15-213 Recitation: Multithreaded Synchronization

Jack Biggs, Section B

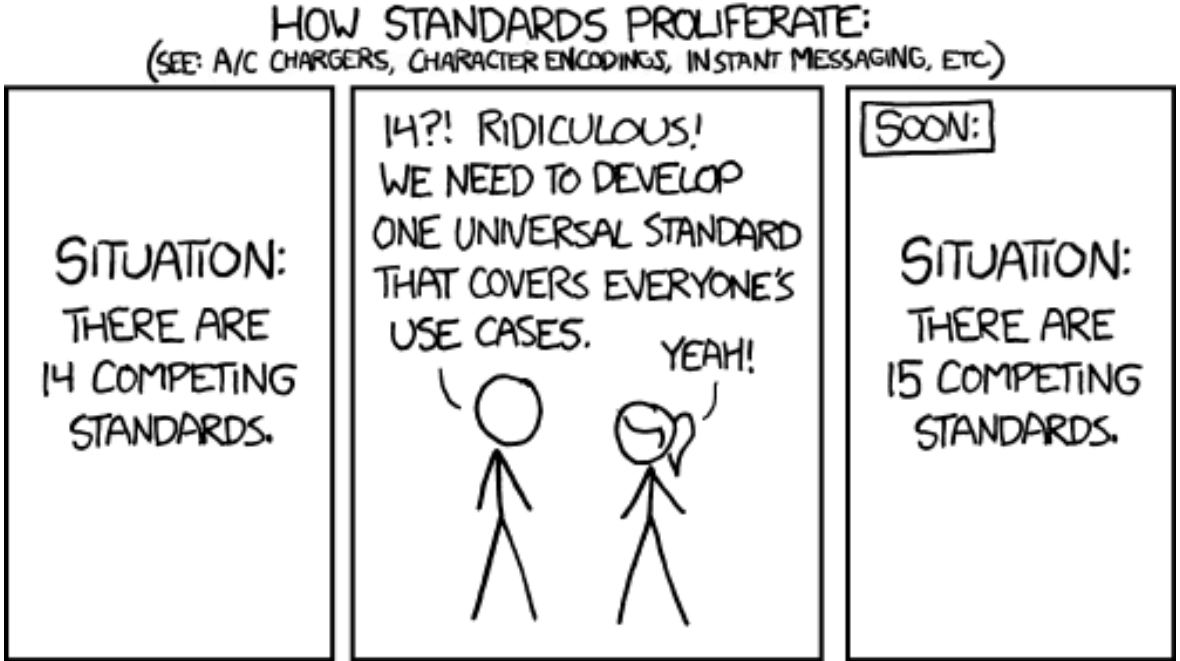
24 Nov 2014

Agenda

- Proxy Lab
  - Basic server code examples
  - Debugging tools
- Concurrency
  - The Good, The Bad, and The Ugly
  - Shared Memory: Synchronization
    - Critical Sections and Locking
  - Common bugs

Proxy Lab

- Due next Thursday
- Make it robust to unexpected hiccups in input
  - The Internet is standardized, but not really



The Echo Server - Iterative

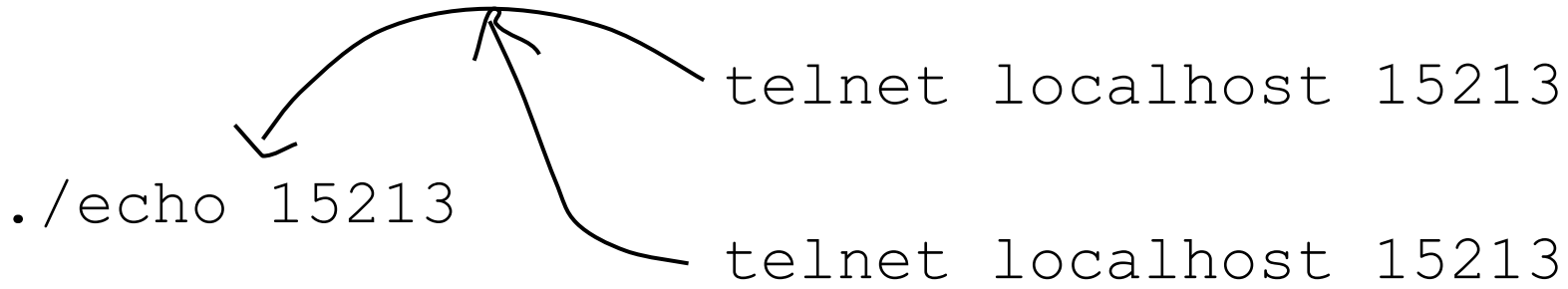
```
void echo(int connfd) {
    size_t n; char buf[MAXLINE]; rio_t rio;
    // initialize robust io for reading on file descriptor
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        // read to buffer, and write it back
        Rio_writen(connfd, buf, n);
    }
}
```

