Concurrency

Other Examples

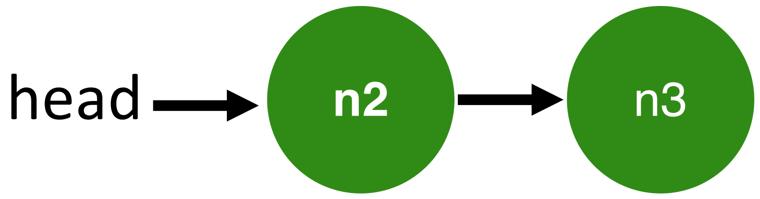
Consider multi-threaded applications that do more than increment shared balance

Multi-threaded application with shared linked-list

- All concurrent:
    - Thread A inserting element a
    - Thread B inserting element b
    - Thread C looking up element c

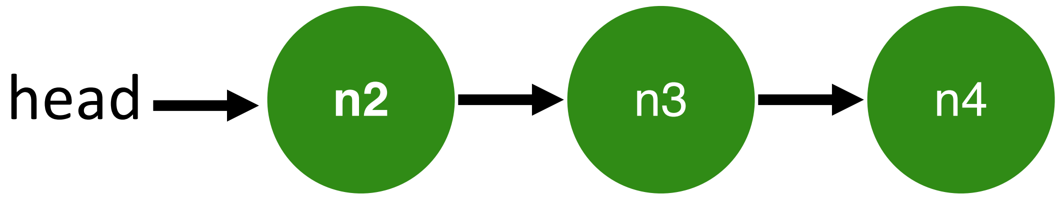
Shared Linked List

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```



Shared Linked List

```
Void List_Insert(list_t *L, int key) {
        node_t *new =
malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}

int List_Lookup(list_t *L,  int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                                return
1;
                tmp = tmp->next;
        }
return 0;
}
```

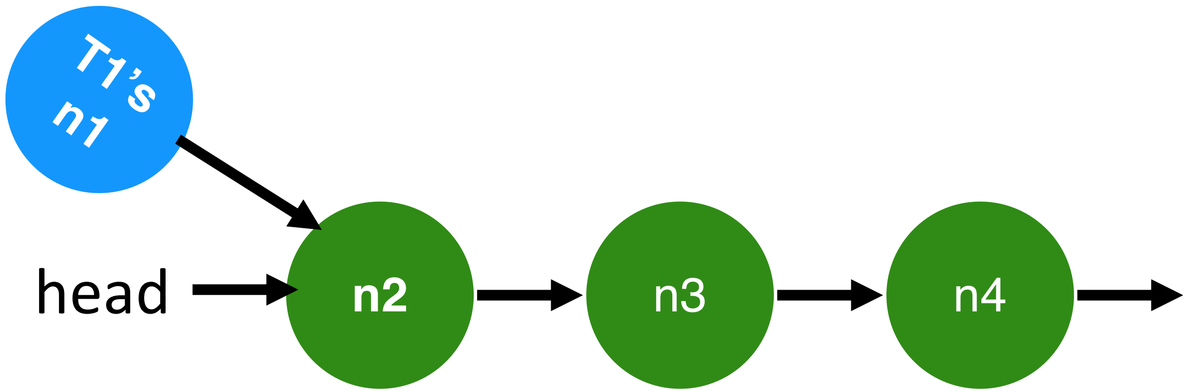


What can go wrong?
Find a schedule that leads to problem?

