15/18-213 Recitation: Multithreaded Synchronization

Isaac Manjarres

Recitation 14: 30 Nov 2015

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 Tuesday on December 8th, 2015
  - You may use a MAX of two late days
- Make it robust to unexpected hiccups in input
  - The Internet is standardized, but not really

The Echo Server – Sequential Handling

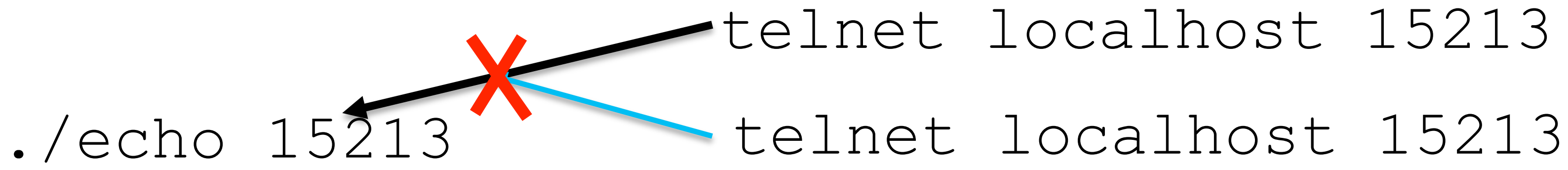
```
void echo(int connfd) {
    size_t n; char buf[MAXLINE]; rio_t rio;
    // initialize robust io for reading on file descriptor
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        // read to buffer,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

The Echo Server – Sequential Handling

```
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 observe a weakpoint within this iterative approach.



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
- Don't forget that binary data is not the same as text data
  - Be careful when using functions that are meant to operate only on strings. They will not work properly with binary data

How Threads Work: Nuances

- Threads run within the same process context
  - Arbitrary interleaving and parallelization similar to processes from Shell Lab
  - But keep in mind they are separate ***logical flows, not separate processes***
  - This implies that there are still context switches, much like with processes, but they are of less duration since threads have less context to store away than processes

Critical Points for Threads

- Threads have their own:
  - Thread ID
  - Stack(contained within the stack area for that process)
  - Stack Pointer
  - Instruction Pointer
  - General Purpose Registers
  - Status Codes
- Threads share:
  - Code sections
  - Heap
  - Open files

Anatomy of pthread_create

- Threads created with `pthread_create`:

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

Pointer to a variable that will hold the new thread's ID (threadID)

NULL for this course (attr)

Pointer to a function that takes in a void pointer, and returns a void pointer. This function is what the new thread will execute. (start_routine)

Pointer to an argument for the function that the thread will execute. Can pass Multiple arguments by putting them in a struct and passing a pointer to the struct (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 provide *mutual exclusion* by going to different sections of the array between threads, but we can't always do this.

