[537] Producer/Consumer Problem

Tyler Harter
10/15/14

Review: Condition Variables

Condition Variables

Condition Variable: queue of sleeping threads.

Threads add themselves to the queue with **wait**.

Threads wake up threads on the queue with **signal**.

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

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

Why must sleep/unlock be atomic?

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

Why must sleep/unlock be atomic?
Why do we need kernel support?

Review: Join

Do problem 1.

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

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

Review: Join

Do problem 1.

Parent: x y z
Child: a b c

GOOD!

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

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

Review: Join

Do problem 1.

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

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

Review: Join

Do problem 1.

Parent: x y
Child: a b c

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

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

Review: Join

Do problem 1.

Parent: x y … sleep forever …
Child: a b c

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

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

Good Rule of Thumb 1

Keep state in addition to CV's!

CV's are used to nudge threads when state changes.

If state is already as needed, don't wait for a nudge!

Review: Join

Do problem 2.

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

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

Review: Join

Do problem 2.

Parent: w x y … sleep forever …
Child: a b

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

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

Review: Join

Do problem 2.

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

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

Review: Join

Do problem 2.

Correct!

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

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

Review: Join

Do problem 2.

Correct?

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

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

Review: Join

Do problem 2.

Correct?

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

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

Review: Join

Do problem 2.

Not only option, but preferred.

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

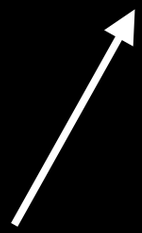
```
void thread_join() {
    Mutex_lock(&m);              // w
    if (done == 0)               // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m);            // z
}
```

Good Rule of Thumb 2

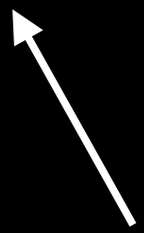
Always do **wait** and **signal** while holding the lock!

Good Rule of Thumb 2

Always do **wait** and **signal** while holding the lock!



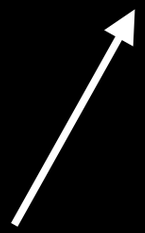
strictly required



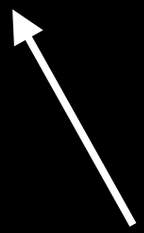
good practice, but in some libraries it's not required.

Good Rule of Thumb 2

Always do **wait** and **signal** while holding the lock!



strictly required



good practice, but in some libraries it's not required.

Doing so will help prevent lost signals.

Producer/Consumer Problem

Example: UNIX Pipes

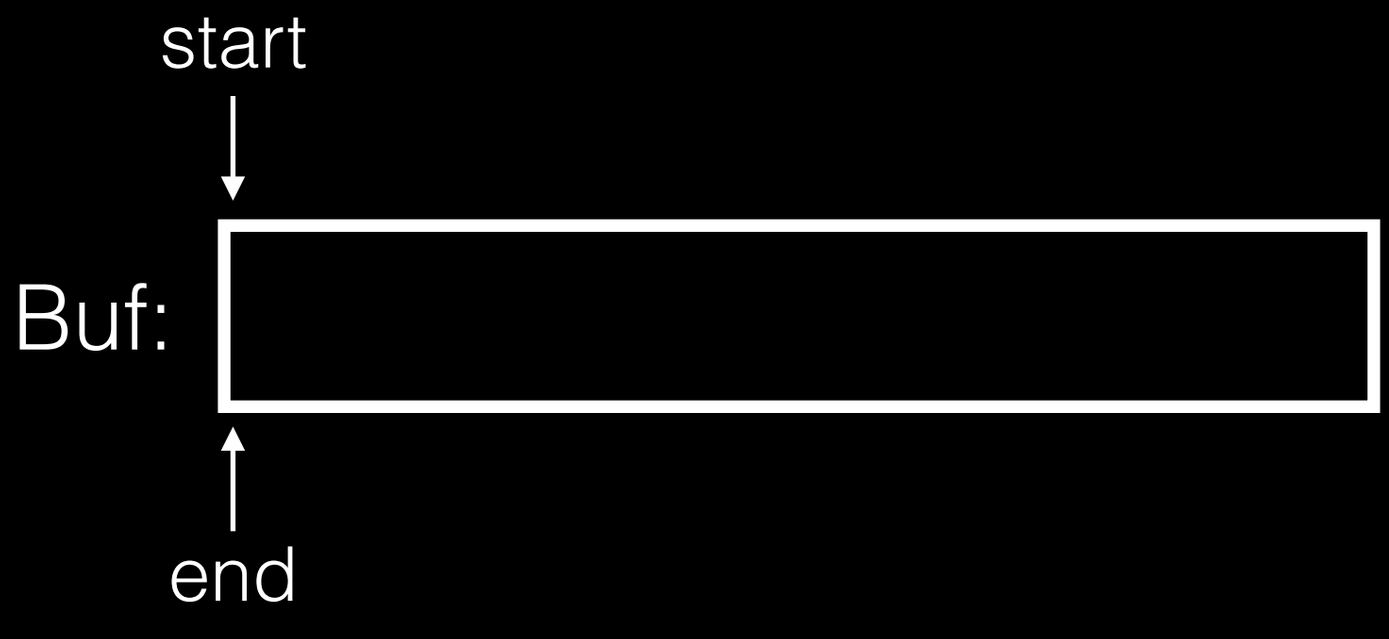
A pipe may have many writers and readers.

Internally, there is a finite-sized buffer.

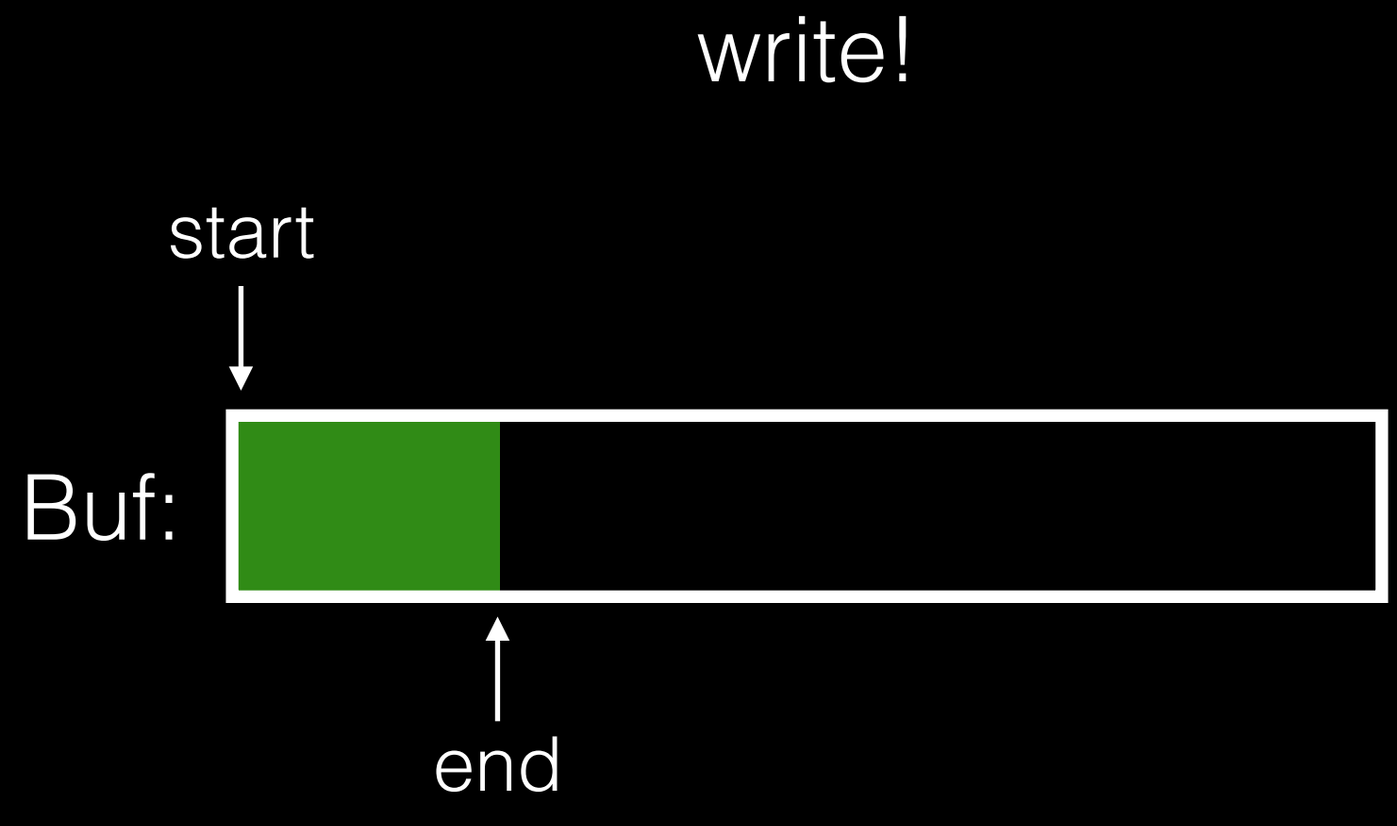
Writers add data to the buffer.

Readers remove data from the buffer.