Linked-List Race

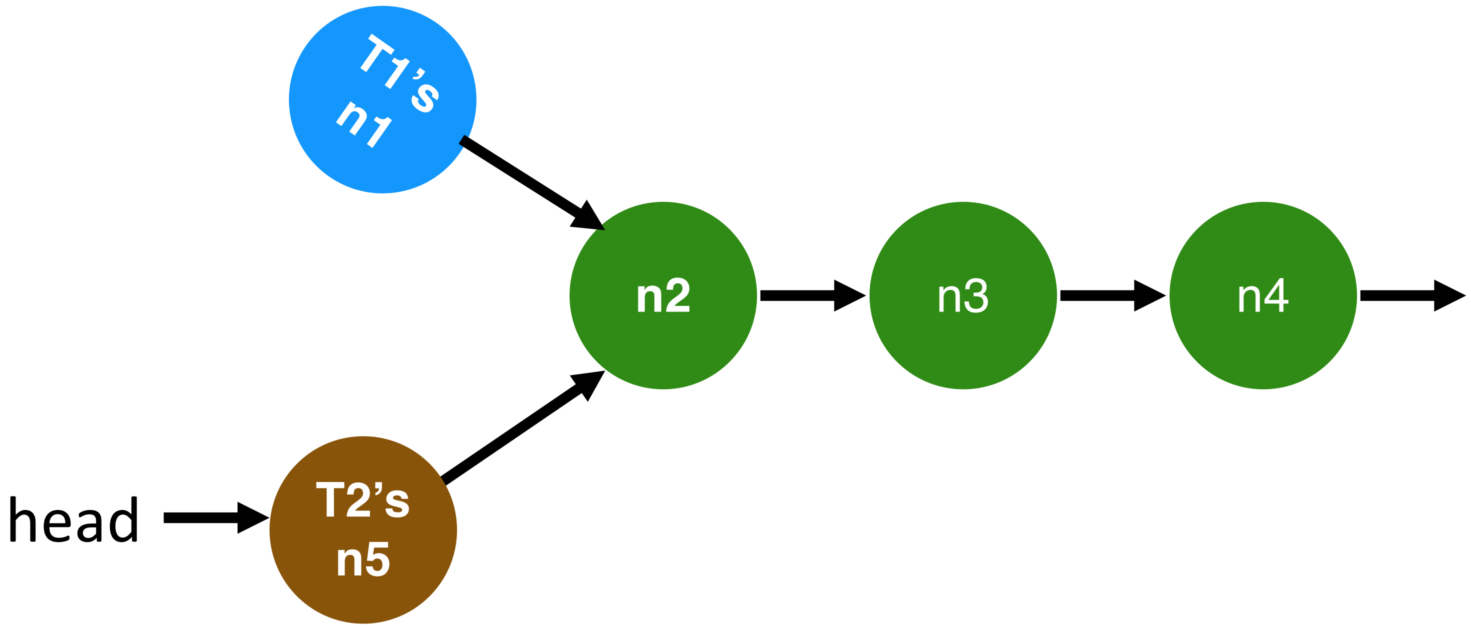
Thread 1	Thread 2
new->key = key	
new->next = L->head	

Linked-List Race



Linked-List Race

Thread 1	Thread 2
new->key = key	
new->next = L->head	
Cntxt_Switch()	new->key = key
	new->next = L->head
	L->head = new


