OSTEP
Locks and
Condition Variables

Questions answered in this lecture:

How can threads **block** instead of **spin-waiting** while waiting for a lock?

When should a waiting thread block and when should it spin?

How can threads enforce **ordering** across operations?

How can **thread_join()** be implemented?

How can **condition** variables be used to support **producer/consumer** apps?

Review: Ticket Lock

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

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

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

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

Are both FAA() instructions needed or can replace with simple ++?

Review: Ticket Lock

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

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

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

void release (lock_t *lock) {
        lock->turn++;
}
```

FAA() used in textbook → conservative

Try this modification in Homework simulations

Lock Evaluation

How to tell if a lock implementation is good?

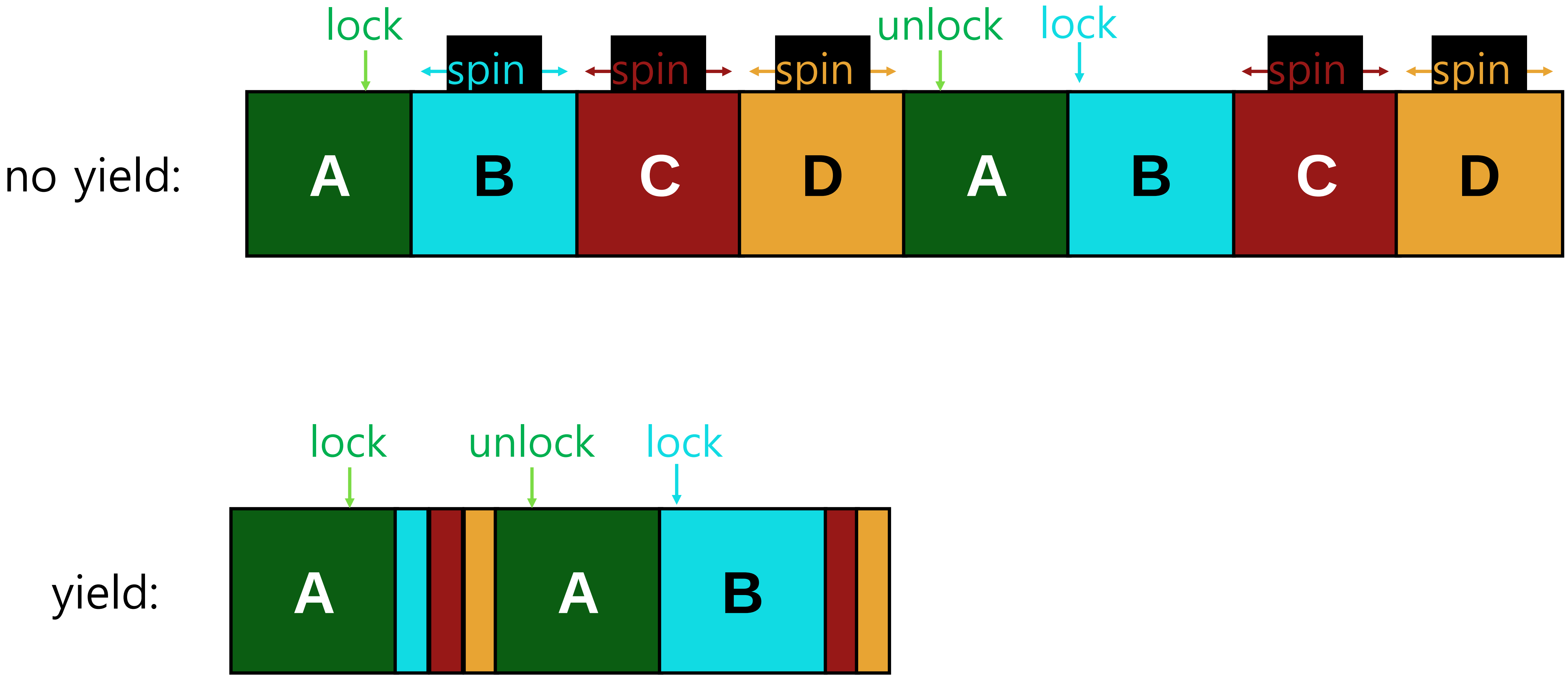
Fairness:

- Do processes acquire lock in same order as requested?

Performance

Two scenarios:

- \- low contention (fewer threads, lock usually available)
- \- high contention (many threads per CPU, each contending)



Why is yield useful?
Why doesn't yield solve all performance problems?

Spinlock Performance

Waste…

Without yield: O(threads * context_switch)

With yield: O(threads * time_slice)

So even with yield, we're slow with high contention.

Race Condition

Thread 1	Thread 2
if (lock->flag == 0)	
queue_push(lock->q, gettid());	
lock->guard = 0;	
	while (xchg(&lock->guard, 1) == 1)
	if (queue_empty(lock->q))
	unpark(queue_pop(lock->q));
	lock->guard = 0;
park();	
(in lock)	(in unlock)

Incorrect Code

```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
            ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
            lock->flag = 1;
            lock->guard = 0;
    } else {              // lock not free: sleep
            queue_push(lock->q, gettid());
            lock->guard = 0;
            park();        // put self to sleep
    }
}
```

```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
            ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
            lock->flag = 1;
            lock->guard = 0;
    } else {              // lock not free: sleep
            queue_push(lock->q, gettid());
            lock->guard = 0;
            park();          // put self to sleep
    }
}
```

Race!

```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
        ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
        lock->flag = 1;
        lock->guard = 0;
    } else {            // lock not free: sleep
        queue_push(lock->q, gettid());
        lock->guard = 0;
        park();         // put self to sleep
    }
}
```

Correct Code