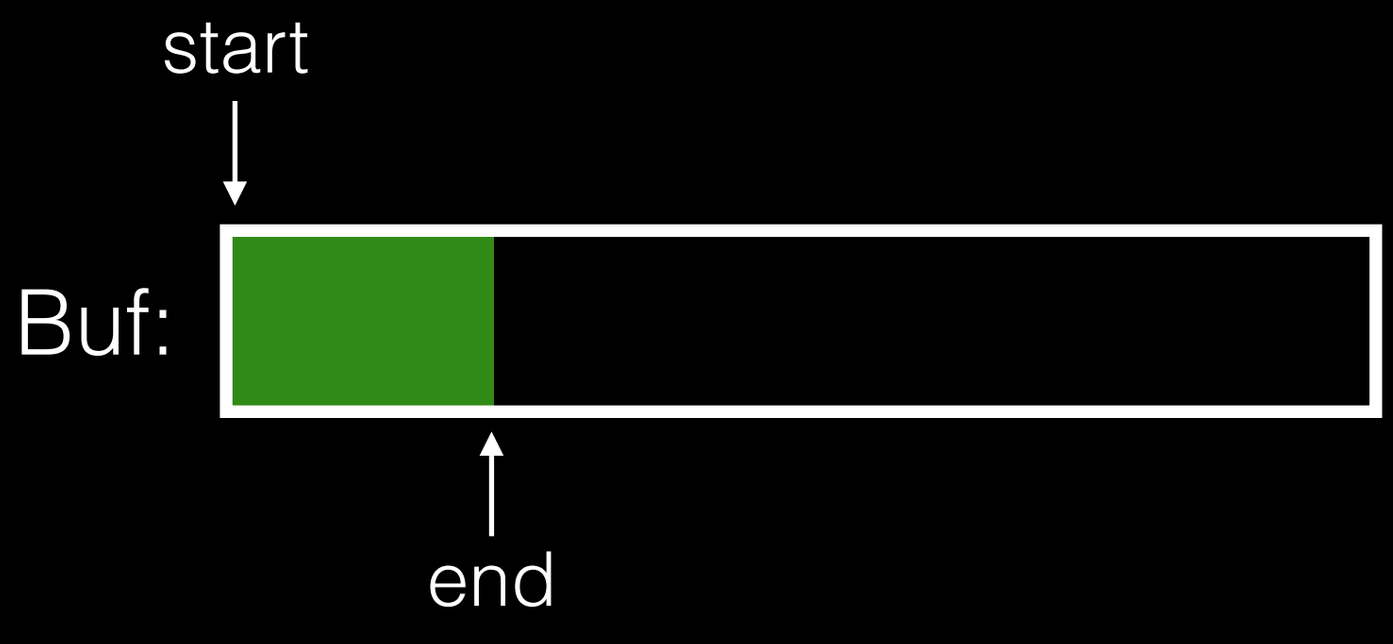
Example: UNIX Pipes



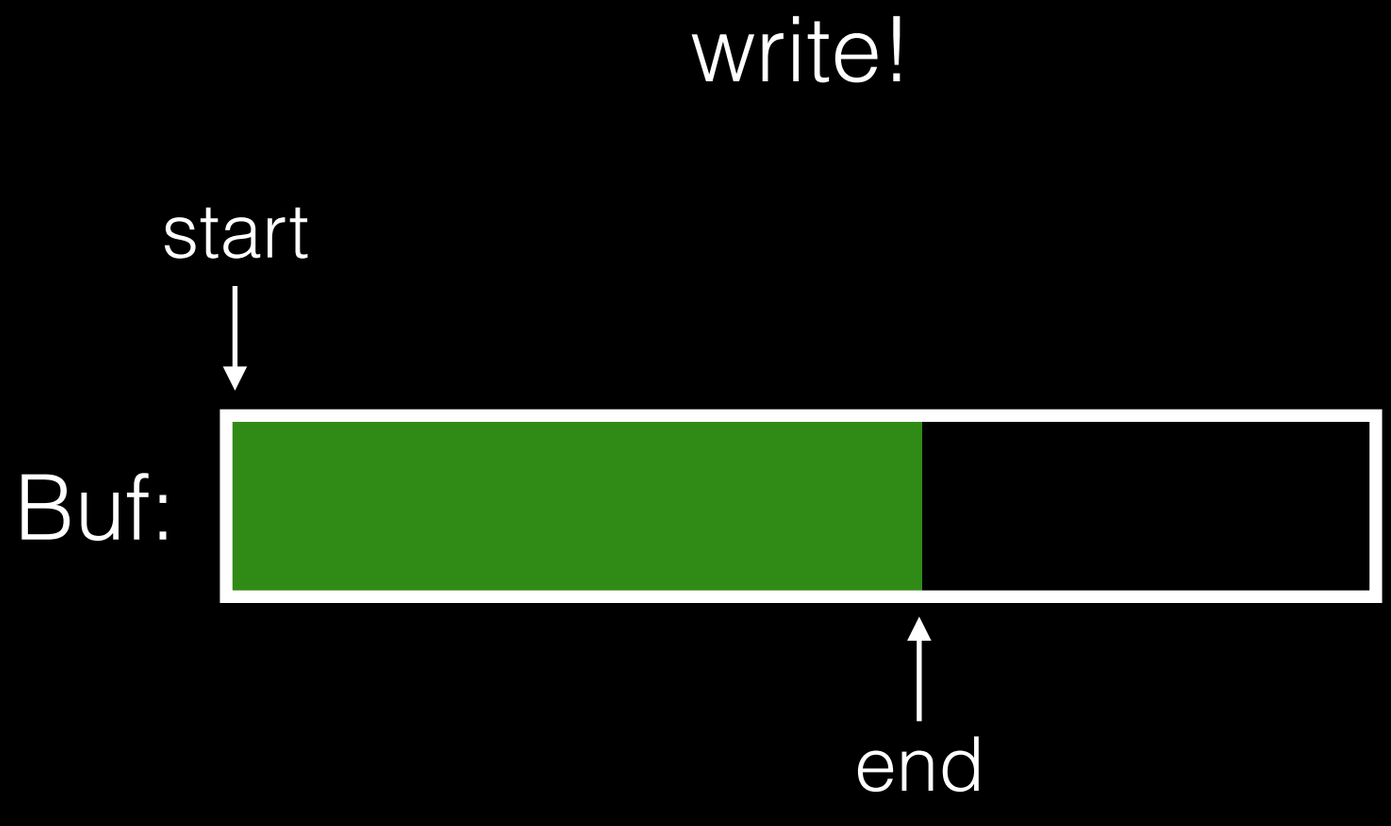
Example: UNIX Pipes



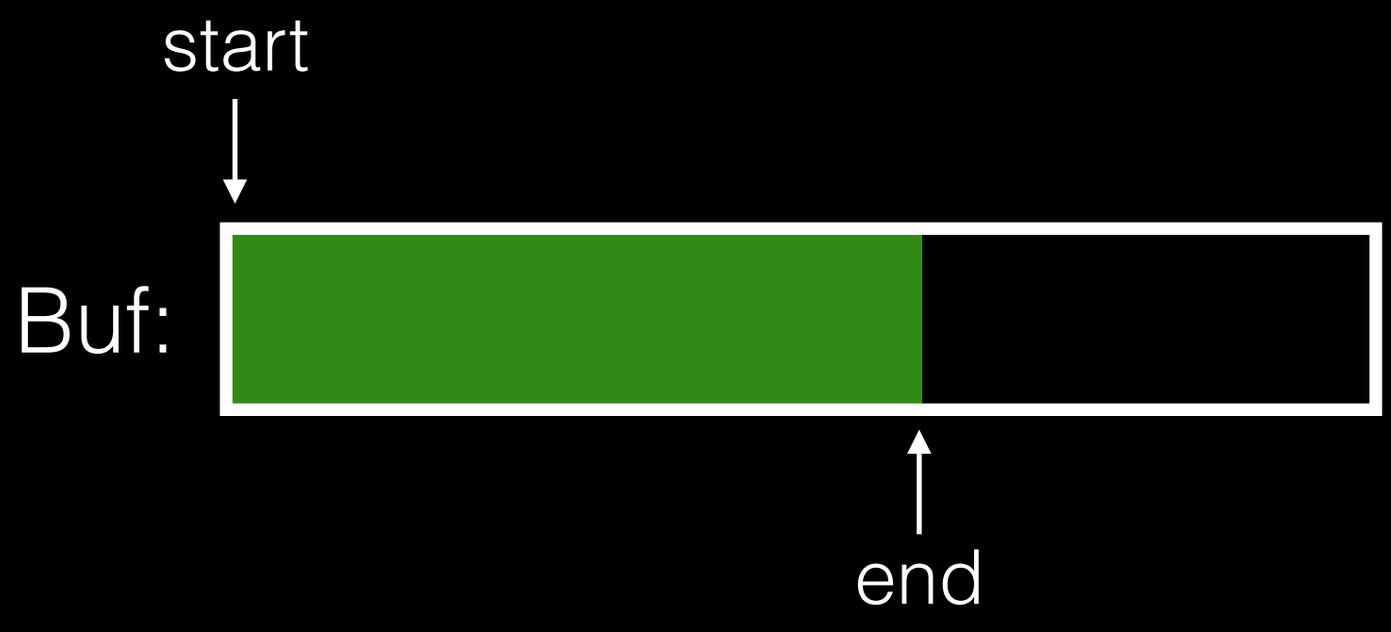
Example: UNIX Pipes



