# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

Running thread:
`worker1`

Ready queue:

- `worker2`
- `scheduler`

Blocked set:
<empty>

numQueuedLock:

numQueued:
`0`

queueCv:

`waiting threads:`

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

Running thread:

`worker1`

Ready queue:

- `worker2`
- `scheduler`

Blocked set:

<empty>

numQueuedLock:

numQueued:

`0`

queueCv:

`waiting threads:`

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker1 acquires mutex

Running thread:
`worker1`

Ready queue:
- `worker2`
- `scheduler`

Blocked set:
`<empty>`

numQueuedLock:

numQueued:
`0`

queueCv:

`waiting threads:`

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker1 acquires mutex

Running thread:
`worker1`

Ready queue:
- `worker2`
- `scheduler`

Blocked set:
`<empty>`

numQueuedLock:

numQueued:
`0`

queueCv:

`waiting threads:`

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker2 starts running

Running thread:
worker2

Ready queue:
- scheduler
- worker1

Blocked set:
<empty>

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

Running thread:
worker2

Ready queue:

- scheduler
- worker1

Blocked set:
<empty>

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:

worker2 attempts to acquire the lock. It's already locked, so worker2 gets moved to the blocked queue.

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

Running thread:
`scheduler`

Ready queue:
- `worker1`

Blocked set:
- `worker2`

numQueuedLock:

numQueued:
`0`

queueCv:

| waiting threads: |
|---|

worker2 attempts to acquire the lock. It's already locked, so worker2 gets moved to the blocked set.

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler starts running

Running thread:
scheduler

Ready queue:
- worker1

Blocked set:
- worker2

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler sleeps. It gets moved to the blocked set.

Running thread:
scheduler

Ready queue:
- worker1

Blocked set:
- worker2

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler sleeps. It gets moved to the blocked set.

Running thread:
worker1

Ready queue:
<empty>

Blocked set:
- worker2
- scheduler

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

Back in worker1…

Running thread:

`worker1`

Ready queue:

<empty>

Blocked set:

- `worker2`
- `scheduler`

numQueuedLock:

numQueued:

`0`

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker1 calls cv.wait(). The predicate function returns false, so worker1 atomically unlocks the mutex and goes to sleep. Unlocking the lock places worker2 back on the ready queue.

Running thread:
worker1

Ready queue:
<empty>

Blocked set:
- worker2
- scheduler

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker1 calls cv.wait(). The predicate function returns false, so worker1 atomically unlocks the mutex and goes to sleep. Unlocking the lock places worker2 back on the ready queue.

Running thread:
<none>

Ready queue:
- worker2

Blocked set:
- scheduler
- worker1

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:
- worker1

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker2 wakes back up and attempts to acquire the lock again. This time, it succeeds.

Running thread:
`worker2`

Ready queue:
<empty>

Blocked set:
- `scheduler`
- `worker1`

numQueuedLock:

numQueued:
`0`

queueCv:

```
waiting threads:
• worker1
```

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

Running thread:
worker2

Ready queue:
<empty>

Blocked set:
- scheduler
- worker1

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:
- worker1

worker2 wakes back up and attempts to acquire the lock again. This time, it succeeds.

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker2 calls cv.wait(). The predicate function returns false, so worker2 atomically unlocks the mutex and goes to sleep

Running thread:
worker2

Ready queue:
<empty>

Blocked set:
- scheduler
- worker1

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:
- worker1

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker2 calls cv.wait(). The predicate function returns false, so worker2 atomically unlocks the mutex and goes to sleep

Running thread:

<empty>

Ready queue:

<empty>

Blocked set:

- scheduler
- worker1
- worker2

numQueuedLock:

numQueued:

0

queueCv:

waiting threads:
- worker1
- worker2

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler wakes up from its sleep

Running thread:
`scheduler`

Ready queue:
<empty>

Blocked set:
- `worker1`
- `worker2`

numQueuedLock:

numQueued:
`0`

queueCv:

```
waiting threads:
• worker1
• worker2
```

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler acquires numQueuedLock and increments numQueued

Running thread:
scheduler

Ready queue:
<empty>

Blocked set:
- worker1
- worker2

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:
- worker1
- worker2

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler acquires numQueuedLock and increments numQueued

Running thread:
scheduler

Ready queue:
<empty>

Blocked set:
- worker1
- worker2

numQueuedLock:

numQueued:
0

queueCv:

waiting threads:
- worker1
- worker2

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler acquires numQueuedLock and increments numQueued

Running thread:
scheduler

Ready queue:
<empty>

Blocked set:
- worker1
- worker2

numQueuedLock:

numQueued:
1

queueCv:

waiting threads:
- worker1
- worker2

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler notifies waiting threads that something has been added to the queue. Those threads are moved to the ready queue

Running thread:
`scheduler`

Ready queue:
<empty>

Blocked set:
- `worker1`
- `worker2`

numQueuedLock:

numQueued:
`1`

queueCv:

```
waiting threads:
• worker1
• worker2
```

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

scheduler notifies waiting threads that something has been added to the queue. Those threads are moved to the ready queue

Running thread:
`scheduler`

Ready queue:
- `worker1`
- `worker2`

Blocked set:
`<empty>`

numQueuedLock:

numQueued:
`1`

queueCv:

| waiting threads: |
| --- |
| |

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

At the bottom of the for loop, the lock_guard goes out of scope, so the scheduler releases numQueuedLock

Running thread:
scheduler

Ready queue:
- worker1
- worker2

Blocked set:
<empty>

numQueuedLock:

numQueued:
1

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

At the bottom of the for loop, the lock_guard goes out of scope, so the scheduler releases numQueuedLock

Running thread:
`scheduler`

Ready queue:
- `worker1`
- `worker2`

Blocked set:
`<empty>`

numQueuedLock:

numQueued:
`1`

queueCv:

```
waiting threads:
```

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

The scheduler repeats the loop and goes back to sleep.

Running thread:
`scheduler`

Ready queue:
- `worker1`
- `worker2`

Blocked set:
<empty>

numQueuedLock:

numQueued:
1

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker1 is put back on the CPU. It re-acquires numQueuedLock and calls the predicate function again, which returns true, so queueCv.wait() returns.

Running thread:
worker1

Ready queue:
- worker2

Blocked set:
- scheduler

numQueuedLock:

numQueued:
1

queueCv:

waiting threads:

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker1 is put back on the CPU. It re-acquires numQueuedLock and calls the predicate function again, which returns true, so queueCv.wait() returns.

Running thread:
`worker1`

Ready queue:
- `worker2`

Blocked set:
- `scheduler`

numQueuedLock:

numQueued:
`1`

queueCv:

`waiting threads:`

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker1 decrements numQueued and unlocks the mutex, then goes to sleep.

Running thread:
worker1

Ready queue:
- worker2

Blocked set:
- scheduler

numQueuedLock:

numQueued:
1

queueCv:

| waiting threads: |
| --- |
| |

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker1 decrements numQueued and unlocks the mutex, then goes to sleep.

Running thread:

`worker2`

Ready queue:

<empty>

Blocked set:

- `scheduler`
- `worker1`

numQueuedLock:

numQueued:

`0`

queueCv:

`waiting threads:`

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker2 is put back on the CPU. It re-acquires numQueuedLock and calls the predicate function again. Sadly, the predicate returns false, so worker2 goes back to sleep until the scheduler wakes up and enqueues another item.

Running thread:
`worker2`

Ready queue:
<empty>

Blocked set:
- `scheduler`
- `worker1`

numQueuedLock:

numQueued:
`0`

queueCv:

`waiting threads:`

# Primitive thread pool (workers.cc)

## Using condition variables

```
static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue "
            << "(numQueued = " << numQueued << ")" << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        cout << oslock << "Scheduler: added to queue (numQueued = "
            << numQueued << ")" << endl << osunlock;
        queueCv.notify_all();
    }
}
```

worker2 is put back on the CPU. It re-acquires numQueuedLock and calls the predicate function again. Sadly, the predicate returns false, so worker2 goes back to sleep until the scheduler wakes up and enqueues another item.