Critical Sections and Shared Variables

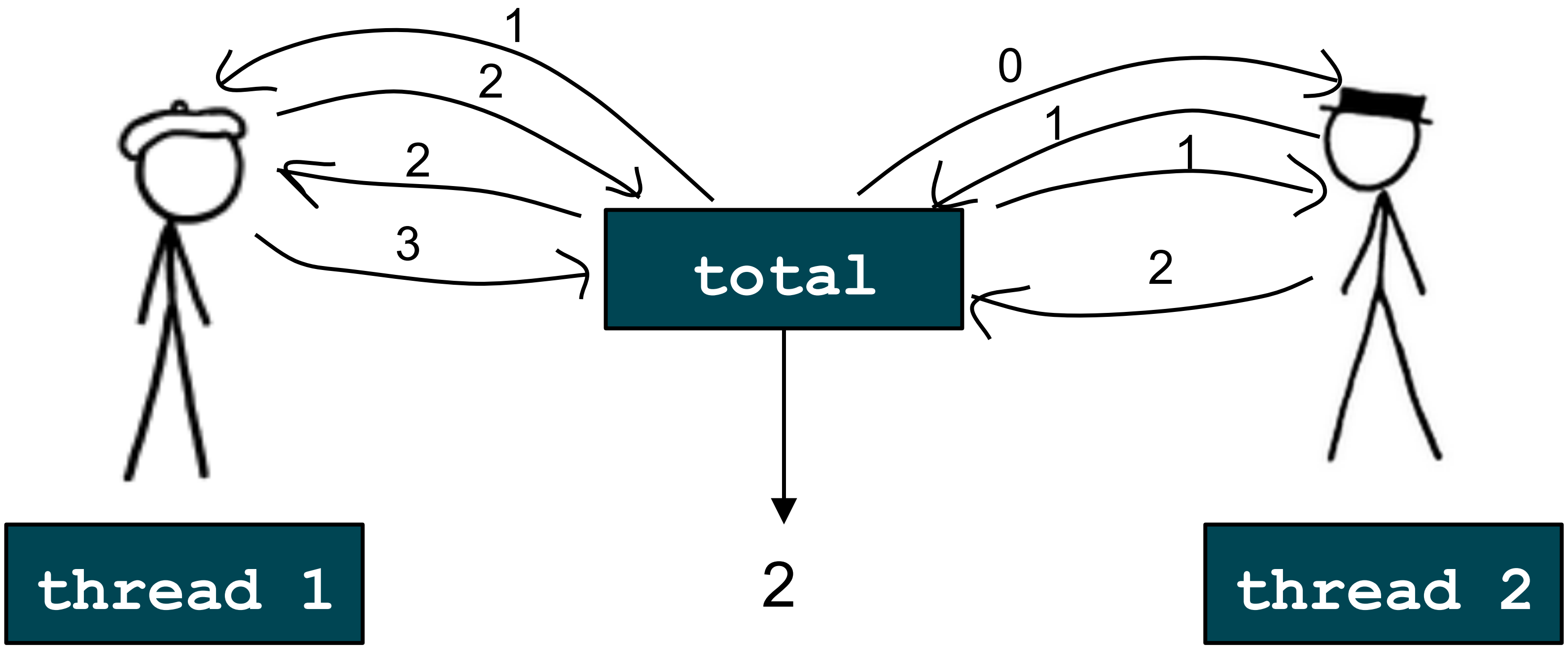
Let's try some more counting with threads.

```
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/suspend** execution of thread until resource is "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);
  ...
}
```

Semaphores and Atomicity

- A semaphore is a special counter with an invariant
  - Let s represent some semaphore, and T be the time domain
  - Invariant: At any given time, the value of a semaphore is non-negative (i.e. $\forall t \in T.\ val(s) \geq 0$)
- Mutexes are a subclass of semaphores in the sense that they either have a value of 0 or 1
- P(s) operation locks a resource (by decrementing the value of s) such that when another thread tries to lock the resource by calling P(s), the value may be 0, and that thread will be suspended until the value of s is greater than 0
- V(s) operation frees a resource (by incrementing the value of s) such that when it is called, s now has a value greater than 0, and any thread that may have been asleep as a result of waiting for s to be free can now be woken up
- Atomic Operation-An operation that is not interrupted once it has begun executing
  - P and V operations are atomic