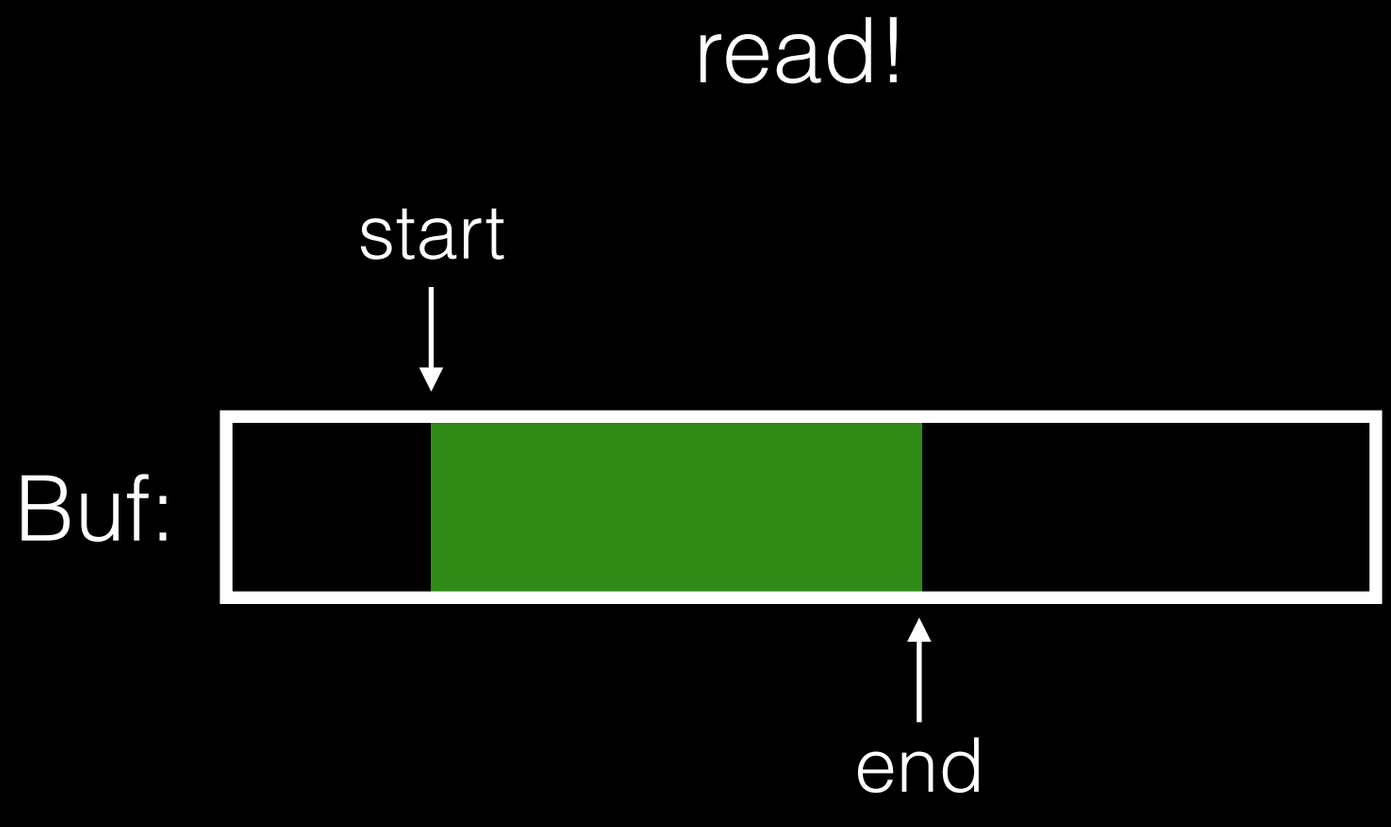
Example: UNIX Pipes



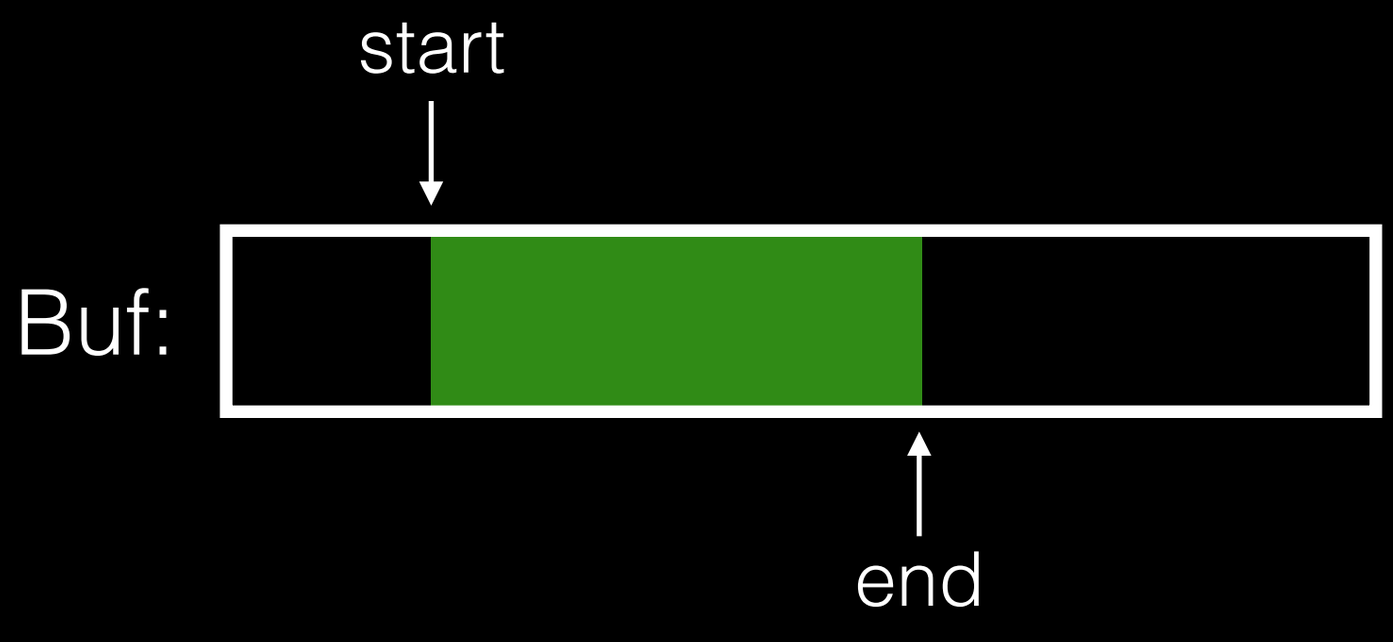
Example: UNIX Pipes



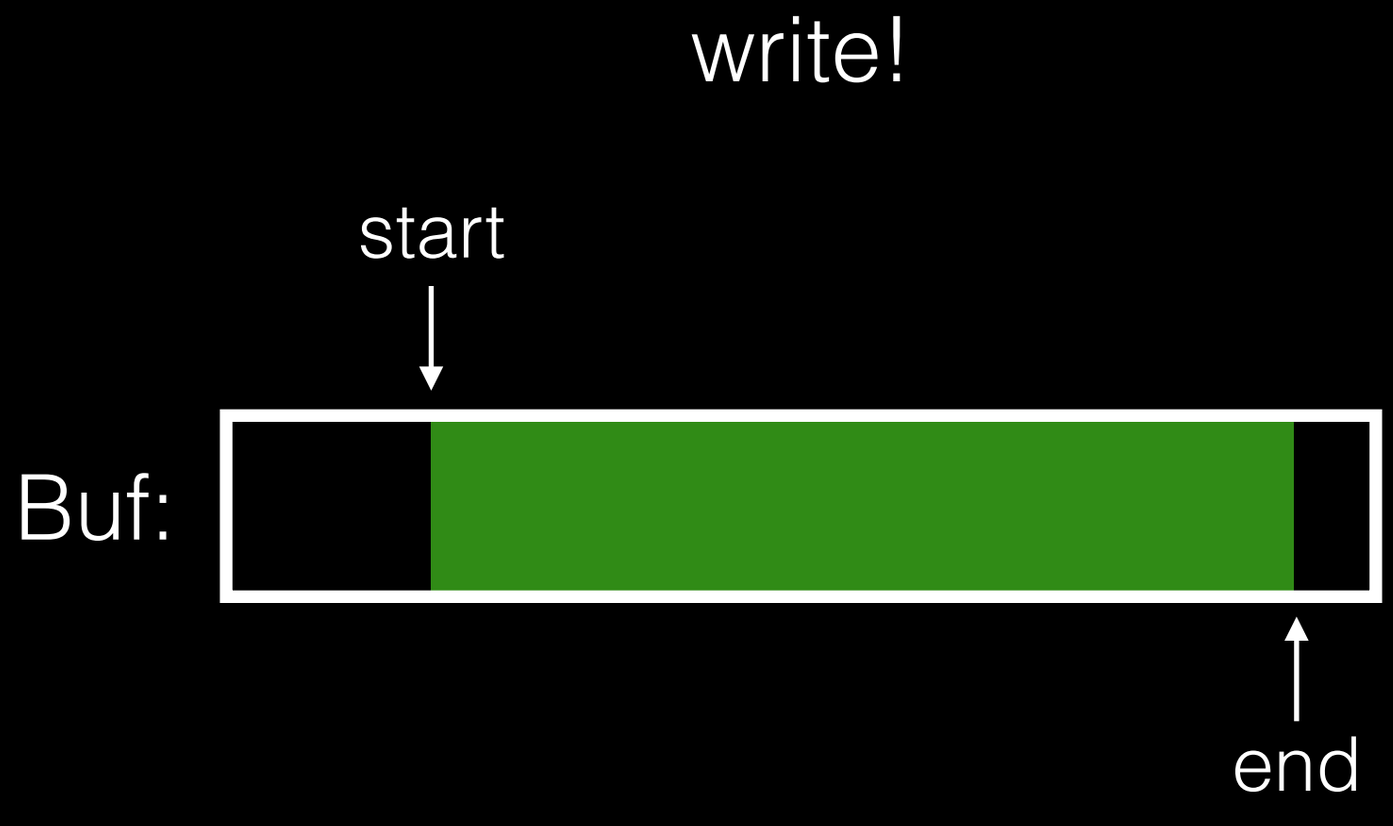