Running thread:
<empty>

Ready queue:
<empty>

Blocked set:
- scheduler
- worker1
- worker2

numQueuedLock:

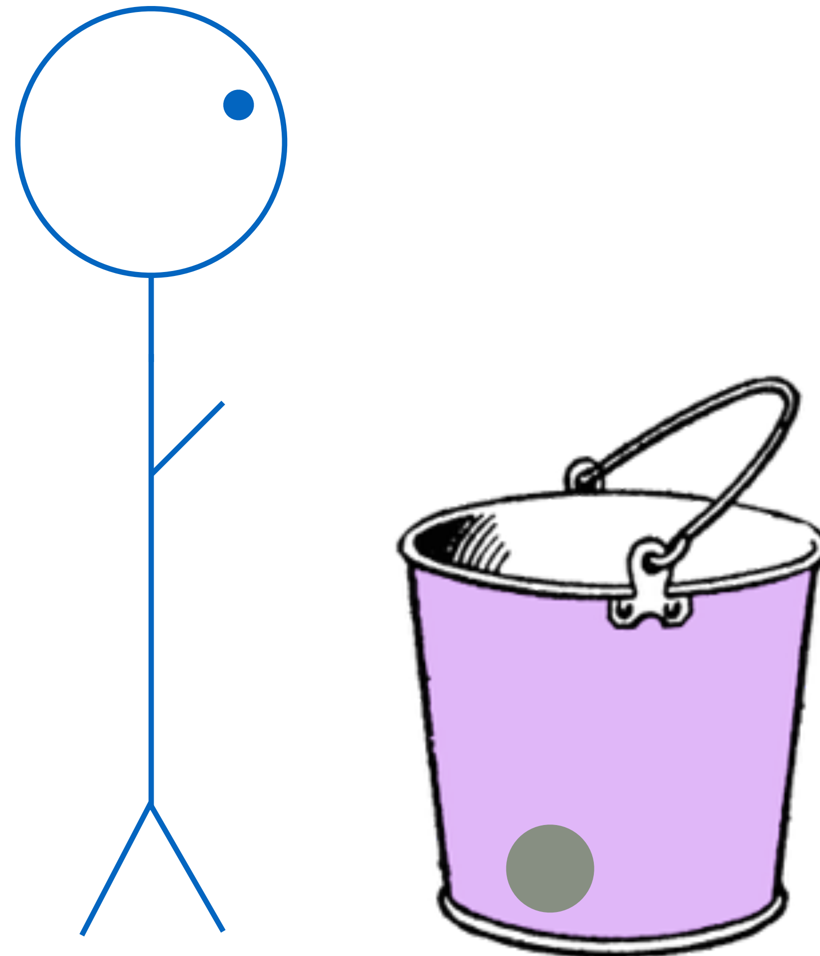
numQueued:
0

queueCv:

waiting threads:
worker2

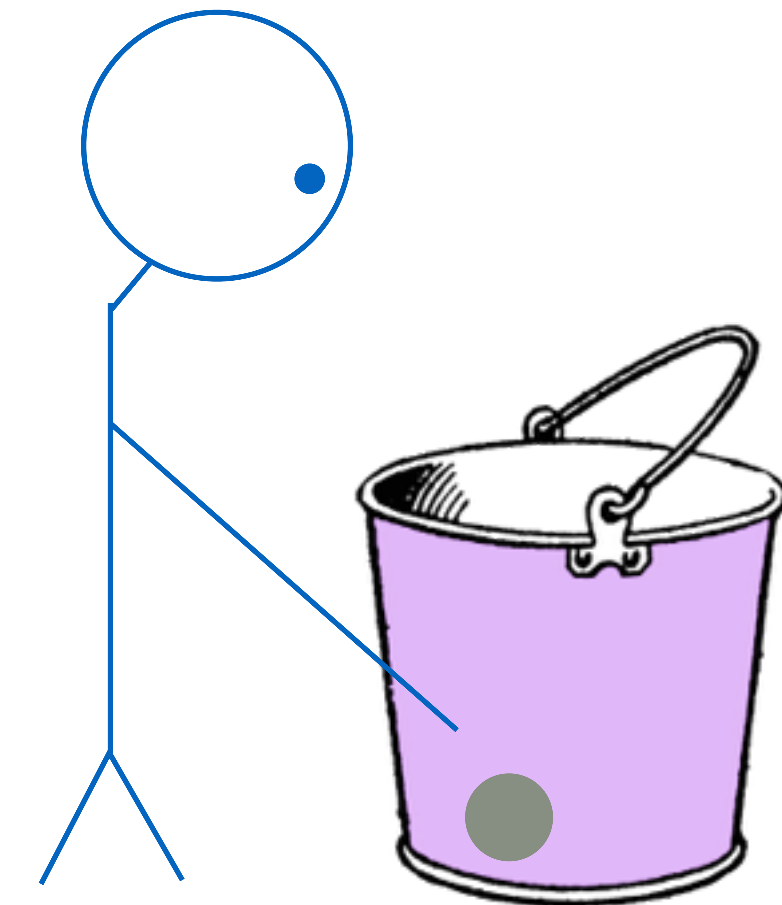
# Semaphores

Like a bucket of balls



# Semaphores

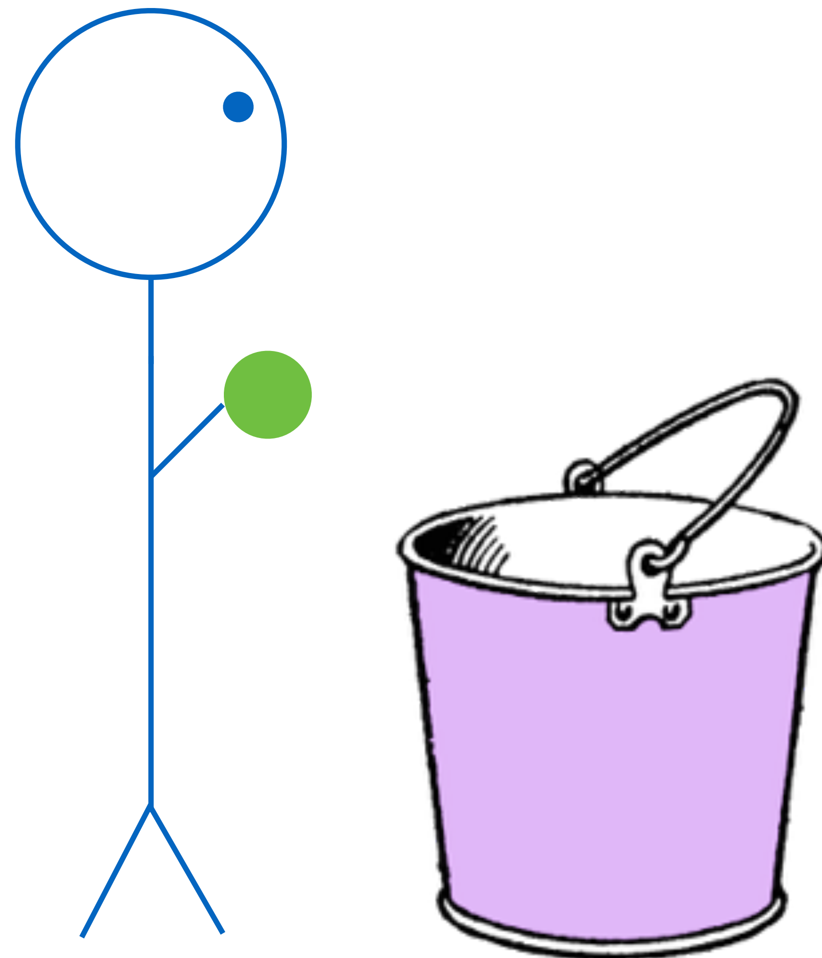
Like a bucket of balls




semaphore.wait()