```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
            ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
            lock->flag = 1;
            lock->guard = 0;
    } else {               // lock not free: sleep
            queue_push(lock->q, gettid());
            setpark();
            lock->guard = 0;
            park();          // put self to sleep
    }
}
```

Lock Implementation: Block when Waiting

Lock implementation removes waiting threads from scheduler r
eady queue (e.g., park() and unpark())

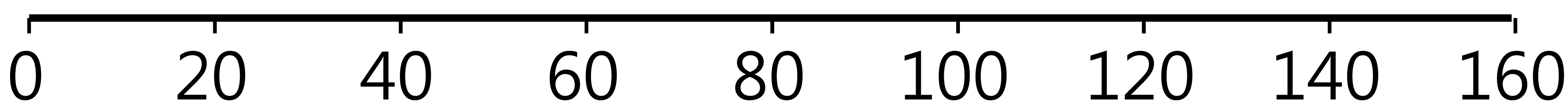
Scheduler runs any thread that is **ready**

Good separation of concerns

RUNNABLE: A, B, C, D

RUNNING: <empty>

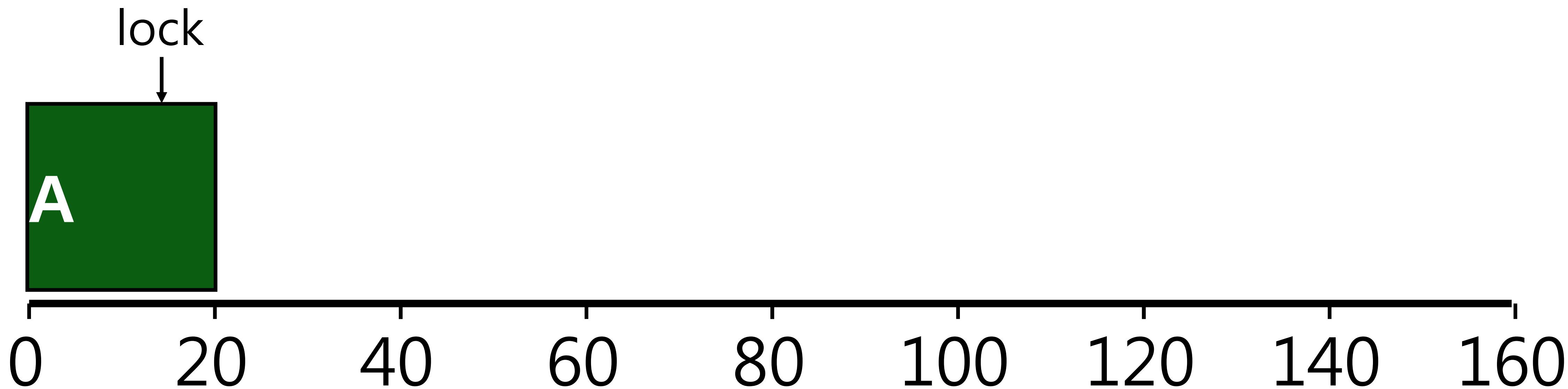
WAITING: <empty>



RUNNABLE: B, C, D

RUNNING: A

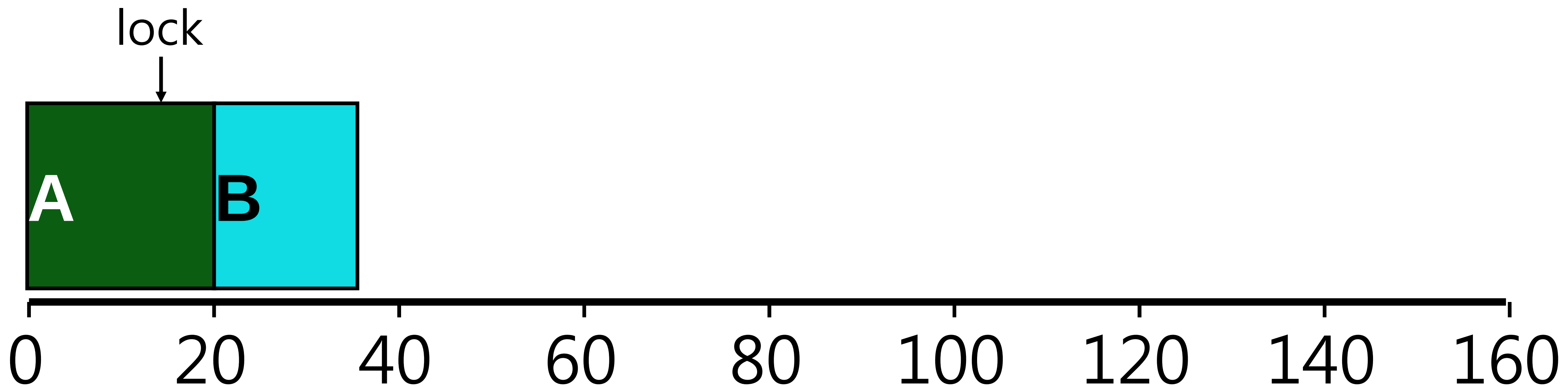
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING: B

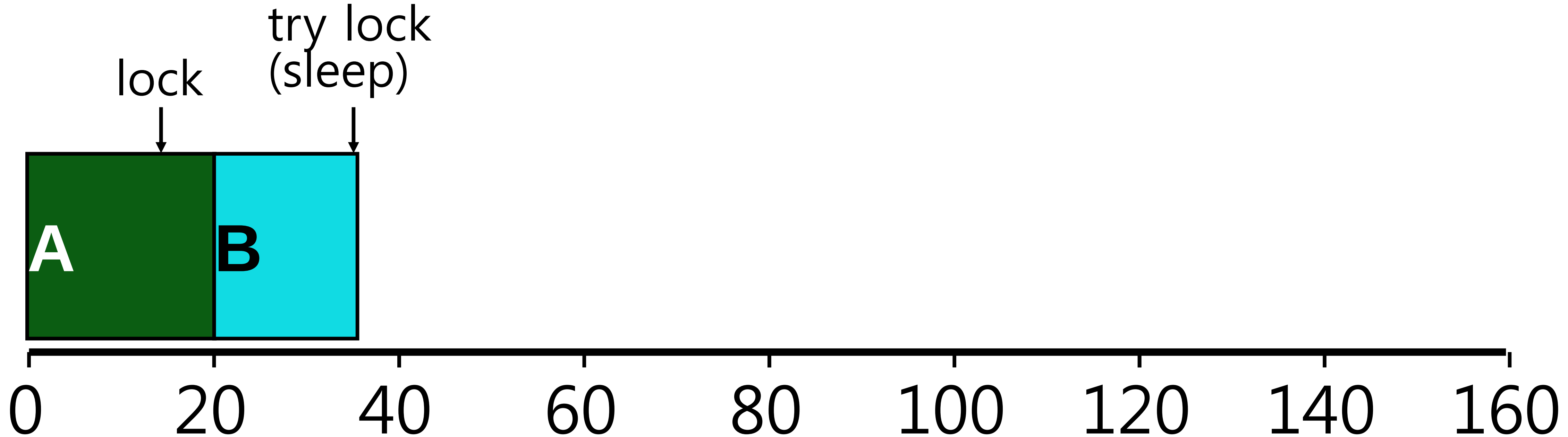
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING:

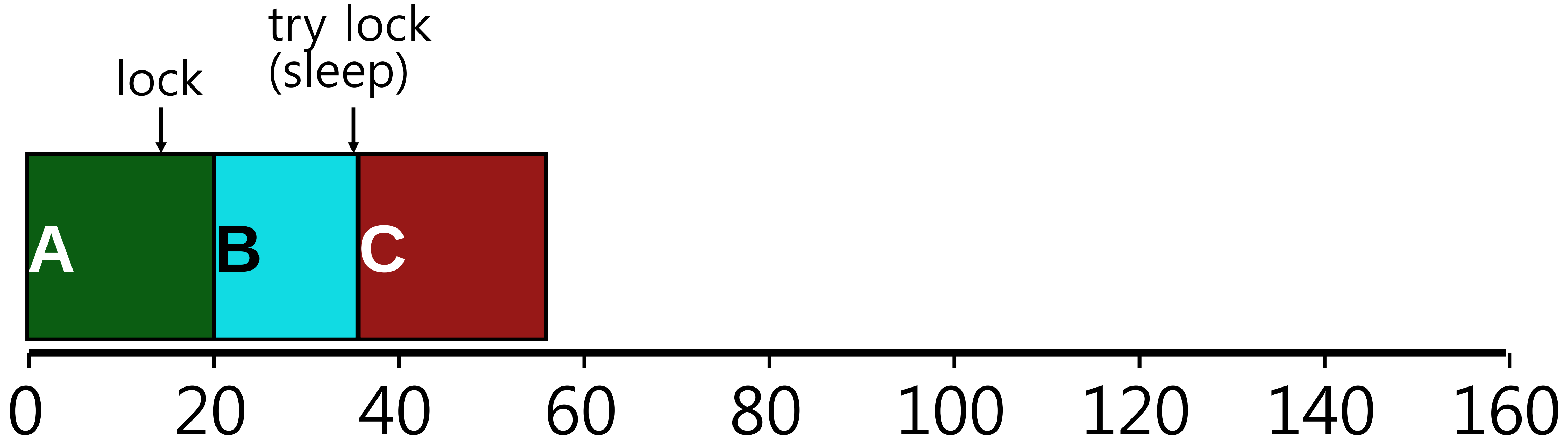
WAITING: B



RUNNABLE: D, A

RUNNING: C

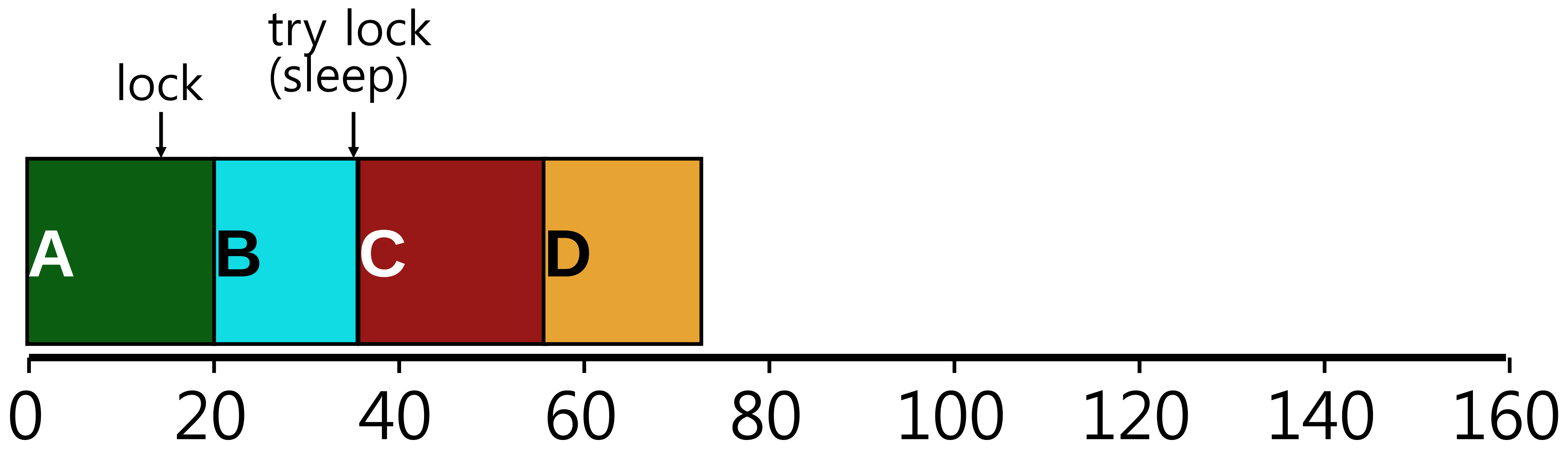
WAITING: B



RUNNABLE: A, C

RUNNING: D

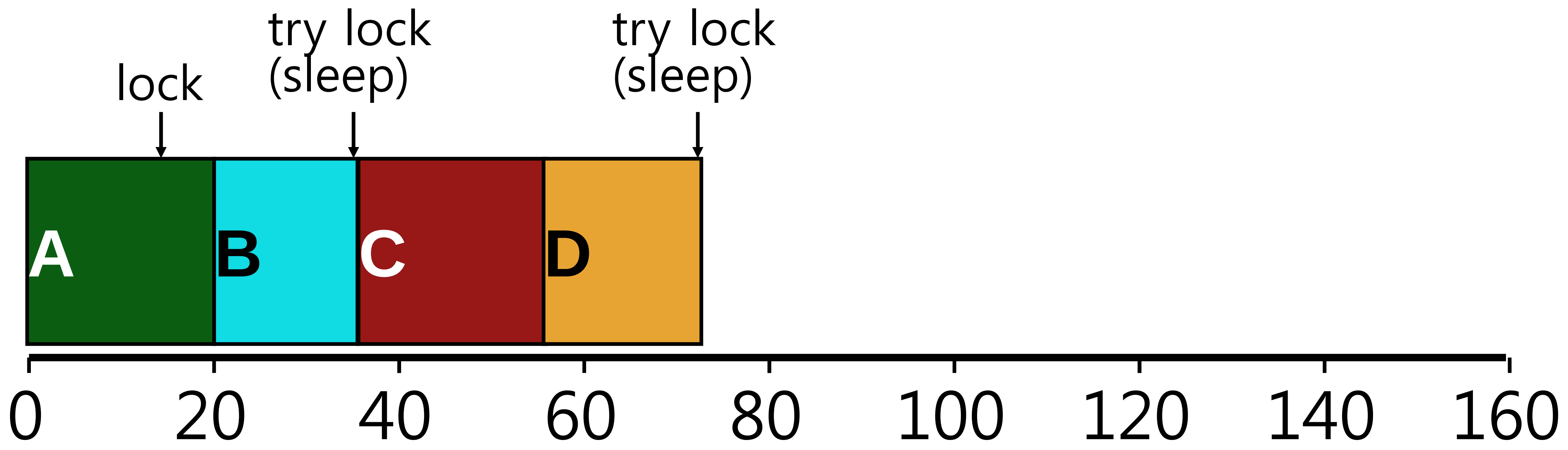
WAITING: B



RUNNABLE: A, C

RUNNING:

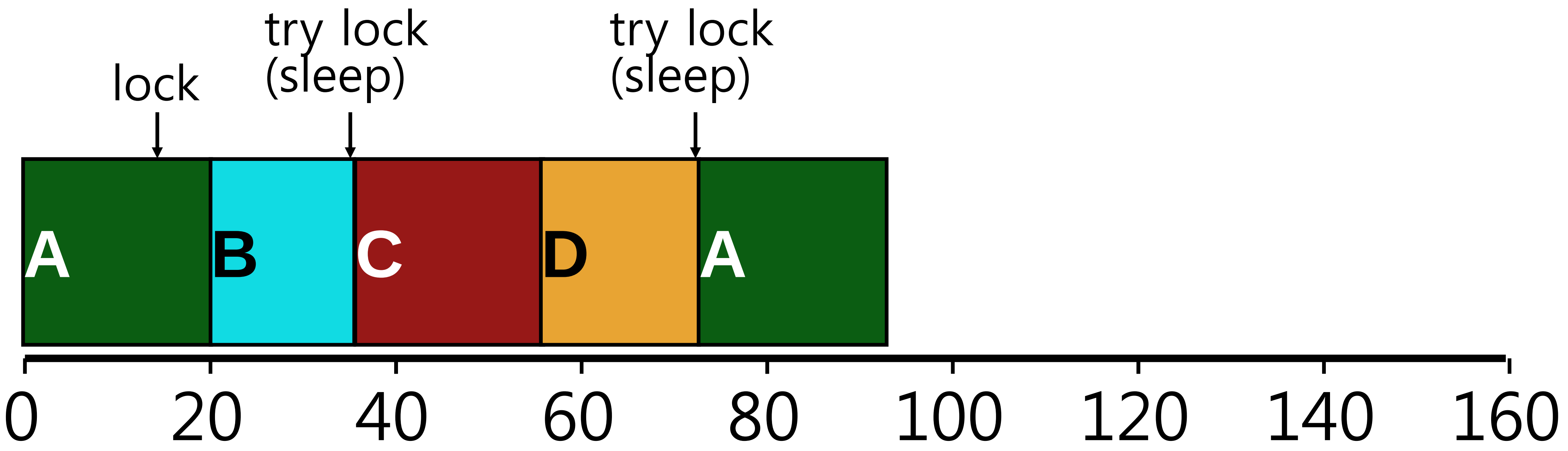
WAITING: B, D



RUNNABLE: C

RUNNING: A

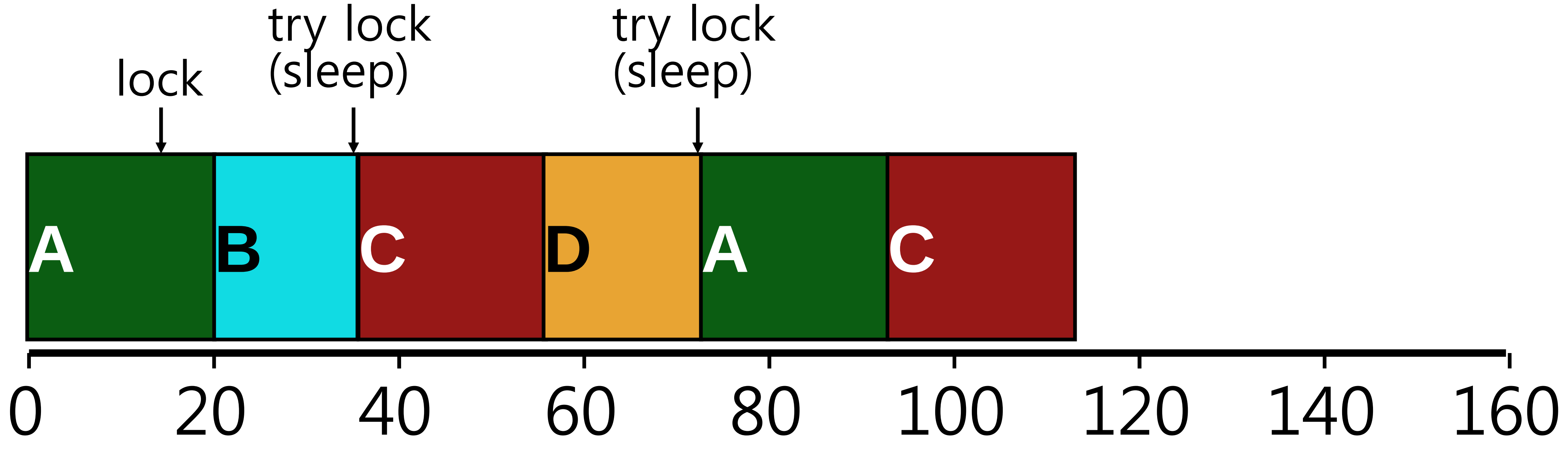
WAITING: B, D



RUNNABLE: A

RUNNING: C

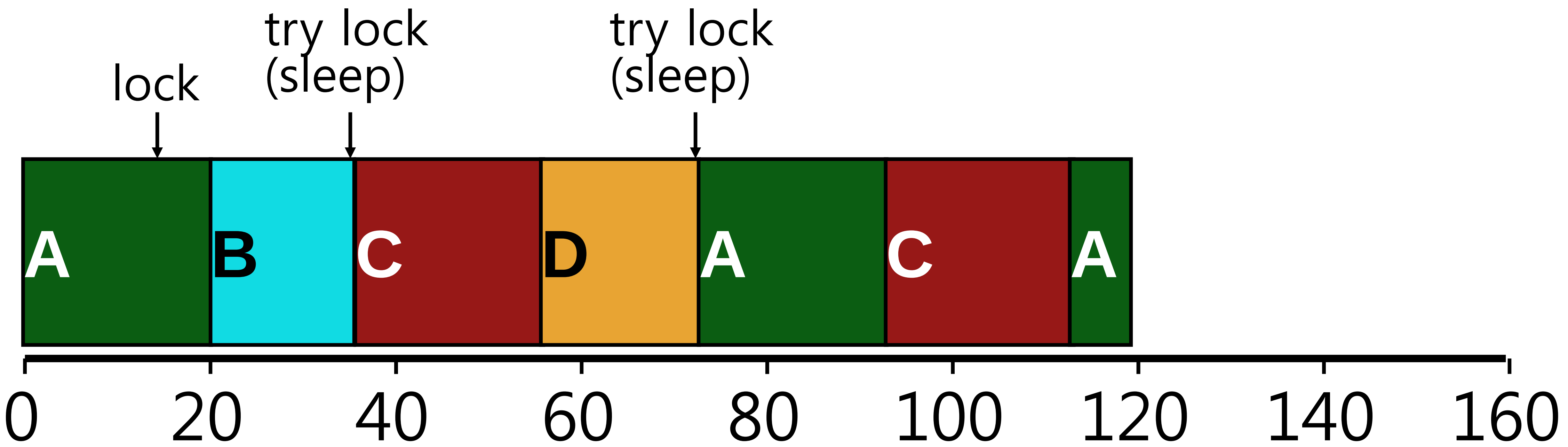
WAITING: B, D



RUNNABLE: C

RUNNING: A

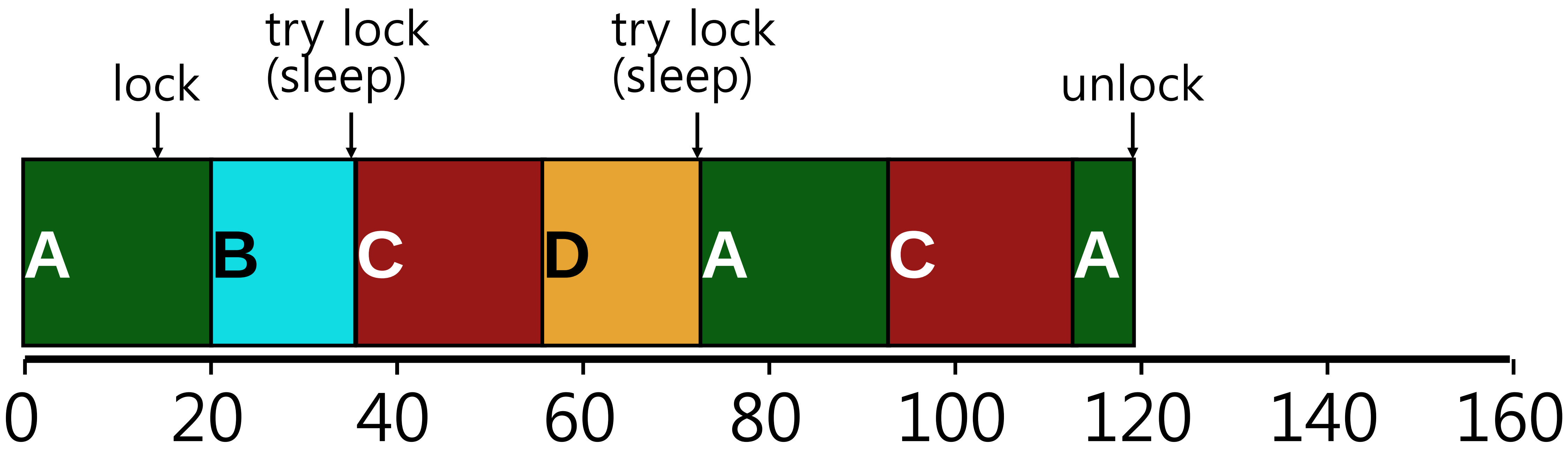
WAITING: B, D



RUNNABLE: B, C

RUNNING: A

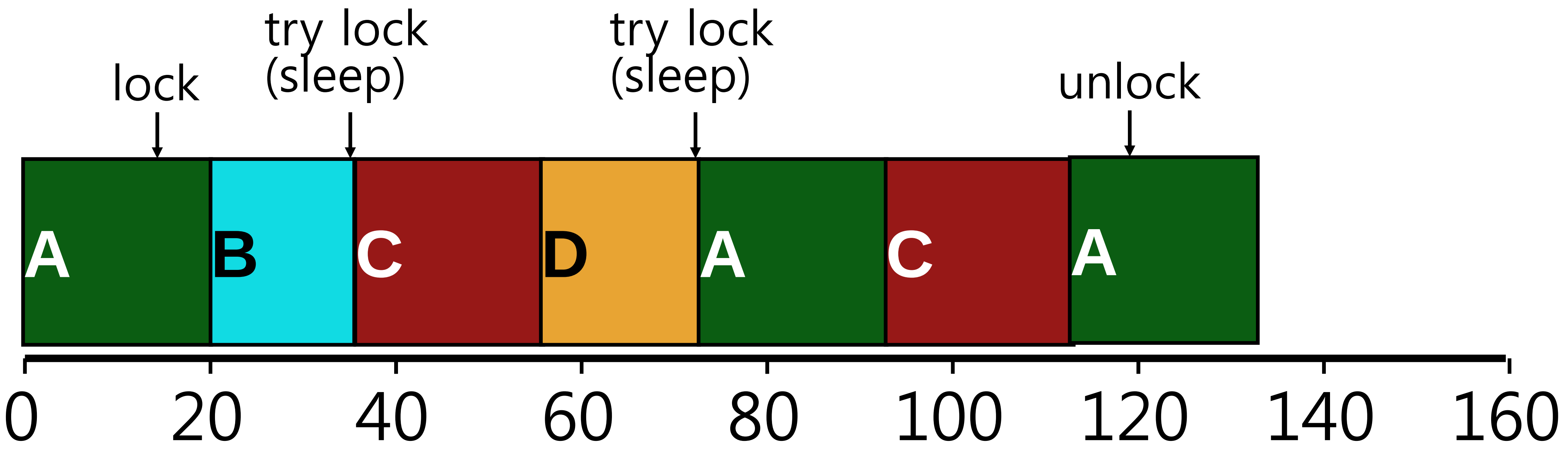
WAITING: D



RUNNABLE: B, C

RUNNING: A

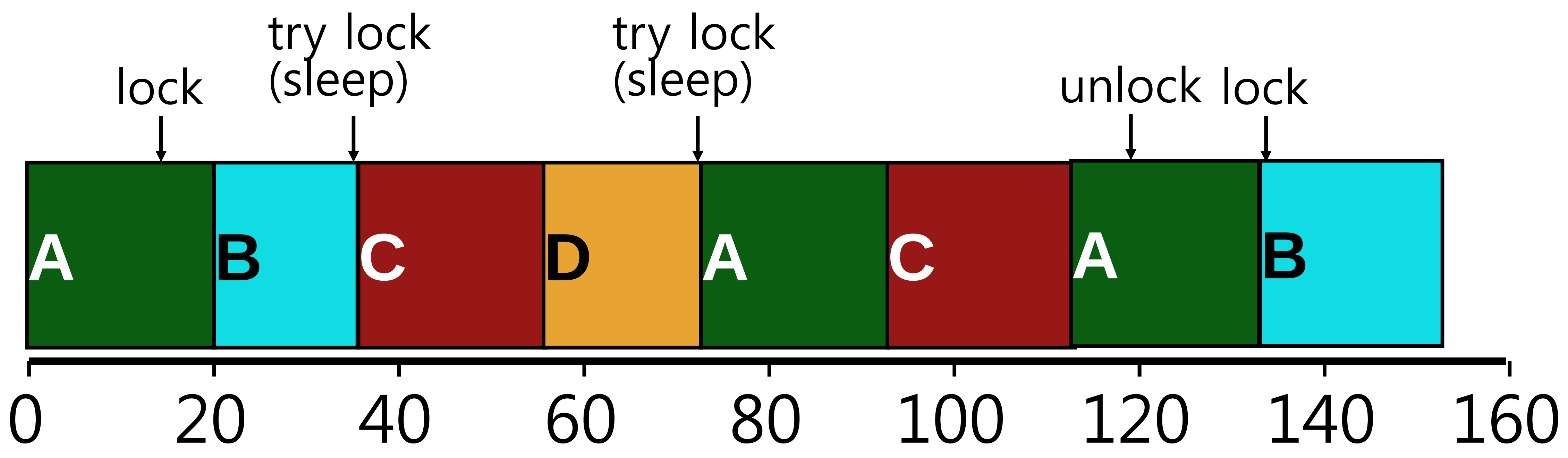
WAITING: D



RUNNABLE: C, A

RUNNING: B

WAITING: D



Lock Implementation: Block when Waiting