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

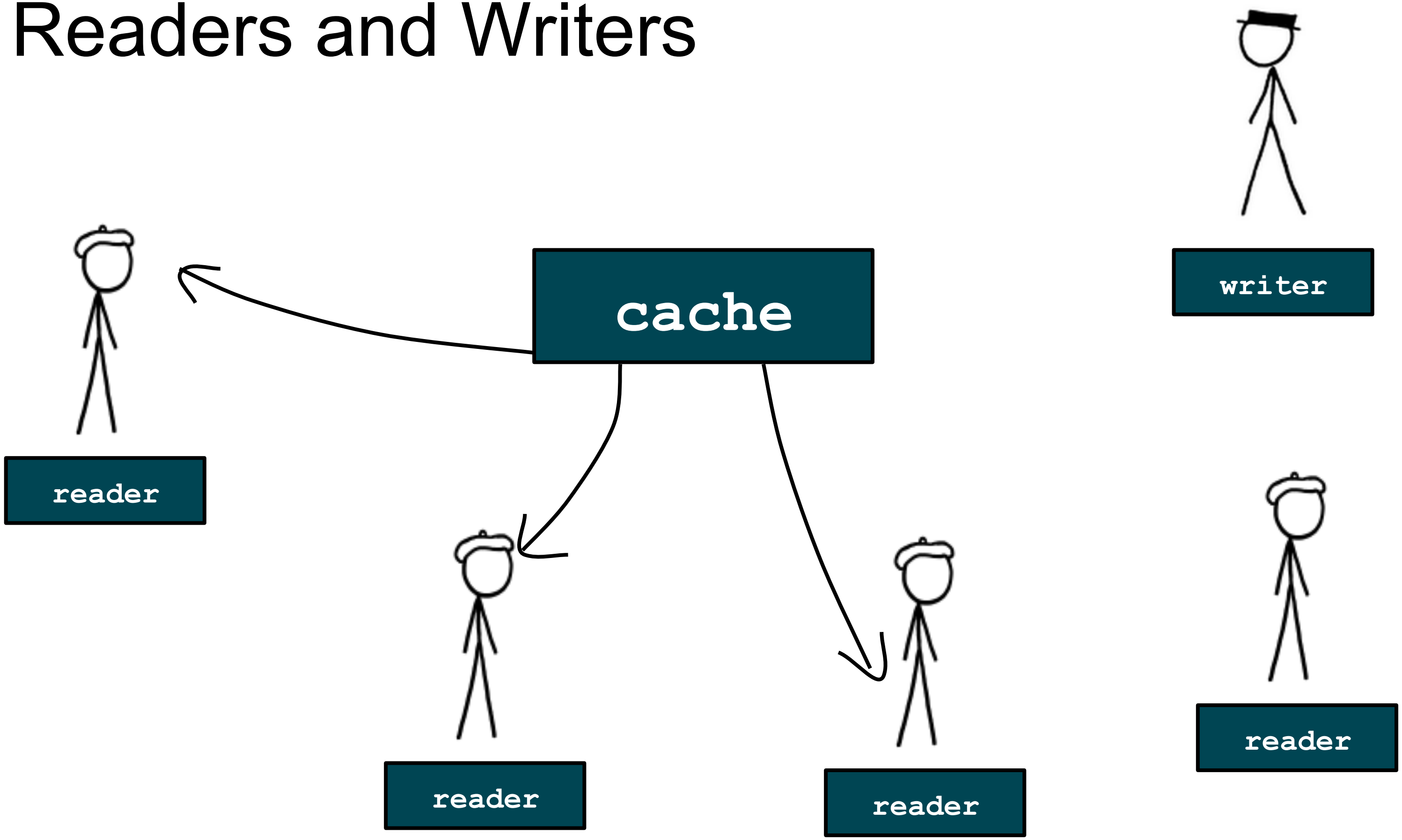
Synchronization Gone Wrong Part 1

```
/*Suppose that we want to perform some calculations concurrently. Also,
  assume that these tasks are done periodically and indefinitely. */

#define THREAD_1_WAIT_TIME 500
#define THREAD_2_WAIT_TIME 200

/*Function that makes the thread spend a certain amount of time in a loop
  Essentially waits for a certain amount of time to elapse. */
void spin_wait(unsigned int time){
  int countDown = time;
  while(countDown != 0){
  }
}

//Should execute for approximately 520 time units
void *thread1Routine(void *arg){
  while(1){
    //Assume this takes 20 time units
    performCalculations(pthread_self());
    spin_wait(THREAD_1_WAIT_TIME);
  }
  /*Control should never reach here*/
  return NULL;
}

//Should execute for approximately 210 time units
void *thread2Routine(void *arg){
  while(1){
    //Assume this takes 10 time units
    performCalculations(pthread_self());
    spin_wait(THREAD_2_WAIT_TIME);
  }
  /*Control should never reach here*/
  return NULL;
}
```