# Semaphores

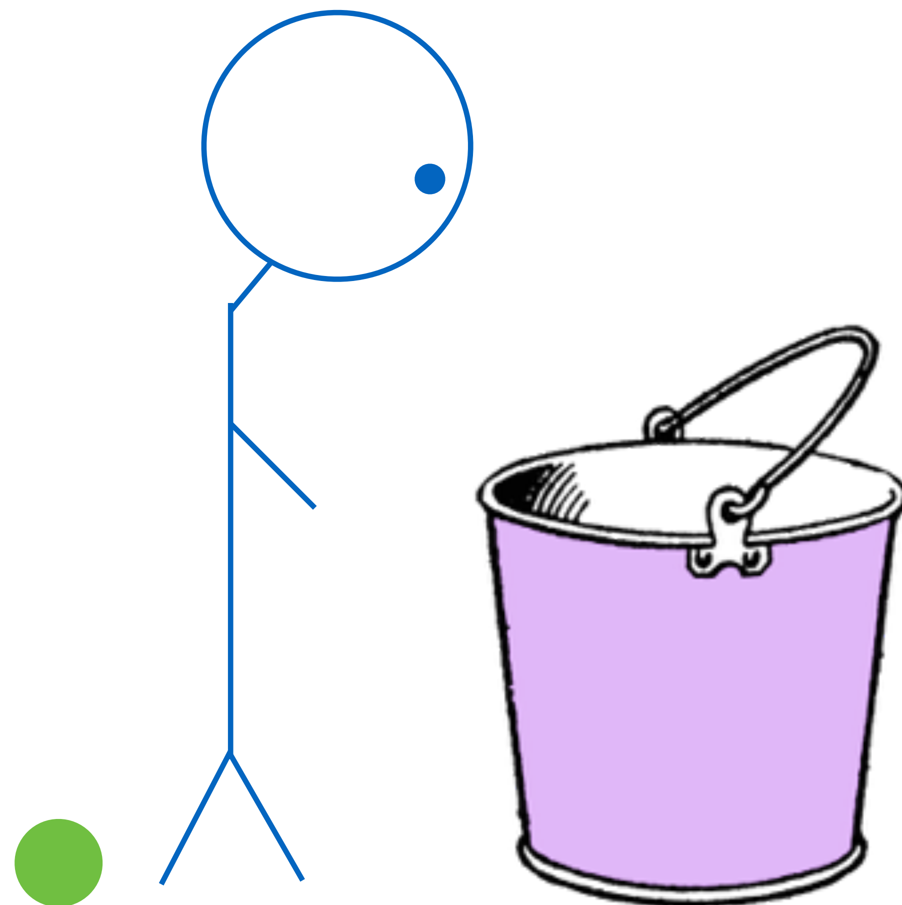
Like a bucket of balls




semaphore.wait()

# Semaphores

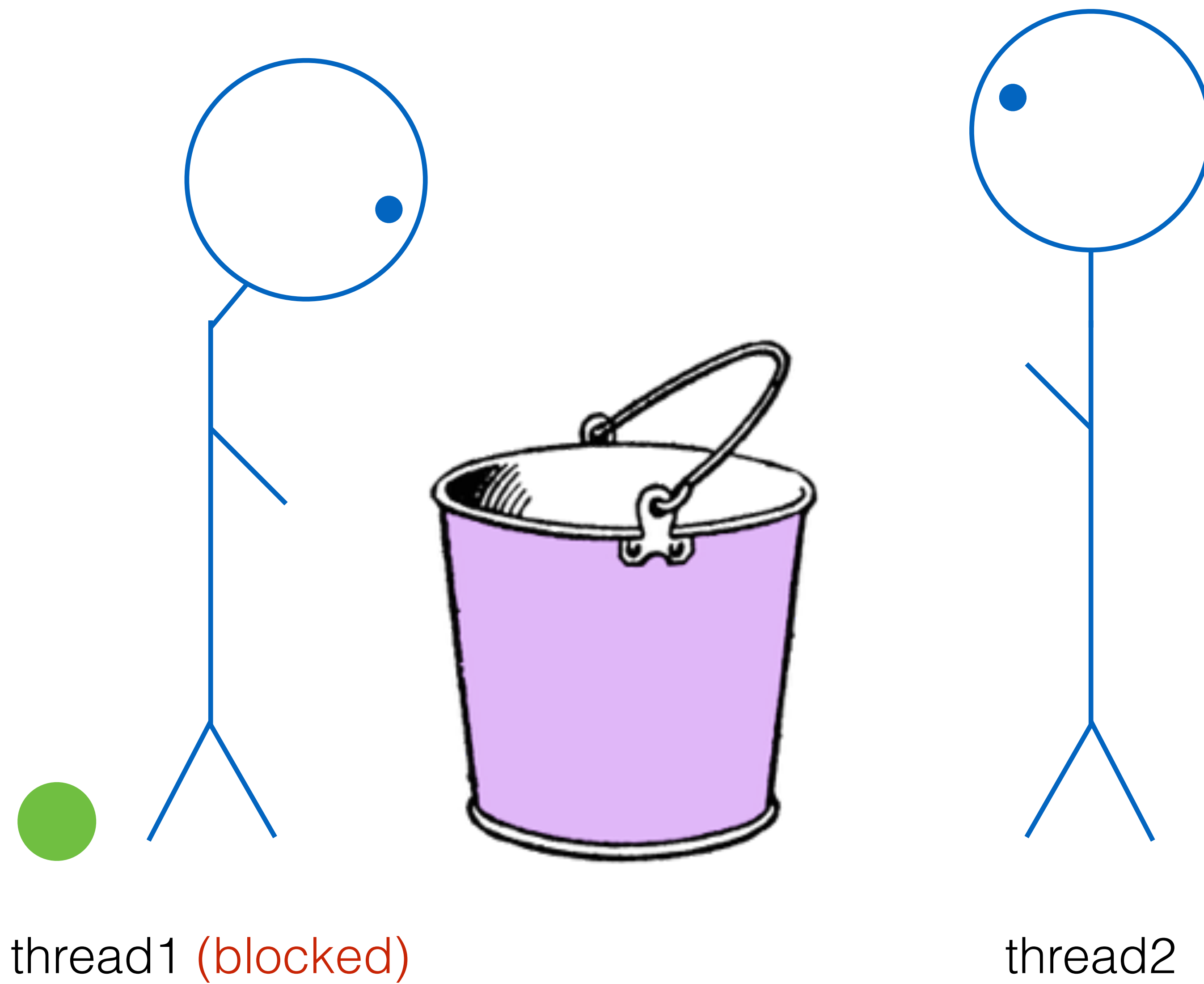
Like a bucket of balls




semaphore.wait() (again)

