# Cond & Threads Wrap-up

## Computer Operating Systems, Fall 2025


Rainy Penn by u/EllMo77
Support Artists! ☺


Instructors: Joel Ramirez

Head TAs: Maya Huizar, Akash Kaukuntla, Vedansh Goenka, Joy Liu

TAs:

| | | | |
|---|---|---|---|
| Eric Zou | Joseph Dattilo | Aniket Ghorpade | Shriya Sane |
| Zihao Zhou | Eric Lee | Shruti Agarwal | Yemisi Jones |
| Connor Cummings | Shreya Mukunthan | Alexander Mehta | Raymond Feng |
| Bo Sun | Steven Chang | Rania Souissi | Rashi Agrawal |
| Sana Manesh | | | |

# Poll Everywhere

pollev.com/cis5480

- What is love?

# Administrivia

- PennOS
  - Milestone 0!
  - Need to meet with your TA next week before end of Friday
    - Late tokens possible to use
  - Covers general planning, understanding of sp-threads, and more!

- Today's Recitation 10/30 (Lead by yours truly & Eric Z perhaps)
  - Will cover threading, locks (spin/mutex), spthread & a baby schedular implementation.
    - Question focused! More practice for the Final! :D
  - If you come, you'll basically be ready to do Milestone 0 on the fly.
  - You can totally do it. I know it's rainy and the day before Halloweenie but pls…

# Administrivia

- Check-In will go out sometime tonight – I'll send an Ed Announcement.

- Don't forget about the Mid Semester Survey! Due Tomorrow @ midnight!
  - Make sure to give us honest feedback. We're not perfect. We are always looking to improve, honestly.

# Lecture Outline

- Multithreaded Cat
  - Producer v Consumer
- Locks Only
- Locks + Boolean Flag
- Posix Condition Variables
- Amdahl's Law
  - How much can threads really help?
  - Live Mandelbrot Example

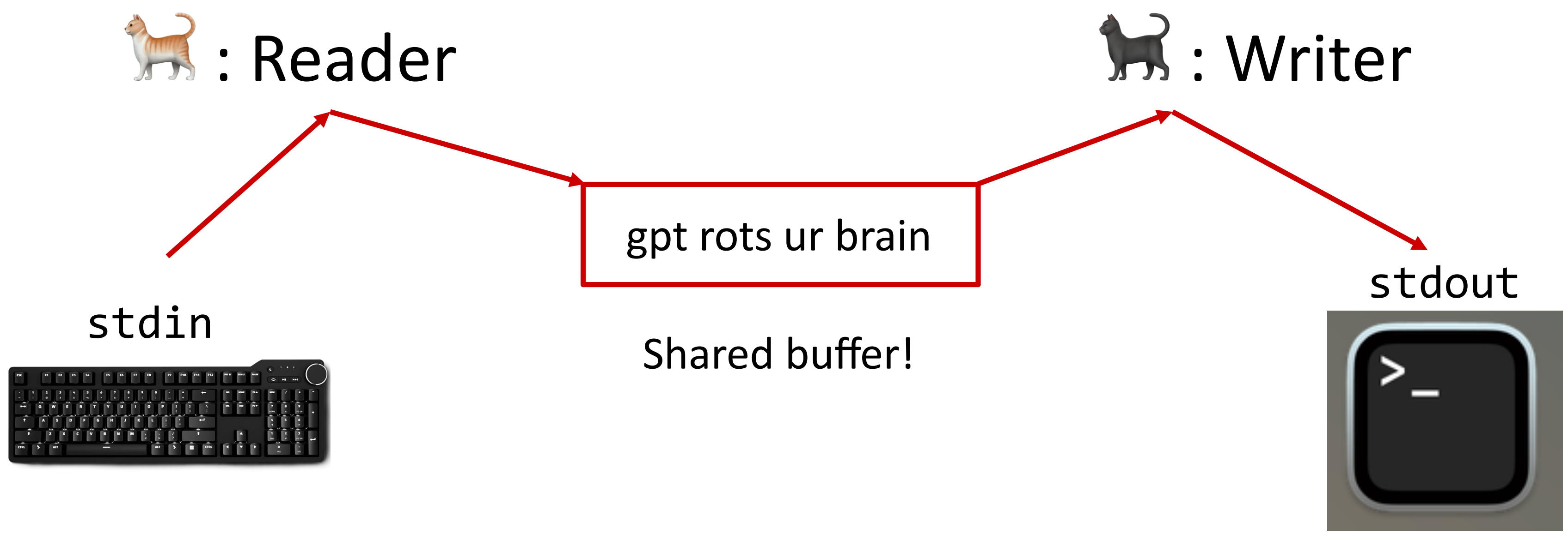
# Lecture Outline

- Multithreaded Cat
  - Producer v Consumer
- Locks Only
- Locks + Boolean Flag
- Posix Condition Variables
- Amdahl's Law
  - How much can threads really help?
  - Live Mandelbrot Example

# Data Race vs Race Condition

🐈 : Reader
🐈‍⬛ : Writer

- Implementing a Double Threaded Cat
  - Thread 1: Responsible for reading from the Terminal
  - Thread 2: Responsible for writing to the Terminal



# Data Race vs Race Condition

- Implementing a Double Threaded Cat
  - Thread 1: Responsible for reading from the Terminal
  - Thread 2: Responsible for writing to the Terminal

- Doesn't seem that bad – at a first glance, it seems all we need to do is protect the buffer/globals from simultaneous accesses!

- Let's take a look at our first implementation.

# Poll Everywhere

pollev.com/cis5480

- Is there a data race here?

```
static bool done = false;
static ssize_t n = 0;
char buffer[BUF_SIZE] = {0};
```

```
void *cat_read(void *arg){
    while (true){
        pthread_mutex_lock(&buf_lock);
        n = read(STDIN_FILENO, buffer, BUF_SIZE);
        if (n == 0){ // Ctrl-D
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_lock(&done_lock);
    done = true;
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

```
void *cat_write(void *arg){
    while(true){
        pthread_mutex_lock(&done_lock);
        if (done) break;
        pthread_mutex_unlock(&done_lock);
        pthread_mutex_lock(&buf_lock);
        if (n > 0){
            write(STDOUT_FILENO, buffer, n);
        }
        pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

# Race Condition vs Data Race

- Data-Race: when there are concurrent accesses to a shared resource, with at least one write, that can cause the shared resource to enter an invalid or "unexpected" state.