```
typedef struct {
  bool lock = false;
  bool guard = false;
  queue_t q;
} LockT;
```

```
void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;
        park();    // blocked
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
```

(a) Why is **guard** used?
(b) Why okay to **spin** on guard?
(c) In release(), why not set lock=false when unpark?
(d) What is the race condition?

Race Condition

Thread 1 (in lock)

```
if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;



        park();    // block
```

Thread 2 (in unlock)

```
while (TAS(&l->guard, true));
if (qempty(l->q)) // false!!
else unpark(qremove(l->q));
l->guard = false;
```

Problem: Guard not held when call park()
Unlocking thread may unpark() before other park()

Block when Waiting: FINAL correct LOCK

```
Typedef struct {
  bool lock = false;
  bool guard = false;
  queue_t q;
} LockT;
```

setpark() fixes race condition

```
void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
            qadd(l->q, tid);
            setpark(); // notify of plan
            l->guard = false;
            park(); // unless unpark()
    } else {
            l->lock = true;
            l->guard = false;
    }
}

void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
```

Spin-Waiting vs Blocking

Each approach is better under different circumstances

- Uniprocessor
  - Waiting process is scheduled --> Process holding lock isn't
  - Waiting process should always relinquish processor
  - Associate queue of waiters with each lock (as in previous implement ation)

- Multiprocessor
  - Waiting process is scheduled --> Process holding lock might be
  - Spin or block depends on how long, t, before lock is released
    - Lock released quickly --> Spin-wait
    - Lock released slowly --> Block
    - Quick and slow are relative to context-switch cost, C

When to Spin-Wait? When to Block?

If know how long**, t,** before lock released, can determine optimal behavior

How much CPU time is wasted when spin-waiting?

t

How much wasted when block?

C

What is the best action when t<C?

spin-wait

When t>C?

block

Problem:

Requires knowledge of future; too much overhead to do any special prediction

Two-Phase Waiting

Theory: Bound worst-case performance; ratio of actual/optimal

When does worst-possible performance occur?

Spin for very long time t >> C

Ratio: t/C (unbounded)

Algorithm: Spin-wait for C then block --> Factor of 2 of optimal

Two cases:

- t < C: optimal spin-waits for t; we spin-wait t too
- t > C: optimal blocks immediately (cost of C); we pay spin C then block (cost of 2 C); 2C / C → 2-competitive algorithm

Example of competitive analysis

Condition Variables

There are many cases where a thread wishes to check whether a **condition** is true before continuing its execution.

Example:

- A parent thread might wish to check whether a child thread has *completed*.
- This is often called a `join()`.

Condition Variables (Cont.)

A Parent Waiting For Its Child

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

What we would like to see here is:

```
parent: begin
child
parent: end
```

Parent waiting fore child: Spin-based Approach

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

- This is hugely <u>inefficient</u> as the parent spins and **wastes CPU time**.

How to wait for a condition

Condition variable

- **Waiting** on the condition
  - An explicit queue that threads can put themselves on when some state of execution is not as desired.
- **Signaling** on the condition
  - Some other thread, *when it changes said state*, can wake one of those waiting threads and allow them to continue.

Definition and Routines

Declare condition variable

```
pthread cond t c;
```

- Proper initialization is required.

Operation (the POSIX calls)

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);   // wait()
pthread_cond_signal(pthread_cond_t *c);                     // signal()
```

- The wait() call takes a mutex as a parameter.
  - The wait() call release the lock and put the calling thread to sleep.
  - When the thread wakes up, it must re-acquire the lock.

Parent waiting for Child: Use a condition variable

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
        Pthread_mutex_lock(&m);
        done = 1;
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
        printf("child\n");
        thr_exit();
        return NULL;
}

void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Parent waiting for Child: Use a condition variable

```
(cont.)
int main(int argc, char *argv[]) {
        printf("parent: begin\n");
        pthread_t p;
        Pthread_create(&p, NULL, child, NULL);
        thr_join();
        printf("parent: end\n");
        return 0;
}
```

Parent waiting for Child: Use a condition variable

Parent:

- Create the child thread and continues running itself.
- Call into `thr_join()` to wait for the child thread to complete.
  - Acquire the lock
  - Check if the child is done
  - Put itself to sleep by calling `wait()`
  - Release the lock

Child:

- Print the message "child"
- Call `thr_exit()` to wake the parent thread
  - Grab the lock
  - Set the state variable `done`
  - Signal the parent thus waking it.

The importance of the state variable `done`

```
void thr_exit() {
        Pthread_mutex_lock(&m);
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void thr_join() {
        Pthread_mutex_lock(&m);
        Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

`thr_exit()` and `thr_join()` without variable `done`

- Imagine the case where the *child runs immediately*.
    - The child will signal, but there is no thread asleep on the condition.
    - When the parent runs, it will call wait and be stuck.
    - No thread will ever wake it.

Another poor implementation

```c
void thr_exit() {
        done = 1;
        Pthread_cond_signal(&c);
}

void thr_join() {
        if (done == 0)
                Pthread_cond_wait(&c);
}
```

- The issue here is a subtle **race condition**.
  - The parent calls `thr_join()`.
    - The parent checks the value of `done`.
    - It will see that it is 0 and try to go to sleep.
    - *Just before* it calls wait to go to sleep, the parent is interrupted and the child runs.
  - The child changes the state variable `done` to 1 and signals.
    - But no thread is waiting and thus no thread is woken.
    - When the parent runs again, it sleeps forever.

Condition Variables

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*

Ordering Example: Join

```
pthread_t p1, p2;
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");
// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);
printf("main: done₩n [balance: %d]₩n [should: %d]₩n",
    balance, max*2);
return 0;
```

how to implement join()?

Condition Variables

Condition Variable: queue of waiting threads

B waits for a signal on CV before running

- wait(CV, …)

A sends signal to CV when time for ***B*** to run

- signal(CV, …)

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)
- 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: Attempt 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);         // x
        Cond_wait(&c, &m);      // y
        Mutex_unlock(&m);       // z
}
```

Child:

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

Example schedule:

Parent:	x	y				z
Child:			a	b	c	

Works!

Join Implementation: Attempt 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // x
        Cond_wait(&c, &m);       // y
        Mutex_unlock(&m);        // z
}
```

Child:

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

Can you construct ordering that does not work?

Example broken schedule:

Parent:				x	y
Child:	a	b	c		

Parent waits forever!

Rule of Thumb 1

Keep state in addition to CV's!

CV's are used to signal threads when state changes

If state is already as needed, thread doesn't wait for a signal!

Join Implementation: Attempt 2

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() {
        done = 1;                 // a
        Cond_signal(&c);          // b
}
```

Fixes previous broken ordering:

Parent:			w	x	y	z
Child:	a	b				

Join Implementation: Attempt 2

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() {
        done = 1;               // a
        Cond_signal(&c);        // b
}
```

Can you construct ordering that does not work?

Parent: w x y … sleep forever …

Child: a b

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
}
```

Parent: w x y z

Child: a b c

Use mutex to ensure no race between interacting with state and wait/signal

Producer/Consumer Problem

Example: UNIX Pipes

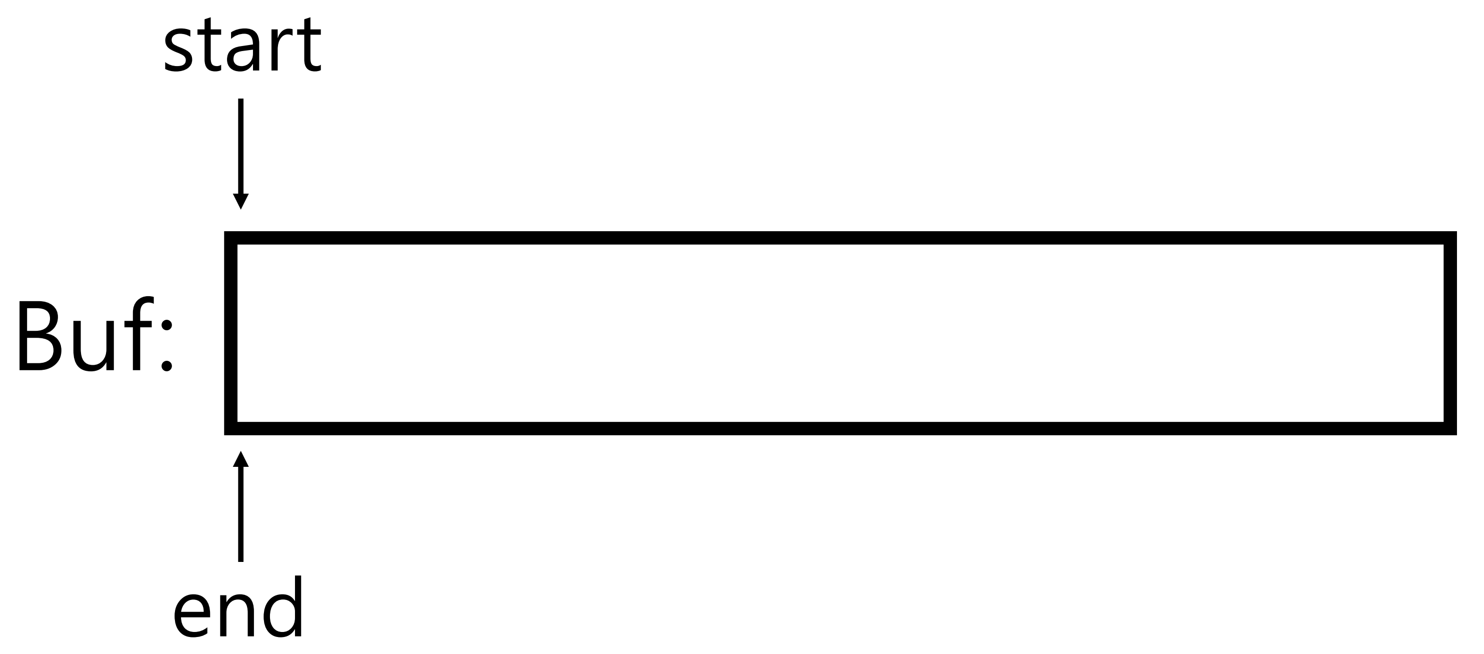
A pipe may have many writers and readers

Internally, there is a finite-sized buffer

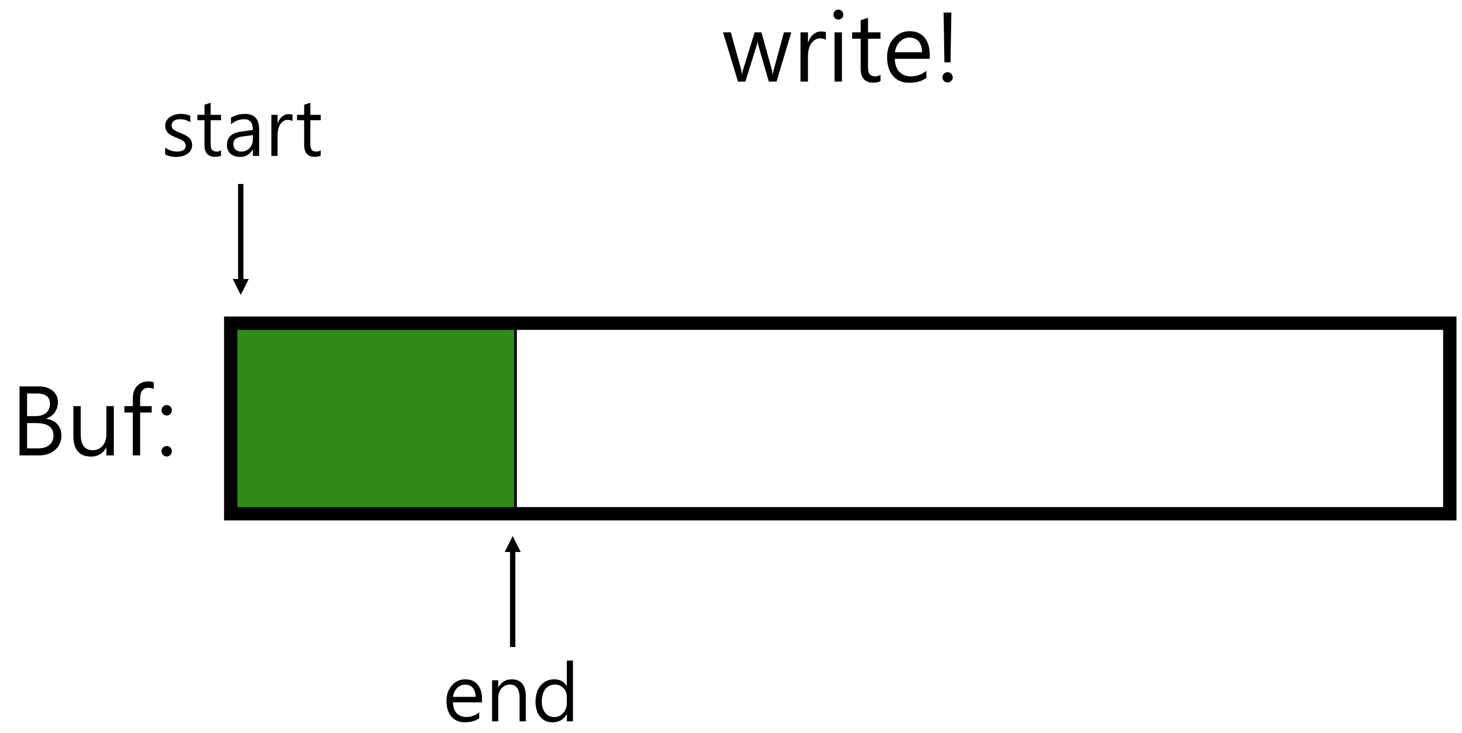
Writers add data to the buffer
- Writers have to wait if buffer is full