T1's
n1
n2
n3
n4
head
T2's
n5

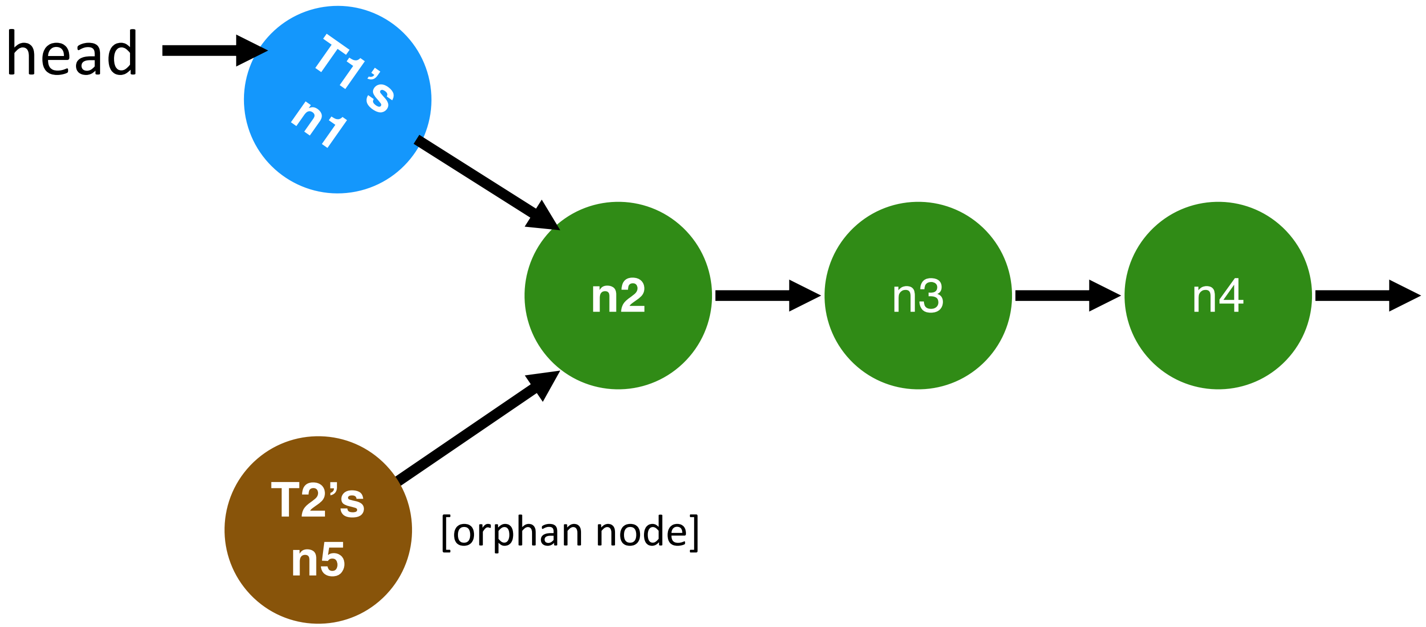
Linked-List Race

Thread 1	Thread 2
new->key = key	
new->next = L->head	
Cntxt_Switch()	new->key = key
	new->next = L->head
	L->head = new Cntxt_Switch()
L->head = new	

Both entries point to old head

Only one entry (which one?) can be the new head.

Resulting Linked List



Locking Linked Lists

```
Void List_Insert(list_t *L, int key) {
        node_t *new =
                malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}

int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                                return
1;
                tmp = tmp->next;
        }
    return 0;
}
```

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

How to add locks?

Locking Linked Lists

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

How to add locks?

pthread_mutex_t lock;

One lock per list

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}
```

Locking Linked Lists : Approach #1

Consider everything critical section

```
Pthread_mutex_lock(&L->lock);      Void List_Insert(list_t *L,  int key) {
                                     node_t *new = malloc(sizeof(node_t));
                                           assert(new);
                                           new->key = key;
                                           new->next = L->head;
                                           L->head = new;
Pthread_mutex_unlock(&L->lock);    }

Pthread_mutex_lock(&L->lock);      int List_Lookup(list_t *L, int key) {
                                           node_t *tmp = L->head;
                                           while (tmp) {
                                                   if (tmp->key == key)

                                                   return 1;
                                                   tmp = tmp->next;
Pthread_mutex_unlock(&L->lock);            }
                                       return 0;
                                   }
```

Locking Linked Lists : Approach #2

```
Void List_Insert(list_t *L,  int key) {
```

Can critical section be smaller?

Pthread_mutex_lock(&L->lock);

```
  node_t *new = malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
```

Pthread_mutex_unlock(&L->lock);

```
}
int List_Lookup(list_t *L, int key) {
```

Pthread_mutex_lock(&L->lock);

```
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)

                return 1;
                tmp = tmp->next;
```

Pthread_mutex_unlock(&L->lock);

```
        }
    return 0;
}
```

Locking Linked Lists : Approach #2

Can critical section be smaller?

```
Void List_Insert(list_t *L,  int key) {
                    Pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
                    Pthread_mutex_unlock(&L->lock);
}
```

```
int List_Lookup(list_t *L, int key) {
                    Pthread_mutex_lock(&L->lock);
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)

                return 1;
                tmp = tmp->next;
        }
                    Pthread_mutex_unlock(&L->lock);
     return 0;
}
```

Locking Linked Lists : Approach #3

Can critical section be smaller?

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

~~Pthread_mutex_lock(&L->lock);~~

If no List_Delete(), locks not needed
(Any non-head element cannot change once inserted)

~~Pthread_mutex_unlock(&L->lock);~~

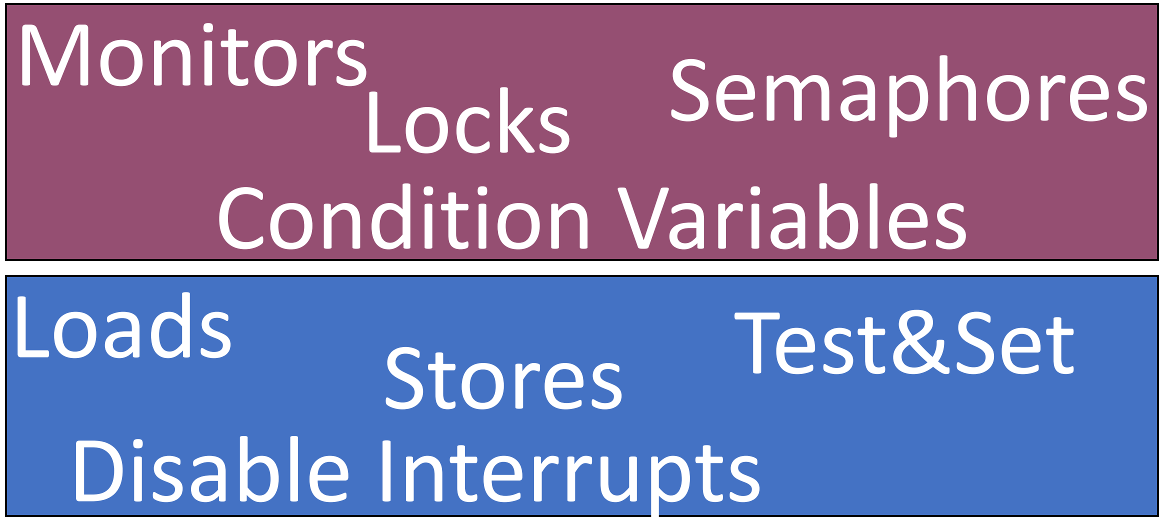
```
Void List_Insert(list_t *L,  int key) {