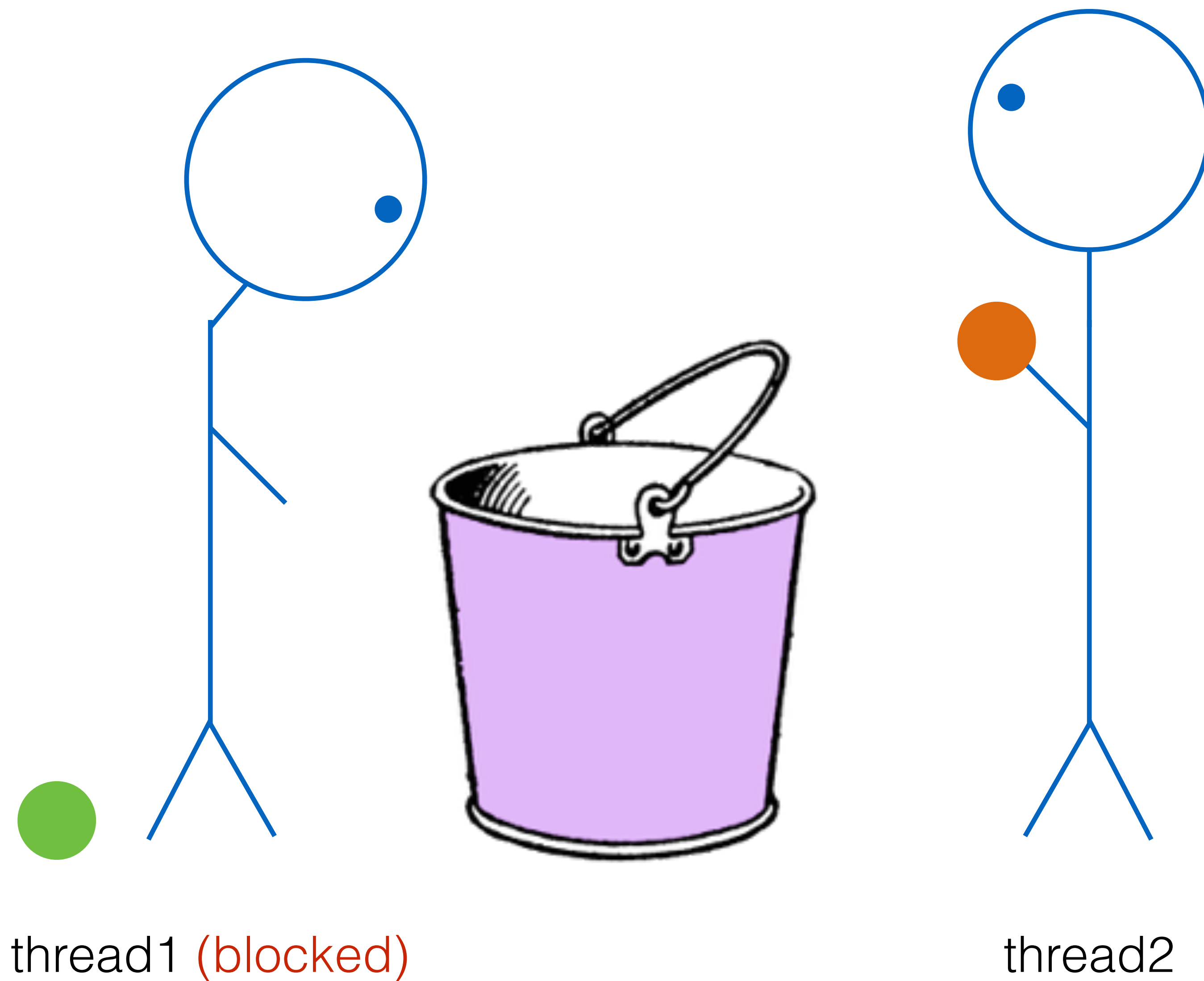
# Semaphores

Like a bucket of balls



# Semaphores

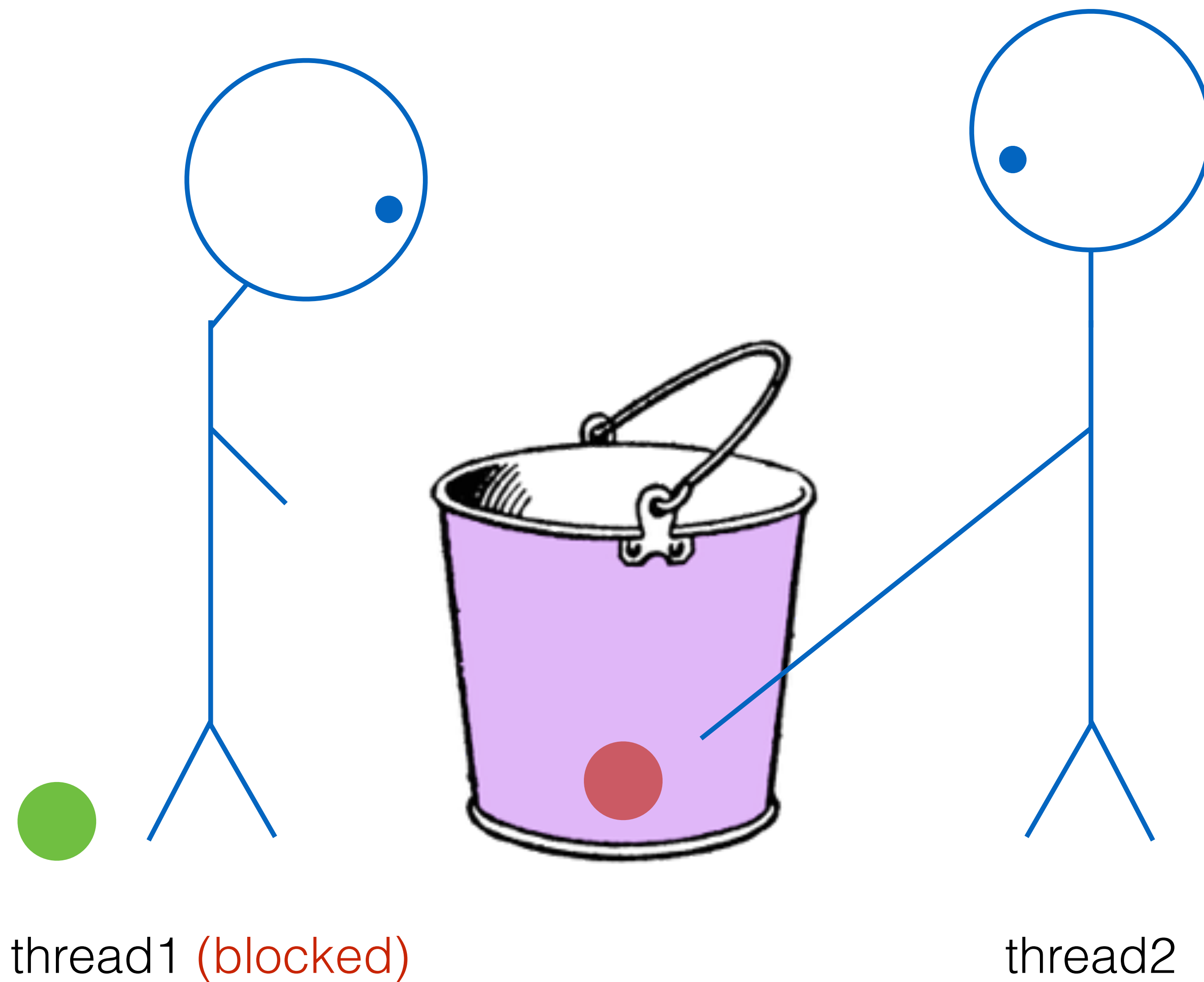
Like a bucket of balls




semaphore.signal()

# Semaphores

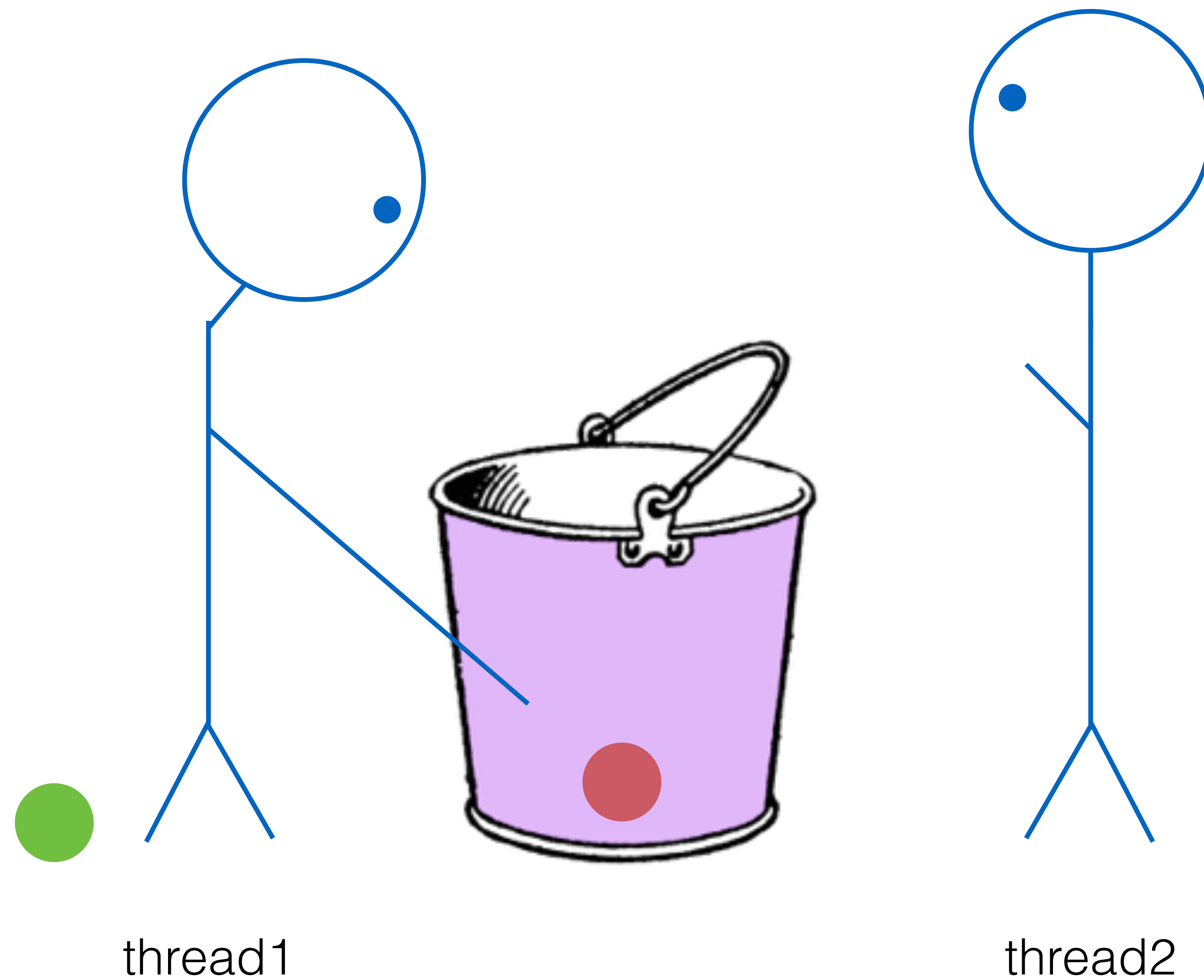
Like a bucket of balls




semaphore.signal()