Readers remove data from the buffer
- Readers have to wait if buffer is empty

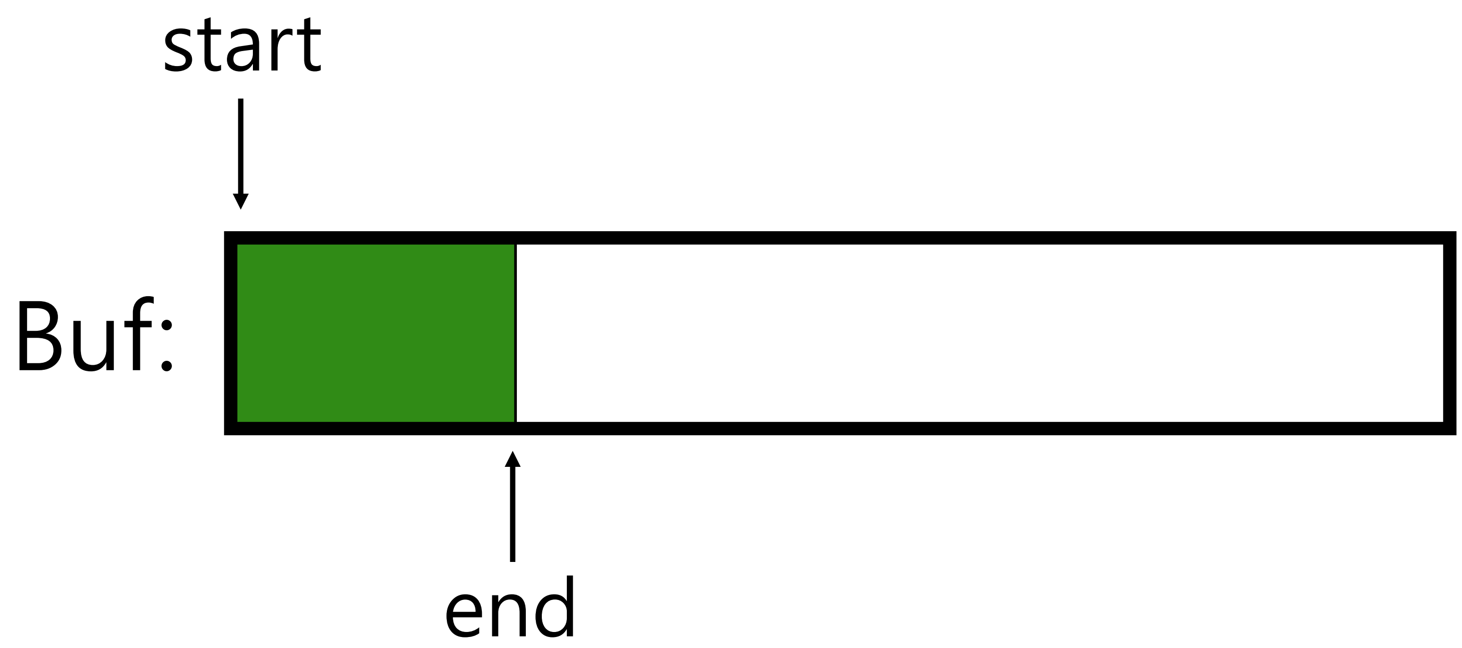
Example: UNIX Pipes



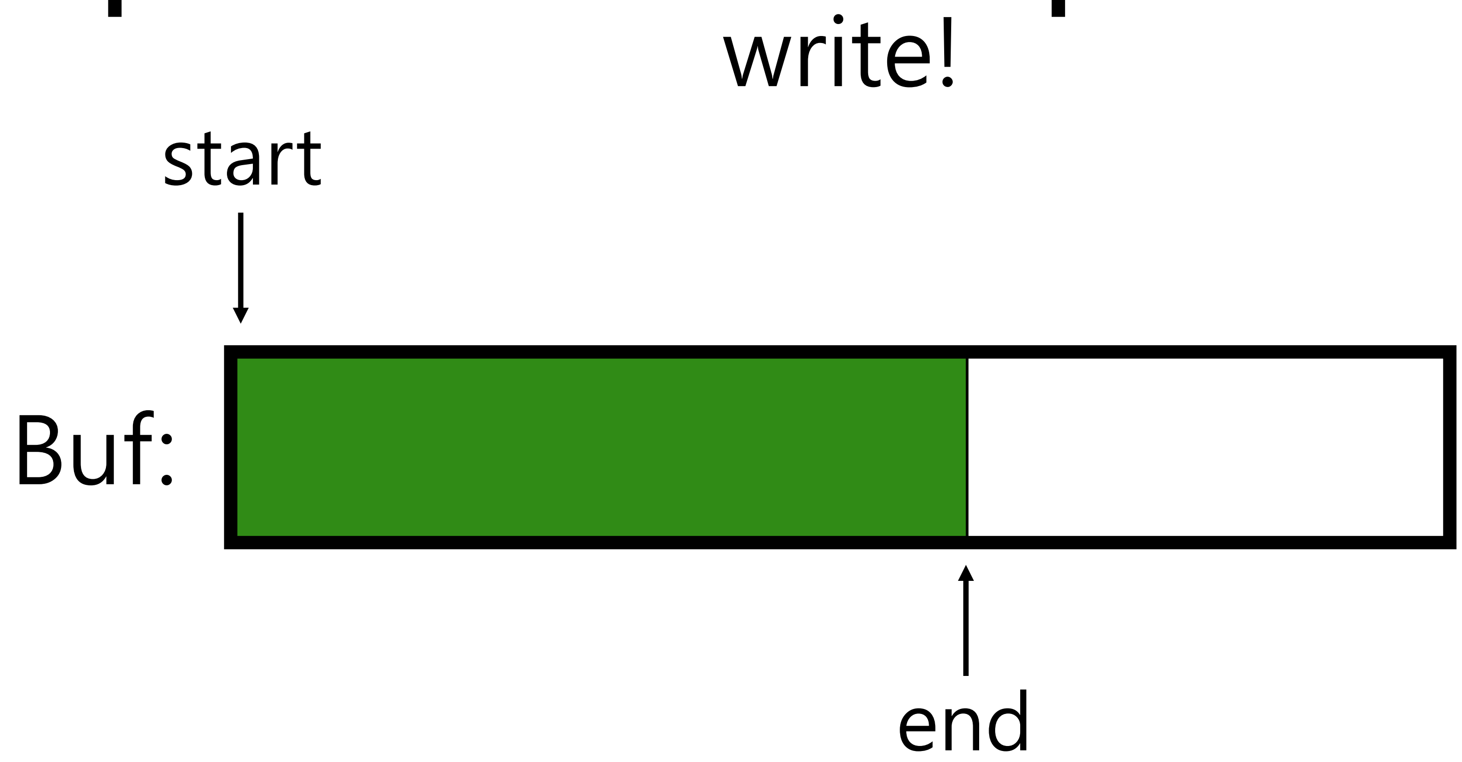
Example: UNIX Pipes



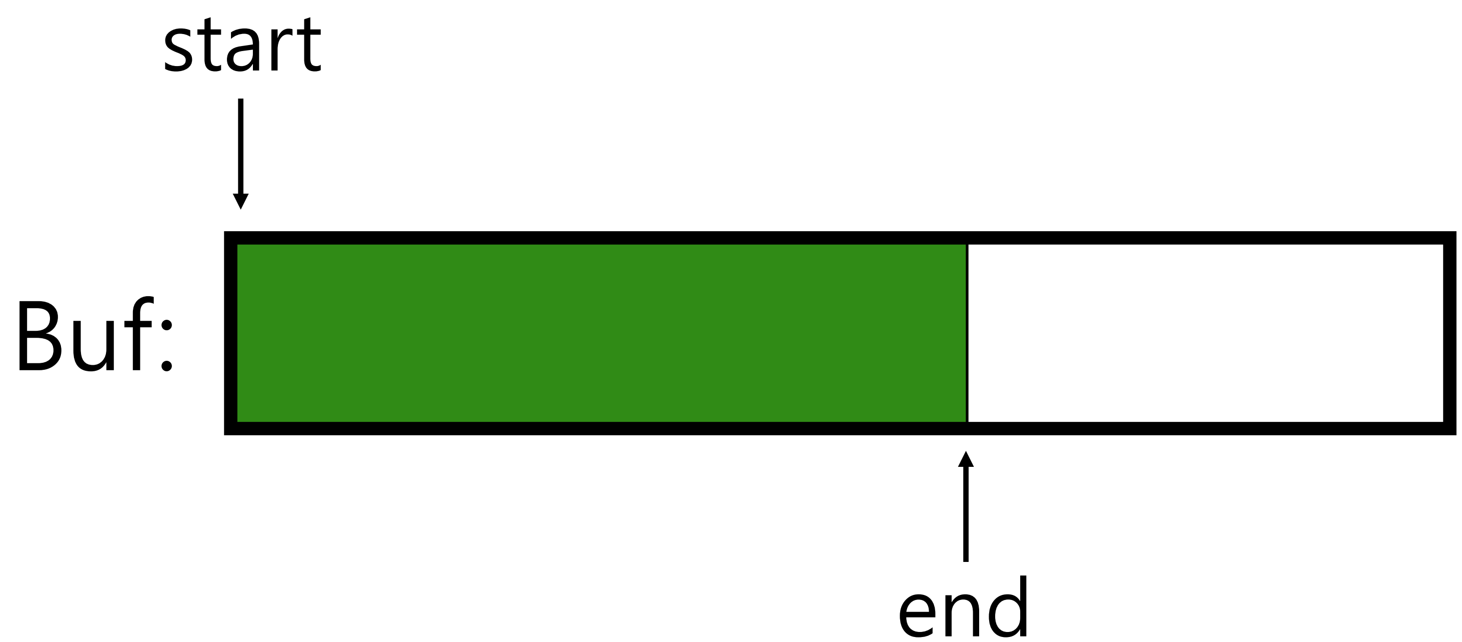
Example: UNIX Pipes



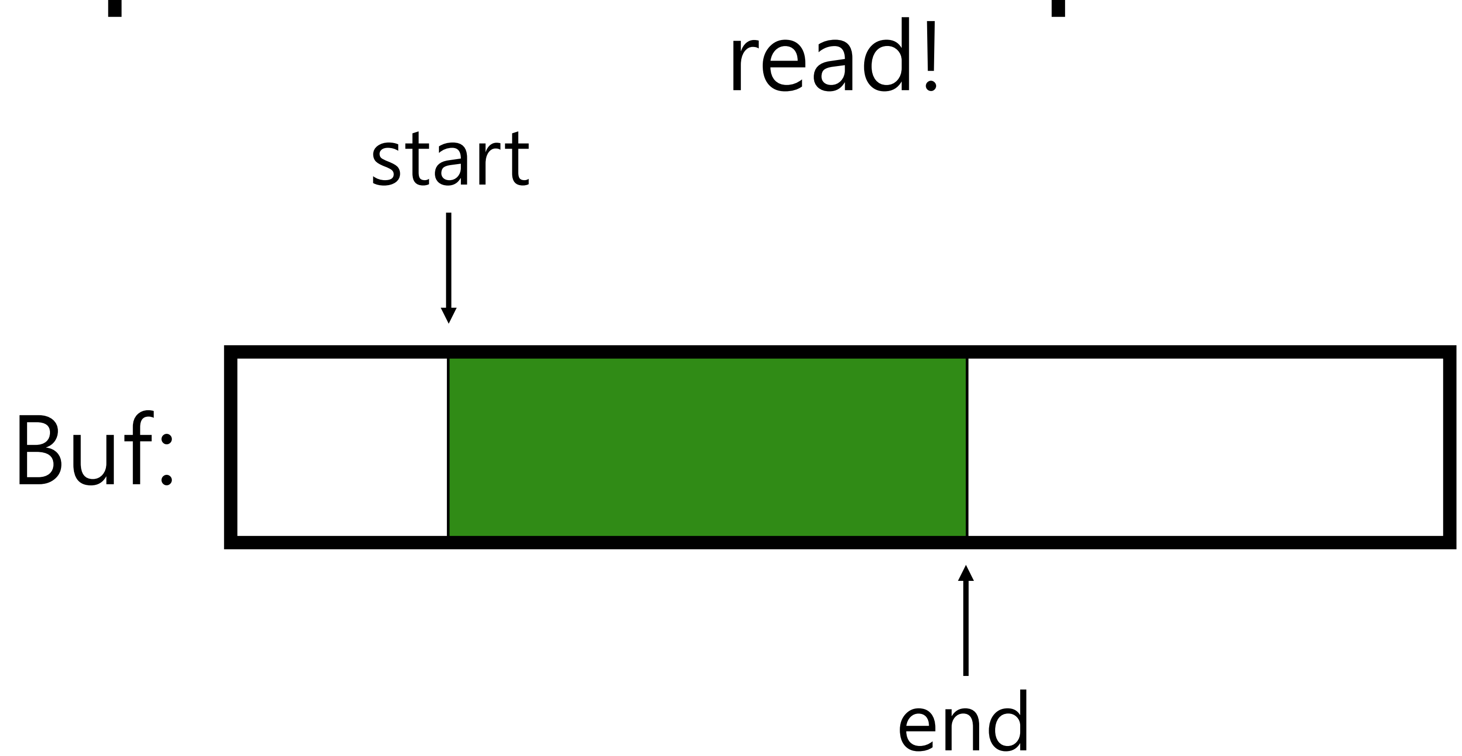
Example: UNIX Pipes