The Echo Server - Iterative

```
int main(int argc, char **argv) {
    int listenfd, connfd;
    struct sockaddr_storage clientaddr; socklen_t clientlen;
    char client_hostname[MAXLINE]; client_port[MAXLINE];
    listenfd = Open_listenfd(argv[1]);
    while(1) { // Handle requests one at a time. I hope I'm not popular!
        clientlen = sizeof(struct sockaddr_storage); // Important!
        connfd = Accept(listenfd, (SA*)&clientaddr, &clientlen);
        Getnameinfo((SA*)&clientaddr, clientlen, client_hostname,
                    MAXLINE, client_port, MAXLINE, 0);
        echo(connfd);
        Close(connfd);
    }
    assert(0);
}
```

The Echo Server: Finding Its Weakness

Using `telnet`, we can disrupt this iterative service.



The second client cannot connect, because echo has not yet closed its connection with the first client.

More Advanced Debugging

- Telnet requires you to type everything yourself
- Web protocols (like HTTP) can be tedious to type
- Use `curl` to form requests for you

```
curl --proxy http://localhost:port url.com
```

- Redirect output using >, for non-text files

How Threads Work: `pthread_create`

- Threads are within the same process
  - Interleave and parallelize similarly to processes
- Threads created with `pthread_create`:

```
int pthread_create(pthread_t *thread,
   const pthread_attr_t *attr,
   void *(*start_routine)(void *), void *arg);
```

Working Together: When to use Threads

Let's sum up the elements in an array.

The boring way:

```
int sum = 0;
for (int i = 0; i < n; i++)
  sum += nums[i];
```

Sums: The Fun Way

```
void *thread_fun(void *vargp) {
   int myid = *((int *)vargp);
   size_t start = myid * nelems_per_thread;
   size_t end = start + nelems_per_thread;
   size_t i;
  size_t index = myid*spacing;
   data_t sum = 0;
   for (i = start; i < end; i++) // sum our section
     sum += i;
   psum[index] = sum;
   return NULL;
}
```

Sums: The Fun Way

```
nelems_per_thread = nelems / nthreads;

// Create threads and wait for them to finish
for (i = 0; i < nthreads; i++) {
    myid[i] = i;
    Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
}

for (i = 0; i < nthreads; i++)
    Pthread_join(tid[i], NULL);
```

Sums: The Fun Way

```
result = 0;
// Add up the partial sums computed by each thread
for (i = 0; i < nthreads; i++)
    result += psum[i*spacing];
// Add leftover elements
for (e = nthreads * nelems_per_thread; e < nelems; e++)
    result += e;

return result;
```

Advantages & Disadvantages

Good:

- We can (potentially) make it faster
- We can exploit better use of the cache

Bad:

- **Hard** to write!
- Shared resources difficult to manage

Here, we avoid *mutual exclusion* by going to different sections of the array between threads, but we can't always do this.