- Race-Condition: Where the program has different behavior depending on the ordering of concurrent threads. This can happen even if all accesses to shared resources are "atomic" or "locked"

# Poll Everywhere

discuss

- ***Is there a way for the writer to never write anything?***

```
static bool done = false;
static ssize_t n = 0;
char buffer[BUF_SIZE] = {0};
```

```
void *cat_read(void *arg){
    while (true){
        pthread_mutex_lock(&buf_lock);
        n = read(STDIN_FILENO, buffer, BUF_SIZE);
        if (n == 0){ // Ctrl-D
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_lock(&done_lock);
    done = true;
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

```
void *cat_write(void *arg){
    while(true){
        pthread_mutex_lock(&done_lock);
        if (done) break;
        pthread_mutex_unlock(&done_lock);
        pthread_mutex_lock(&buf_lock);
        if (n > 0){
            write(STDOUT_FILENO, buffer, n);
        }
        pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

# Thread Communication

- Threads may need to communicate with each other to know when they can perform operations
  - (almost like giving each other for permission to proceed)

- Example: Producer and consumer threads
  - One thread creates tasks/data (Our reader)
  - One thread consumes the produced tasks/data to perform some operation (Our writer)
  - The consumer thread can only consume things once the producer has produced them

- ***Need to make sure this communication has no data race or race condition.***
  - ***This is the hardest thing to reason about. Lots of sketching and tracing.***

# Lecture Outline

- Multithreaded Cat
  - Producer v Consumer
- Locks Only
- Locks + Boolean Flag
- Posix Condition Variables
- Amdahl's Law
  - How much can threads really help?
  - Live Mandelbrot Example

# Producer & Consumer Problem

- Common design pattern in concurrent programming.
  - There are at least two threads, at least one producer and at least one consumer.
  - The producer threads create some data that is then added to a shared data structure
  - Consumers will process and remove data from the shared data structure

- Let's try to go ahead and fix our example doubly threaded cat as best we can.

# Producer Consumer Example

- **Which needs to proceed first, the reader or the writer in our example?**

- We have to ensure that **our reader reads before the other thread writes.**

- Currently, we don’t have anything to change here. So let’s check out our writer.

```
void *cat_read(void *arg){
     while (true){
         pthread_mutex_lock(&buf_lock);
         n = read(STDIN_FILENO, buffer, BUF_SIZE);
         if (n == 0){ // Ctrl-D
             pthread_mutex_unlock(&buf_lock);
             break;
         }
          pthread_mutex_unlock(&buf_lock);
      }
      pthread_mutex_lock(&done_lock);
      done = true;
      pthread_mutex_unlock(&done_lock);
      return NULL;
}
```

# Producer Consumer Example

```
void *cat_write(void *arg){
    while(true){
         pthread_mutex_lock(&done_lock);
         if (done) break;
         pthread_mutex_unlock(&done_lock);

         pthread_mutex_lock(&buf_lock);
         if (n > 0){
             write(STDOUT_FILENO, buffer, n);
         }
         pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

- Would be great to not attempt to write before we're sure the reader has read so lets go ahead and do this.

# Producer Consumer Example

```
void *cat_write(void *arg){
    while(true){
        pthread_mutex_lock(&done_lock);
        if (done) break;
        pthread_mutex_unlock(&done_lock);

        pthread_mutex_lock(&has_read);
        while(!done_reading){
            pthread_mutex_unlock(&has_read);
            pthread_mutex_lock(&has_read);
        }
        done_reading = false;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&buf_lock);
        if (n > 0){
            write(STDOUT_FILENO, buffer, n);
        }
        pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

- Would be great to not attempt to write before we're sure the reader has read so lets go ahead and do this.
- We'll use a done_reading bool to indicate if reader is done reading and use a mutex for that variable.
  - Writer will reset the variable. *(Consume the permission)*
- We pass this red square once the reader has read into the buffer. So let's go back to the reader to ensure we set this flag.

# Producer Consumer Example

- Where should we indicate that we are done reading?
  - Once we are actually done reading!
- But what else do we need to do?

```
void *cat_read(void *arg){
    while (true){
        pthread_mutex_lock(&buf_lock);
        n = read(STDIN_FILENO, buffer, BUF_SIZE);
        if (n == 0){ // Ctrl-D
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_read);
        done_reading = true;
        pthread_mutex_unlock(&has_read);

    }
    pthread_mutex_lock(&done_lock);
    done = true;
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

# Poll Everywhere

**discuss**

```
void *cat_read(void *arg){
    while (true){
        pthread_mutex_lock(&buf_lock);
        n = read(STDIN_FILENO, buffer, BUF_SIZE);
        if (n == 0){ // Ctrl-D
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_read);
        done_reading = true;
        pthread_mutex_unlock(&has_read);

    }
    pthread_mutex_lock(&done_lock);
    done = true;
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

```
void *cat_write(void *arg){
    while(true){
        pthread_mutex_lock(&done_lock);
        if (done) break;
        pthread_mutex_unlock(&done_lock);

        pthread_mutex_lock(&has_read);
        while(!done_reading){
            pthread_mutex_unlock(&has_read);
            pthread_mutex_lock(&has_read);
        }
        done_reading = false;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&buf_lock);
        if (n > 0){
            write(STDOUT_FILENO, buffer, n);
        }
        pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

- ***What is missing here?***

# Poll Everywhere

**pollev.com/cis5480**

```c
void *cat_read(void *arg){
    while (true){
        pthread_mutex_lock(&buf_lock);
        n = read(STDIN_FILENO, buffer, BUF_SIZE);
        if (n == 0){ // Ctrl-D
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_read);
        done_reading = true;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&has_written);
        while(!done_writing){
            pthread_mutex_unlock(&has_written);
            pthread_mutex_lock(&has_written);
        }
        done_writing = false;
        pthread_mutex_unlock(&has_written);