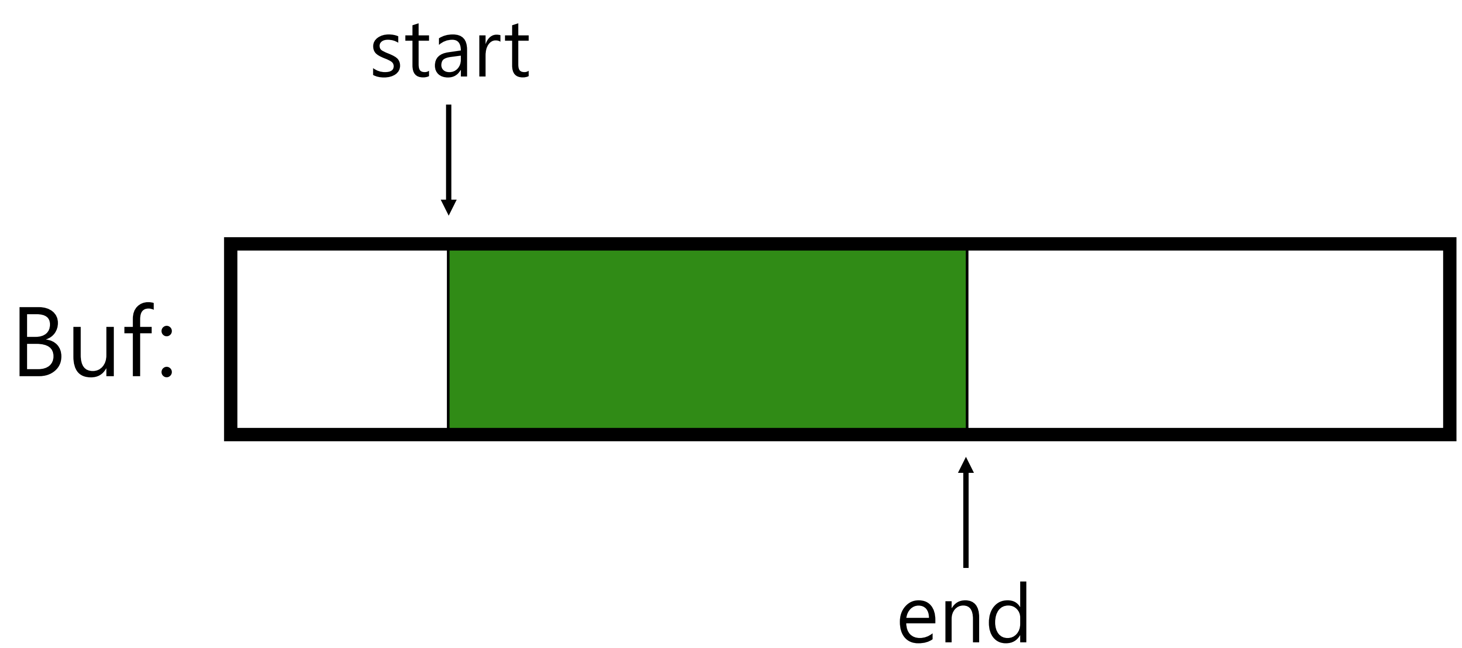
Example: UNIX Pipes



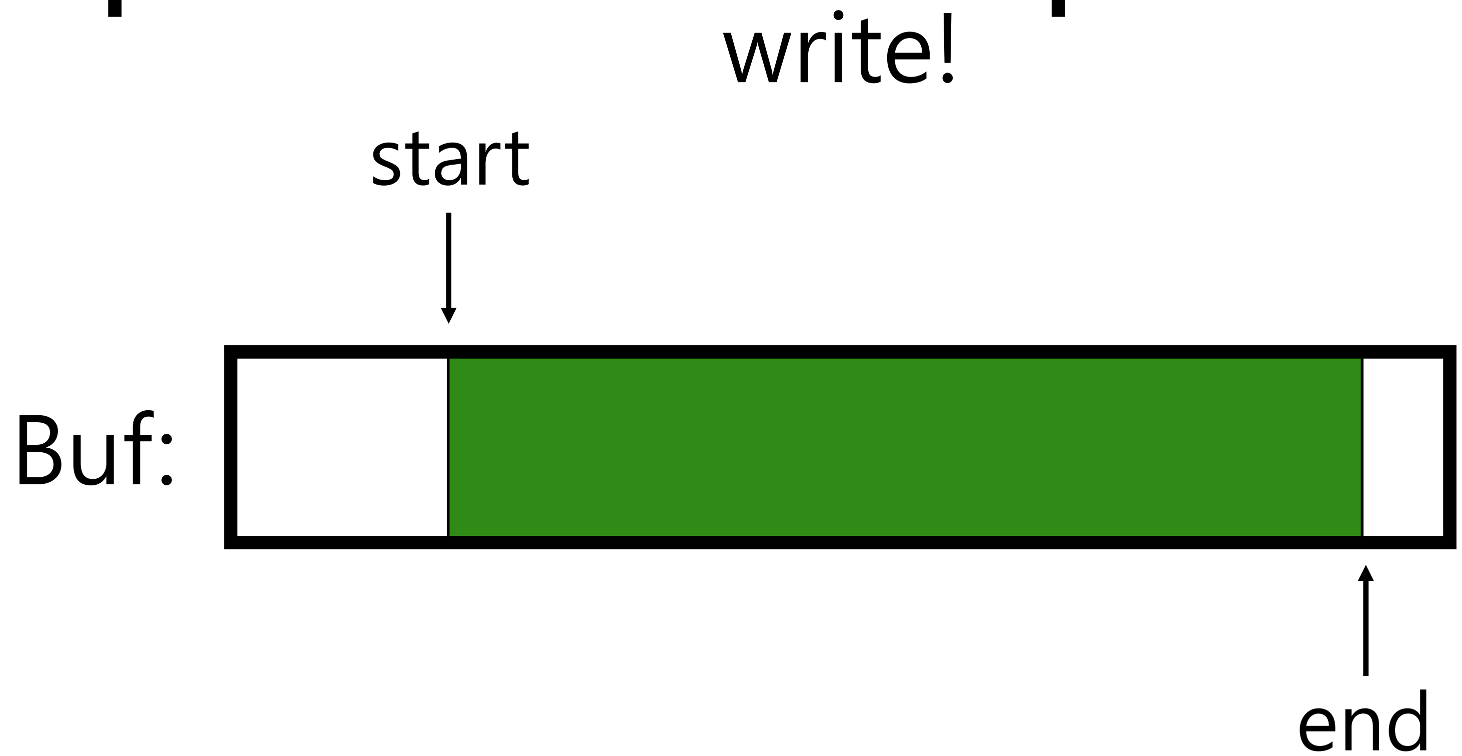
Example: UNIX Pipes



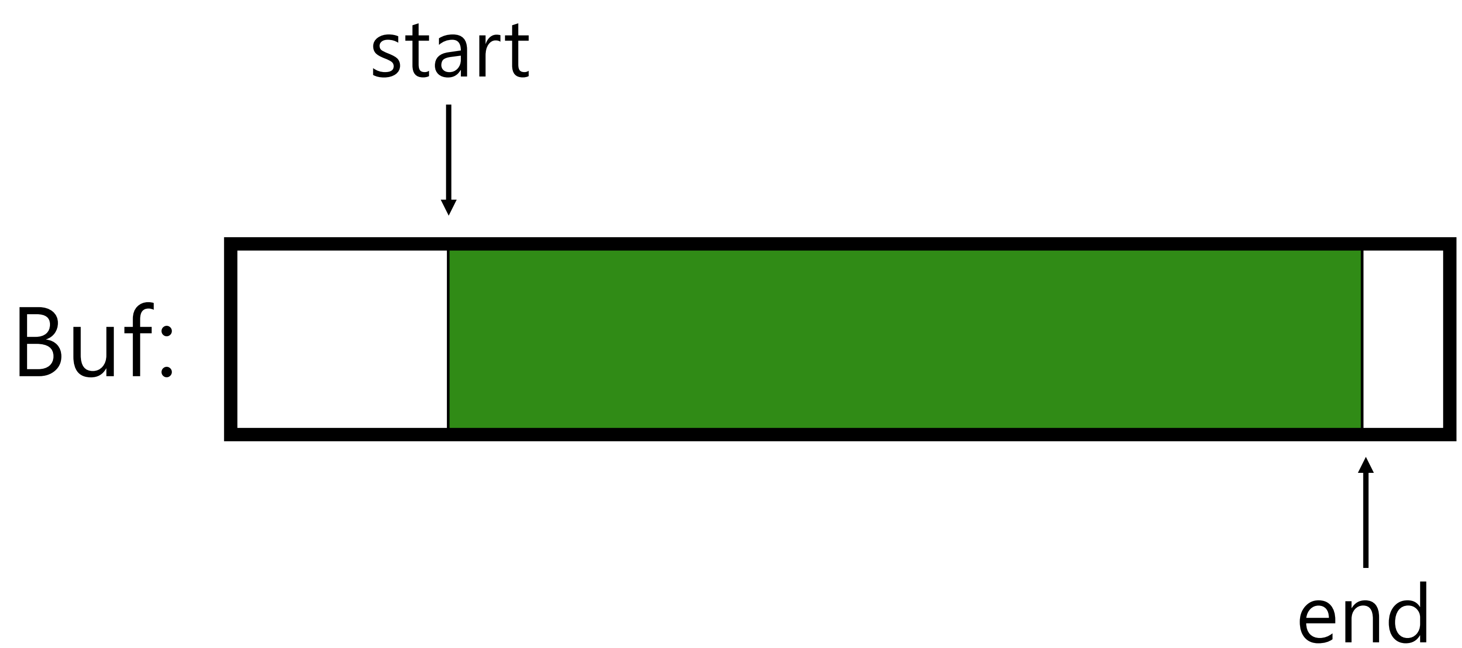
Example: UNIX Pipes



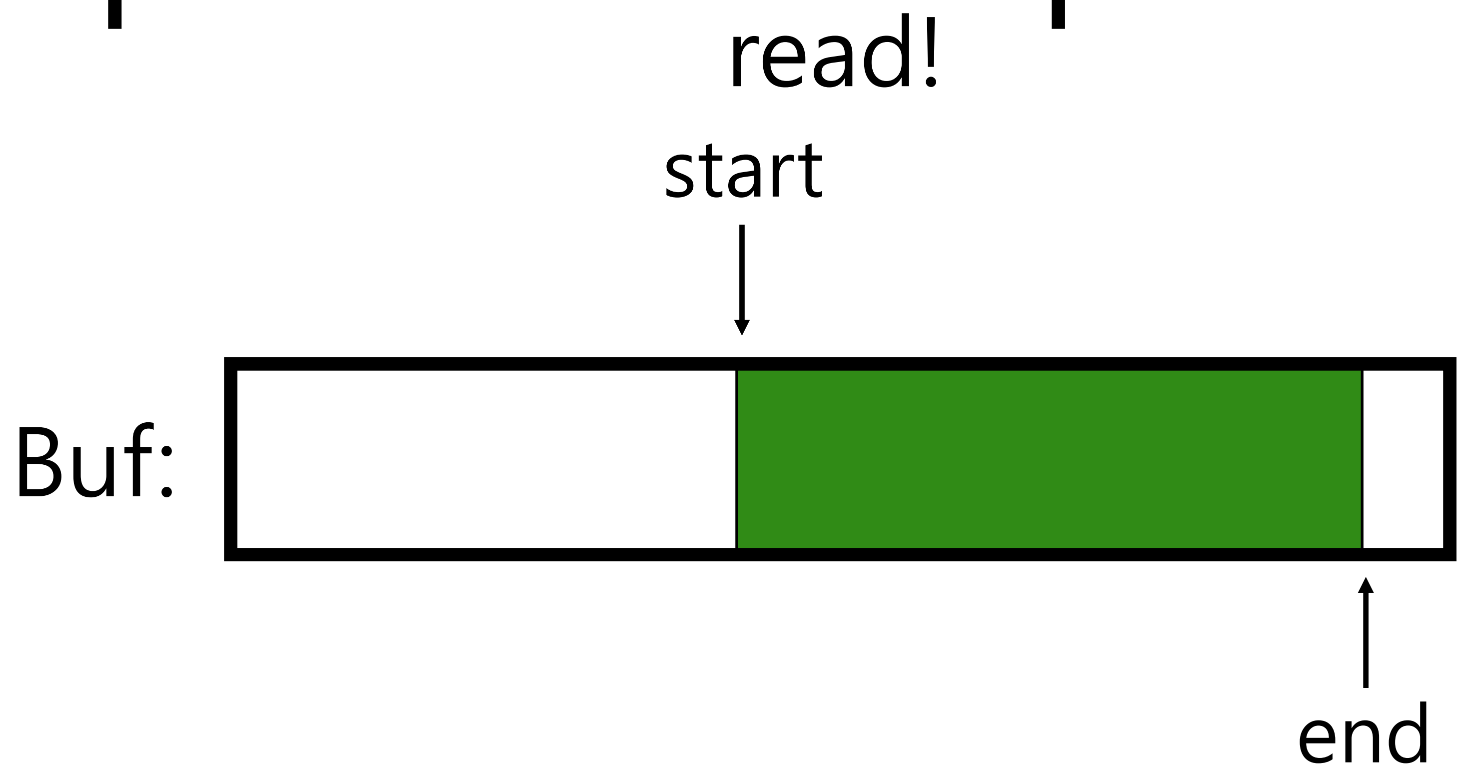
Example: UNIX Pipes



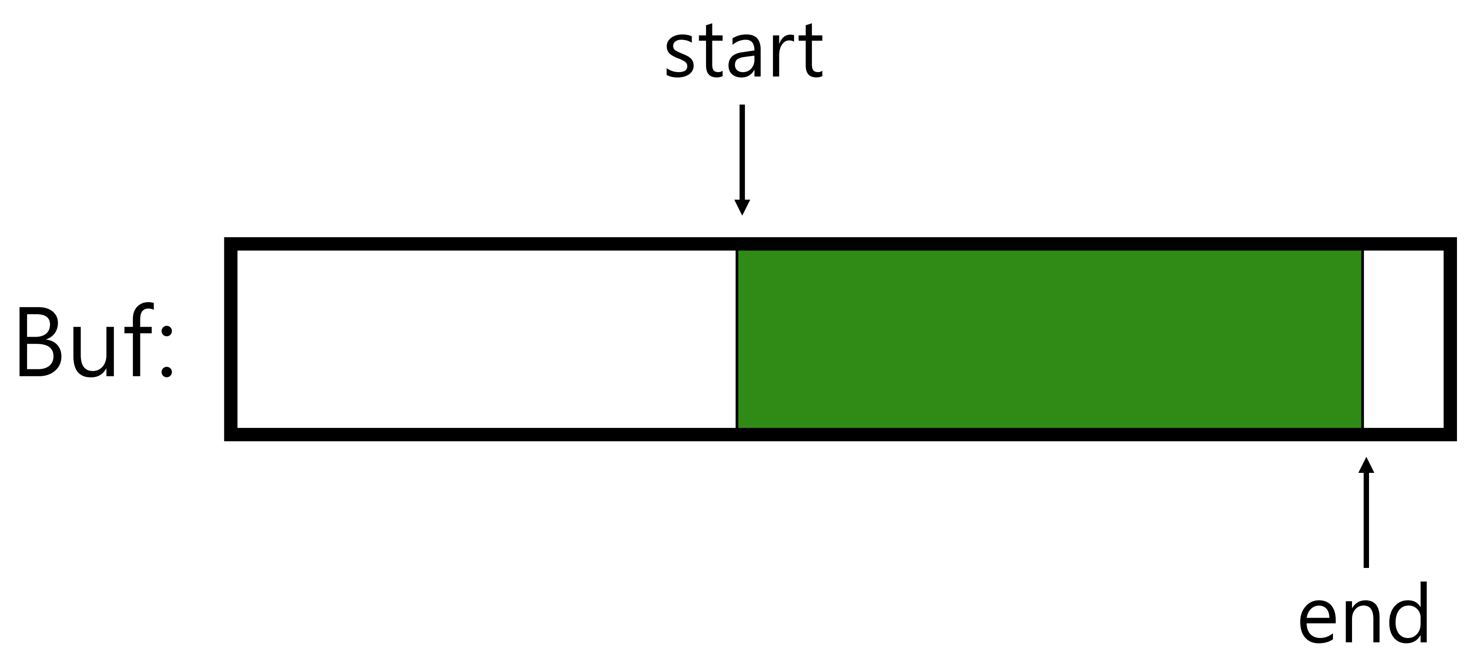
Example: UNIX Pipes