Critical Sections and Shared Variables

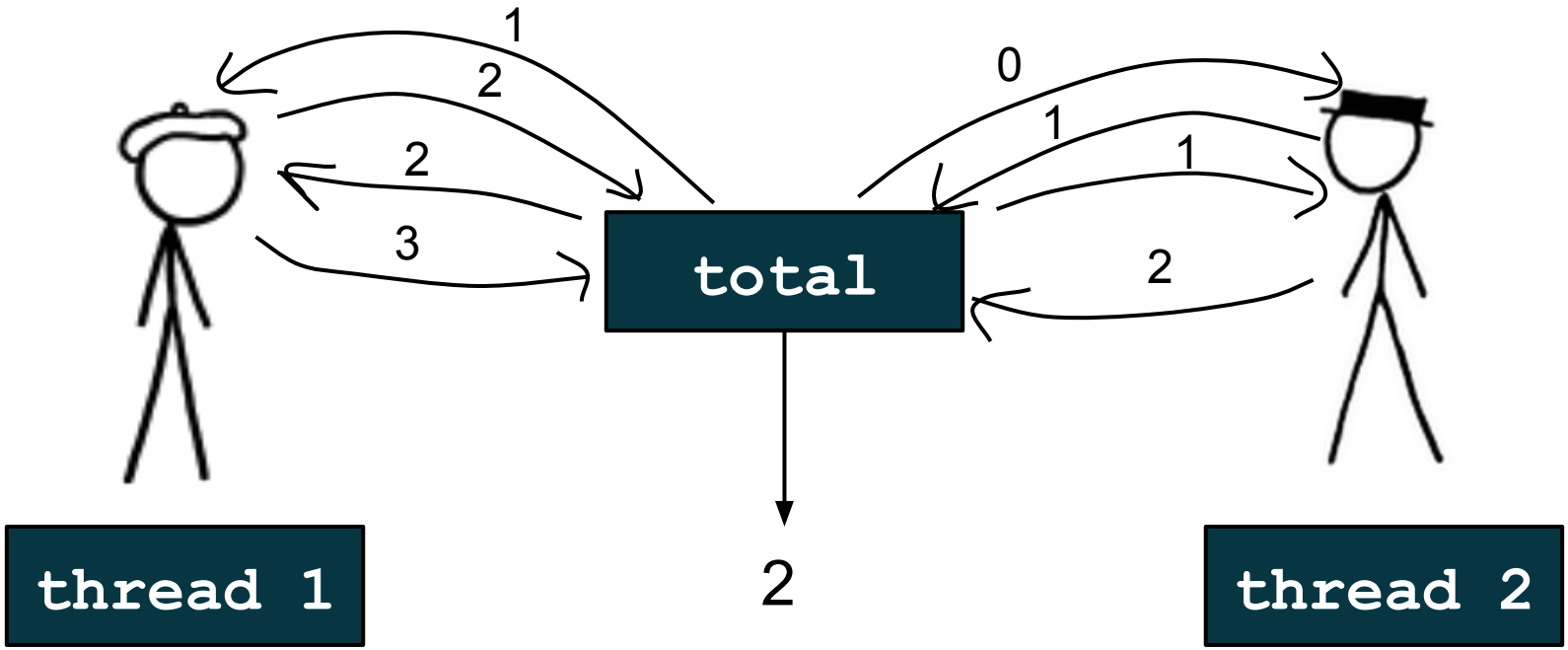
Let's try some more counting with threads.

```
volatile int total = 0;
void incr(void *ptr) {
   pthread_detach(pthread_self());
   for (int i = 0; i < *ptr; i++)
       total++;
}
```

Critical Sections and Shared Variables

```
#define NTHREADS 2
#define NINCR 100
int main() {
   pthread_t tids[NTHREADS];
   int y = NINCR;
   for (int i = 0; i < NTHREADS; i++)
          pthread_create(&tids[i], NULL, incr, &y);
   // output will range between NTHREADS-y*NTHREADS
   printf("total is: %d", total);
}
```

What happens



Mutexes

Solution: **Lock** execution of thread until resource “acquired”

```
volatile int total = 0;
pthread_mutex_t M;

void incr(void *ptr) {
   pthread_detach(pthread_self());
   for (int i = 0; i < *ptr; i++) {
          pthread_mutex_lock(&M);
       total++; // CRITICAL SECTION
       pthread_mutex_unlock(&M);
   }
}
```