  node_t *new = malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}
int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)

                return 1;
                tmp = tmp->next;
        }
     return 0;
}
```

Implementing Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right



Lock Implementation Goals

Correctness

- Mutual exclusion
    - Only one thread in critical section at a time
- Progress (deadlock-free)
    - If several simultaneous requests, must allow one to proceed
- Bounded (starvation-free)
    - Must eventually allow each waiting thread to enter

Fairness

Each thread waits for same amount of time

Performance

CPU is not used unnecessarily (e.g., spinning)

Implementing Synchronization

To implement, need atomic operations (with hardware support)

Atomic operation: No other instructions can be interleaved

Examples of atomic operations

- Code “between” interrupts on uniprocessors
  - Disable timer interrupts, don’t do any I/O
- Loads and stores of words (aligned, hardware-native load/store size)
  - Load r1, B
  - Store r1, A
- **Special hardware instructions**
  - **Test&Set**
  - **Compare&Swap**
  - **Fetch&Add**

Implementing Locks with Interrupts

Turn off interrupts for critical sections

- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

```
void lock(lock_t *l) {
      disableInterrupts();
}
void unlock(lock_t *l) {
      enableInterrupts();
}
```

Disadvantages??

Don’t want to allow user-space code disable interrupts

What if process crashes or runs away with the processor without yielding?

Stopping interrupts for too long is bad (can’t perform other useful work)

Only works on uniprocessors

Implementing LOCKS: w/ Load+Store

Code uses a single **shared** lock variable

```
Boolean flag = false; // shared variable
Void lock(Boolean *flag) {
     while (*flag) /* wait */ ;
     *flag = true;
}
Void unlock(Boolean *flag) {
     *flag = false;
}
```

Does this work?

Can you produce an example schedule that fails with 2 threads?

Race Condition with LOAD and STORE

```
*flag == 0 initially

Thread 1                              Thread 2
-------------------------------------------------------
while(*flag == 1);
                                      while(*flag== 1);
                                      *flag = 1
*flag = 1
```

Both threads grab lock!

Problem: Testing lock and setting lock are not atomic

Next solutions: Build on hardware atomic instructions

xchg: atomic exchange, or test-and-set

```
// xchg(int *addr, int newval)
// return what was pointed to by addr
// at the same time, atomically store newval into addr

int xchg(int *addr, int newval) {
  int old = *addr;
  *addr = newval;
  return old;
}

static inline uint
xchg(volatile unsigned int *addr, unsigned int newval)
{
    uint result;
    asm volatile("lock; xchgl %0, %1" :
                 "+m" (*addr), "=a" (result) :
                 "1" (newval) : "cc");
    return result;
}
```

LOCK Implementation with XCHG

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t *lock) {
        lock->flag = ??;
}

void lock(lock_t *lock) {
        ????;
        // hint: spin-wait (do nothing)
}

void unlock(lock_t *lock) {
        lock->flag = ??;
}
```

Use

```
int xchg(int *addr, int newval)
```

LOCK implementation with XCHG

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t *lock) {
        lock->flag = 0;
}

void acquire(lock_t *lock) {
        while(xchg(&lock->flag, 1) == 1) ;
        // spin-wait (do nothing)
}

void release(lock_t *lock) {
        lock->flag = 0; // no atomics here
}
```

More atomic HW: Compare and Swap