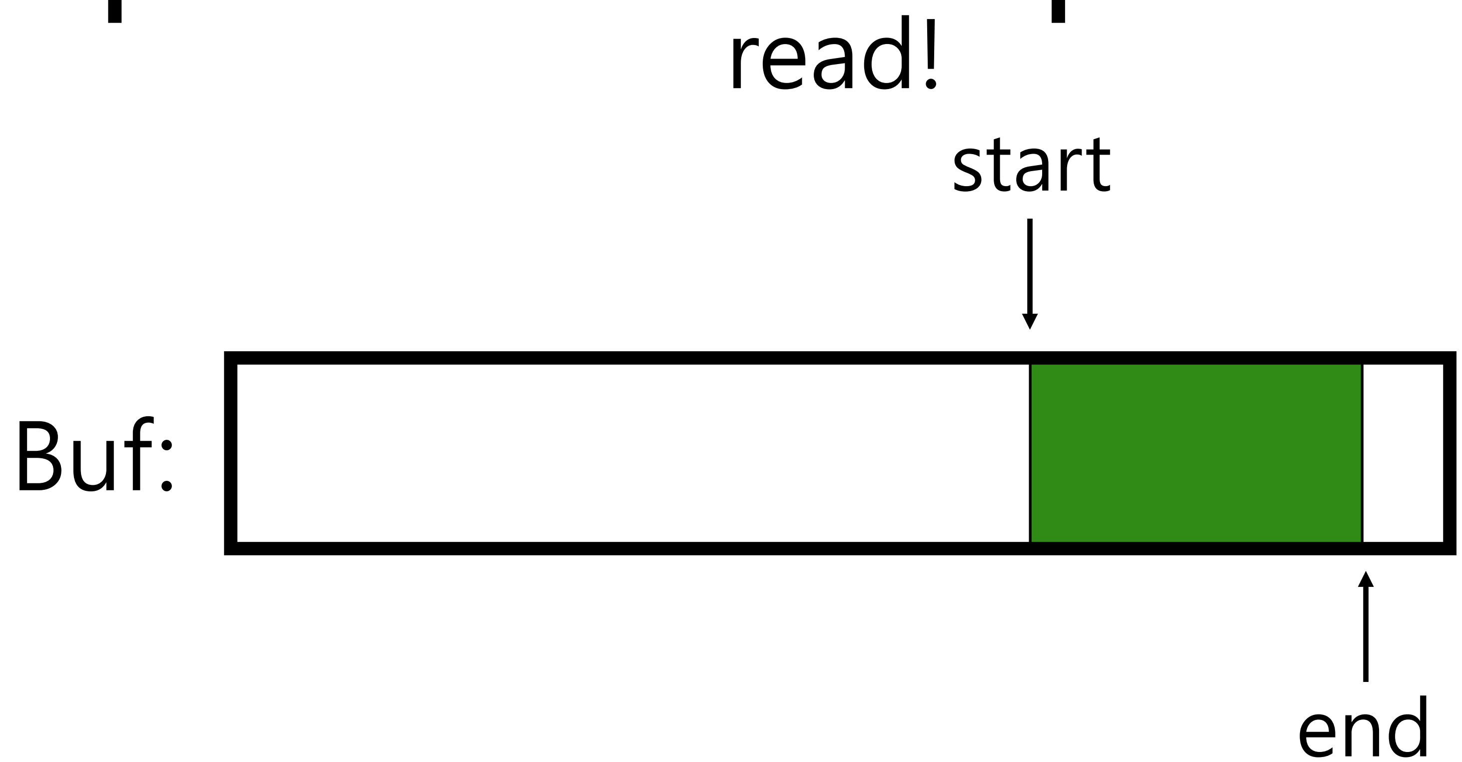
Example: UNIX Pipes



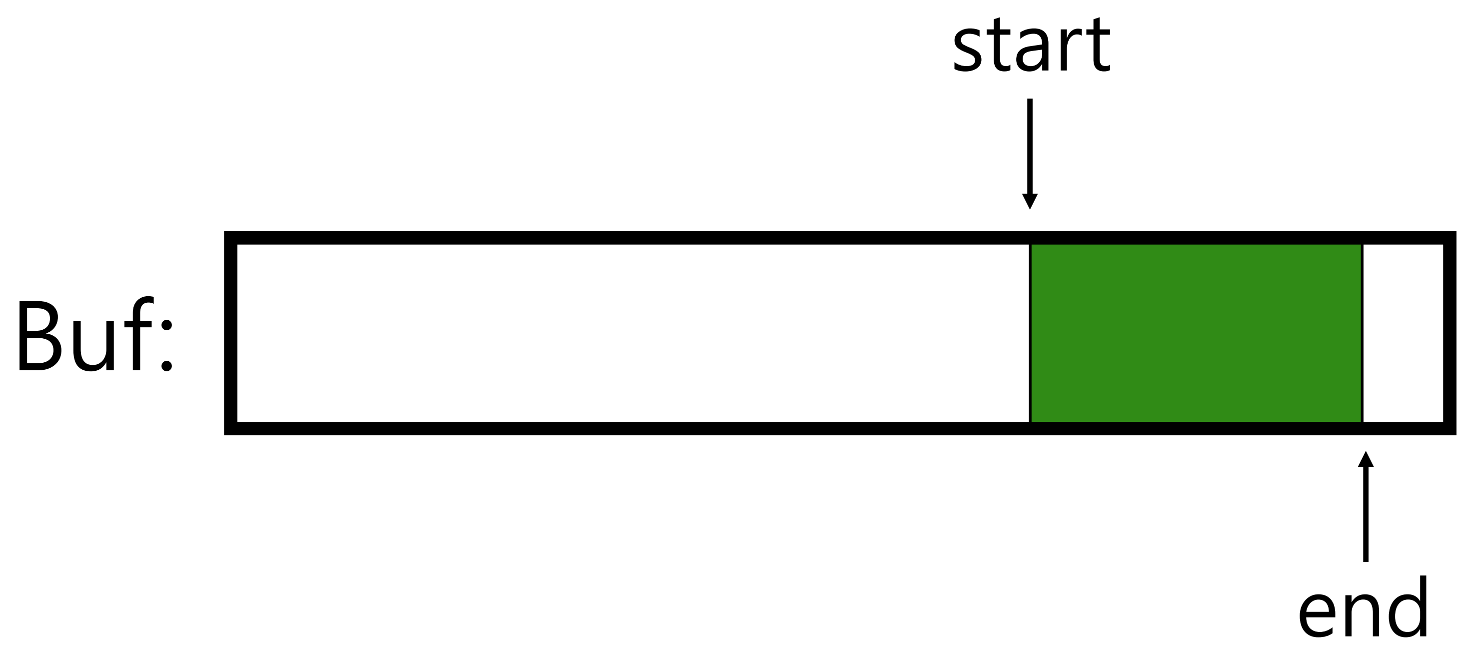
Example: UNIX Pipes



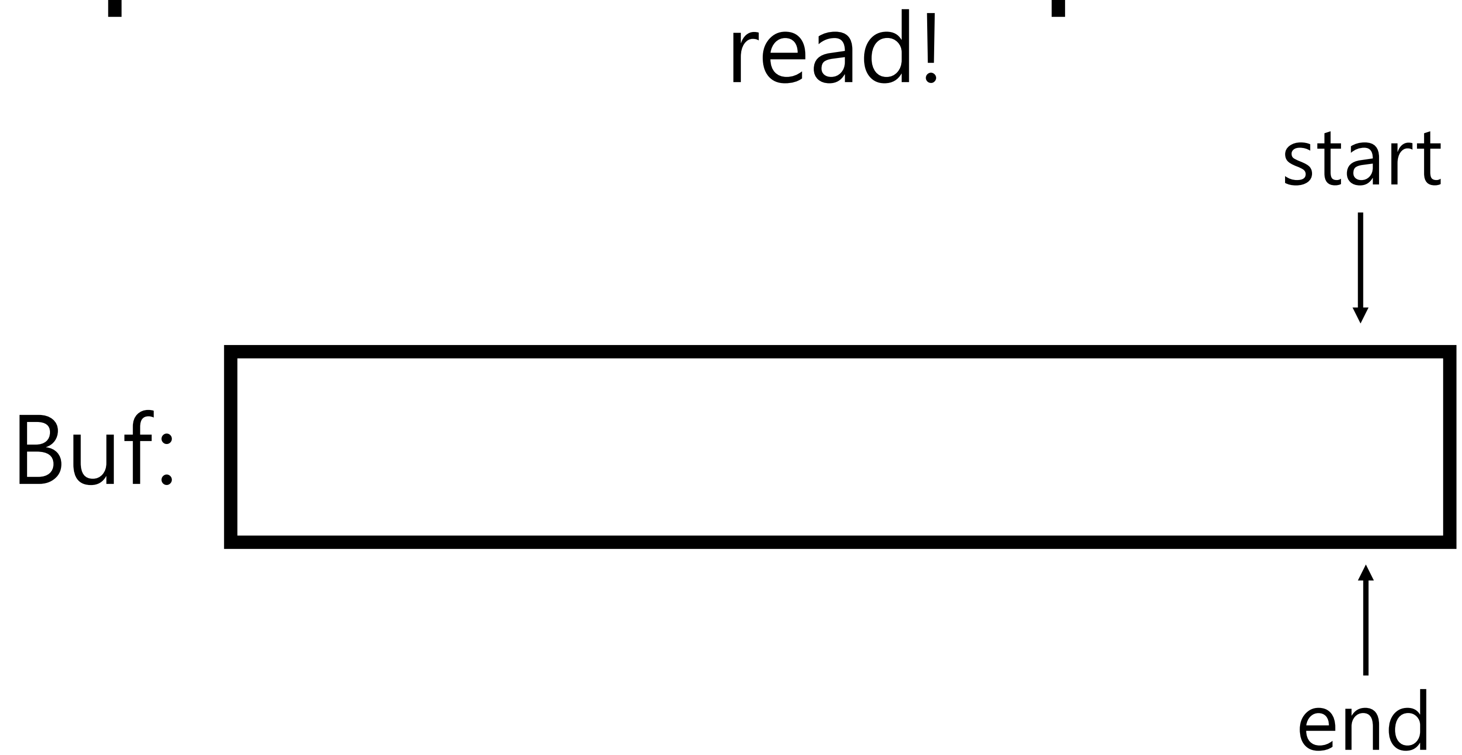
Example: UNIX Pipes



Example: UNIX Pipes

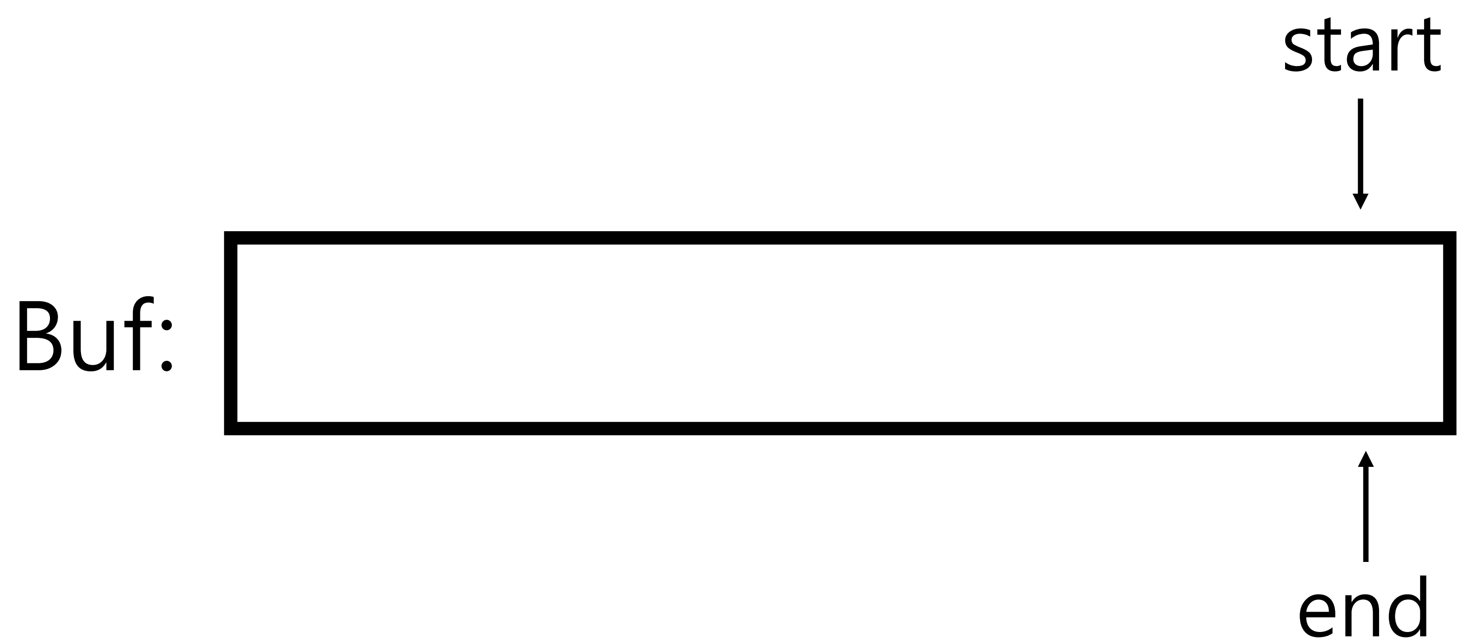


Example: UNIX Pipes



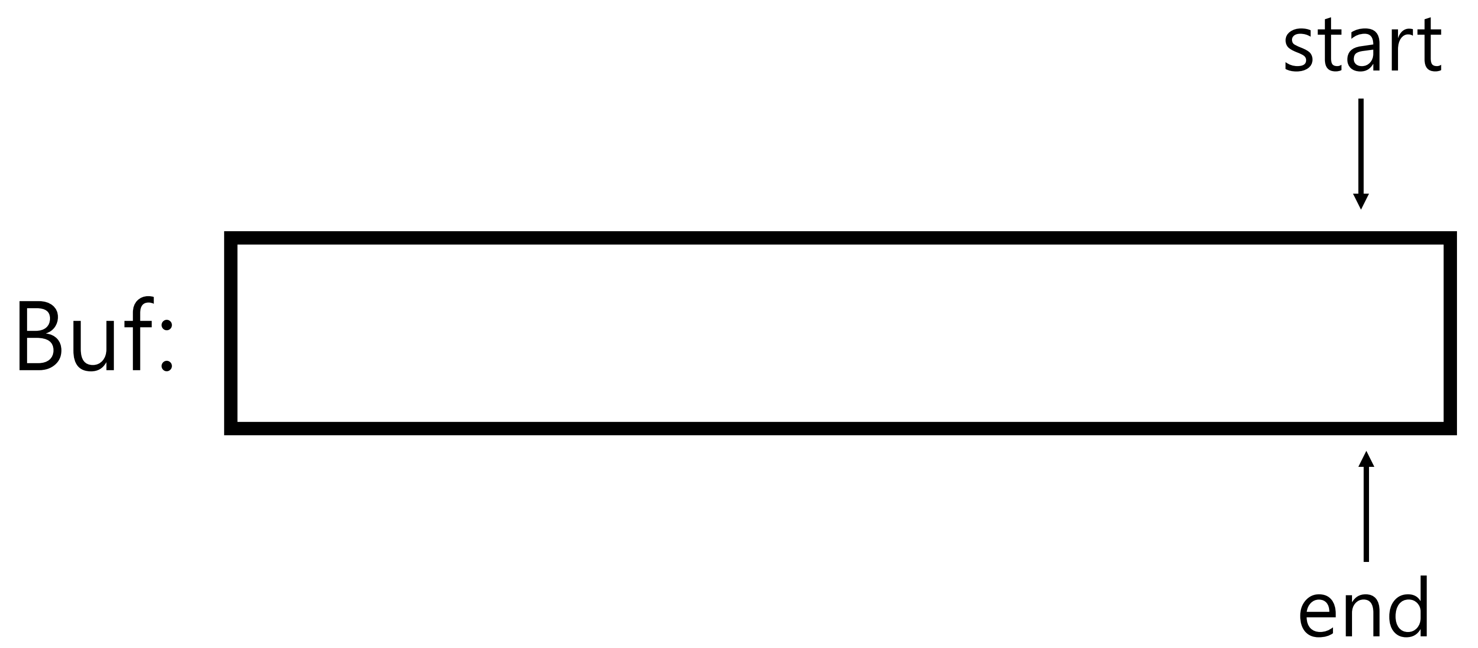
Example: UNIX Pipes

read!