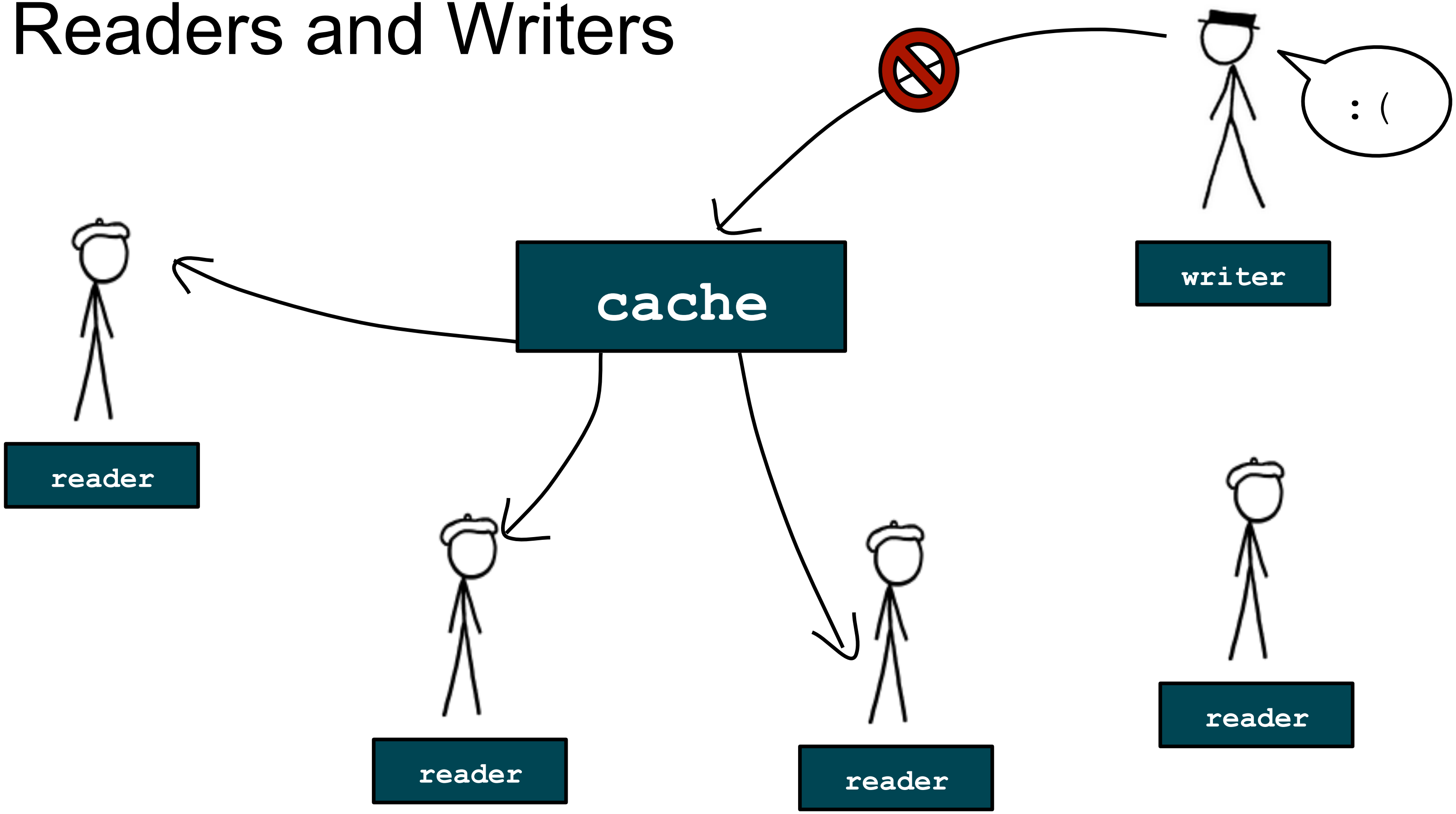
Synchronization Gone Wrong Part 2

- What’s wrong with this?
  - Note how there is only a small fraction of time per thread where calculations are actually being done
  - The great majority of the time is spent waiting to execute calculations again
  - This is a waste of valuable processor time
  - Do not do this
- Solution
  - Write code that will minimize the amount of time that the processor is not doing anything useful