```
int CompareAndSwap(int *addr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}
```

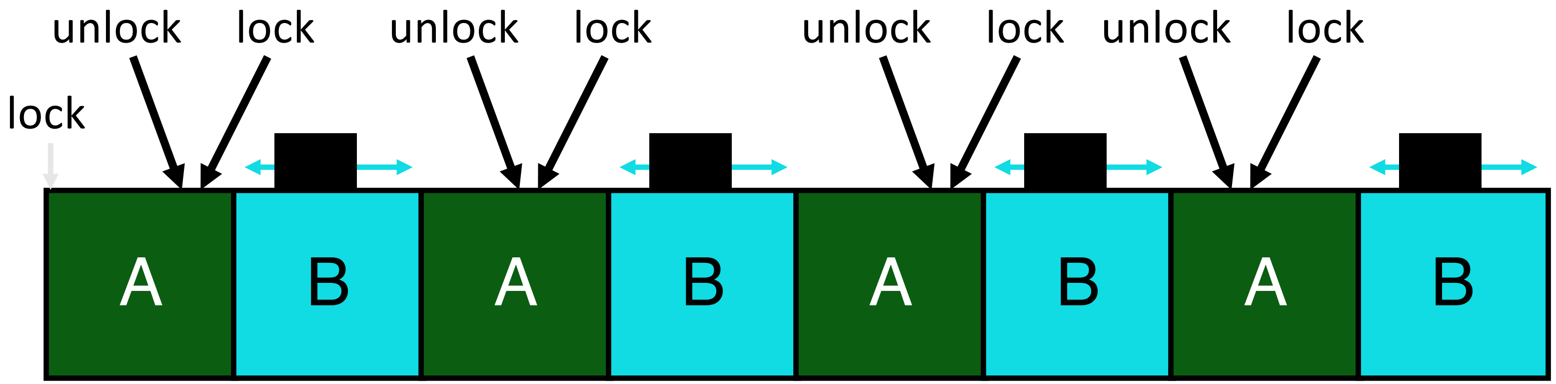
```
void lock(lock_t *lock) {
      ??;
     // use CompareAndSwap(&lock->flag, ?, ?)
     // hint: spin-wait (do nothing)
}
```

More atomic HW: Compare and Swap

```
int CompareAndSwap(int *addr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}
```

```
void lock(lock_t *lock) {
     while(CompareAndSwap(&lock->flag, 0, 1)
                    == 1) ;
     // spin-wait (do nothing)
}
```

Basic Spinlocks are Unfair



OS Scheduler is typically independent of locks/unlocks

Fairness: Ticket Locks

Idea: reserve each thread's turn to use a lock.

Each thread spins until their turn.

Use yet another hardware atomic primitive, fetch-and-add:

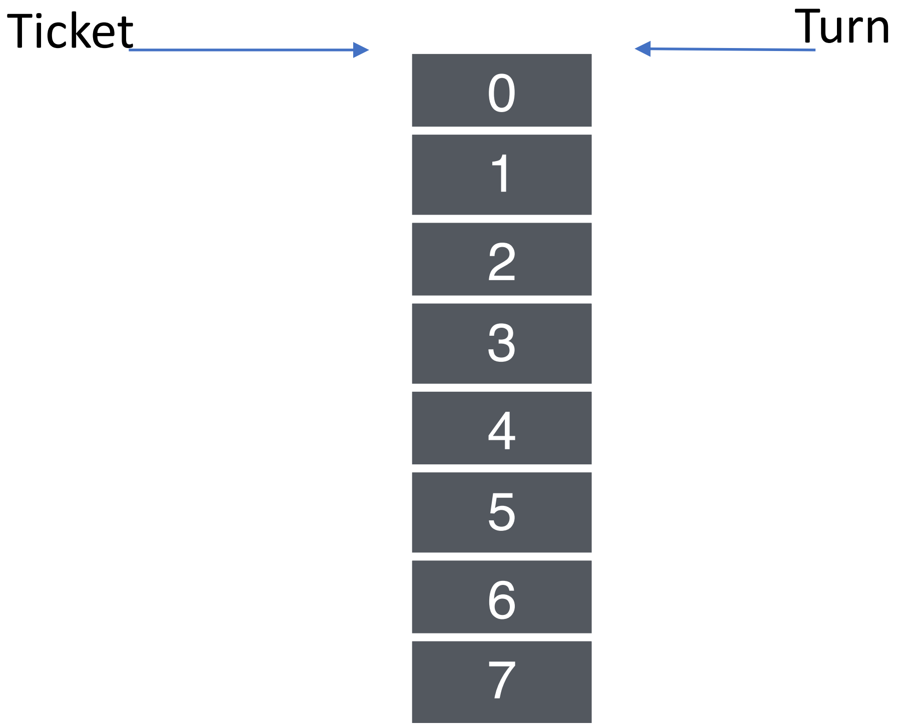
```
int FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

Lock: Grab ticket; Spin while not thread's ticket != turn

Unlock: Advance to next turn

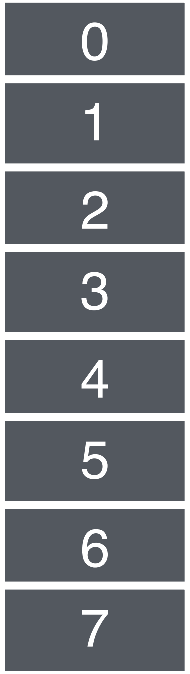
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():