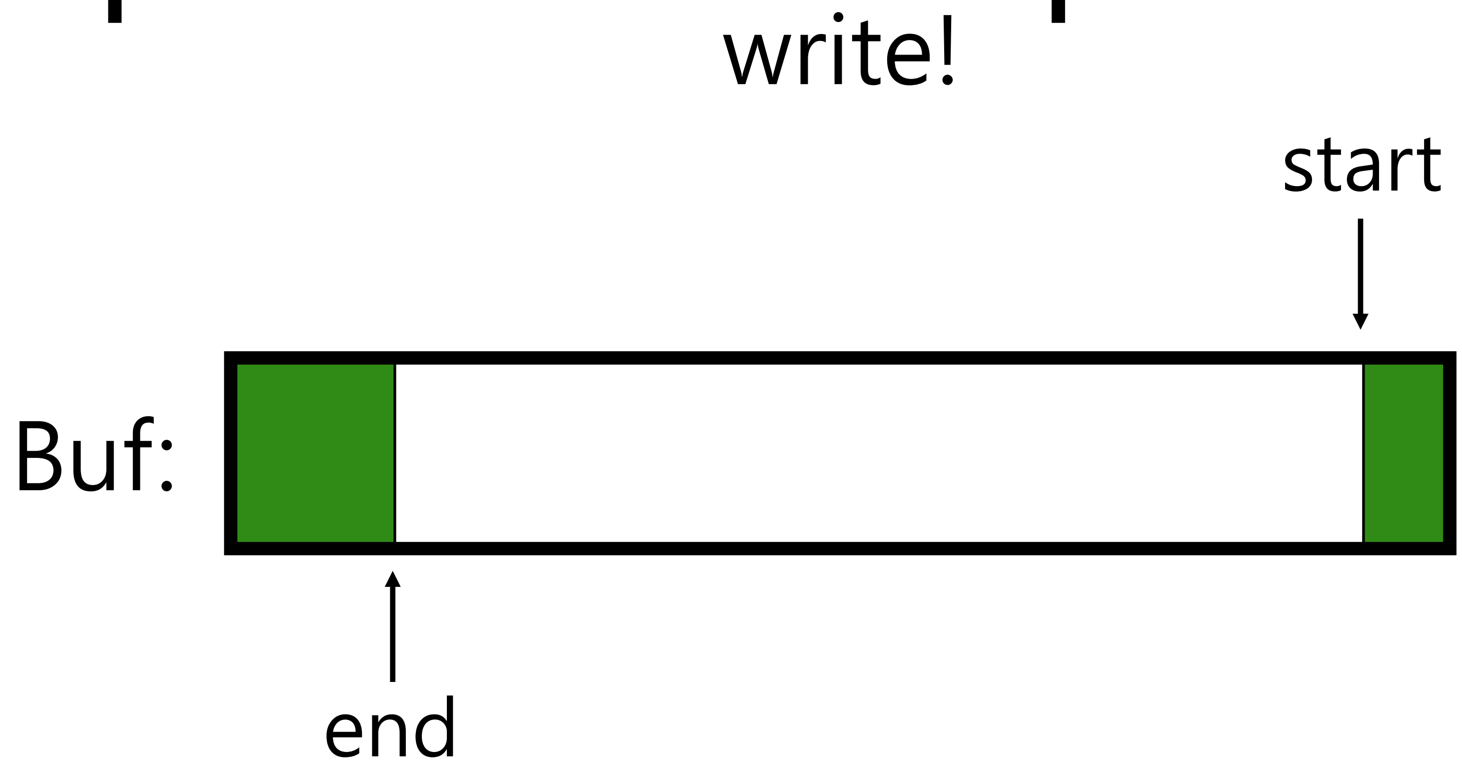


note: readers must wait

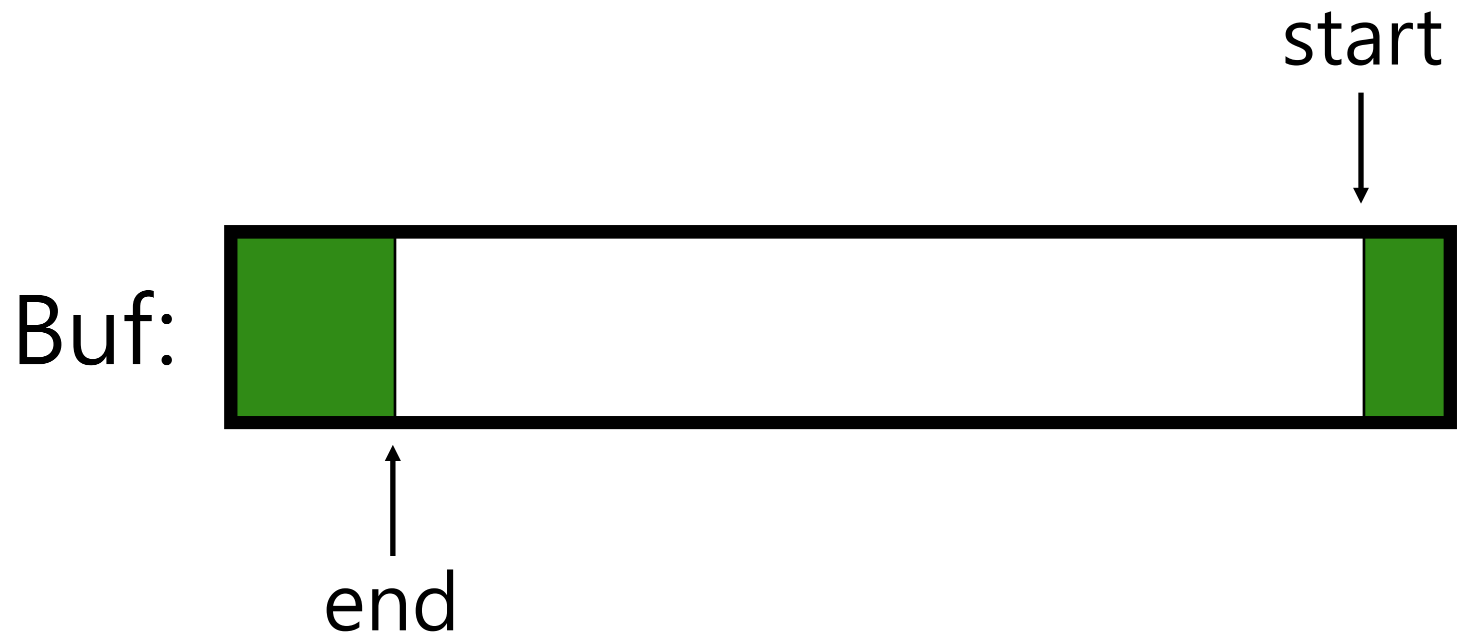
Example: UNIX Pipes



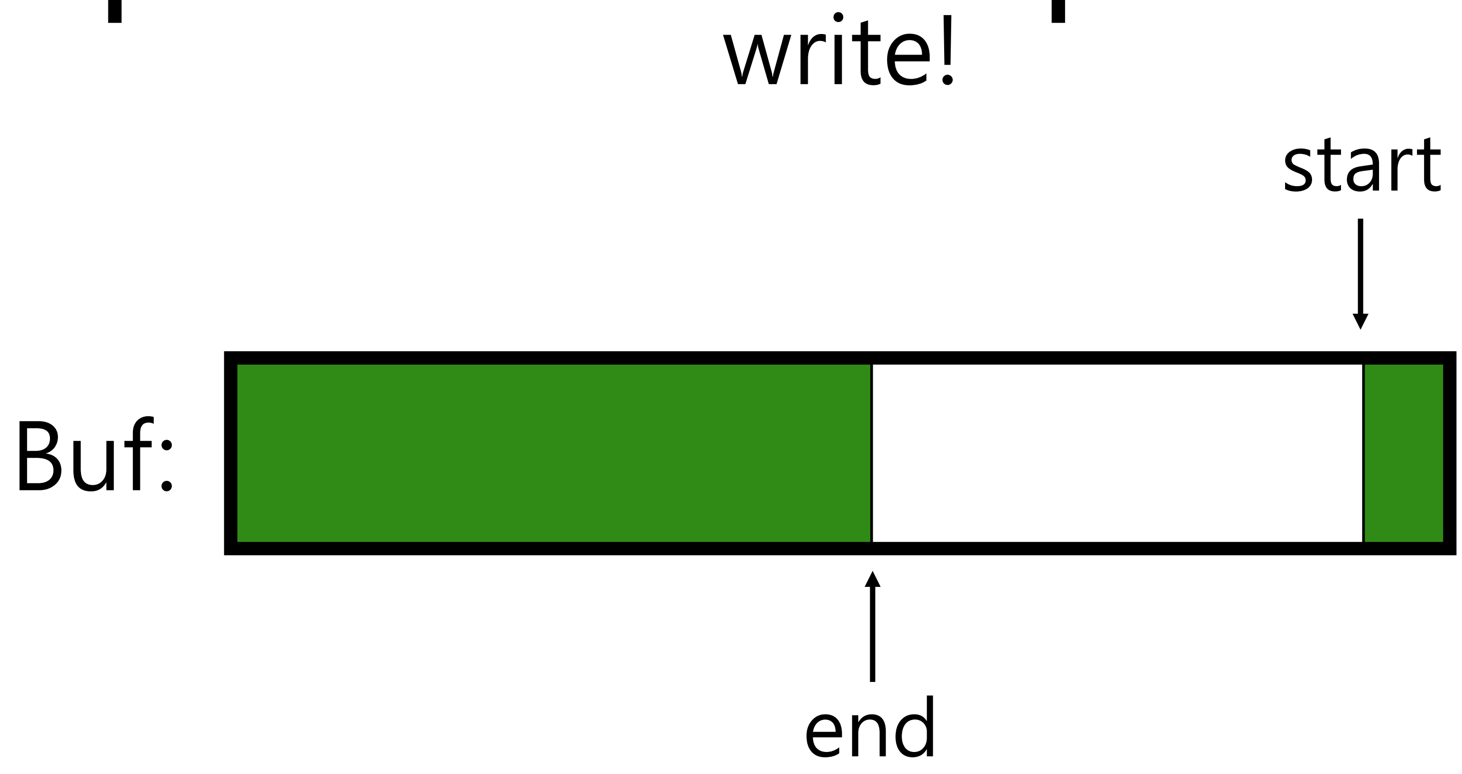
Example: UNIX Pipes



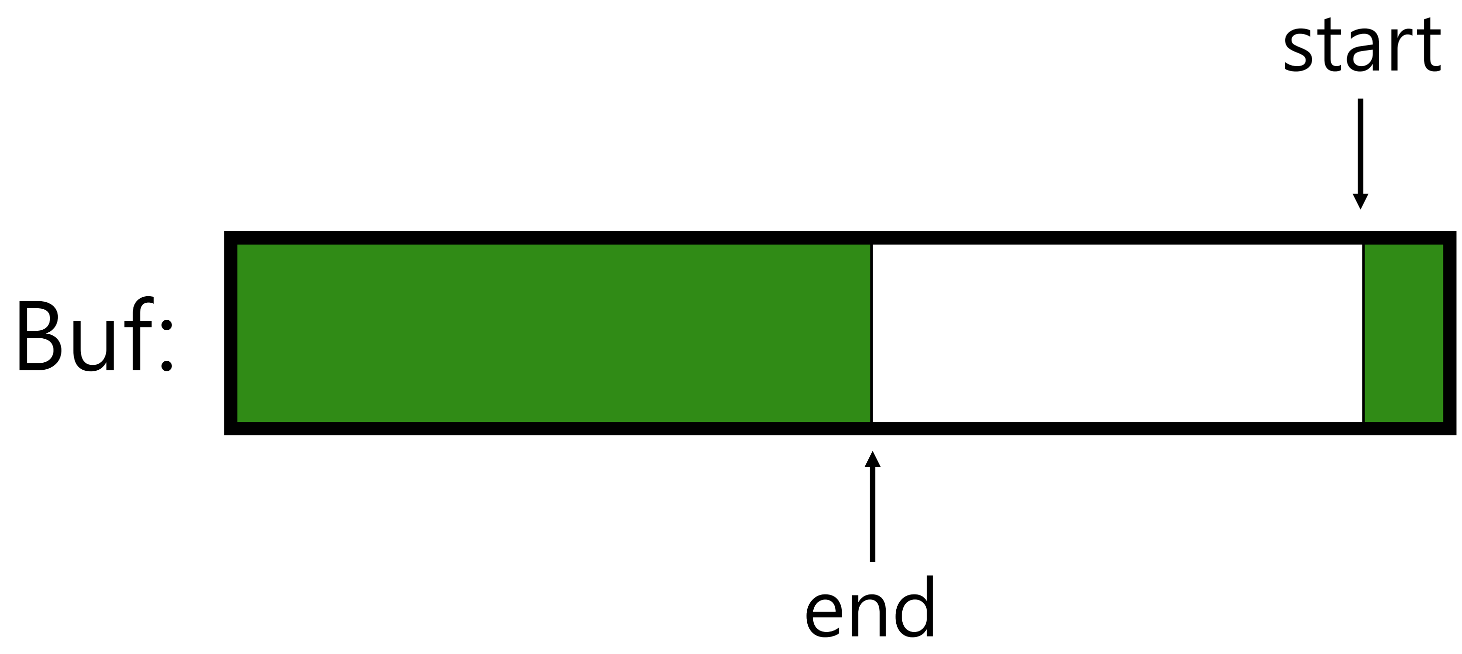
Example: UNIX Pipes



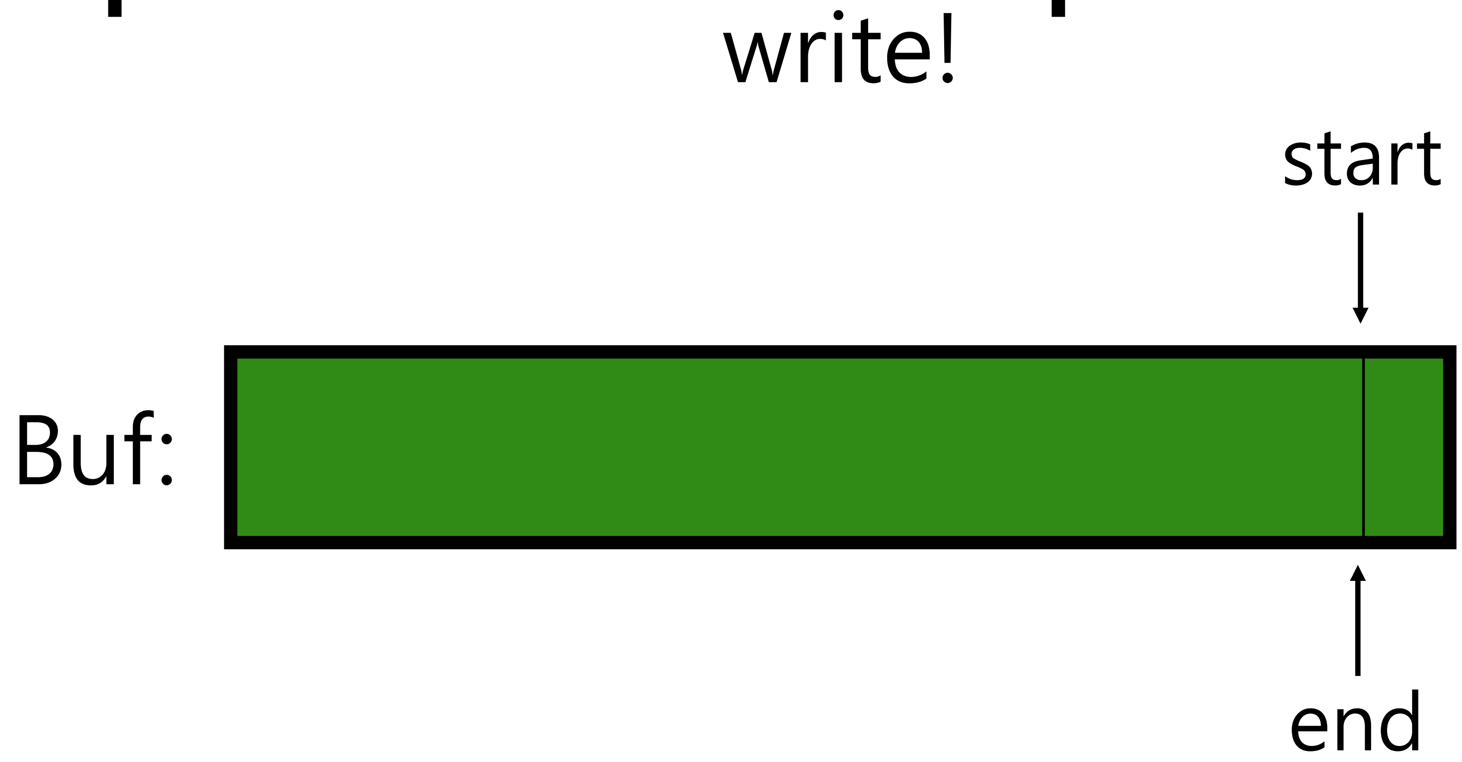
Example: UNIX Pipes



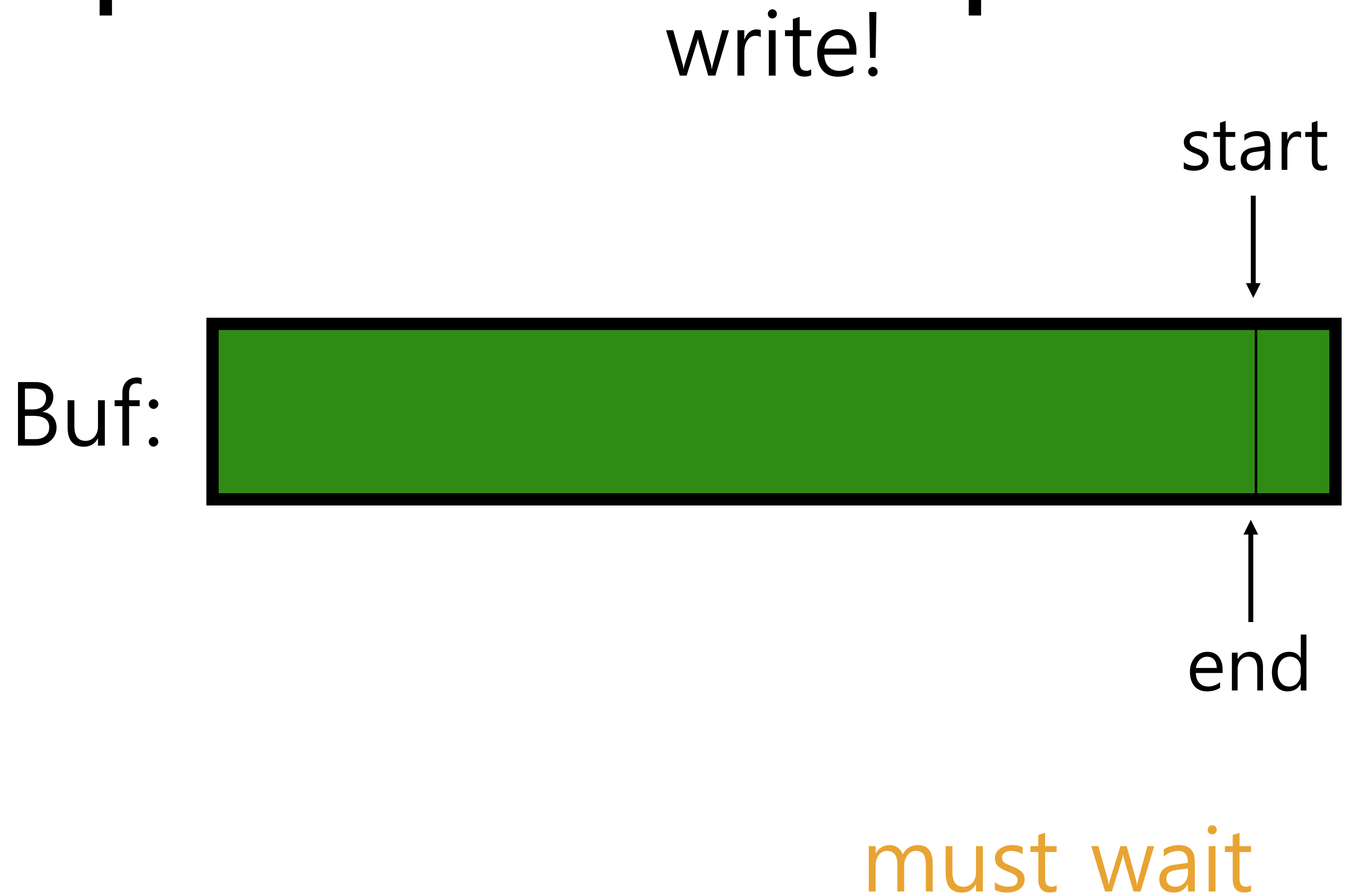
Example: UNIX Pipes



Example: UNIX Pipes

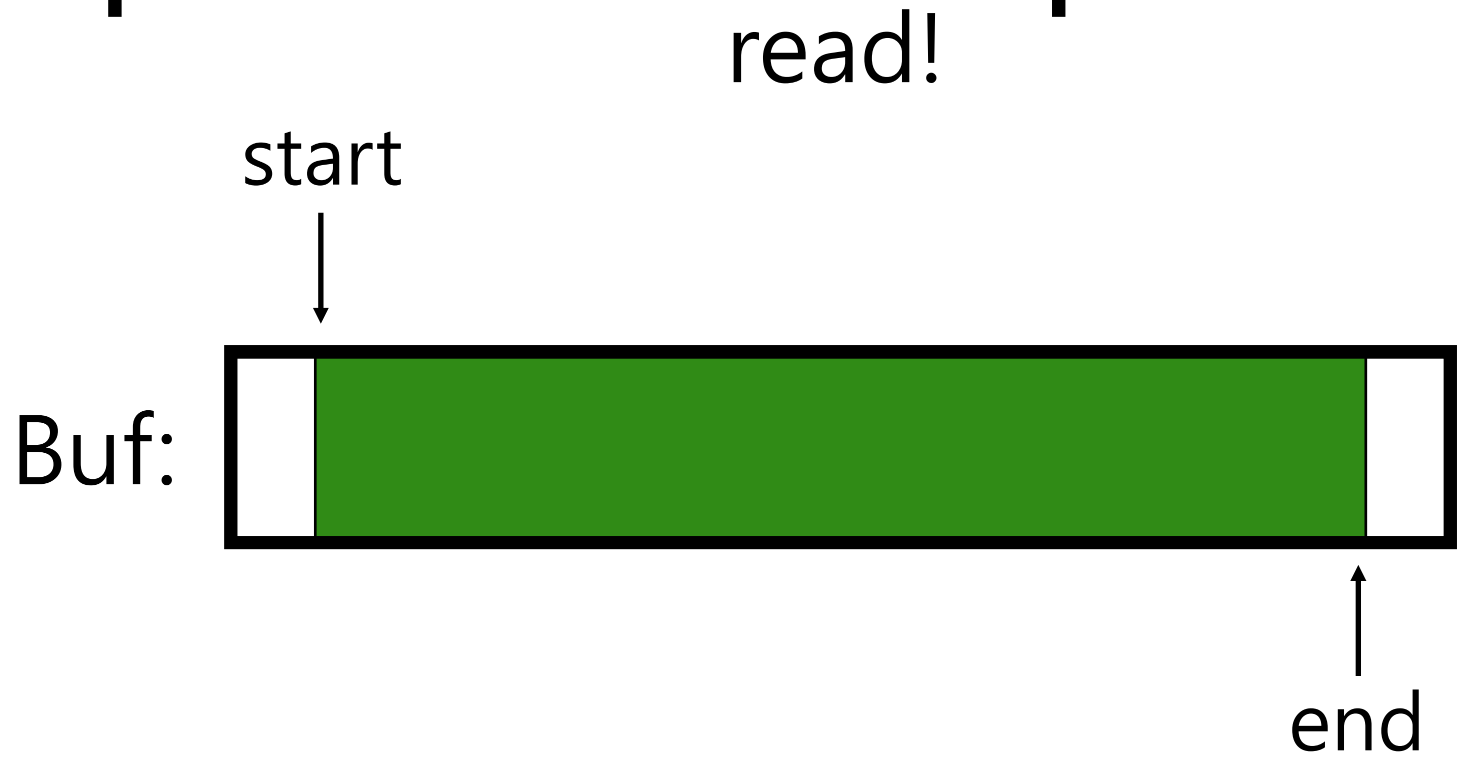


Example: UNIX Pipes



Example: UNIX Pipes

Example: UNIX Pipes



Example: UNIX Pipes

Implementation:

- reads/writes to buffer require locking
- when buffers are full, writers must wait
- when buffers are empty, readers must wait

The Producer / Consumer (Bound Buffer) Problem

Producer

- Produce data items
- Wish to place data items in a buffer

Consumer

- Grab data items out of the buffer consume them in some way

Example: Multi-threaded web server

- *A producer* puts HTTP requests in to a work queue
- *Consumer threads* take requests out of this queue and process them

Bounded buffer

A bounded buffer is used when you pipe the output of one program into another.

- Example: `grep foo file.txt | wc -l`
  - The `grep` process is the producer.
  - The wc process is the consumer.
  - Between them is an in-kernel bounded buffer.
- Bounded buffer is Shared resource → **Synchronized access** is required.

The Put and Get Routines (Version 1)

```
int buffer;
int count = 0;        // initially, empty

void put(int value) {
        assert(count == 0);
        count = 1;
        buffer = value;
}

int get() {
        assert(count == 1);
        count = 0;
        return buffer;
}
```

- Only put data into the buffer when `count` is zero.
  - i.e., when the buffer is *empty*.
- Only get data from the buffer when `count` is one.
  - i.e., when the buffer is *full*.

Producer/Consumer Threads (Version 1)

```c
void *producer(void *arg) {
        int i;
        int loops = (int) arg;
        for (i = 0; i < loops; i++) {
                put(i);
        }
}

void *consumer(void *arg) {
        int i;
        while (1) {
                int tmp = get();
                printf("%d\n", tmp);
        }
}
```

- **Producer** puts an integer into the shared buffer loops number of times.
- **Consumer** gets the data out of that shared buffer.

Producer/Consumer: Single CV and If Statement

A single condition variable `cond` and associated lock `mutex`

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // p1
        if (count == 1)                             // p2
            Pthread_cond_wait(&cond, &mutex);       // p3
        put(i);                                     // p4
        Pthread_cond_signal(&cond);                 // p5
        Pthread_mutex_unlock(&mutex);               // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // c1
```

Producer/Consumer: Single CV and If Statement

```
            if (count == 0)                          // c2
                Pthread_cond_wait(&cond, &mutex);    // c3
            int tmp = get();                         // c4
            Pthread_cond_signal(&cond);              // c5
            Pthread_mutex_unlock(&mutex);            // c6
            printf("%d\n", tmp);
        }
    }
```

- p1-p3: A producer waits for the buffer to be empty.
- c1-c3: A consumer waits for the buffer to be full.
- With just *a single producer* and *a single consumer*, the code works.

If we have more than one of producer and consumer?

Thread Trace: Broken Solution (Version 1)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
C1	Running		Ready		Ready	0	
C2	Running		Ready		Ready	0	
C3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in …
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	… and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	**Oh oh! No data**

Thread Trace: Broken Solution (Version 1)

The problem arises for a simple reason:

- After the producer woke T_{c1}, but before T_{c1} ever ran, the state of the bounded buffer *changed by* T_{c2}.

- There is no guarantee that when the woken thread runs, the state will still be as desired → Mesa semantics.
  - Virtually every system ever built employs *Mesa semantics*.

- Hoare semantics provides a stronger guarantee that the woken thread will run immediately upon being woken.

Producer/Consumer: Single CV and While

Consumer T_{c1} wakes up and re-checks the state of the shared variable.

- If the buffer is empty, the consumer simply goes back to sleep.

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // p1
        while (count == 1)                      // p2
            Pthread_cond_wait(&cond, &mutex);   // p3
        put(i);                                 // p4
        Pthread_cond_signal(&cond);             // p5
        Pthread_mutex_unlock(&mutex);           // p6
    }
}

```

Producer/Consumer: Single CV and While

```
(Cont.)
    void *consumer(void *arg) {
        int i;
        for (i = 0; i < loops; i++) {
            Pthread_mutex_lock(&mutex);                 // c1
            while (count == 0)                          // c2
                Pthread_cond_wait(&cond, &mutex);       // c3
            int tmp = get();                            // c4
            Pthread_cond_signal(&cond);                 // c5
            Pthread_mutex_unlock(&mutex);               // c6
            printf("%d\n", tmp);
        }
    }
```

- A simple rule to remember with condition variables is to **always use while loops**.
- However, this code still has a bug (*next page*).

Thread Trace: Broken Solution (Version 2)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	**Oops! Woke T_{c2}**

Thread Trace: Broken Solution (Version 2) (Cont.)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
…	…	…	…	…	…	…	(*cont.*)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	**Everyone asleep …**

- A consumer should not wake other consumers, only producers, and vice-versa.

The single Buffer Producer/Consumer Solution

- Use two condition variables and while
  - **Producer** threads wait on the condition `empty`, and signals `fill`.
  - **Consumer** threads wait on `fill` and signal `empty`.

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

```

The single Buffer Producer/Consumer Solution

```
(Cont.)
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

The Final Producer/Consumer Solution

- More **concurrency** and **efficiency** → Add more buffer slots.