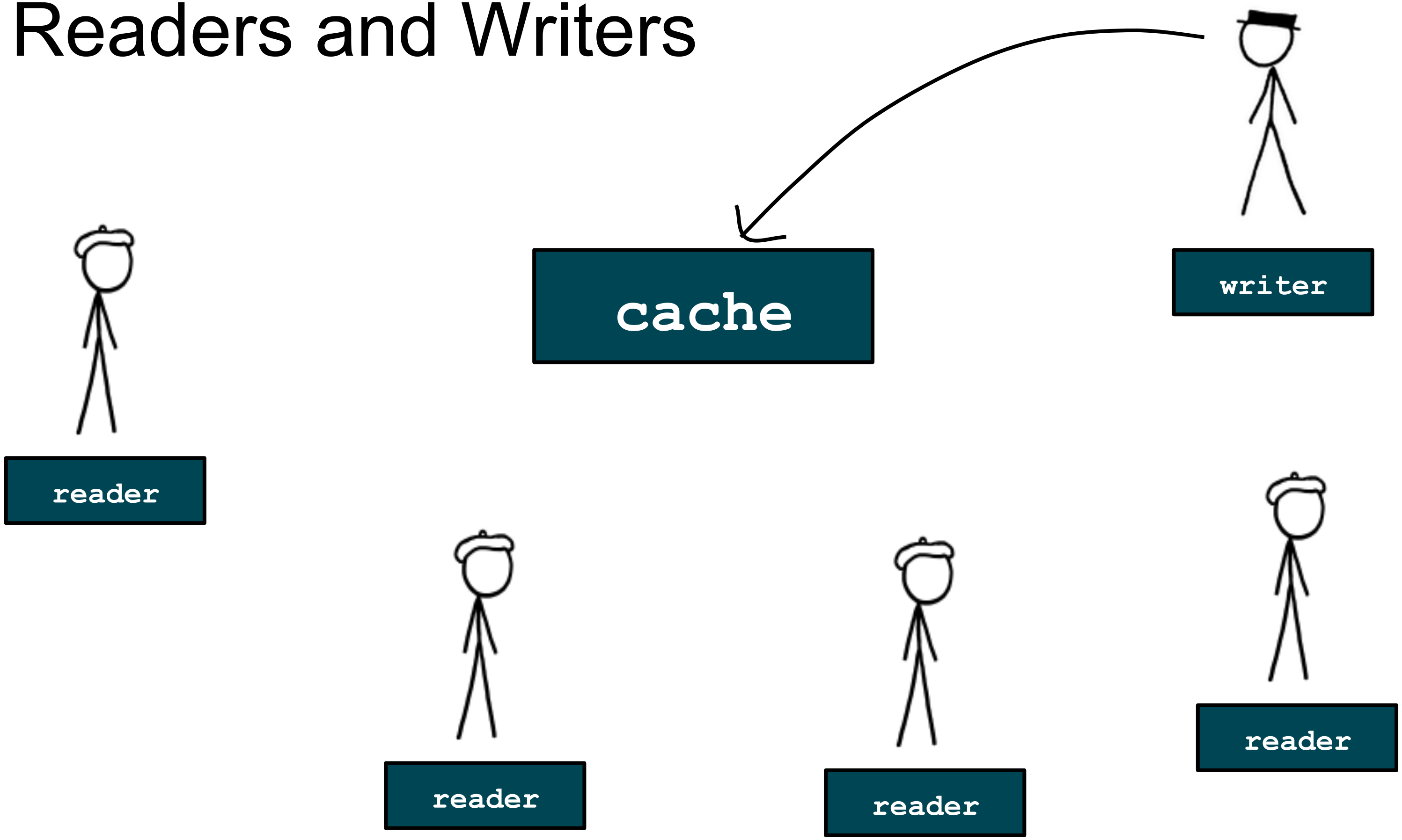
Readers and Writers



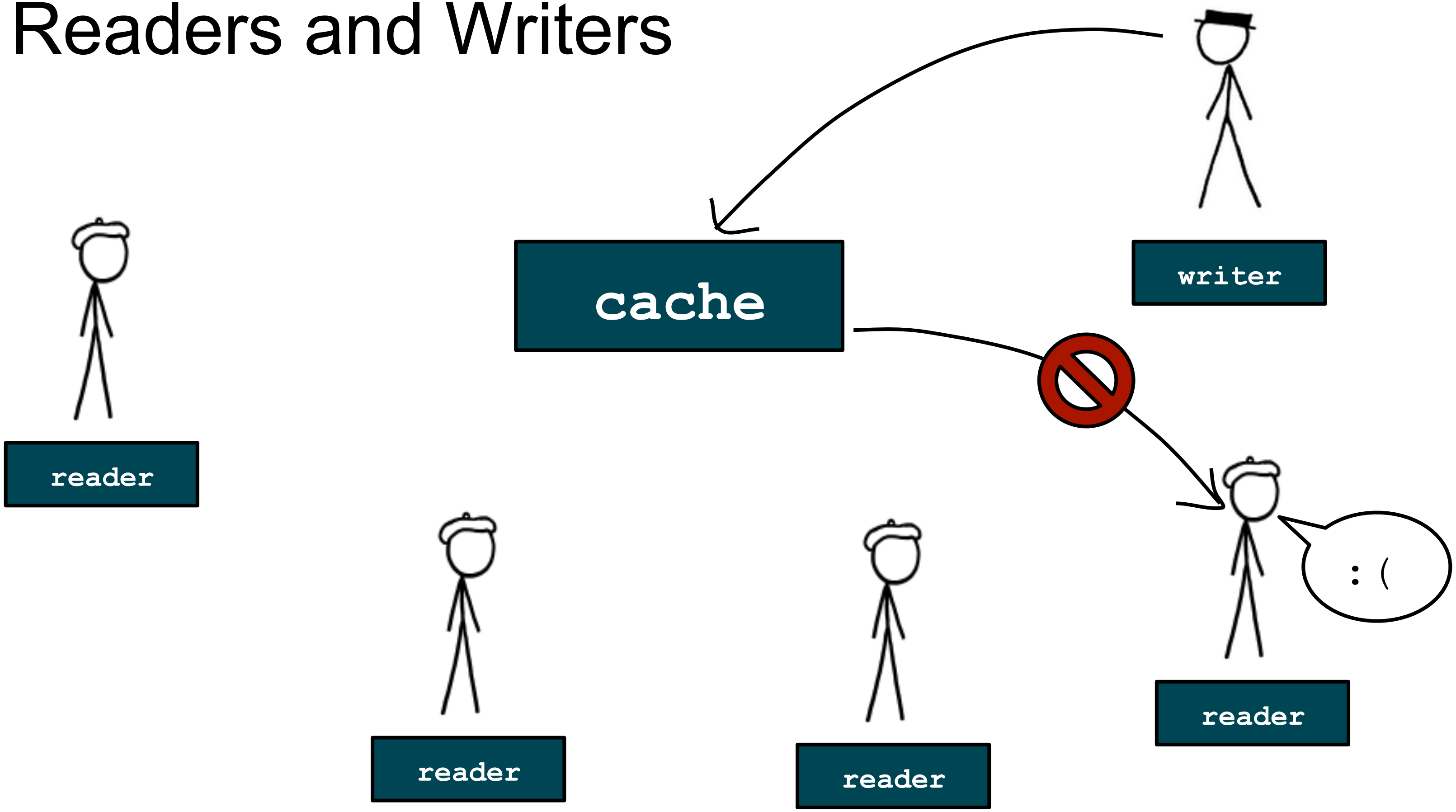