Mutexes

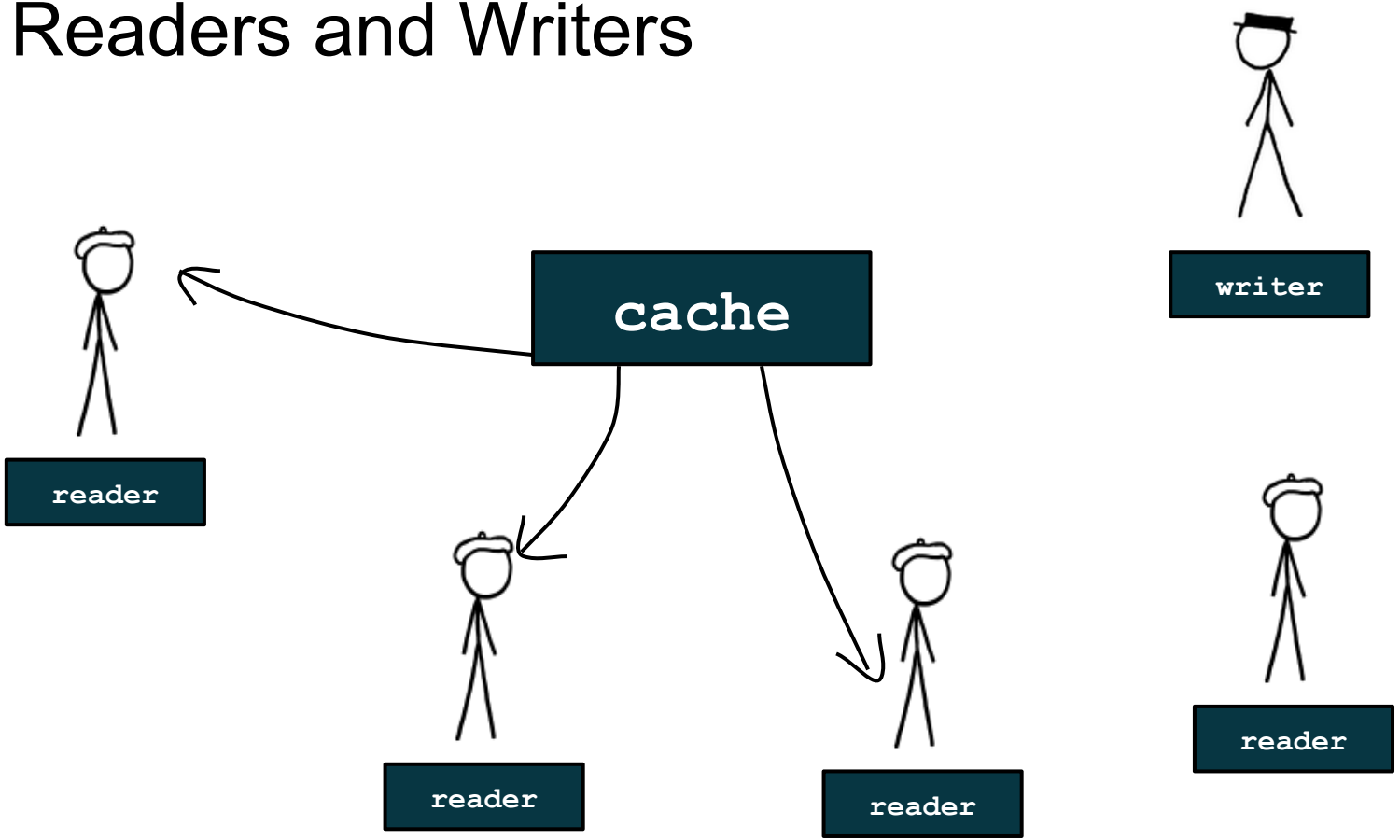
Remember to initialize the mutex first!

```
#define NTHREADS 2
#define NINCR 100
volatile int total = 0;