Ticket Lock Example

A lock(): gets ticket 0, spins until turn = 0 →runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++ (turn = 1)
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++ (turn = 2)
C runs
C unlock(): turn++ (turn = 3)
A runs
A unlock(): turn++ (turn = 4)
C lock(): gets ticket 4, runs



Ticket Lock Implementation

```
typedef struct __lock_t {
        int ticket;
        int turn;
}

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
```

```
void lock(lock_t *lock) {
        int myturn = FAA(&lock->ticket);
        while (lock->turn != myturn); // spin
}

void unlock (lock_t *lock) {
        FAA(&lock->turn);
}
```

Spinlock Performance

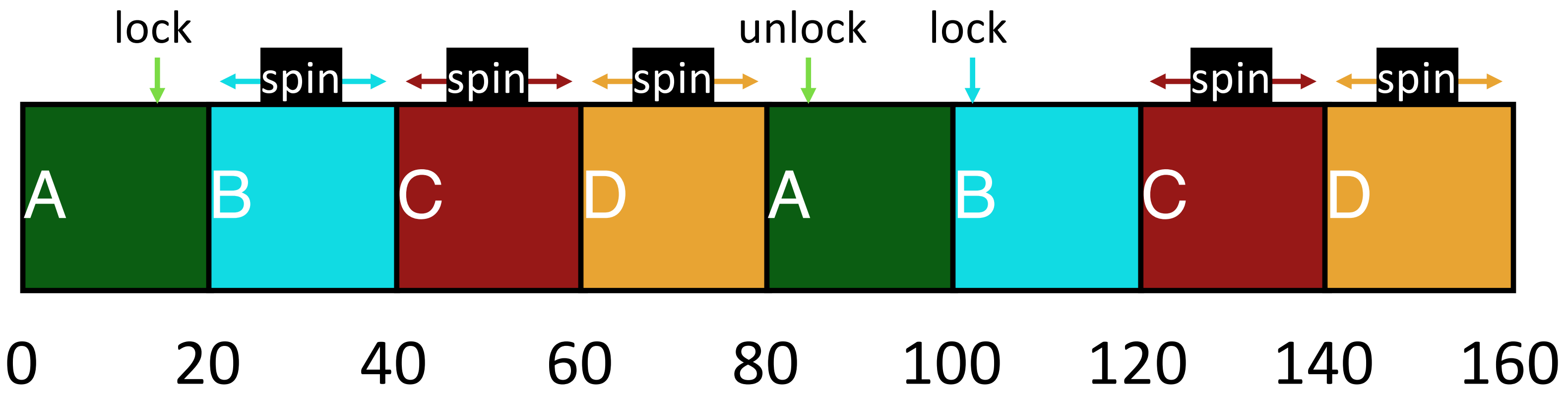
Fast when…

- many CPUs
- locks held a short time
- advantage: avoid context switch

Slow when…

- one CPU
- locks held a long time
- disadvantage: spinning is wasteful

Spinlocks may be CPU-inefficient



CPU scheduler may run **B** instead of **A**
even though **B** is waiting for **A**