    }

}
```

```c
void *cat_write(void *arg){
    while(true){
        pthread_mutex_lock(&done_lock);
        if (done) break;
        pthread_mutex_unlock(&done_lock);

        pthread_mutex_lock(&has_read);
        while(!done_reading){
            pthread_mutex_unlock(&has_read);
            pthread_mutex_lock(&has_read);
        }
        done_reading = false;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&buf_lock);
        if (n > 0){
            write(STDOUT_FILENO, buffer, n);
        }
        pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

***What is missing here? One more piece left.***

# Let’s Simplify and demo!

```
void *cat_read(void *arg){
    while (true){
        pthread_mutex_lock(&buf_lock);
        n = read(STDIN_FILENO, buffer, BUF_SIZE);
        if (n == 0){ // Ctrl-D
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_read);
        done_reading = true;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&has_written);
        while(!done_writing){
            pthread_mutex_unlock(&has_written);
            pthread_mutex_lock(&has_written);
        }
        done_writing = false;
        pthread_mutex_unlock(&has_written);

    } //checkout code for edge case…
    return NULL;
}
```

```
void *cat_write(void *arg){
    while(true){

        pthread_mutex_lock(&has_read);
        while(!done_reading){
            pthread_mutex_unlock(&has_read);
            pthread_mutex_lock(&has_read);
        }
        done_reading = false;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&buf_lock);
        if (n > 0){
            write(STDOUT_FILENO, buffer, n);
        } else {
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_written);
        done_writing = true;
        pthread_mutex_unlock(&has_written);
    }
    return NULL;
}
```

# Can we do better?

- The code is correct, but do we notice anything wrong with this code?
- Maybe a common inefficiency that I have told you about several times before (just in other contexts?)
- The consumer code "busy waits" when there is nothing for it to consume and the producer "busy waits" for the consumer to consume
- For us, the Writer busy waits until there is something to write and the Reader busy waits until the Writer is done writing.

# Do we need buf_lock here?

```
void *cat_read(void *arg){
    while (true){
        pthread_mutex_lock(&buf_lock);
        n = read(STDIN_FILENO, buffer, BUF_SIZE);
        if (n == 0){ // Ctrl-D
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_read);
        done_reading = true;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&has_written);
        while(!done_writing){
            pthread_mutex_unlock(&has_written);
            pthread_mutex_lock(&has_written);
        }
        done_writing = false;
        pthread_mutex_unlock(&has_written);

    } //checkout code for edge case…
    return NULL;
}
```

```
void *cat_write(void *arg){
    while(true){

        pthread_mutex_lock(&has_read);
        while(!done_reading){
            pthread_mutex_unlock(&has_read);
            pthread_mutex_lock(&has_read);
        }
        done_reading = false;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&buf_lock);
        if (n > 0){
            write(STDOUT_FILENO, buffer, n);
        } else {
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_written);
        done_writing = true;
        pthread_mutex_unlock(&has_written);
    }
    return NULL;
}
```

# Thread Communication: Naïve Solution

- Consider the previous example where a thread must wait to be notified before it can continue…

- Possible solution: "Spinning"
  - Infinitely loop until the producer thread notifies that the consumer thread can print

- See `cat_spin.c`
  - The thread in the loop uses A LOT of cpu just checking until the value is safe
  - Use h`top` to see CPU util

- Alternative: Condition variables

# Lecture Outline

- Multithreaded Cat
  - Producer v Consumer
- Locks Only
- Locks + Boolean Flag
- Posix Condition Variables
- Amdahl's Law
  - How much can threads really help?
  - Live Mandelbrot Example

# Condition Variables

- Variables that allow for a thread to wait (suspend) until they are ***notified (continued)*** to resume
- Avoids waiting clock cycles "spinning"
- Done in the context of mutual exclusion (That's how you check the condition…)
  - A thread must already have a lock, which it will temporarily release while waiting
  - Once notified, the thread will re-acquire a lock and resume execution
- Honestly, the look much nicer in C++ but we are limited to our lovely C.

# pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_init(pthread_cond_t* cond,
                        const pthread_condattr_t* attr);
  ```
  - Initializes a condition variable with specified attributes

- ```
  int pthread_cond_destroy(pthread_cond_t* cond);
  ```
  - “Uninitializes” a condition variable – clean up when done

- Just do this to statically initialize:
  ```
  pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
  ```

# Condition Variables

- Done in the context of mutual exclusion
  - A thread must already have a lock, which it will temporarily release while waiting
  - Once notified, the thread will re-acquire a lock and resume execution

```
//assume these are already initialized
condition_var cv;
mutex m;

lock(&m); //lock m first
while(some_cond_is_false){
    wait(&cv, &m); //wait here, release the mutex m and suspend
    // when re-woken, we lock m again if we can…if not, we block…
}
// m is locked when we leave this loop!
unlock(&m);
```

# pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);
  ```
  - Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked

```
int pthread_cond_signal(pthread_cond_t* cond);
```

  - Wakes up at least one of the threads which is waiting on the condition cond

```
int pthread_cond_broadcast(pthread_cond_t* cond);
```

  - Wakes up at all of the threads waiting on the condition cond
```

# pthread_cond_t Internal Pseudo-Code

- Here is some pseudo code to help understand condition variables

```
int pthread_cond_wait(pthread_cond_t* cond, pthead_mutex_t* lock) {
  pthread_mutex_unlock(&lock);
  sleep_on_cond(cond);  // this and previous line happen atomically
  pthread_mutex_lock(&lock);
  return 0;
}
```

```
int pthread_cond_signal(pthread_cond_t* cond) {
  wakeup_a_thread(cond);  // wake up a thread sleeping on the cond
  return 0;
}
```

```
int pthread_cond_broadcast(pthread_cond_t* cond) {
  for (thread_sleeping : cond->asleep) { // wake's up all threads
    wakeup(thread_sleeping);
  }
  return 0;
}
```

# A Condition is Necessary

- This appears to force a thread to be suspended until another signals it.