Example: UNIX Pipes



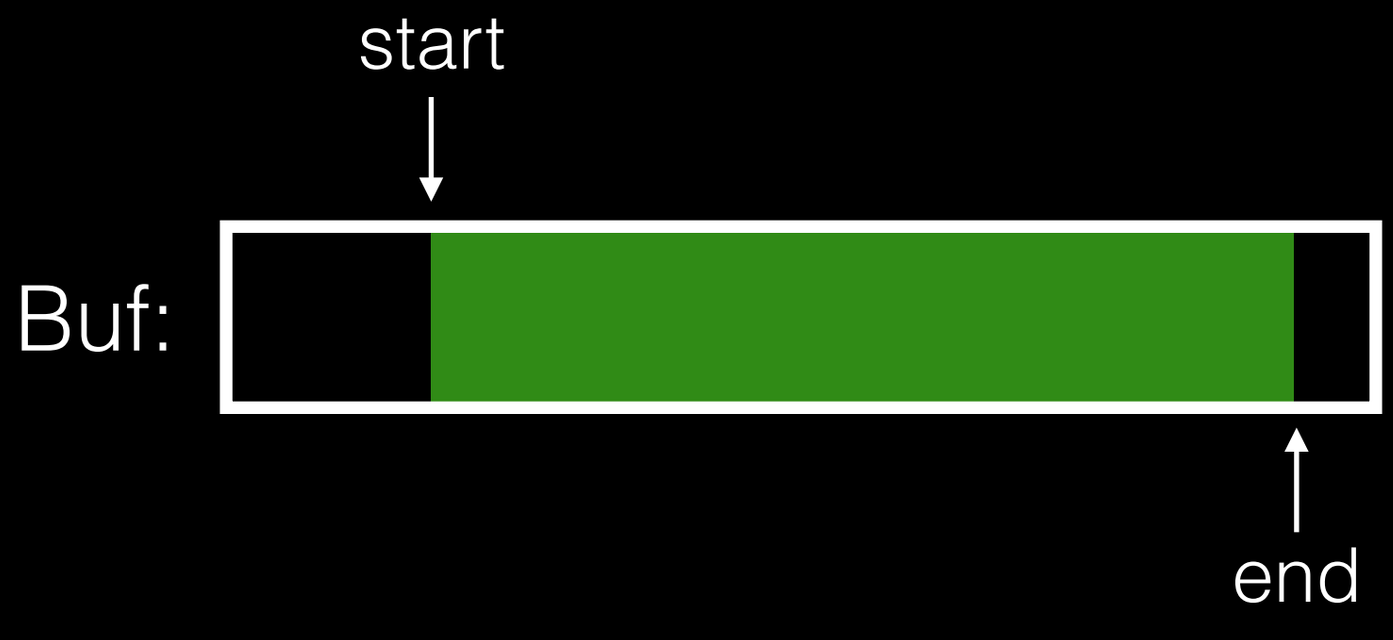
Example: UNIX Pipes



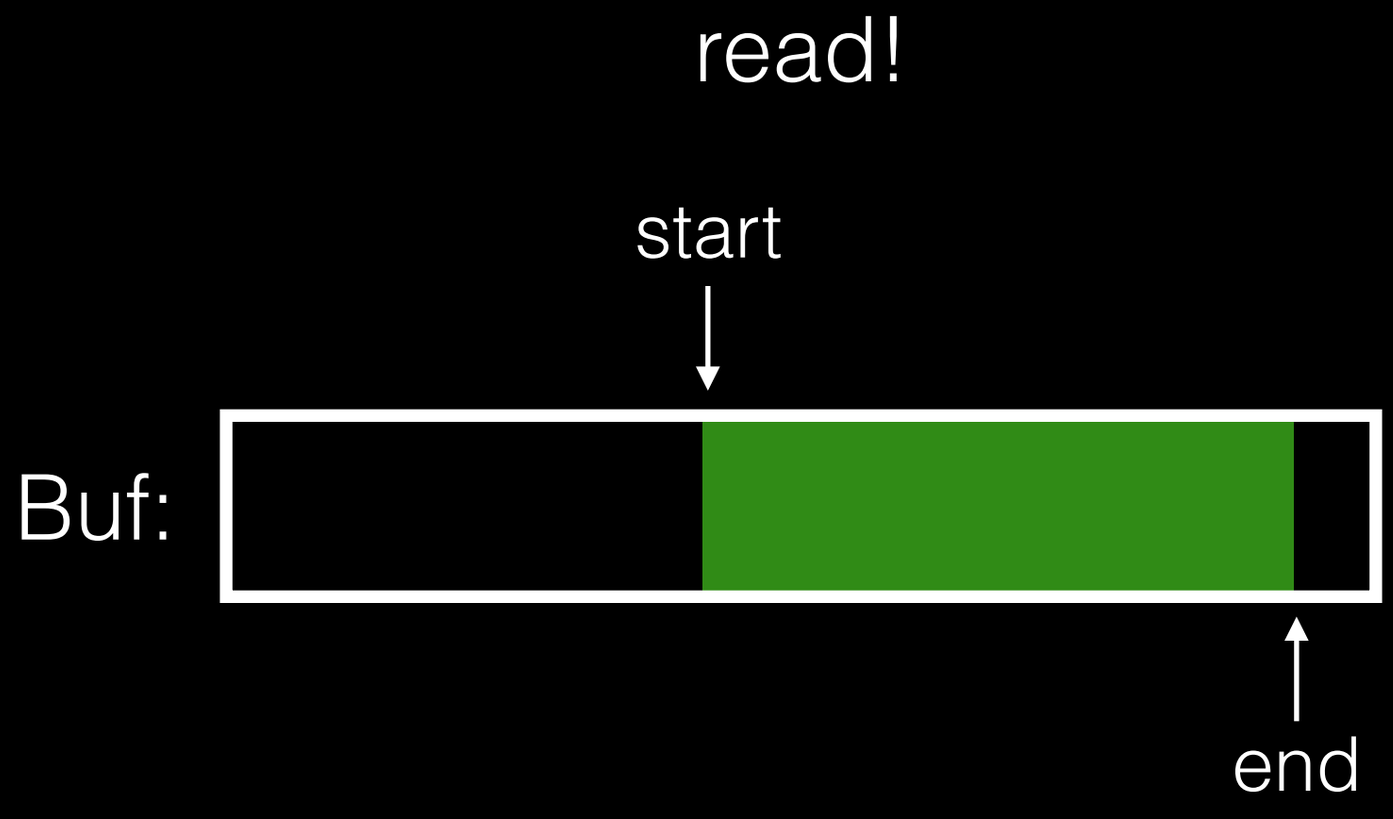
Example: UNIX Pipes



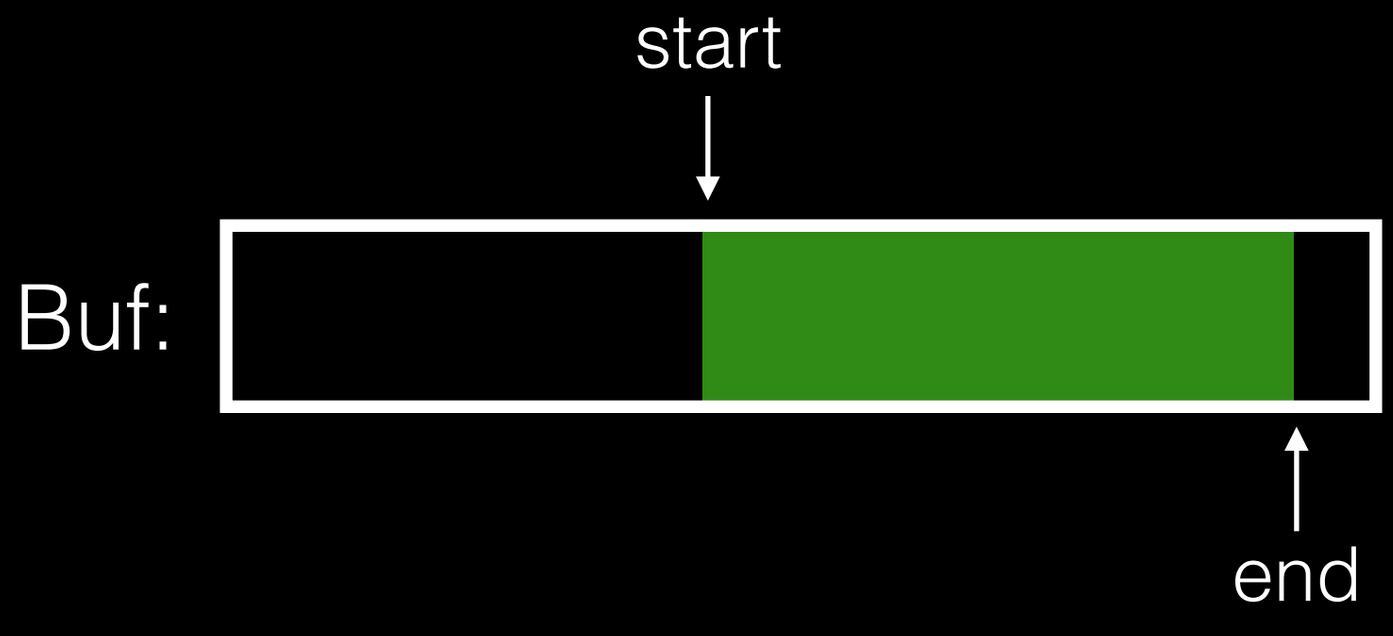