Readers and Writers



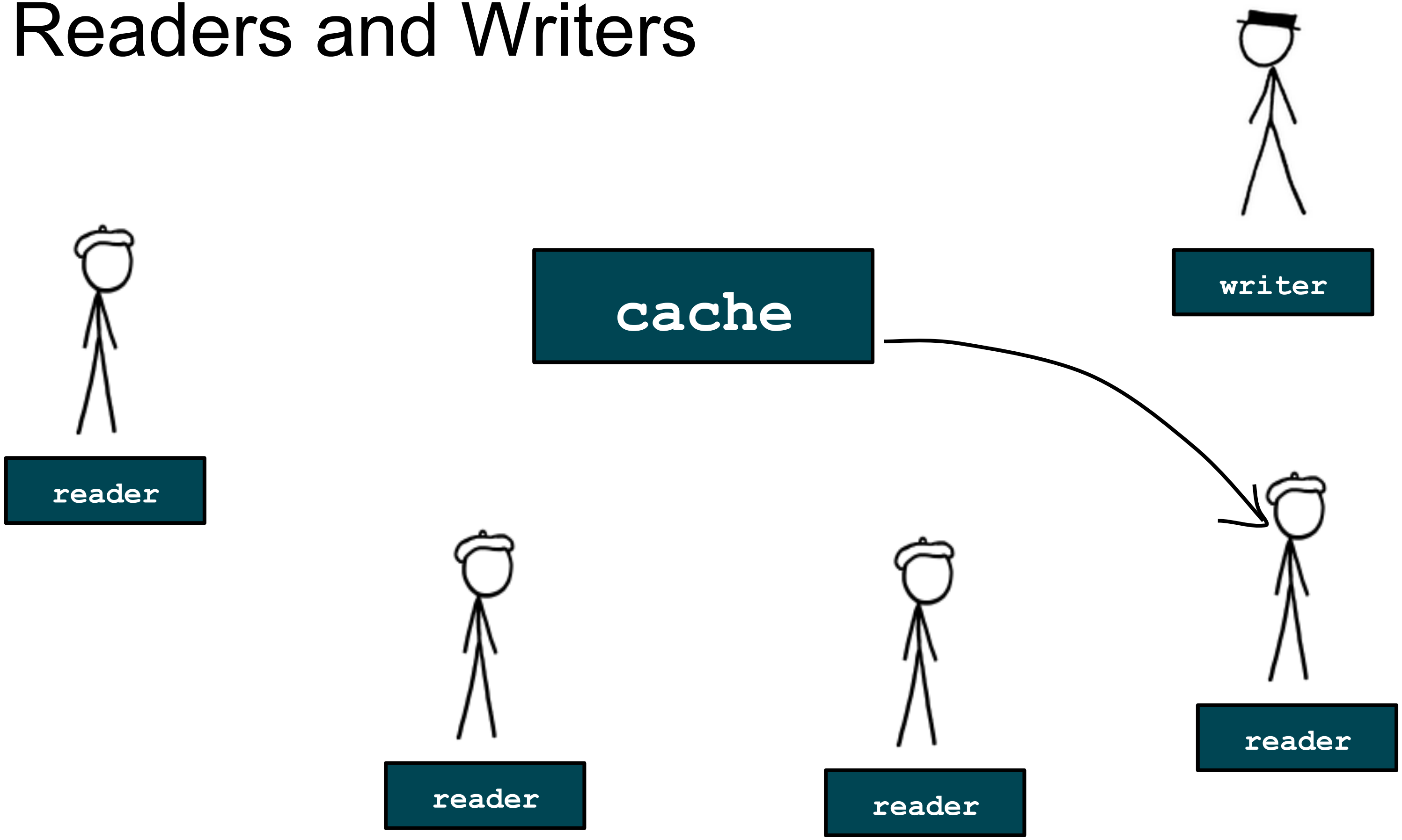
Readers and Writers



Readers and Writers