```
pthread_mutex_lock(&lock);
pthread_cond_wait(&cv, &lock);

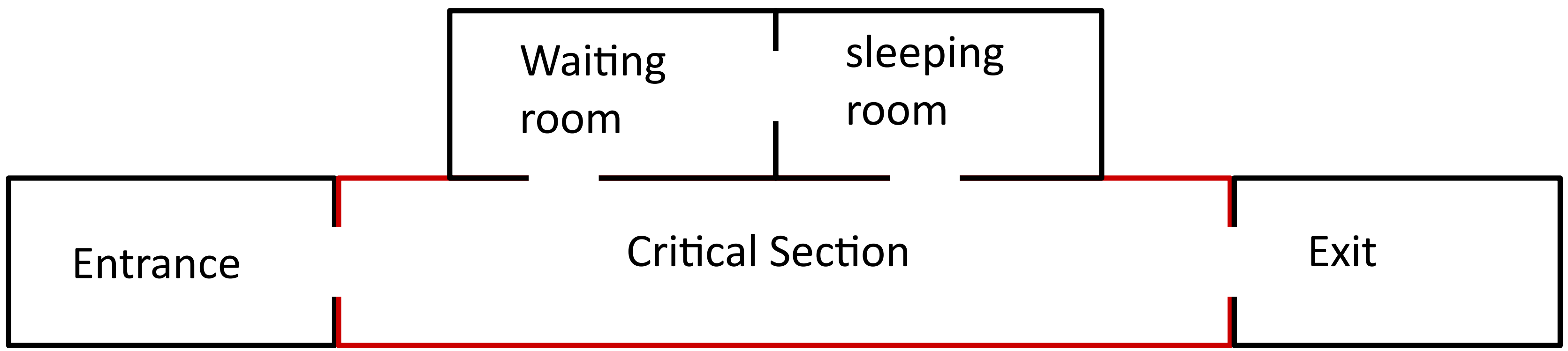
pthread_mutex_unlock(&lock);
```

- However, threads can have “spurious” wakeups. The thread that is suspended on the condition can wakeup even before another signal signals it.

- All to say, make sure to protect your condition variables with a condition….

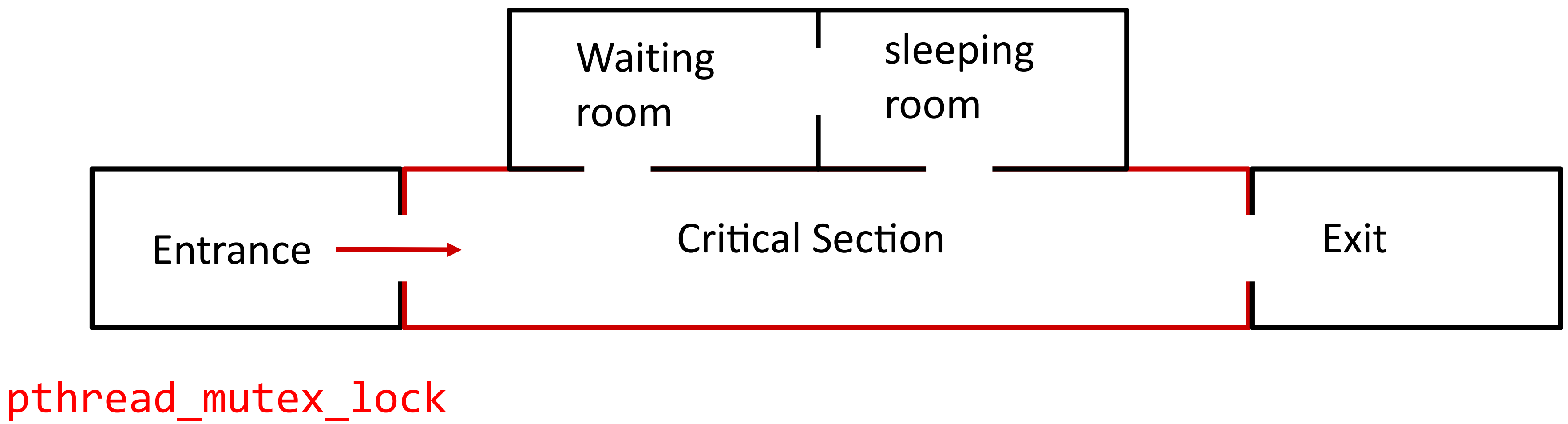