Example: UNIX Pipes



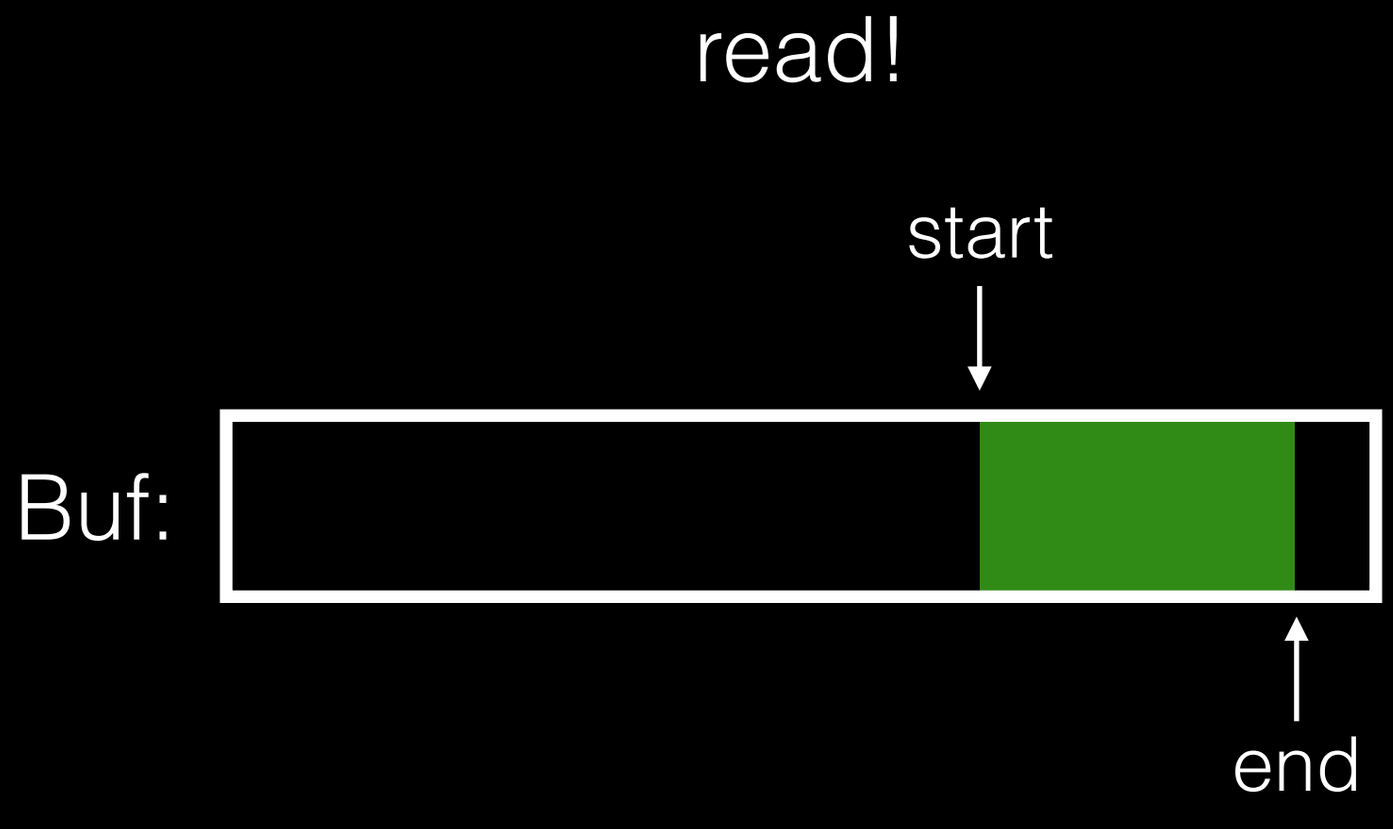
Example: UNIX Pipes



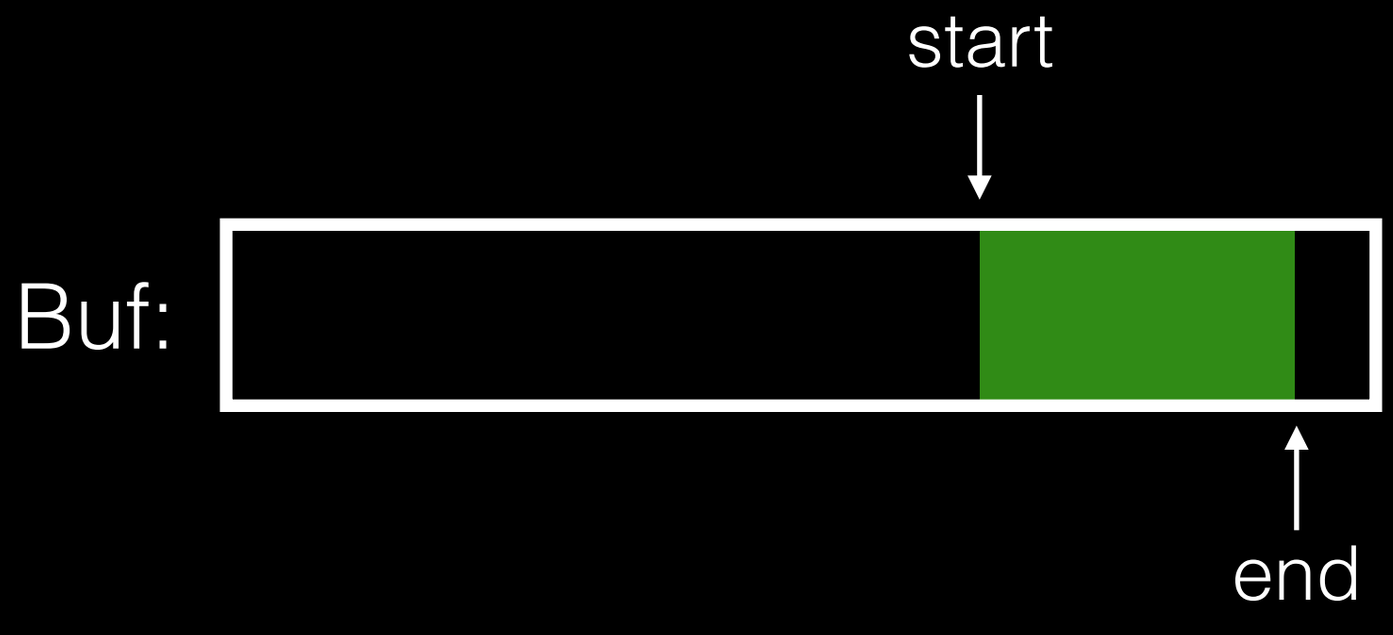
Example: UNIX Pipes



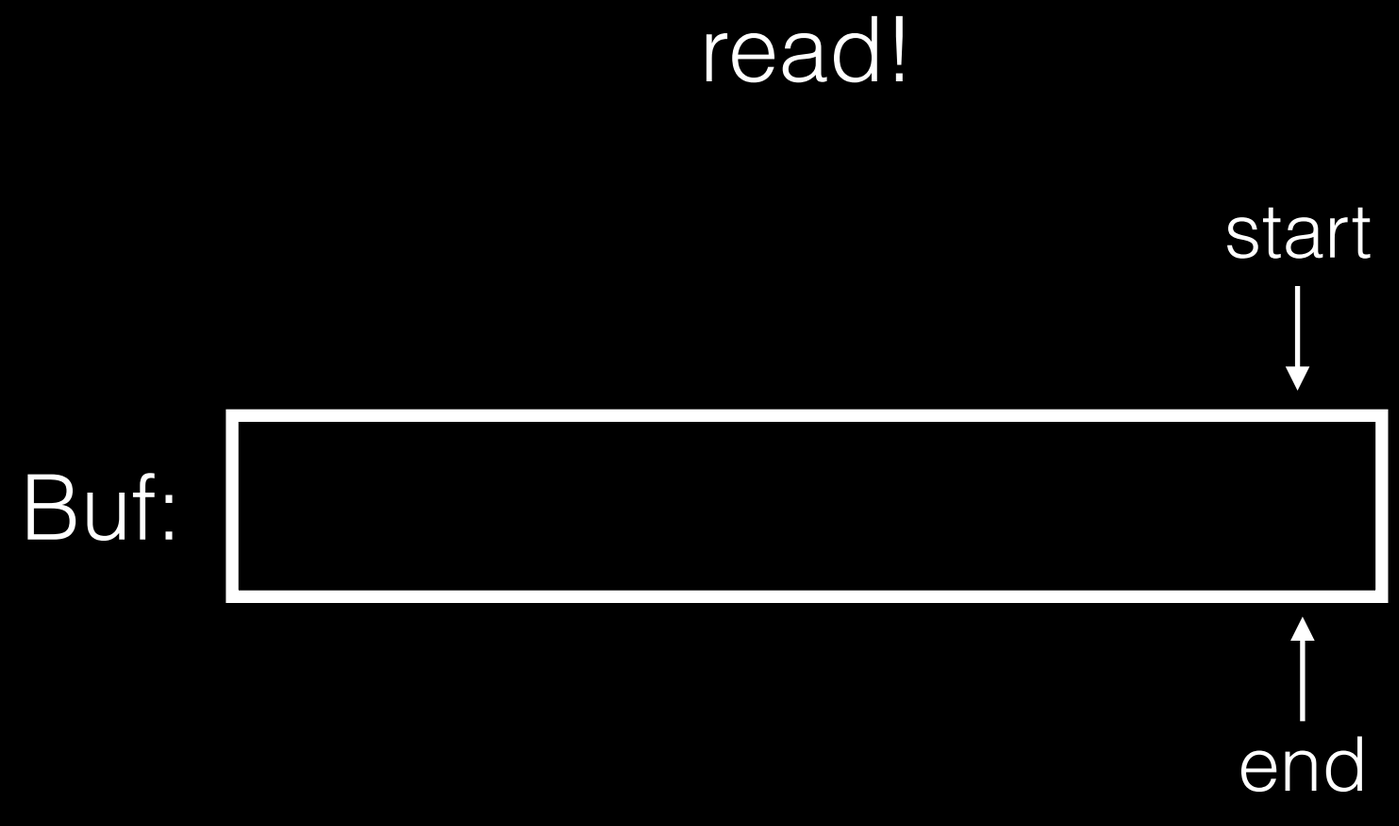