  - Allow concurrent production or consuming to take place.
  - Reduce context switches.

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```

The Final Put and Get Routines

The Final Producer/Consumer Solution (Cont.)

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // p1
        while (count == MAX)                    // p2
            Pthread_cond_wait(&empty, &mutex);        // p3
        put(i);                                 // p4
        Pthread_cond_signal(&fill);             // p5
        Pthread_mutex_unlock(&mutex);           // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // c1
        while (count == 0)                      // c2
            Pthread_cond_wait(&fill, &mutex);   // c3
        int tmp = get();                        // c4
```

The Final Producer/Consumer Solution (Cont.)

```
(Cont.)
            Pthread_cond_signal(&empty);             // c5
            Pthread_mutex_unlock(&mutex);            // c6
            printf("%d\n", tmp);
        }
    }
```

The Final Working Solution (Cont.)

- p2: **A producer** only sleeps if all buffers are currently filled.
- c2: **A consumer** only sleeps if all buffers are currently empty.

Covering Conditions

- Assume there are zero bytes free
  - Thread T_a calls `allocate(100)`.
  - Thread T_b calls `allocate(10)`.
  - Both T_a and T_b wait on the condition and go to sleep.
  - Thread T_c calls `free(50)`.

Which waiting thread should be woken up?

Covering Conditions (Cont.)

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...;                // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c);        // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

Covering Conditions (Cont.)

- Solution (Suggested by Lampson and Redell)
  - Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`
  - `pthread_cond_broadcast()`
    - Wake up **all waiting threads.**
    - Cost: too many threads might be woken.
    - Threads that shouldn’t be awake will simply wake up, re-check the condition, and then go back to sleep.

Producer/Consumer Problem

Producers generate data (like pipe writers)

Consumers grab data and process it (like pipe readers)

Producer/consumer problems are frequent in systems

- Web servers

General strategy use condition variables to:
make producers wait when buffers are full
make consumers wait when there is **nothing to consume**

Produce/Consumer Example

Start with easy case:

- 1 producer thread
- 1 consumer thread
- 1 shared buffer to fill/consume (max = 1)

Numfill = number of buffers currently filled

Examine slightly broken code to begin…

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNING]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNING]

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

[SLEEPING]

```
void *consumer(void *arg) {
	while(1) {
		Mutex_lock(&m);
		while(numfull == 0)
			Cond_wait(&cond, &m);
		int tmp = do_get();
		Cond_signal(&cond);
		Mutex_unlock(&m);
		printf(“%d₩n”, tmp);
	}
}
```

numfull=0

[RUNNING]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNING]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[RUNNING]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[RUNNING]

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

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[RUNNING]

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

[RUNNABLE]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d₩n", tmp);
    }
}
```

numfull=1

[RUNNING]

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

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[RUNNING]

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

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[RUNNING]

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

[RUNNABLE]

```
void *consumer(void *arg) {
	while(1) {
		Mutex_lock(&m);
		while(numfull == 0)
			Cond_wait(&cond, &m);
		int tmp = do_get();
		Cond_signal(&cond);
		Mutex_unlock(&m);
		printf("%d₩n", tmp);
	}
}
```

numfull=1

[SLEEPING]

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

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[SLEEPING]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[SLEEPING]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[SLEEPING]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNING]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=0

[RUNNING]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[RUNNING]

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

[SLEEPING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

numfull=1

[RUNNING]

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

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                while(numfull == 0)
                        Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d₩n", tmp);
        }
}
```

What about 2 consumers?

Can you find a problematic timeline with 2 consu mers (still 1 producer)?

```
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
        }
}
```

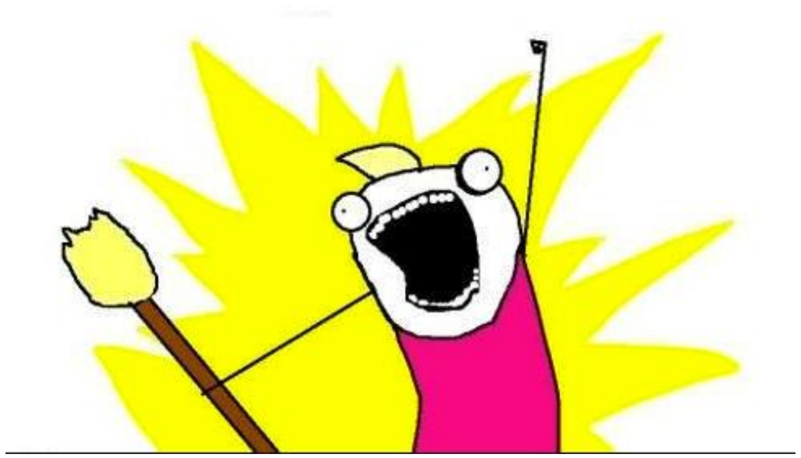
			wait()			wait()				signal()				wait()			signal()
Producer:							p1	p2	p4	p5	p6	p1	p2	p3			
Consumer1:	c1	c2	c3														
Consumer2:				c1	c2	c3									c2	c4	c5

does last signal wake producer or consumer2?

How to wake the right thread?

One solution:

wake all the threads!



Waking All Waiting Threads

wait(cond_t *cv, mutex_t *lock)

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

broadcast(cond_t *cv)

any disadvantage?

- - wake **all** waiting threads (if >= 1 thread is waiting)
- - if there are no waiting thread, just return, doing nothing

Example Need for Broadcast

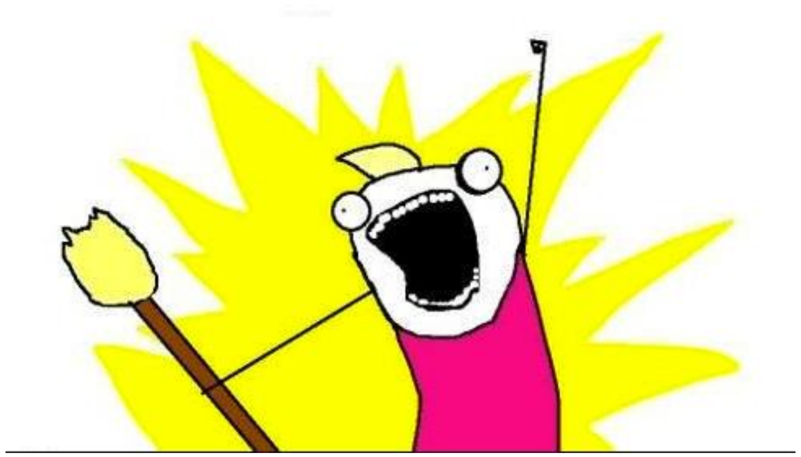
```
void *allocate(int size) {
  mutex_lock(&m);
  while (bytesLeft < size)
        cond_wait(&c);
  …
}
```

```
void free(void *ptr, int size) {
        …
        cond_broadcast(&c)
        …
}
```

How to wake the right thread?

One solution:

wake all the threads!



Better solution (usually): use two condition variables

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);
                if (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?

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

Good Rule of Thumb 3

Whenever a lock is acquired, recheck assumptions about state!

Possible for another thread to grab lock in between signal and wak eup from wait

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