# Condition Variable & Mutex Visualization

- A possible condition variable visualization.



# Condition Variable & Mutex Visualization

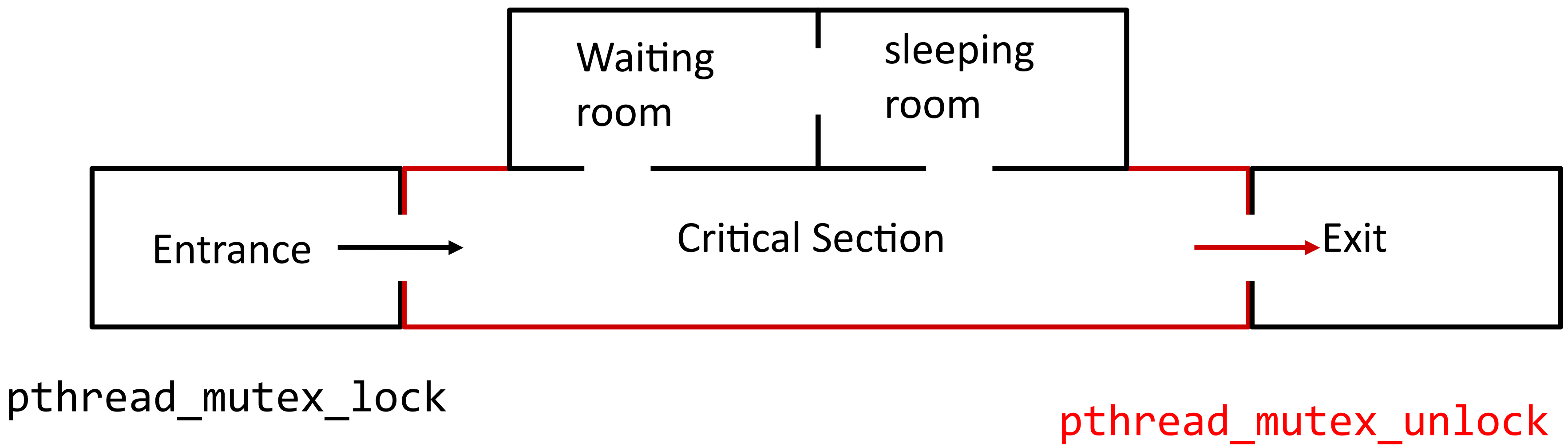
- A possible condition variable visualization.




A thread enters the critical section by acquiring a lock

# Condition Variable & Mutex Visualization

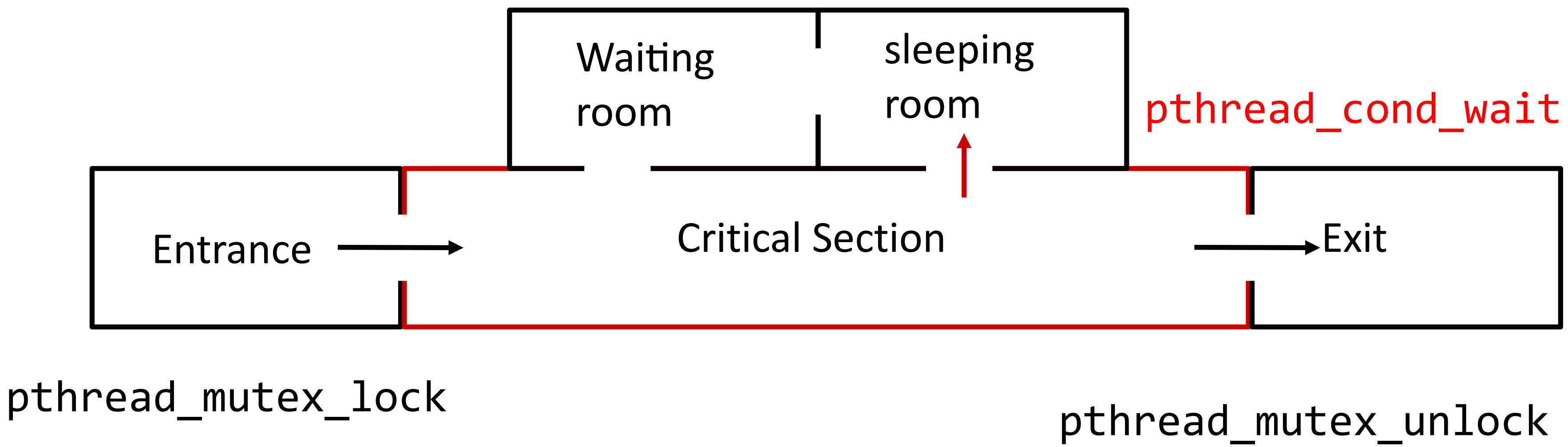
- A possible condition variable visualization.