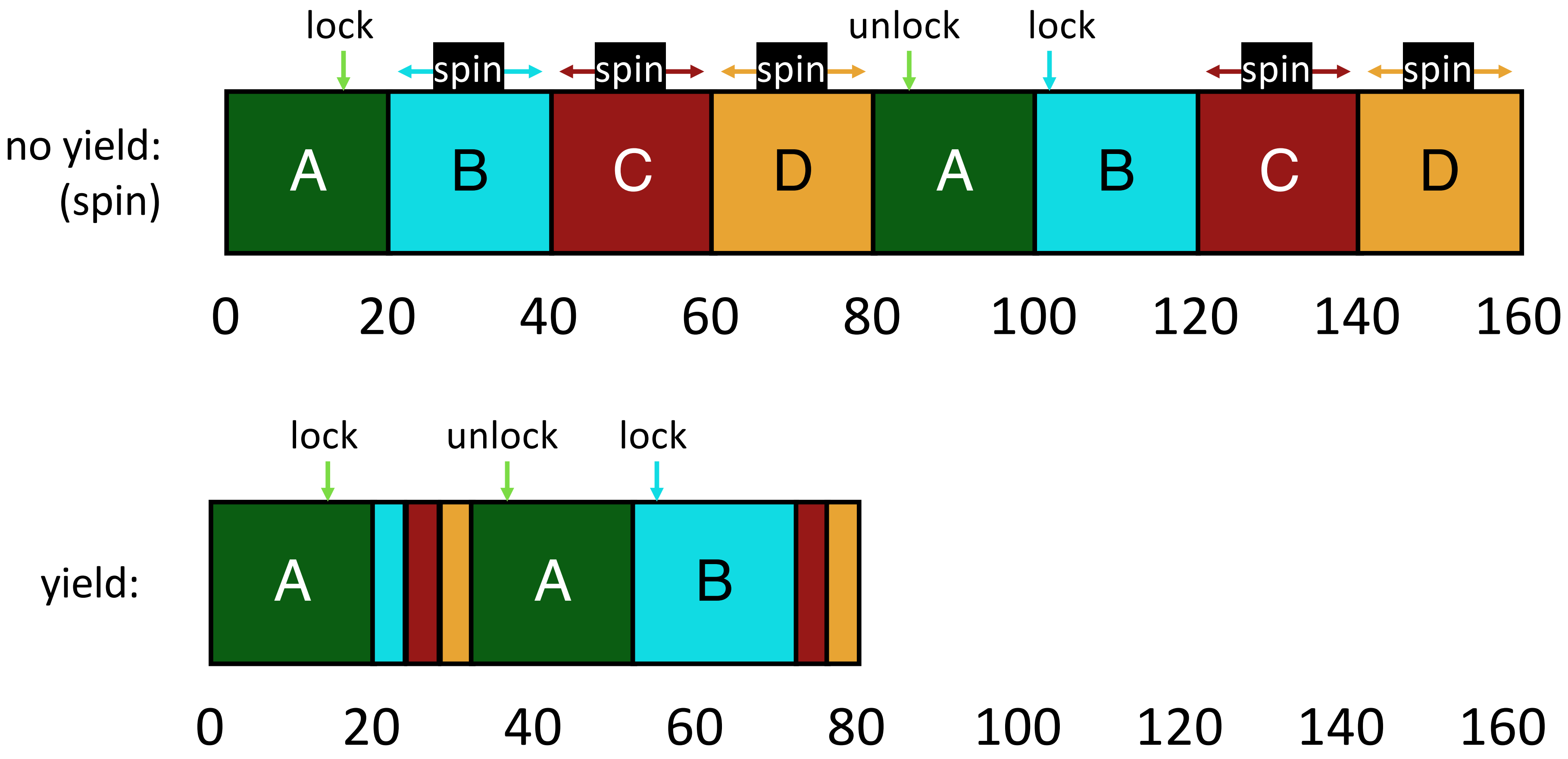
Significant inefficiency on uniprocessors
(e.g. N-1 threads waiting one quantum each for N’th thread to release the lock)

Could we just `yield()` when waiting?

```
typedef struct __lock_t {
        int ticket;
        int turn;
}

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
```

```
void lock(lock_t *lock) {
        int myturn = FAA(&lock->ticket);
        while(lock->turn != myturn)
                yield();
}

void unlock(lock_t *lock) {
        FAA(&lock->turn);
}
```

`Yield` Instead of Spin



Spinlock vs. `yield` Performance

How much CPU wasted?

Without yield: O(threads * **time_slice**)

With yield: O(threads * **context_switch**)

So even with yield, spinning is slow with high thread contention

Next improvement: Block (don’t schedule a thread until unlocked) and put thread on waiting queue instead of check and yield each time

OS support in implementing locks is helpful!

Useful to also enforce fairness (e.g., unblock one at a time)

See text for example implementations of blocking-based locks

Concurrency Objectives

Mutual exclusion (e.g., A and B don't run at same time)

- solved with *locks*

Ordering (e.g., B runs after A does something)

- solved with *condition variables* and *semaphores*

Condition Variables

wait(cond_t *cv, mutex_t *lock)

- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)

  Don’t spin wait for condition to become true
- when awoken, reacquires lock before returning

signal(cond_t *cv)

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing

Join Implementation: Correct

Parent:

```
void thread_join() {
        Mutex_lock(&m);             // w
        if (done == 0)              // x
                Cond_wait(&c, &m);  // y
        Mutex_unlock(&m);           // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);             // a
        done = 1;                   // b
        Cond_signal(&c);            // c
        Mutex_unlock(&m);           // d
}
```

Note: Cond_wait also releases mutex before waiting.
Provided condition is not met yet

Parent: w x y z

Child: a b c d

Use mutex (lock) m to ensure no race between interacting with state (`done`) and the wait/signal

Producer/Consumer Problem

Producers generate data (like pipe writers)