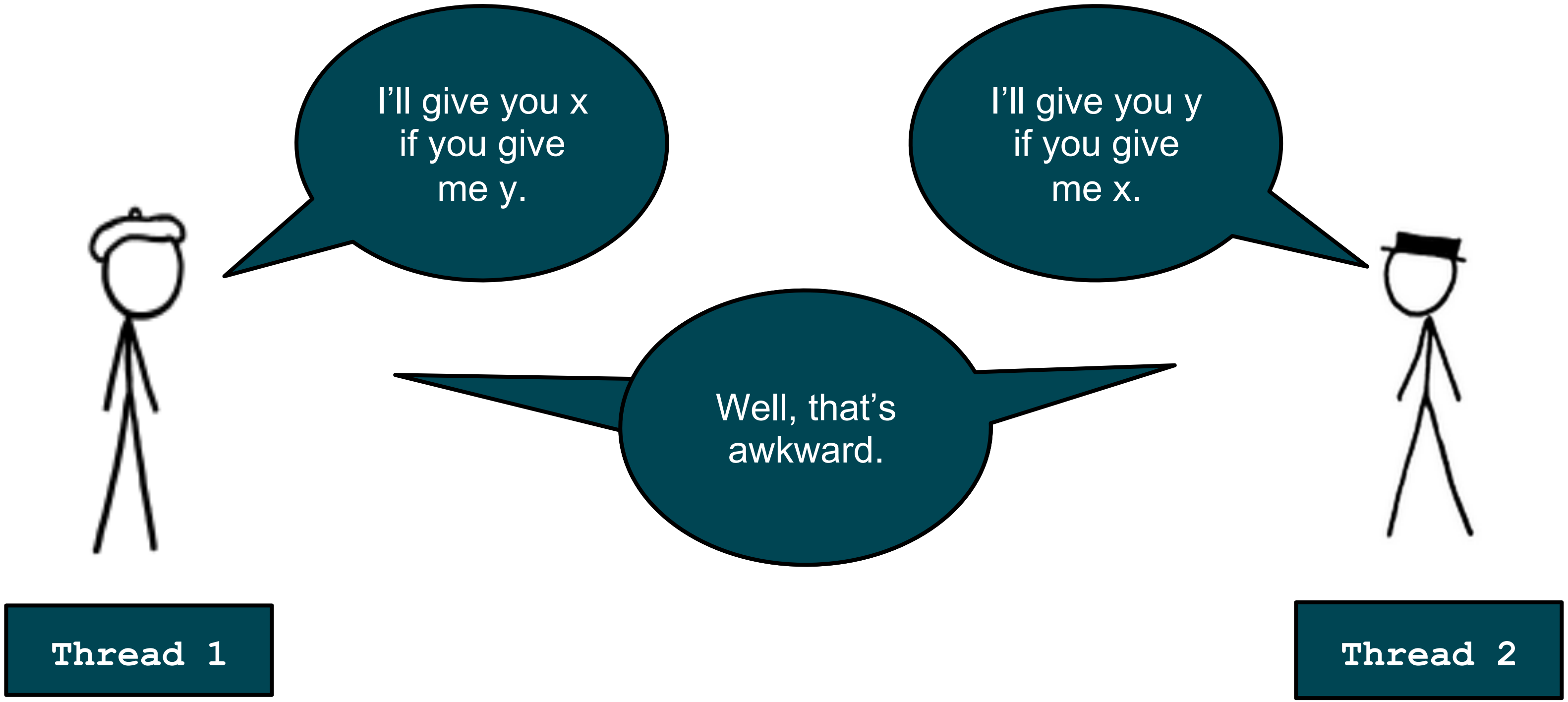
Readers and Writers



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

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

Recognitions

- Slides adapted from Jack Biggs