A thread can exit the critical section by releasing the lock

# Condition Variable & Mutex Visualization

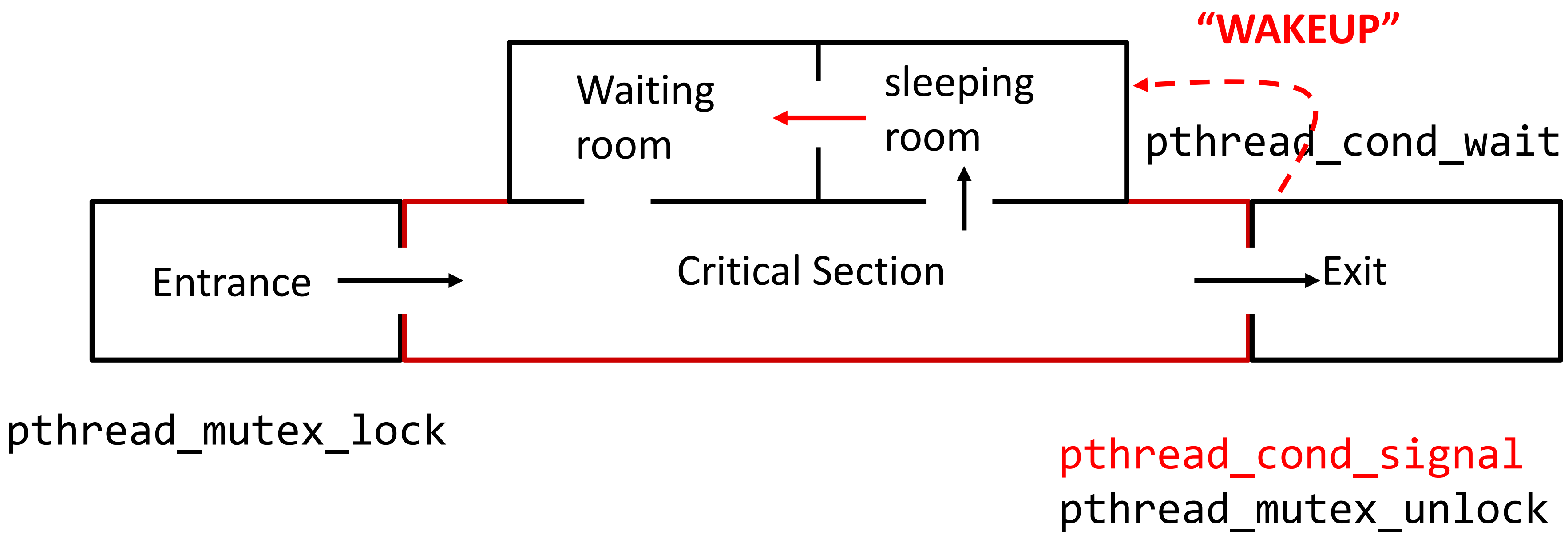
- A possible condition variable visualization.




If a thread can't complete its action, or must wait for some change in state, it can "go to sleep" until someone wakes it up later.
It will release the lock implicitly when it goes to sleep