# Semaphores

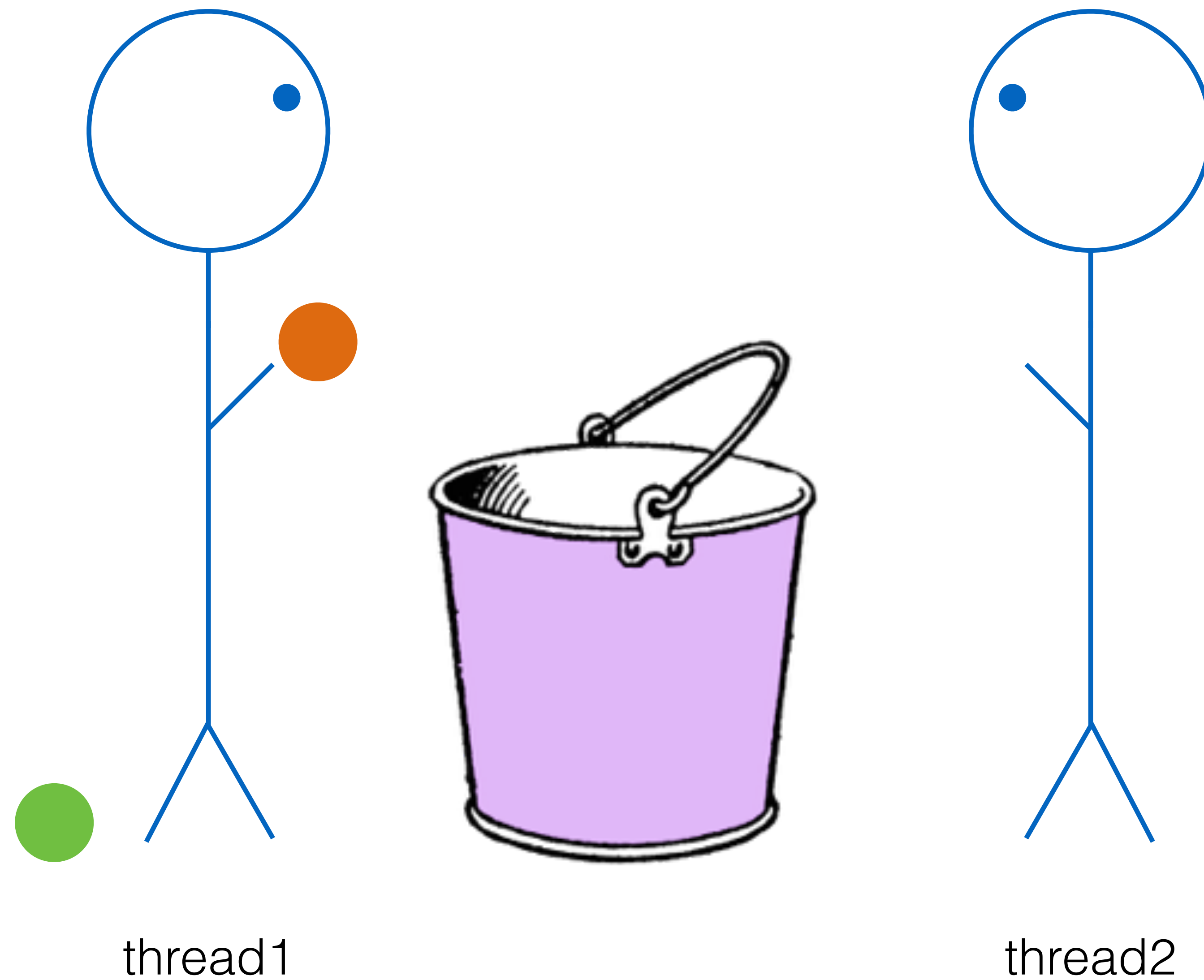
Like a bucket of balls




semaphore.signal()

# Semaphores

Like a bucket of balls




semaphore.signal()

# Semaphores

semaphore.signal():

- Adds a ball to the bucket
- **Never blocks**

semaphore.wait():

- If a ball is in the bucket, takes the ball and returns immediately
- If no ball is in the bucket, waits until one is available, then takes the ball and returns

# Primitive thread pool (workers.cc)

## Now with semaphores

```
size_t numQueued = 0;
mutex numQueuedLock;
conditional_variable_any queueCv;

static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue."
            << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        queueCv.notify_all();
        cout << oslock << "Scheduler: added to queue."
            << endl << osunlock;
    }
}
```

# Primitive thread pool (workers.cc)

## Now with semaphores

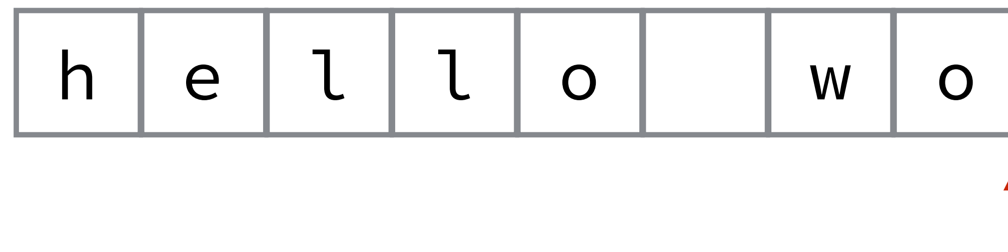
```
semaphore sem;



static void runWorker(size_t id) {
    while (true) {
        sem.wait();






        cout << oslock << "Worker #" << id << ": popped from queue."
            << endl << osunlock;

        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        sem.signal();


        cout << oslock << "Scheduler: added to queue."
            << endl << osunlock;
    }
}
```

# Semaphores and condition variables

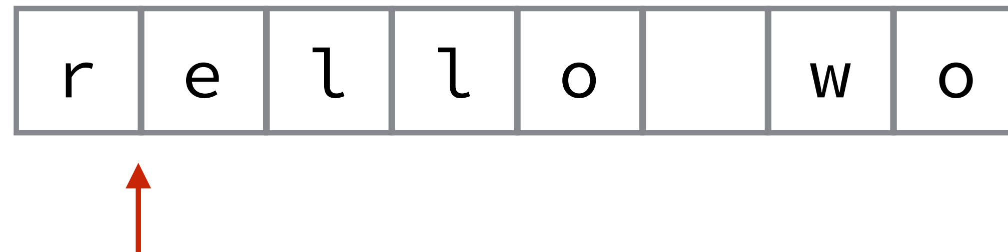
- Anything you can do with a semaphore, you can also do with a condition variable
- If you can build it using a semaphore, build it using a semaphore.

