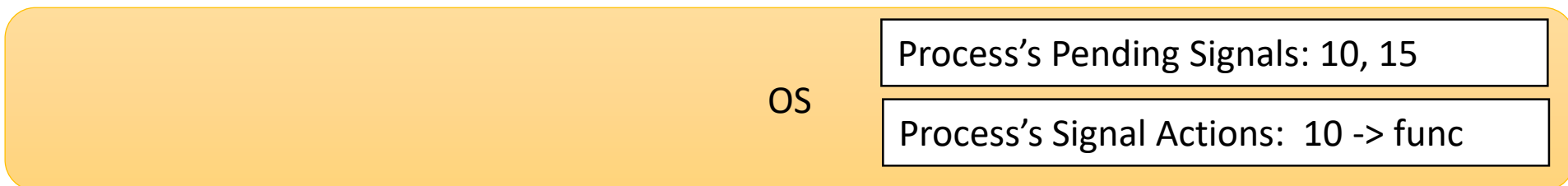
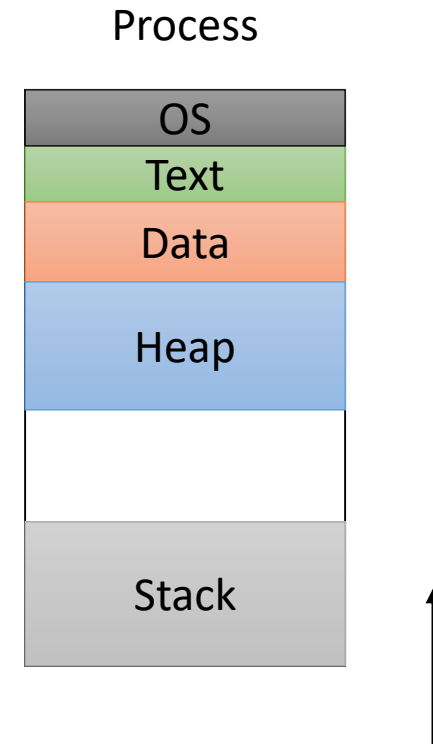
When versions indicate another thread has written, must handle how to proceed!

Beyond CS 31

- In 31, you saw the pthreads library.
- So far, we've seen the OS support it by providing atomic primitives.
- Up next:
 1. Monitors: built-in language support for automatic synchronization
 2. Optimistic concurrency: concurrency without locking
 3. Mixing signals and threads

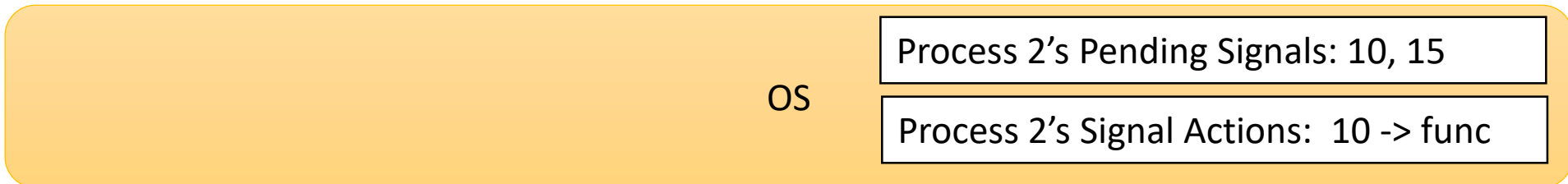
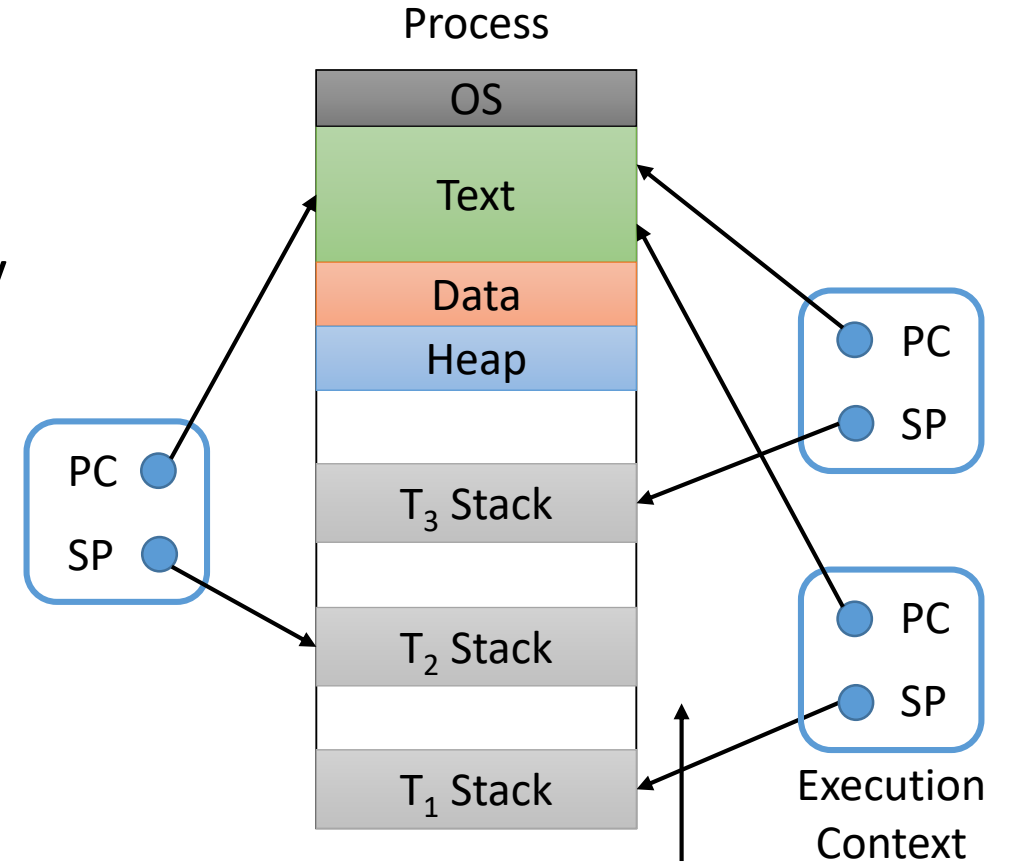
Recall: Signals