# Condition Variable & Mutex Visualization

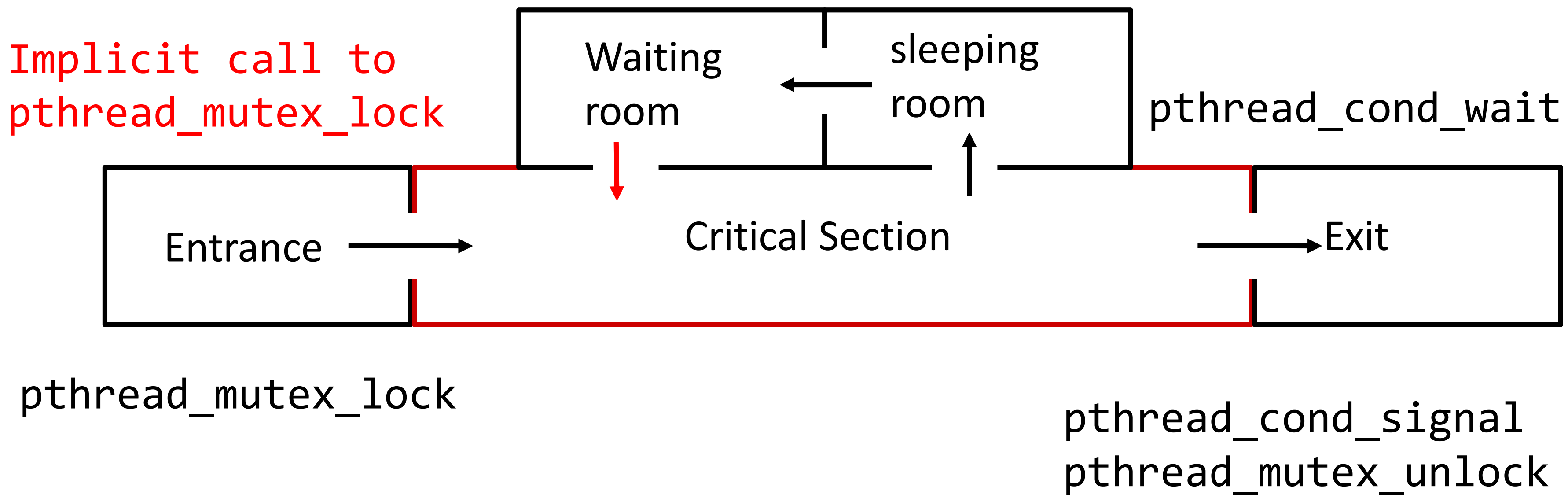
- A possible condition variable visualization.




When a thread modifies state and then leaves the critical section, it can also call pthread_cond_signal to wake up threads sleeping on that condition variable

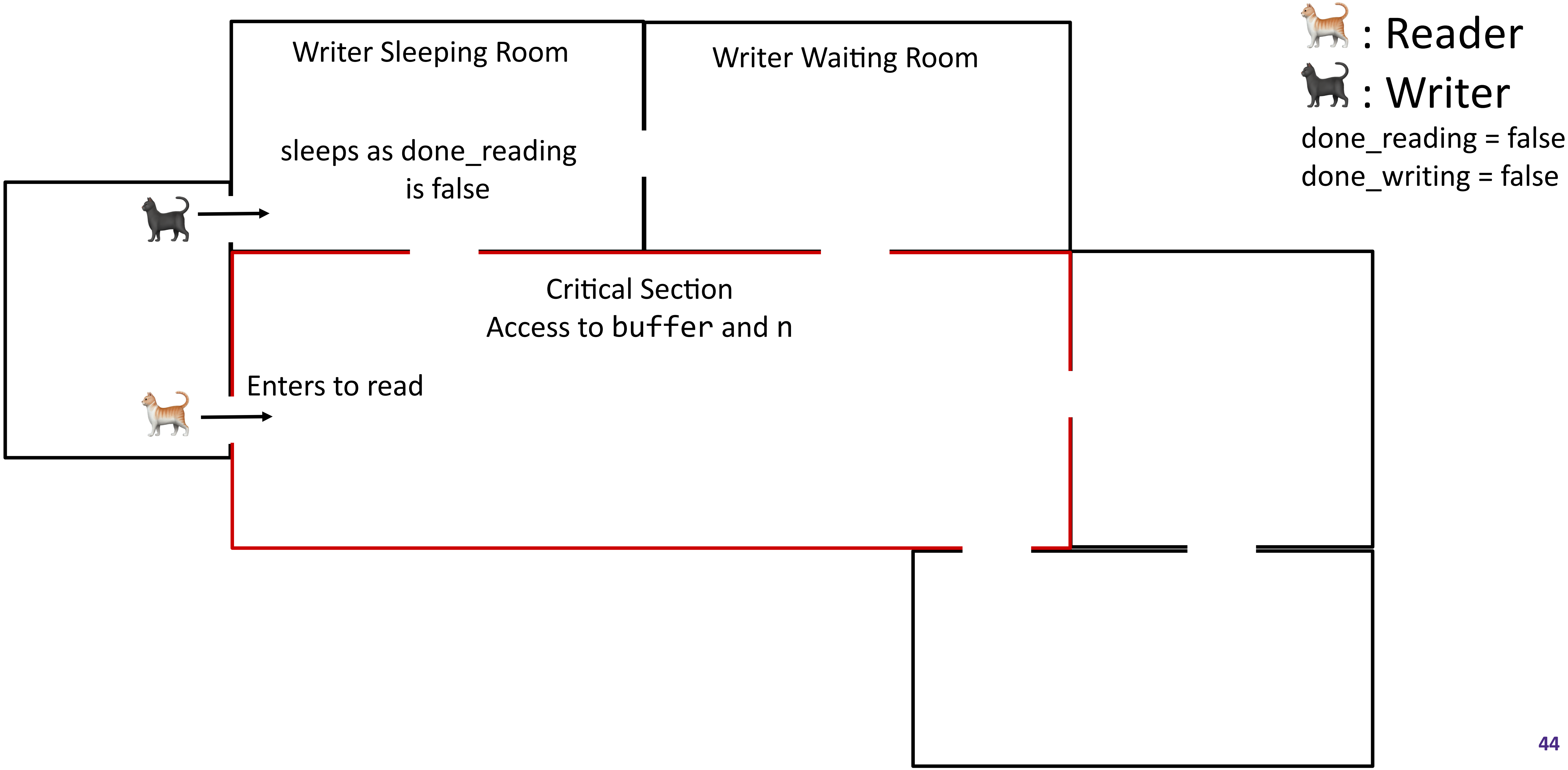
# Condition Variable & Mutex Visualization

- A possible condition variable visualization.