pthread_mutex_t M;
...
int main() {
   ...
   pthread_mutex_init(&M);
   ...
}
```

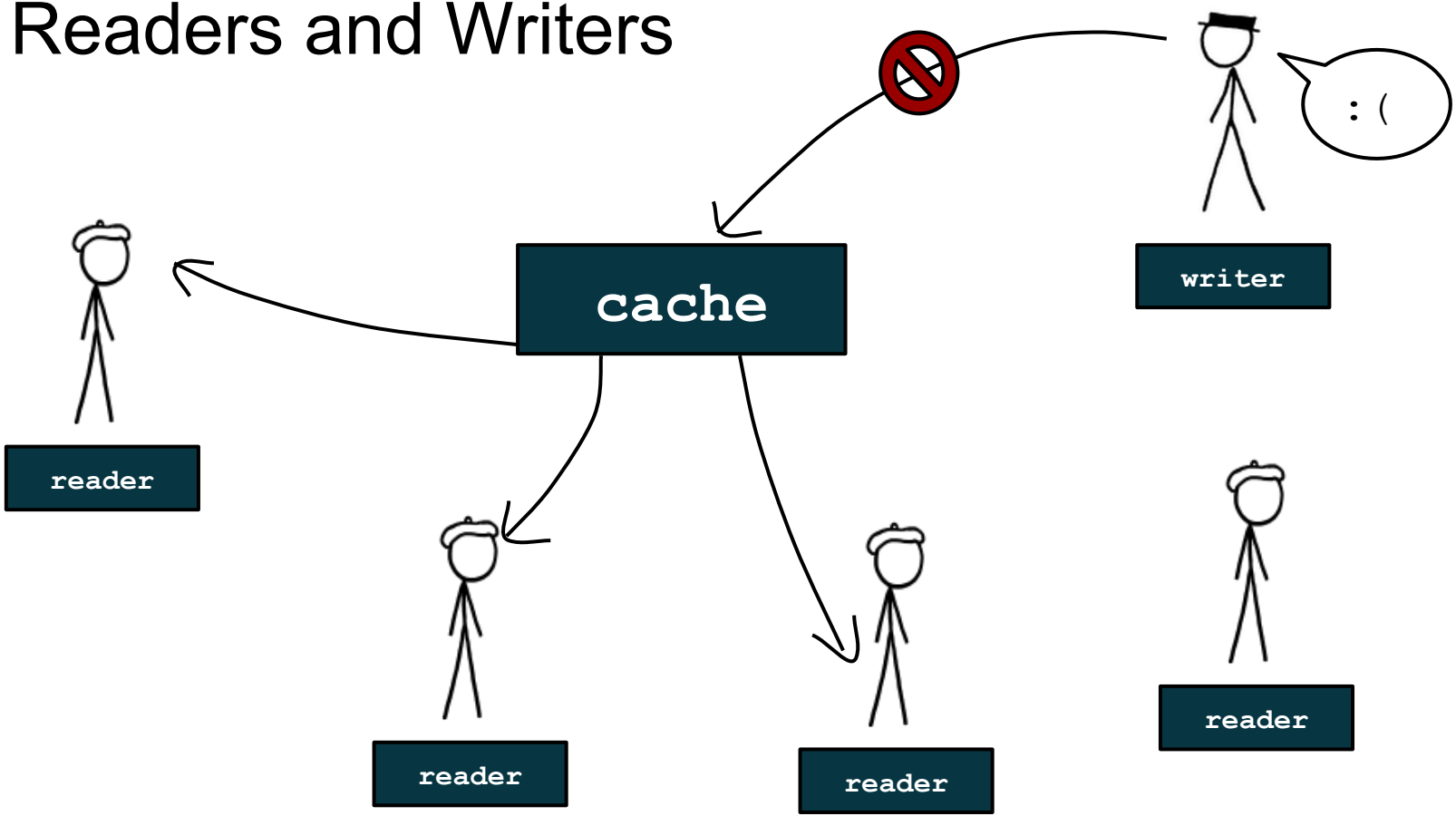
Problem solved… right?