# Ring buffer with semaphores

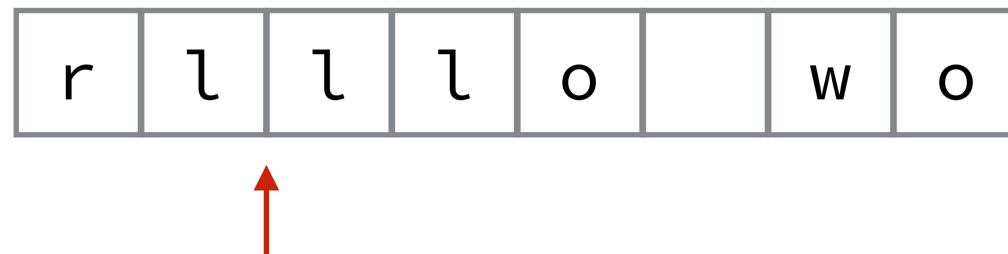
# Ring buffer with semaphores



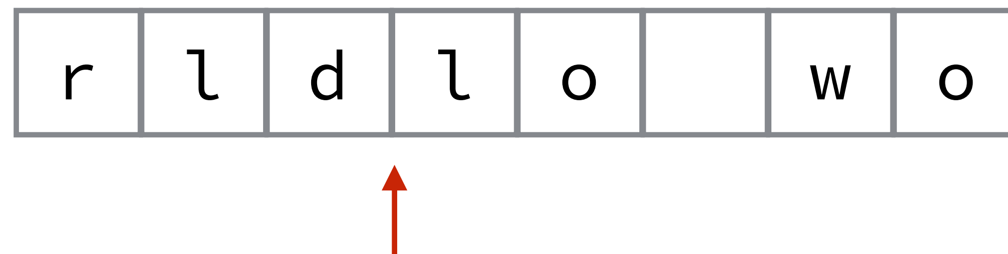
# Ring buffer with semaphores



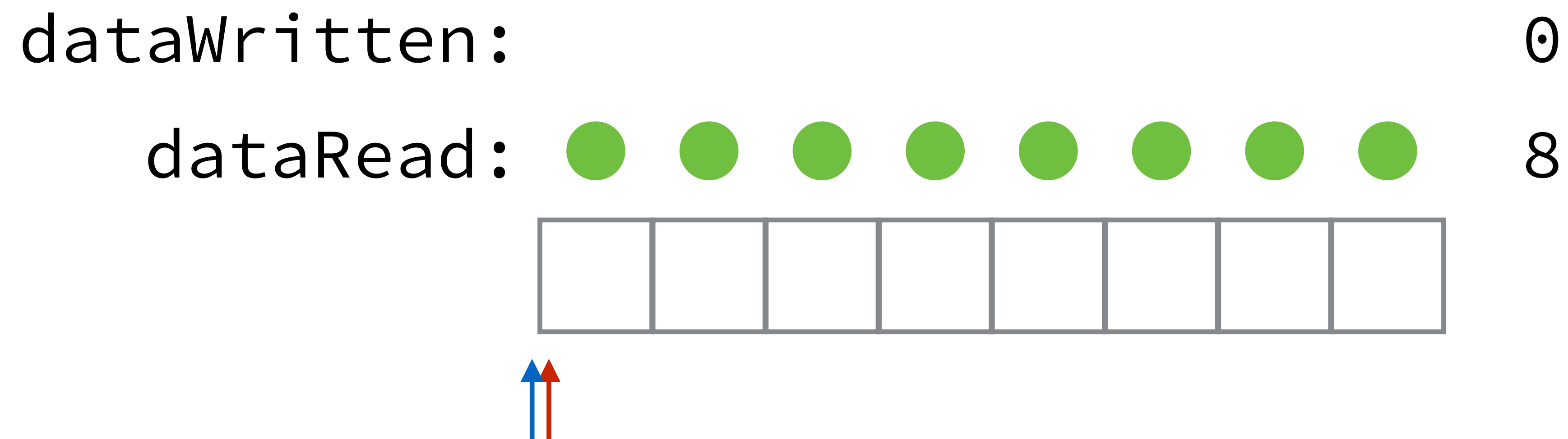
# Ring buffer with semaphores



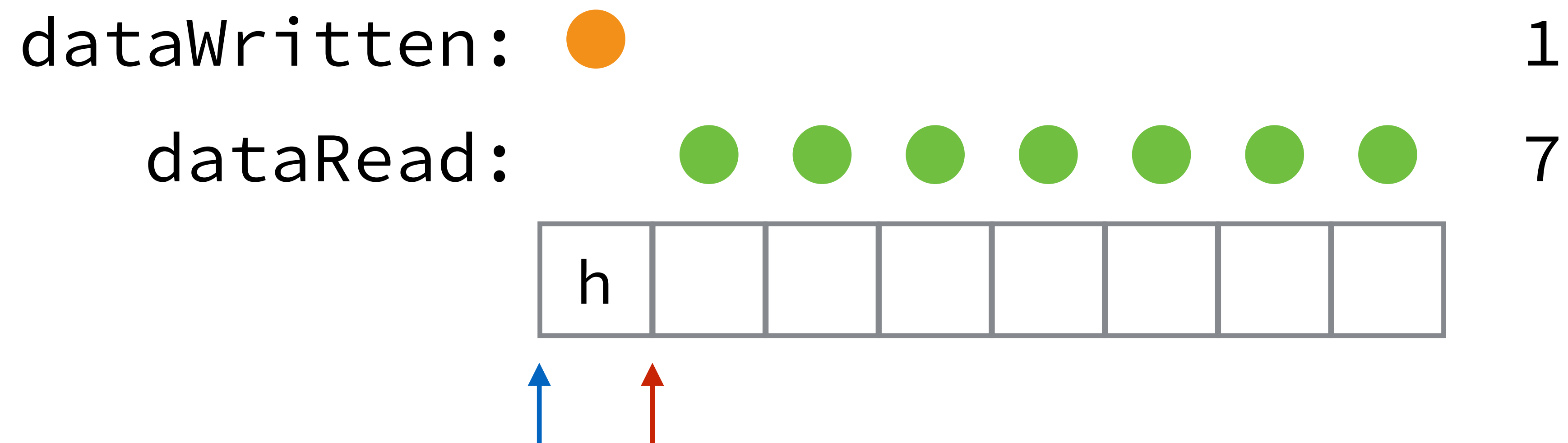
# Ring buffer with semaphores



# Ring buffer with semaphores

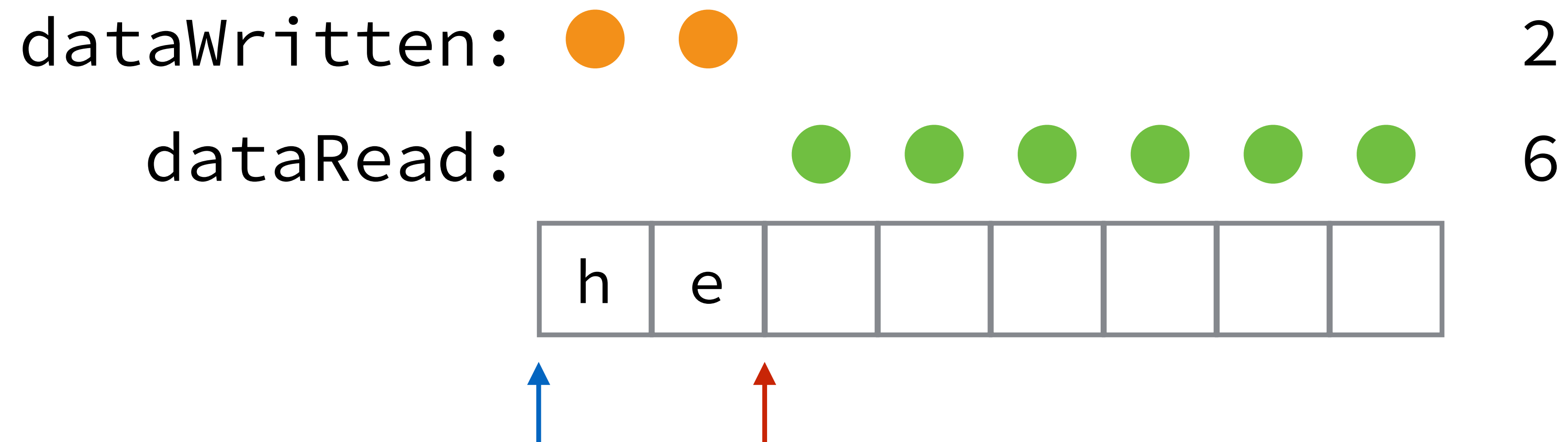


# Ring buffer with semaphores



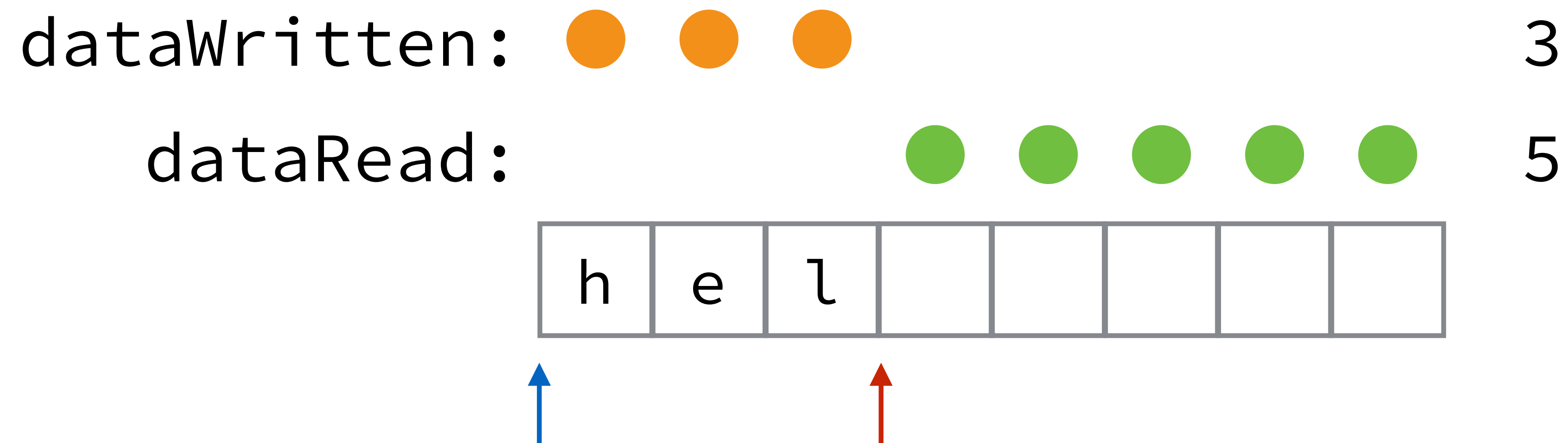

1. write h

# Ring buffer with semaphores




1. write h
2. write e