- Data transfer: send one integer.
- Synchronization: OS marks signals as pending, delivers signals when it feels like it (when scheduler chooses target)
- Used for:
 - terminating processes
 - asking long-running processes to do maintenance
 - debugging
 - OS sending simple message to process (e.g., SIGPIPE)



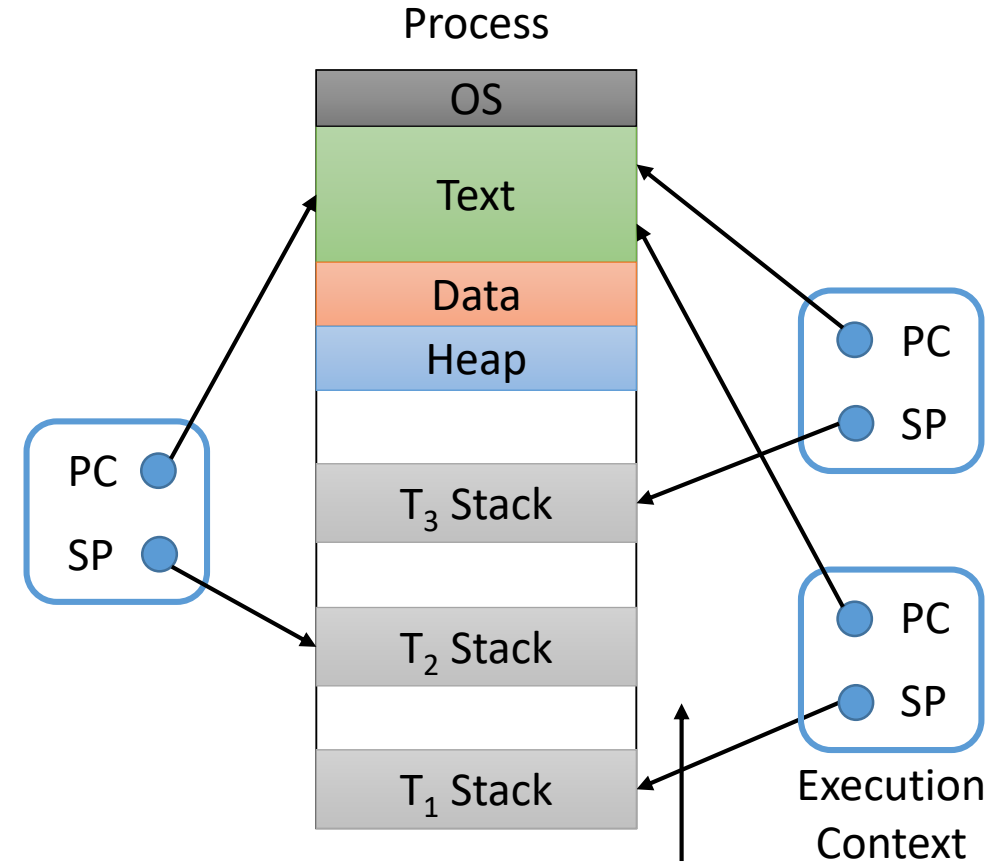
Signals & Threads

- Signals are delivered to a process (sent to a PID)
- But...now a process has multiple, independently scheduled execution contexts.
- Which should handle the signal?



Which thread should get the signal? Why?

- A. The signal should always go to the first/main thread.
- B. The OS should choose one thread to handle all subsequent signals.
- C. The user should choose one thread to handle all signals.
- D. The signal should go to any available thread.



OS

Process 2's Pending Signals: 10, 15

Process 2's Signal Actions: 10 -> func

Take Care Mixing Signals and Threads!

- You might end up interrupting a thread you didn't want to interrupt.
- If you're lucky: all threads are doing the same thing, so any of them can handle a signal.
- Otherwise: you can disable signals on a per-thread basis.
 - Tell all threads but one to ignore all signals.
 - Designate one thread to sit around and only wait for signals: `sigwait()`

Summary

- We extended our process model:
 - Process is state (PCB), a virtual address space, and open files
 - Process has multiple execution contexts, 1:1 with threads (most OSes)
- Concurrency gives nice performance, but leads to race conditions.
- Atomicity helps to solve races, but we want atomic hardware instructions.
- OS provides low-level synchronization primitives, which serve as basis for higher-level libraries, language support (monitors) and other abstractions.