- Locks in threads are *slow*
  - This is a **terrible** way to sum up to 200
  - Avoid shared memory if you can
- This is one of the simpler models for thread sync.
  - Reading vs. Writing
  - Producers and Consumers

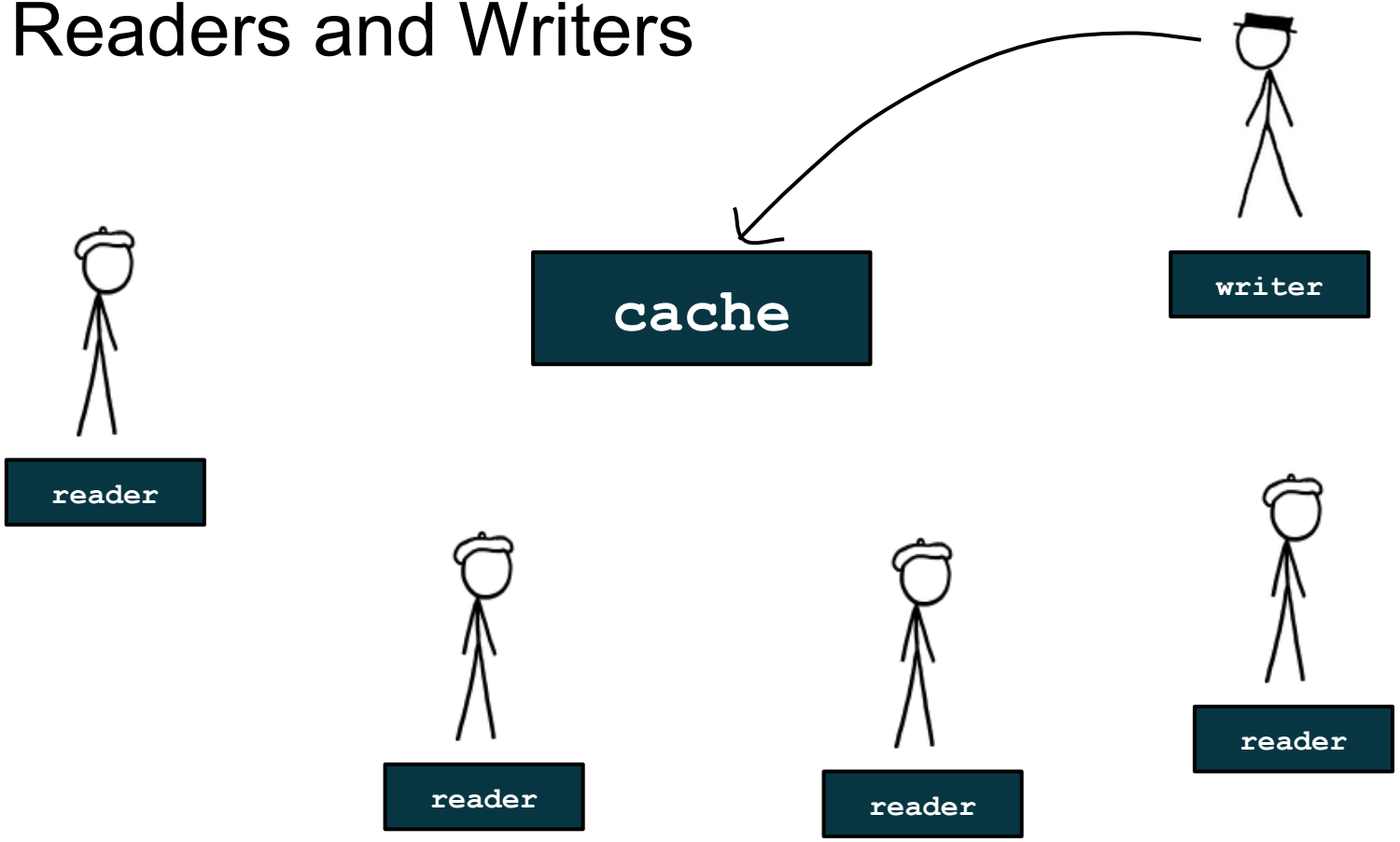
Readers and Writers



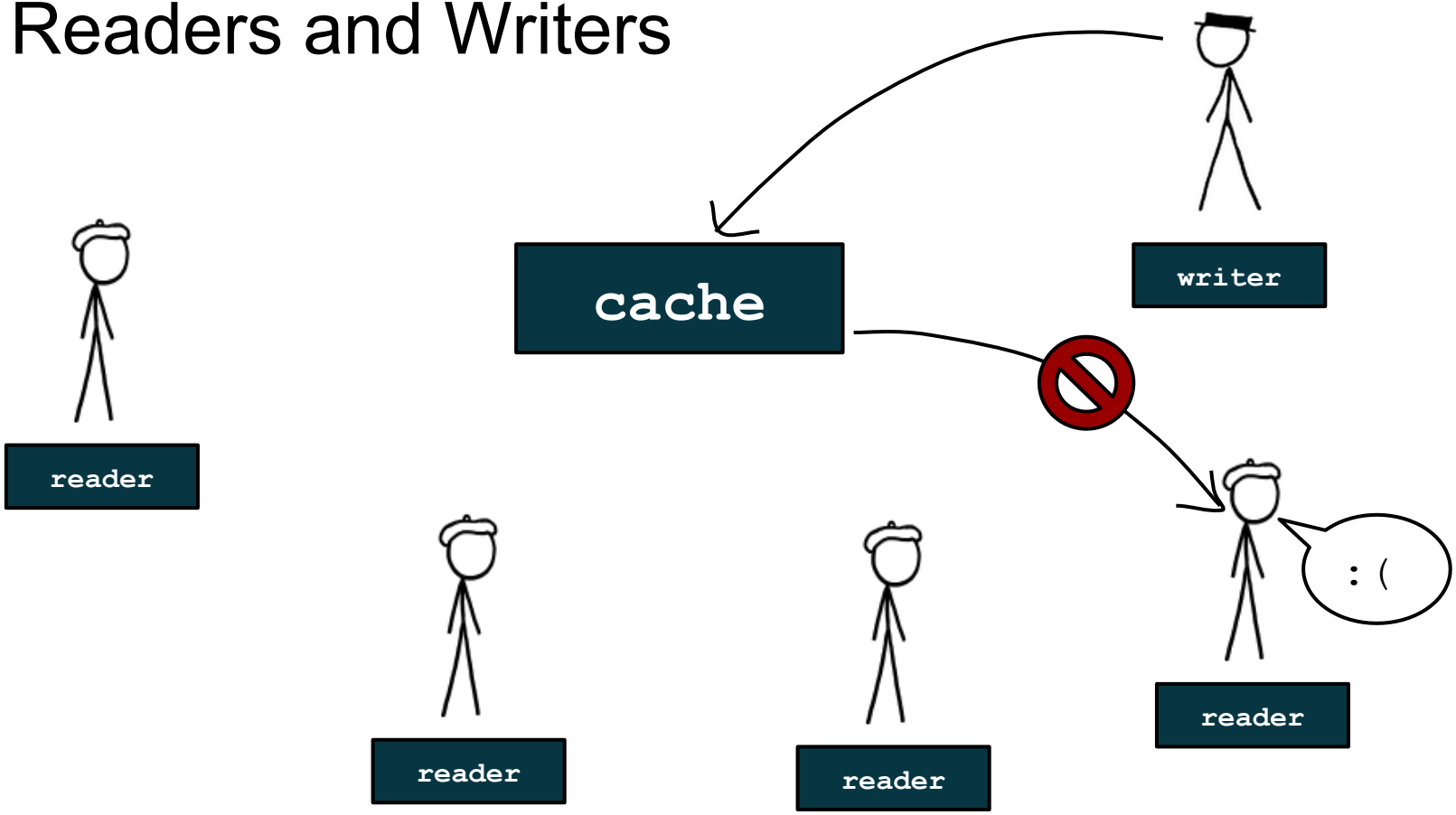
Readers and Writers



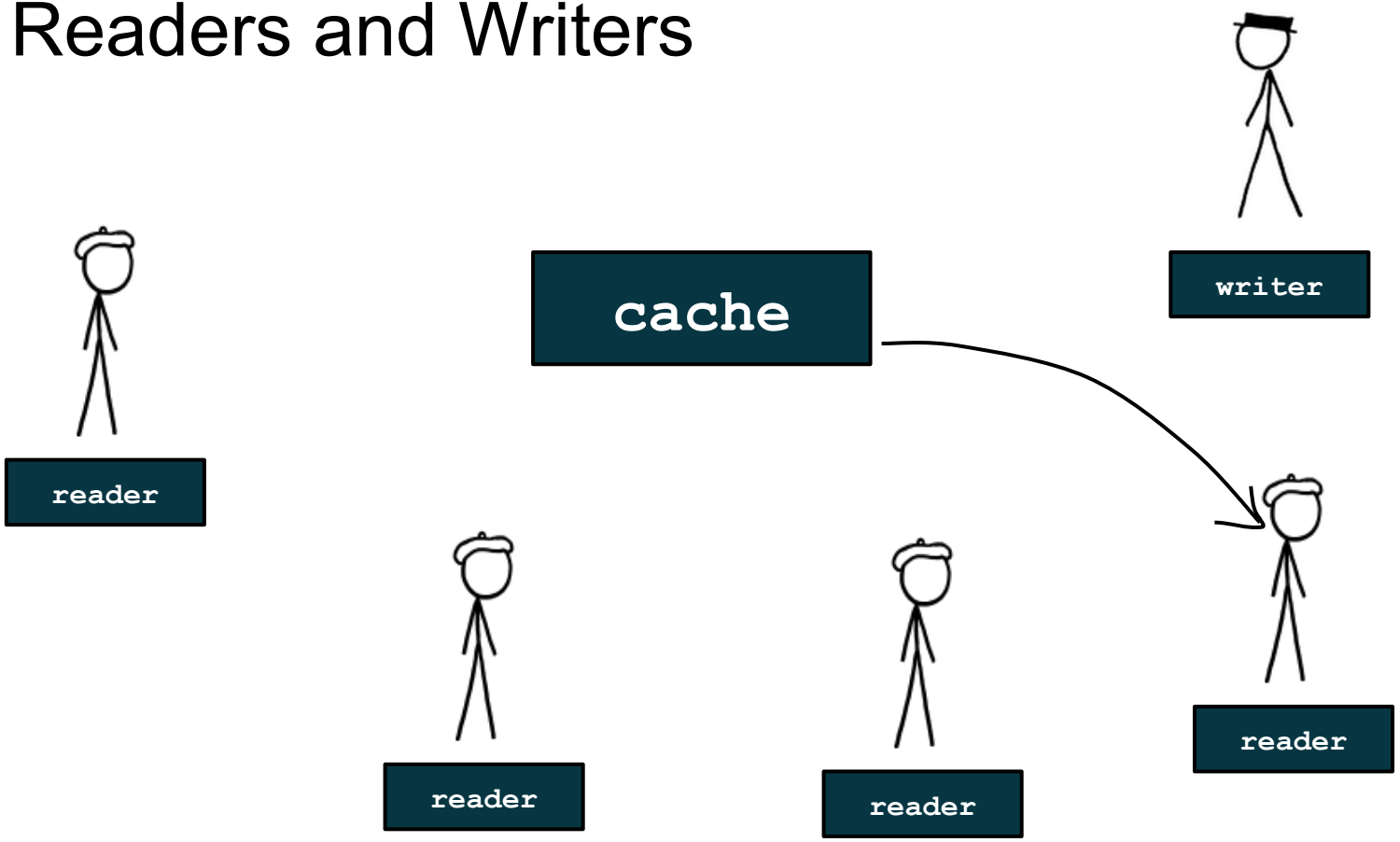
Readers and Writers



Readers and Writers



Readers and Writers



Readers / Writers done for you!

The `pthread` library solves this for you with `rwlocks`:

```
pthread_rwlock_t;
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
  const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Concurrency and Starvation

- In previous example, readers block one writer
  - Writer might not get the resource
  - Writer is being **starved** of resource
- Make sure that readers don’t hold resource for long

Producers / Consumers: Semaphores

- Buffer objects between threads
- One (or more) thread(s) **produce** and **consume** resources between each other.
- Semaphores: A mutually-exclusive counter.