Example: UNIX Pipes



Example: UNIX Pipes

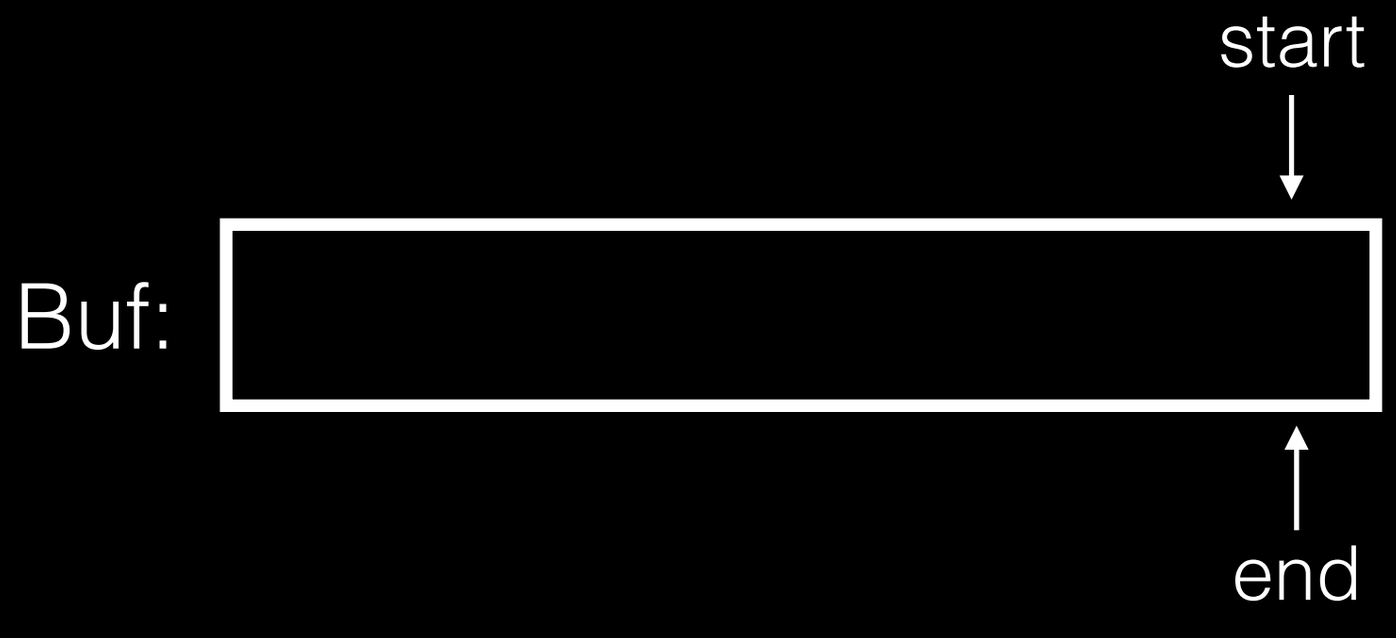


Example: UNIX Pipes



Example: UNIX Pipes

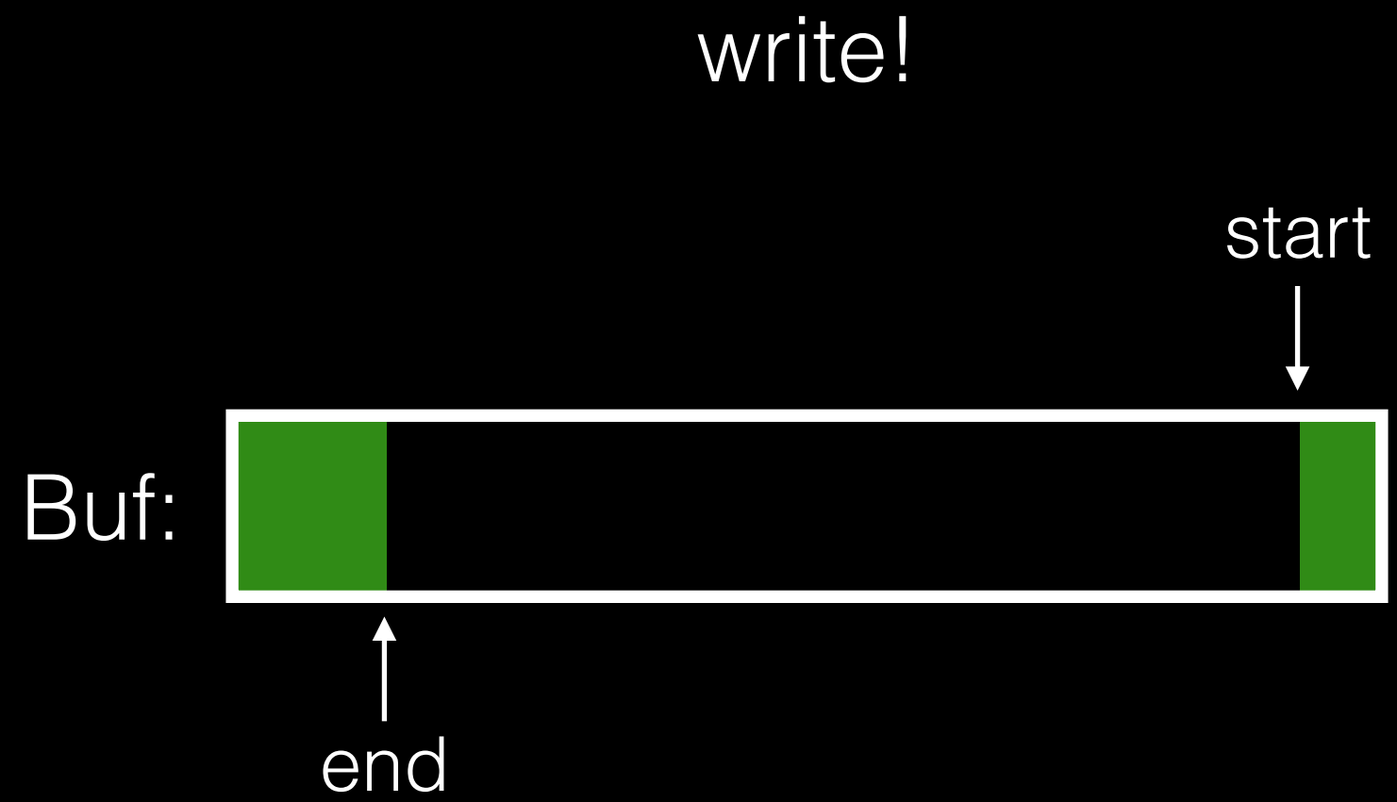
read!