# Ring buffer with semaphores



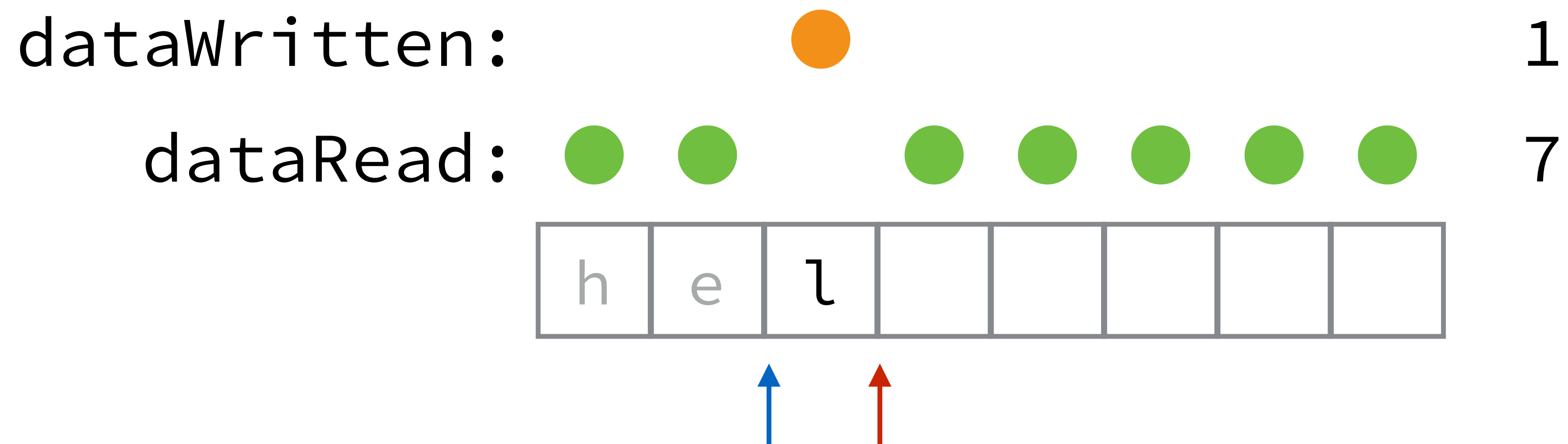

1. write h
2. write e
3. write l

# Ring buffer with semaphores



1. write h
2. write e
3. write l
4. read h

# Ring buffer with semaphores




1. write h
2. write e
3. write l
4. read h
5. read e

# Ring buffer with semaphores

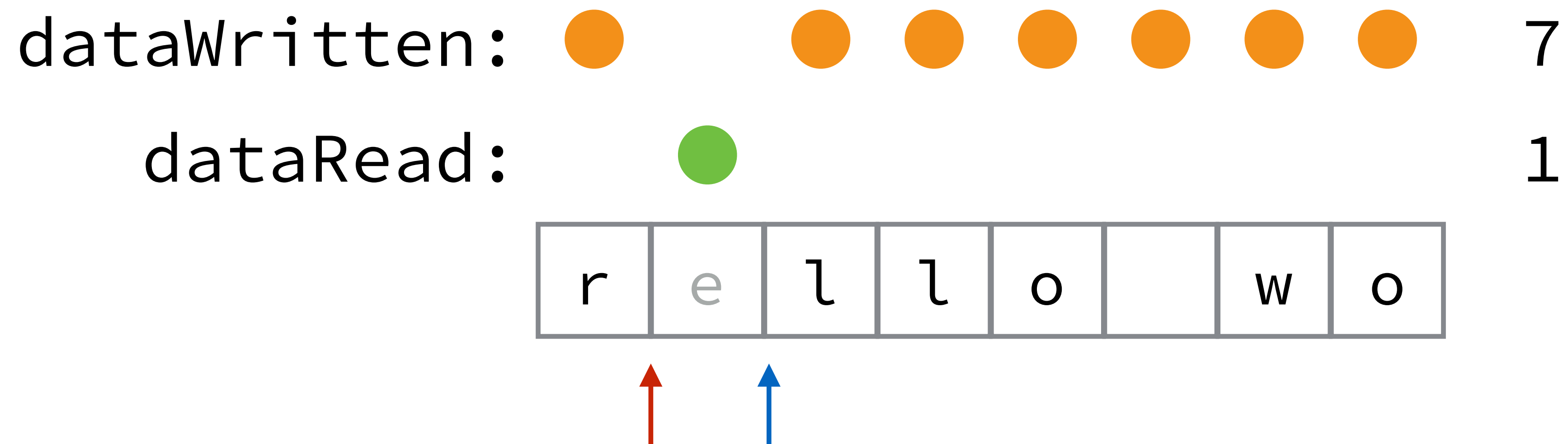
dataWritten: 6

dataRead: 2

| h | e | l | l | o |  | w | o |
|---|---|---|---|---|---|---|---|

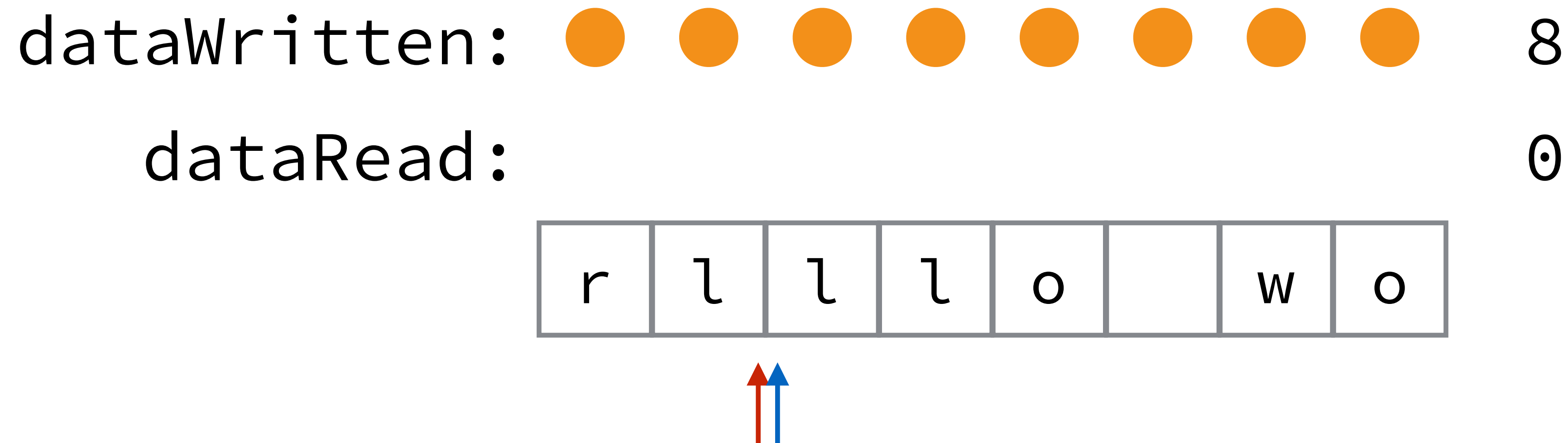
1. write h
2. write e
3. write l
4. read h
5. read e
6. write l
7. write o
8. write _
9. write w
10. write o