```
sem_t semaphore;
// value = starting value of semaphore
int sem_init(sem_t *sem, int pshared, unsigned int value);
void P(sem_t *sem); // decrement
void V(sem_t *sem); // increment
```

The Producer

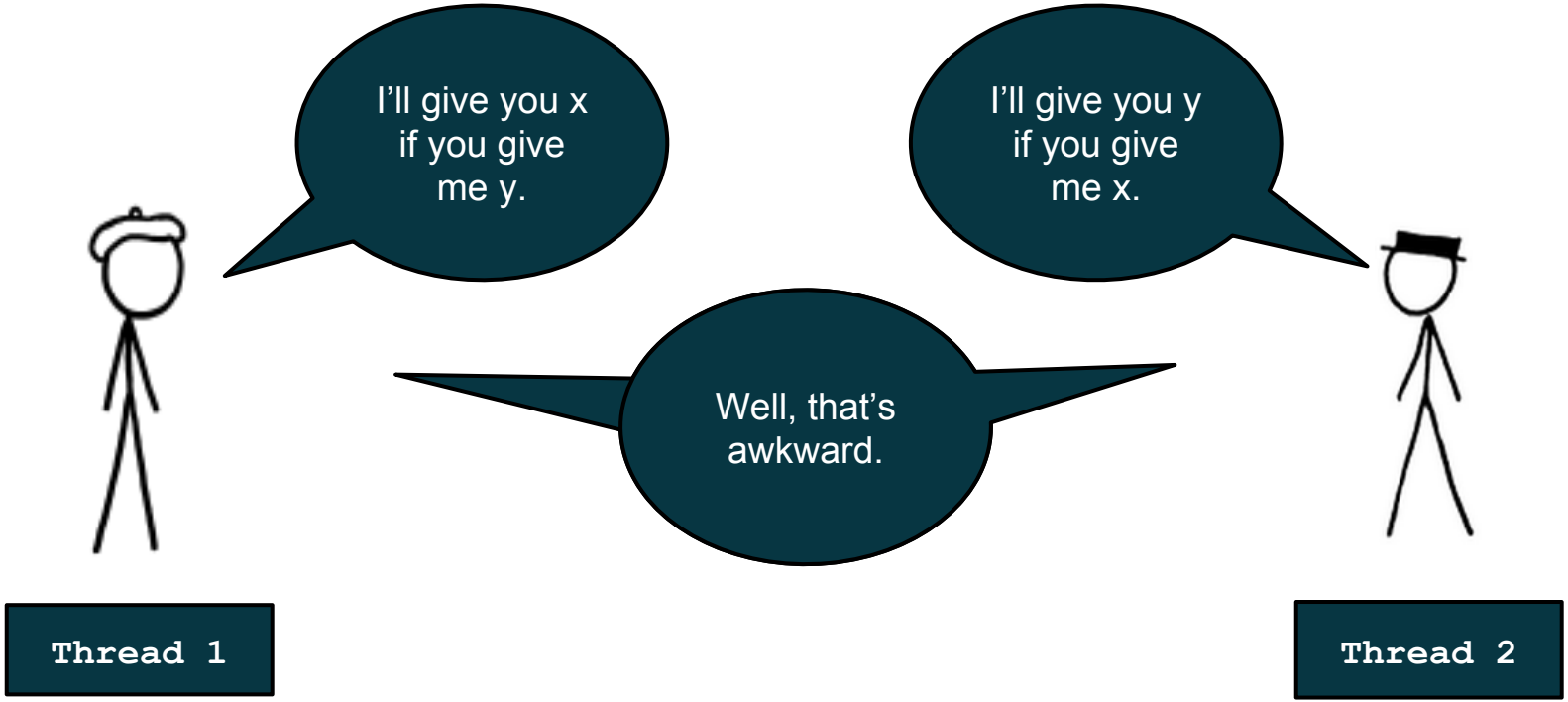
```
void init() {
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFMAX);
    sem_init(&full, 0, 0);
}
void producer() {
    while(1) {
        int item = produce();
        P(empty); // wait for more space
        P(mutex); // store the item
        buffer[in] = item;
        in = (in + 1) % BUFMAX;
        count++;
        V(mutex);
        V(full); // report new empty slot
    }
}
```

The Consumer

```
while(1) {
   P(full); // wait for something to be stored
   P(mutex);
   item = buffer[out];
   out++;
   count = (count - 1) % BUFMAX;
   V(mutex);

   V(empty); // let em know we have an open slot
   consume(item); // nom nom nom
}
```

Problem: **Deadlock**



Problem: **Deadlock**

- Entire program will hang
- Pay attention to how and where you lock/unlock
- Program may or may not hang predictably
  - Thread scheduling is non-deterministic
  - Similar to race conditions, usually worse
- Critical section should be as **small** as possible

Problem: **Livelock**

- Similar to Deadlock
  - Two programs feed back on one another
  - Spins indefinitely instead of hanging
- Two people trying to get past each other in a hallway
  - Both move the same direction simultaneously
  - Both do an awkward dance from side to side
  - Dance continues indefinitely
- Often happens when threads attempt to compensate for deadlock