note: readers must wait

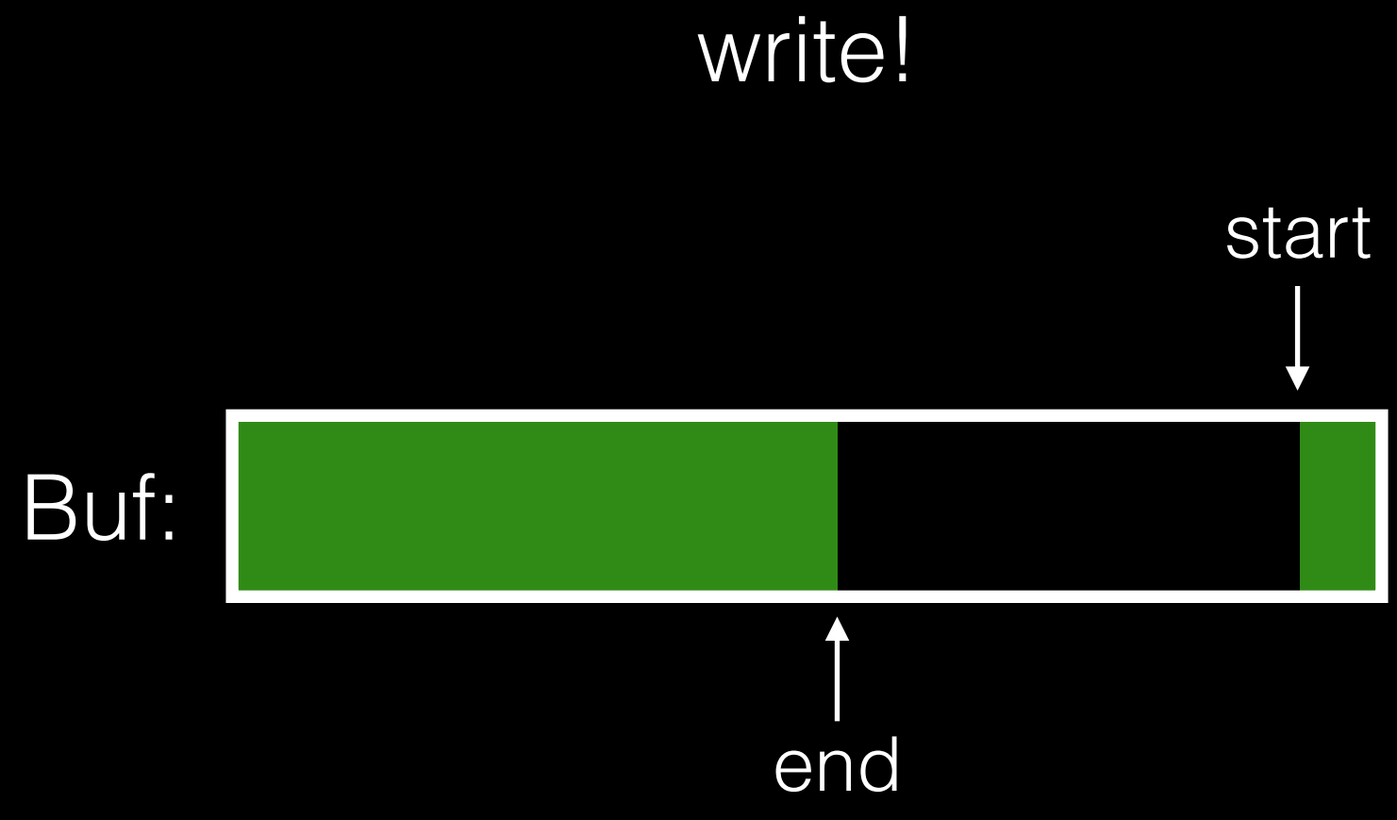
Example: UNIX Pipes

Example: UNIX Pipes

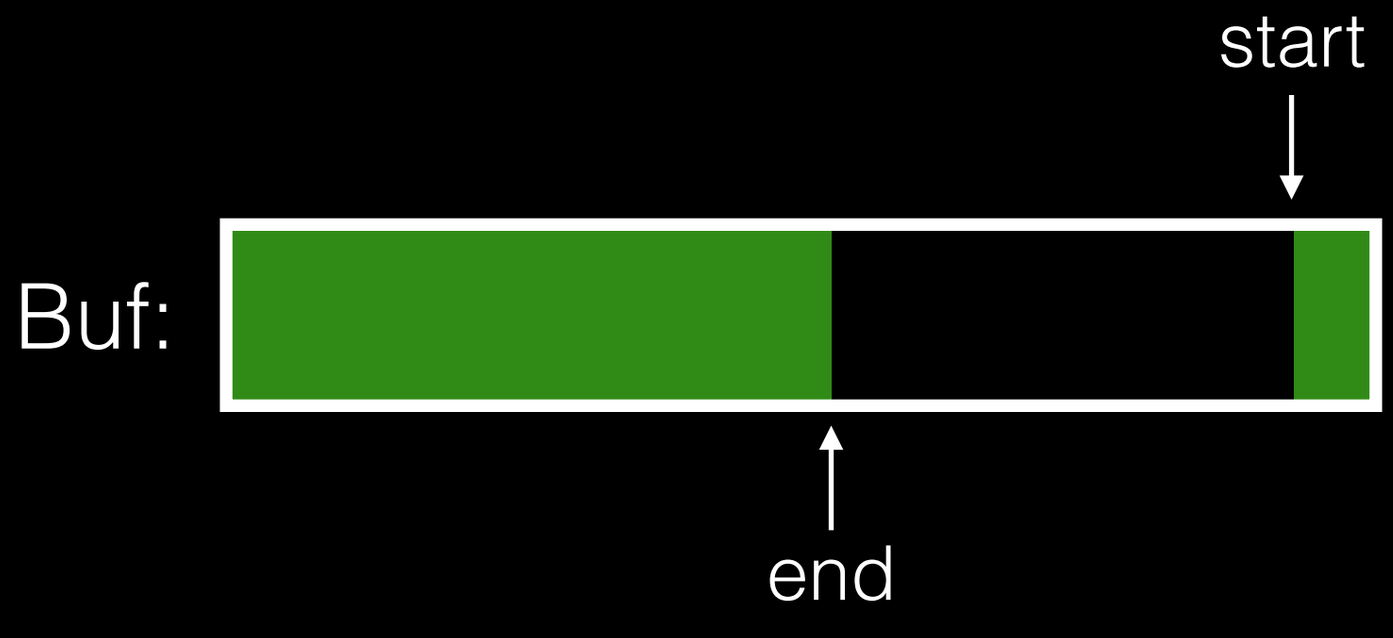


Example: UNIX Pipes

Example: UNIX Pipes



Example: UNIX Pipes



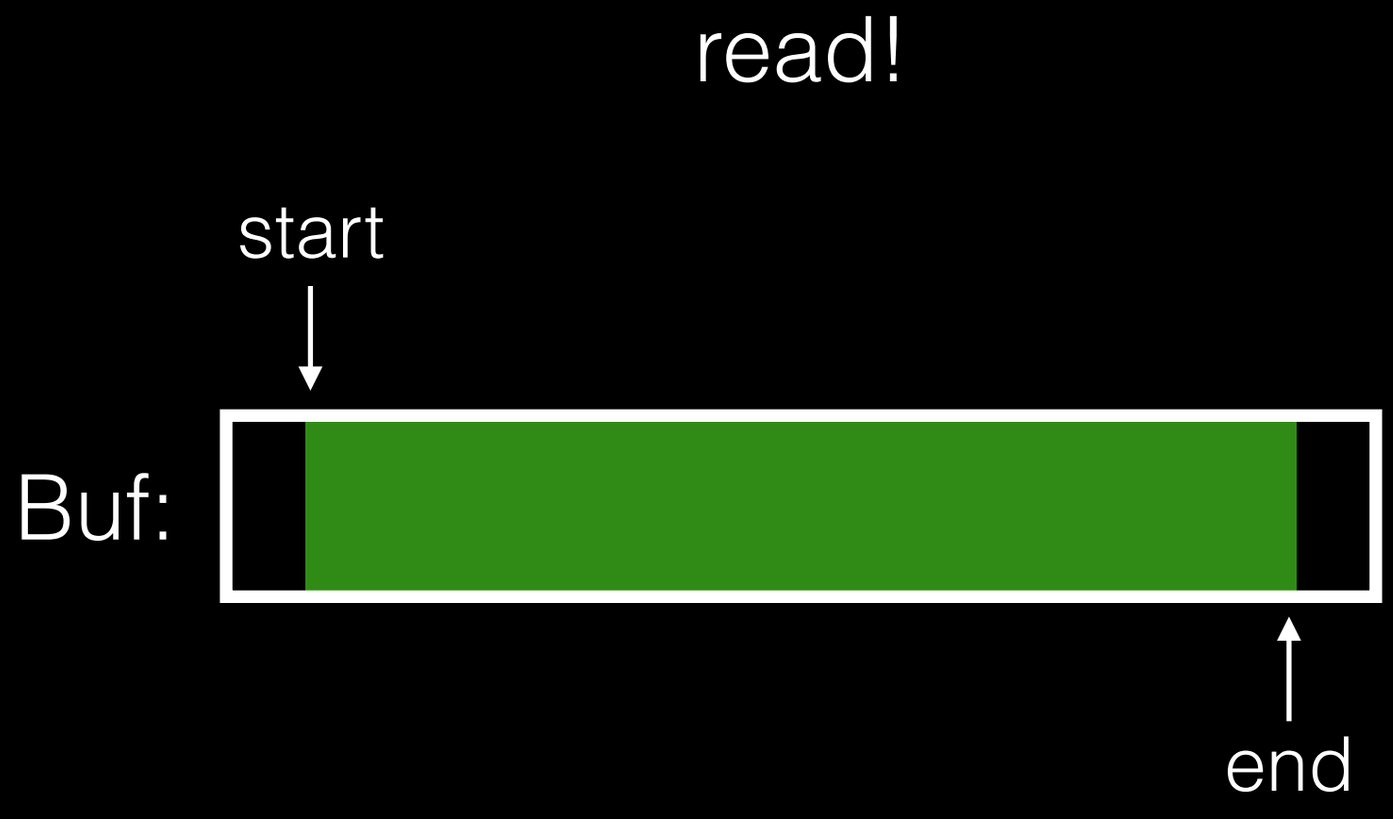
Example: UNIX Pipes

Example: UNIX Pipes

note: writers must wait

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

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.

Other Examples

Pipes
Web servers
Memory allocators
Device I/O
…

Other Examples

Pipes
Web servers
Memory allocators
Device I/O
…

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

Code Examples

Worksheet

main() creates **1** producer and **N** consumers.

The producer calls **do_fill**().

The consumers call **do_get**().

Example v1

Let’s first assume:

- max = 1
- there is one consumer

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

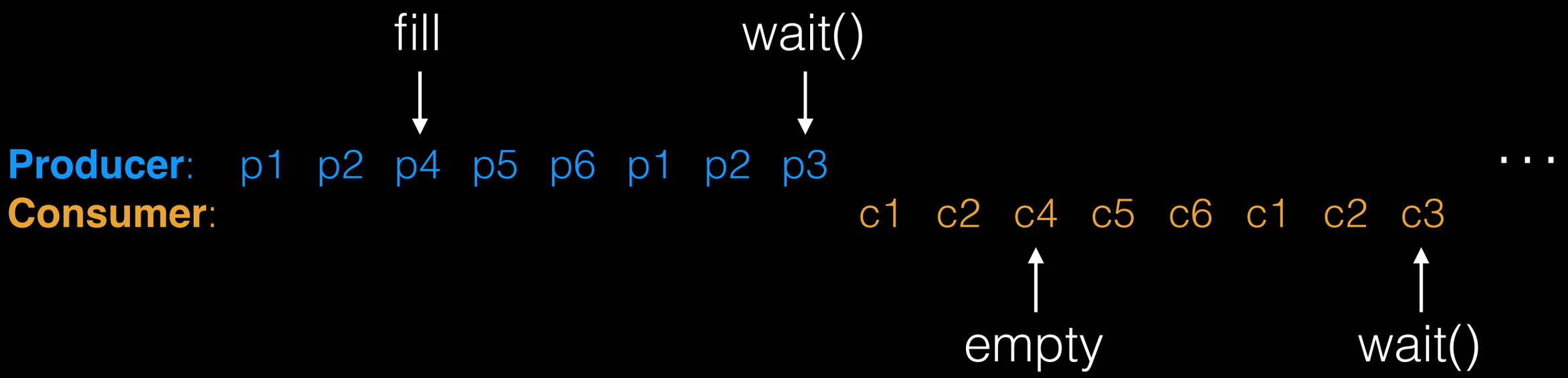
[RUNNING]

```
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);
    }
}
```

Timeline

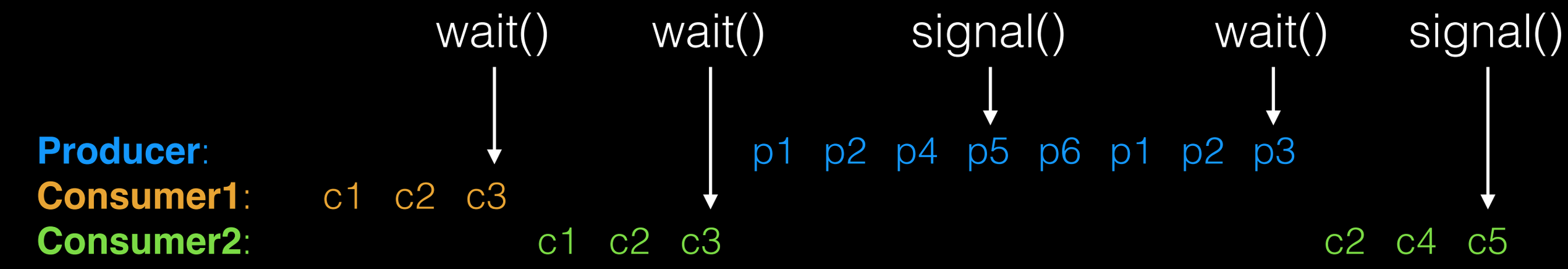

fill
wait()
Producer: p1 p2 p4 p5 p6 p1 p2 p3 ...
Consumer: c1 c2 c4 c5 c6 c1 c2 c3
empty
wait()

What about 2 consumers (v1)?

Can you find a problematic timeline?

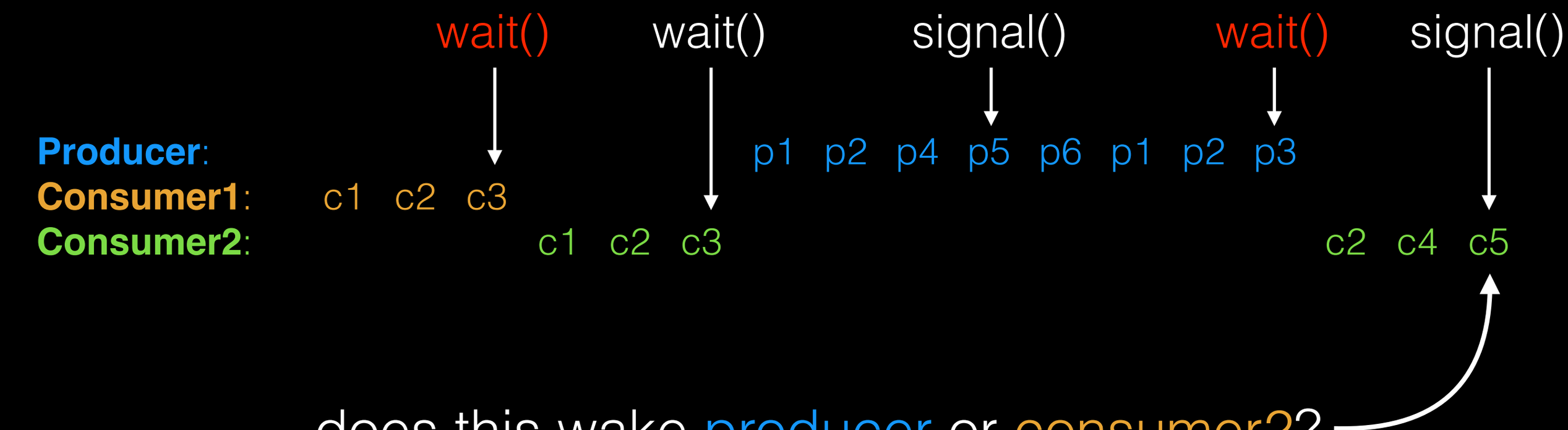
What about 2 consumers (v1)?

Can you find a problematic timeline?



What about 2 consumers (v1)?

Can you find a problematic timeline?



does this wake producer or consumer2?

How to wake the right thread?

How to wake the right thread?

One solution:

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

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

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

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

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

How to wake the right thread?

One solution:

How to wake the right thread?

One solution:

Better solution (usually): use two condition variables.

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)
    ...
}
```

Example v2

Is this correct? Can you find a bad schedule?

Example v2

Is this correct? Can you find a bad schedule?

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

Good Rule of Thumb 3

Whenever a lock is acquired, recheck assumptions about state!

Locks are acquired explicitly with Mutex_lock().

Locks are acquired implicitly with Cond_wait().

Note that some libraries also have “spurious wakeups”

Example v3

Is this correct? Can you find a bad schedule?

Example v3

Is this correct? Yes!

Can you find a bad schedule? No!

We observe:

- 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

Keep state in addition to CV's

Always do wait/signal with lock held

Whenever you acquire a lock, recheck state

Announcements

Exam this Friday.
- Oct 17, 7-9pm, in CHEM 1351.
- Read OSTEP!

Review tonight.
- Oct 15, 7-9pm, CS 1221, come with questions!

Office hours at 1pm.