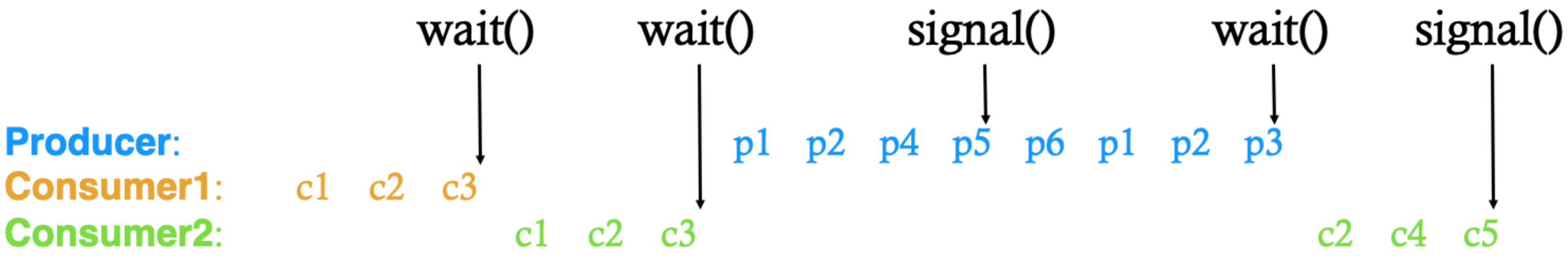
Consumers grab data and process it (like pipe readers)

Use condition variables to:
make producers wait when buffers are full
make consumers wait when there is nothing to consume

A broken Implementation of Producer Consumer

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        while(numfull == max) //p2
            Cond_wait(&cond, &m); //p3
        do_fill(i); // p4
        Cond_signal(&cond); //p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);  // c1
        while(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```



does last signal wake producer or consumer2?

Producer/Consumer: Two CVs

```
void *producer(void *arg) {
        for (int i = 0; i < loops; i++) {
                Mutex_lock(&m); // p1
                if (numfull == max) // p2
                    Cond_wait(&empty, &m); // p3
                do_fill(i);  // p4
                Cond_signal(&fill); // p5
                Mutex_unlock(&m); //p6
        }
}
```

```
void *consumer(void *arg) {
        while (1) {
                Mutex_lock(&m);  // c1
                if (numfull == 0) // c2
                        Cond_wait(&fill, &m); // c3
                int tmp = do_get(); // c4
                Cond_signal(&empty); // c5
                Mutex_unlock(&m);  // c6
        }
}
```

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer1 then reads bad data.

Is this correct? Can you find a bad schedule?

Producer:				p1	p2	p4	p5	p6								
Consumer1:	c1	c2	c3													c4! ERROR
Consumer2:									c1	c2	c4	c5	c6			

CV Rule of Thumb 3

Whenever a lock is acquired, recheck assumptions about state!

  Use “while” instead of “if”

Possible for another thread to grab lock between signal and wakeup from wait

- Difference between Mesa (practical implementation) and Hoare (theoretical) semantics
- Signal() simply makes a thread runnable, does not guarantee thread run next

Note that some libraries also have “spurious wakeups”

- May wake multiple waiting threads at signal or at any time

Producer/Consumer: Two CVs and WHILE

```
void *producer(void *arg) {
        for (int i = 0; i < loops; i++) {
                Mutex_lock(&m); // p1
                while (numfull == max) // p2
                        Cond_wait(&empty, &m); // p3
                do_fill(i);  // p4
                Cond_signal(&fill); // p5
                Mutex_unlock(&m); //p6
        }
}
```

```
void *consumer(void *arg) {
        while (1) {
                Mutex_lock(&m);
                while (numfull == 0)
                        Cond_wait(&fill, &m);
                int tmp = do_get();
                Cond_signal(&empty);
                Mutex_unlock(&m);
        }
}
```

Is this correct? Can you find a bad schedule?

Correct!

- no concurrent access to shared state
- every time lock is acquired, assumptions are reevaluated
- a consumer will get to run after every do_fill()
- a producer will get to run after every do_get()

Summary: rules of thumb for CVs

Keep state in addition to CV’s

Always do wait/signal with lock held

Whenever thread wakes from waiting, recheck state

Common concurrency bugs

- Violations of atomicity
- Violations of ordering
- Deadlocks
  - `T1: lock(l1); lock(l2);`
  - `T2: lock(l2); lock(l1);`

Conclusions

Concurrency is needed to obtain high performance by utilizing multiple cores

Threads are multiple execution streams within a single process or address space (share PID and address space, own registers and stack)

Context switches within a critical section can lead to non-deterministic bugs (race conditions)

Use locks to provide mutual exclusion, CVs for ordering

Improving performance requires reducing critical section cost