# Ring buffer with semaphores




1. write h
2. write e
3. write l
4. read h
5. read e
6. write l
7. write o
8. write _
9. write w
10. write o
11. write r

# Ring buffer with semaphores

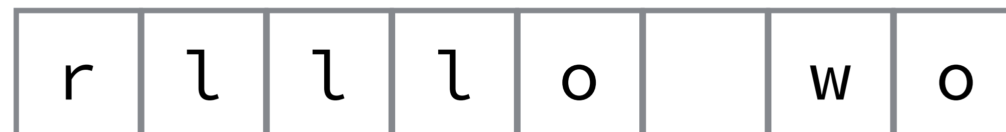



1. write h
2. write e
3. write l
4. read h
5. read e
6. write l
7. write o
8. write _
9. write w
10. write o
11. write r

# Ring buffer with semaphores

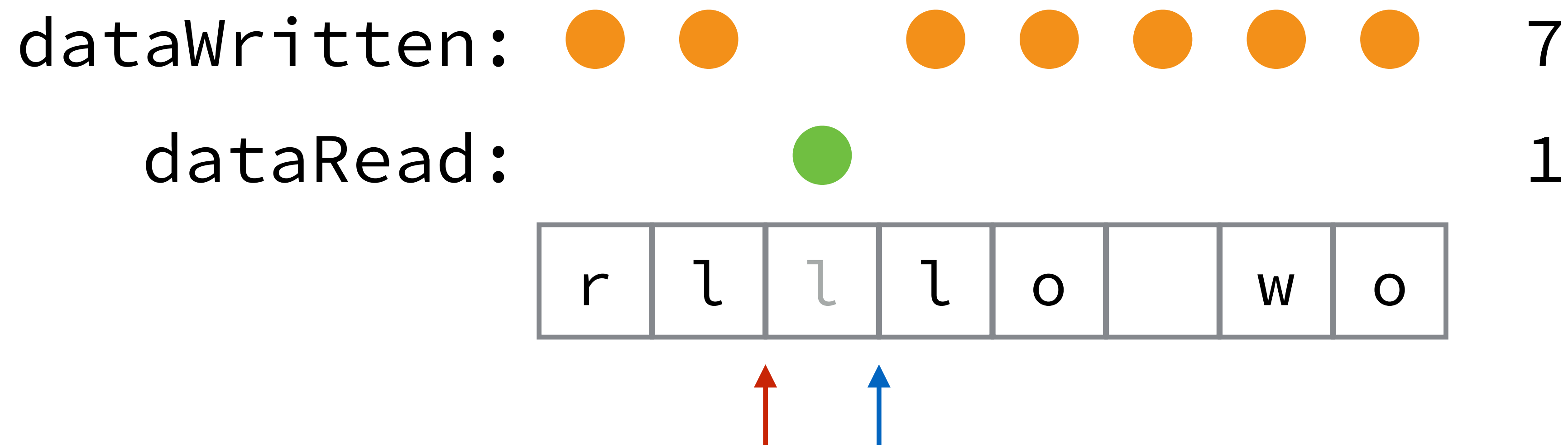
At this point, reader could read 8 bytes. writer has to wait for dataRead to be signaled (i.e. has to wait for the reader to read stuff before it overwrites more characters in the buffer)

```
dataWritten: ● ● ● ● ● ● ● ●   8
   dataRead:                   0
```




```
1. write h      7. write o
2. write e      8. write _
3. write l      9. write w
4. read   h     10.write o
5. read   e     11.write r
6. write l
```

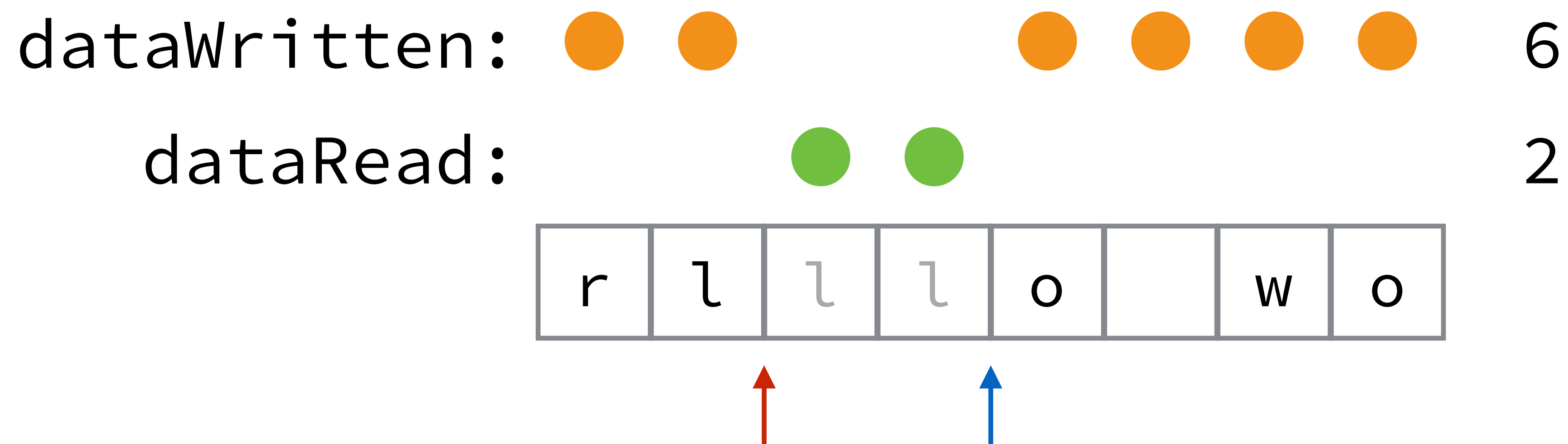
# Ring buffer with semaphores




1. write h
2. write e
3. write l
4. read h
5. read e
6. write l
7. write o
8. write _
9. write w
10. write o
11. write r
12. read l

# Ring buffer with semaphores

dataRead could be wait()ed twice, so the writer thread can write 2 more characters now

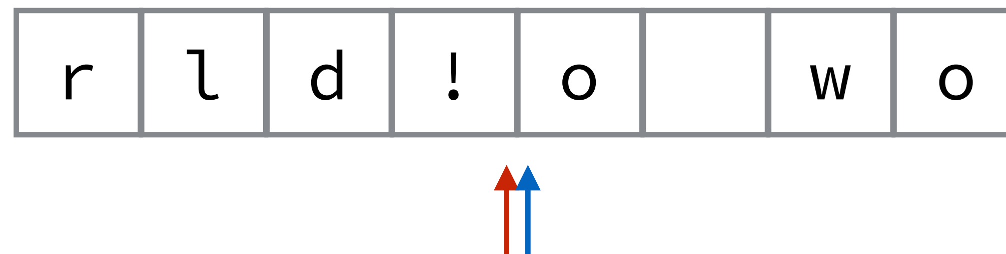



1. write h
2. write e
3. write l
4. read h
5. read e
6. write l
7. write o
8. write _
9. write w
10. write o
11. write r
12. read l
13. read l

# Ring buffer with semaphores

dataWritten: ● ● ● ● ● ● ● ● 8

dataRead: 0




1. write h
2. write e
3. write l
4. read h
5. read e
6. write l
7. write o
8. write _
9. write w
10. write o
11. write r
12. read l
13. read l
14. write d
15. write !