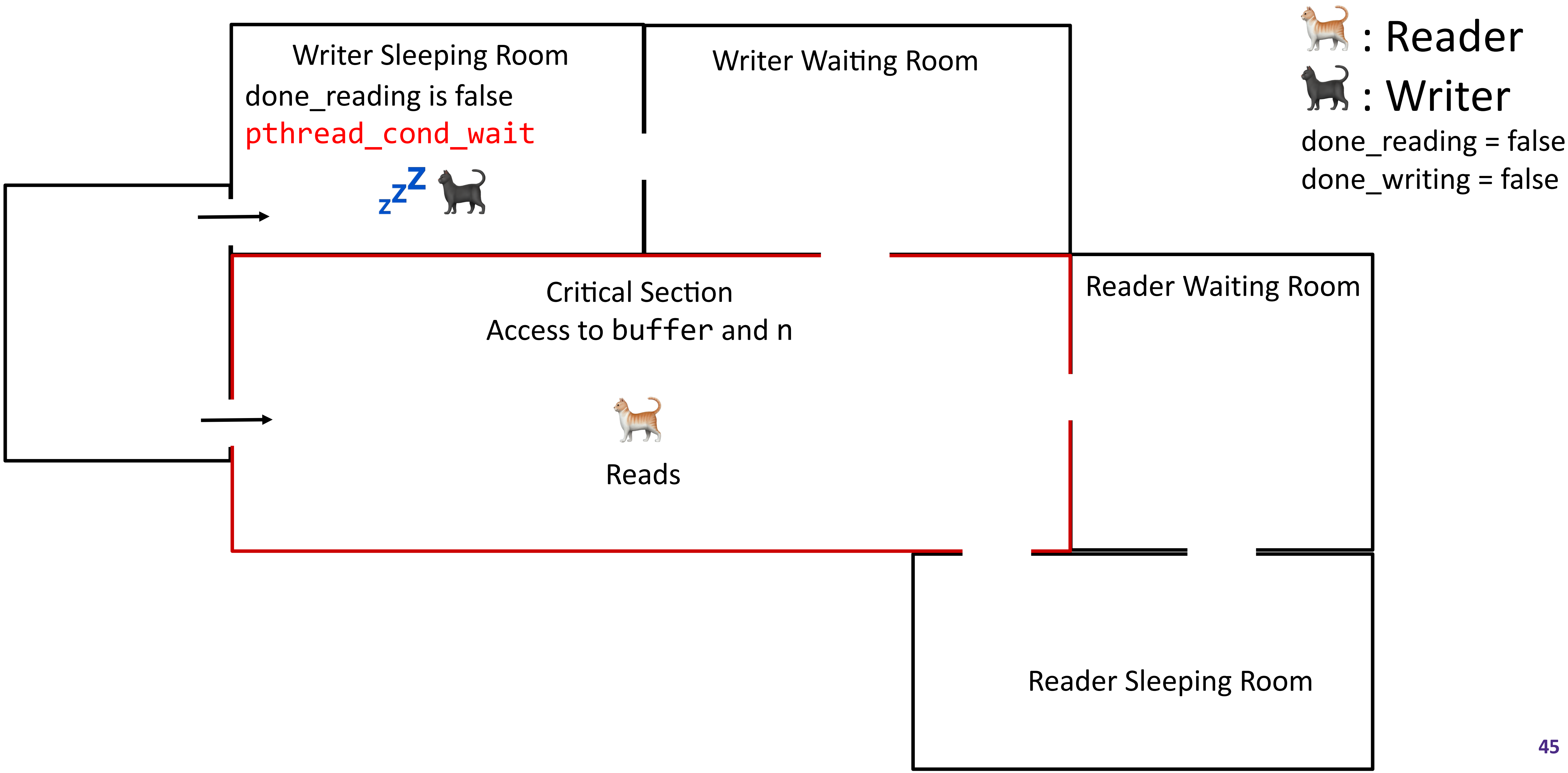



One or more sleeping threads wake up and attempt to acquire the lock.
Like a normal call to pthread_mutex_lock the thread will block until it can acquire the lock

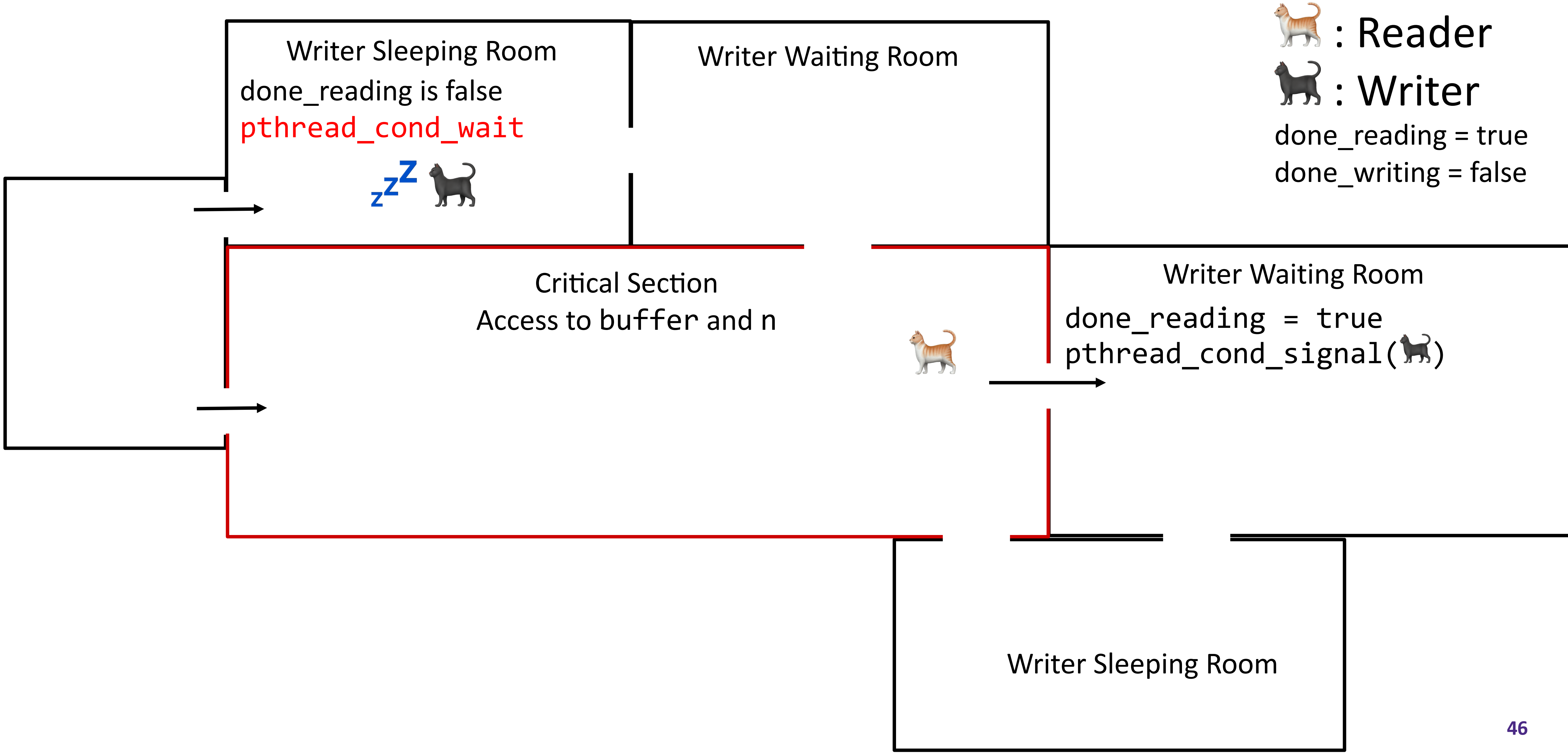
# Condition Variable & Mutex Visualization with our CAT!



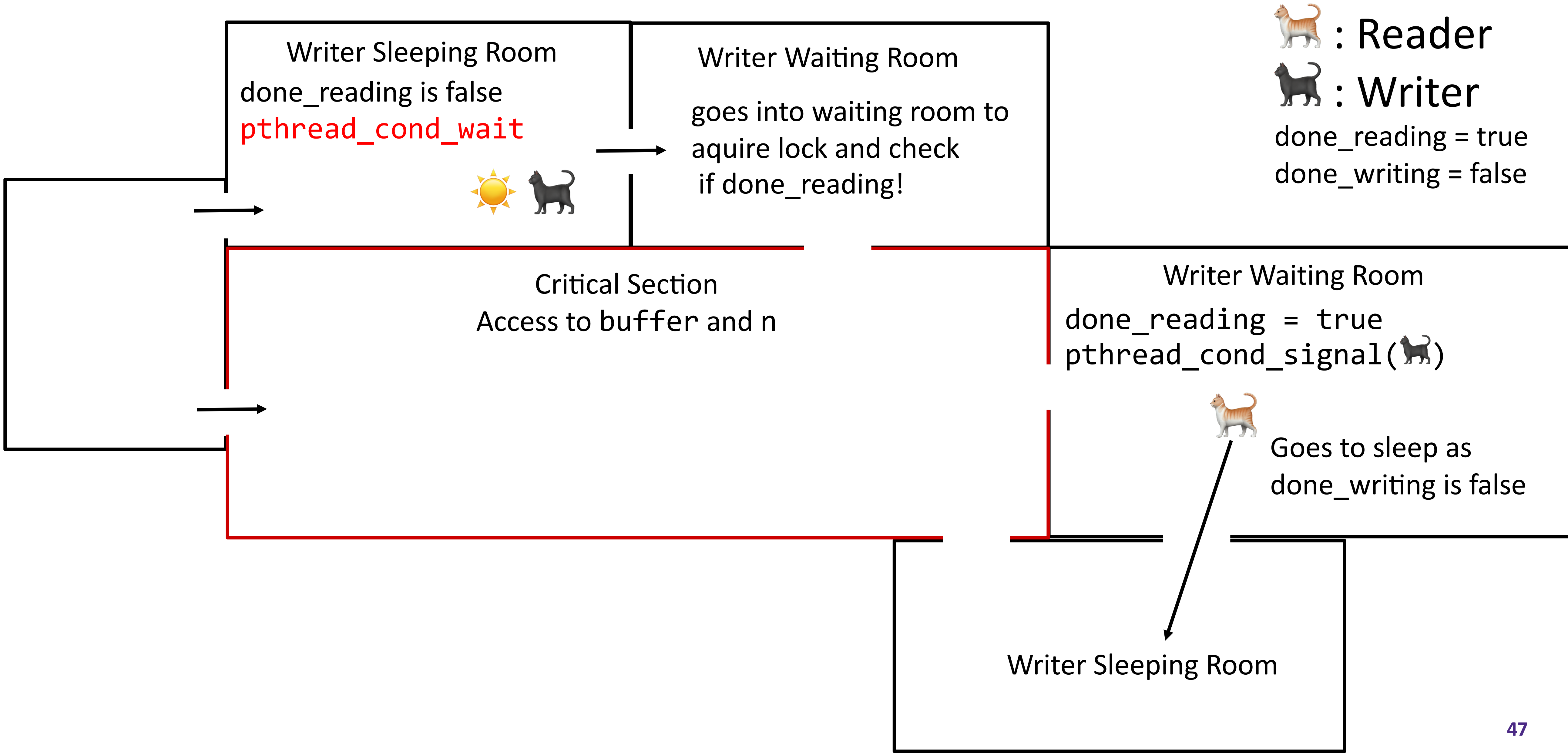
# Condition Variable & Mutex Visualization with our CAT!



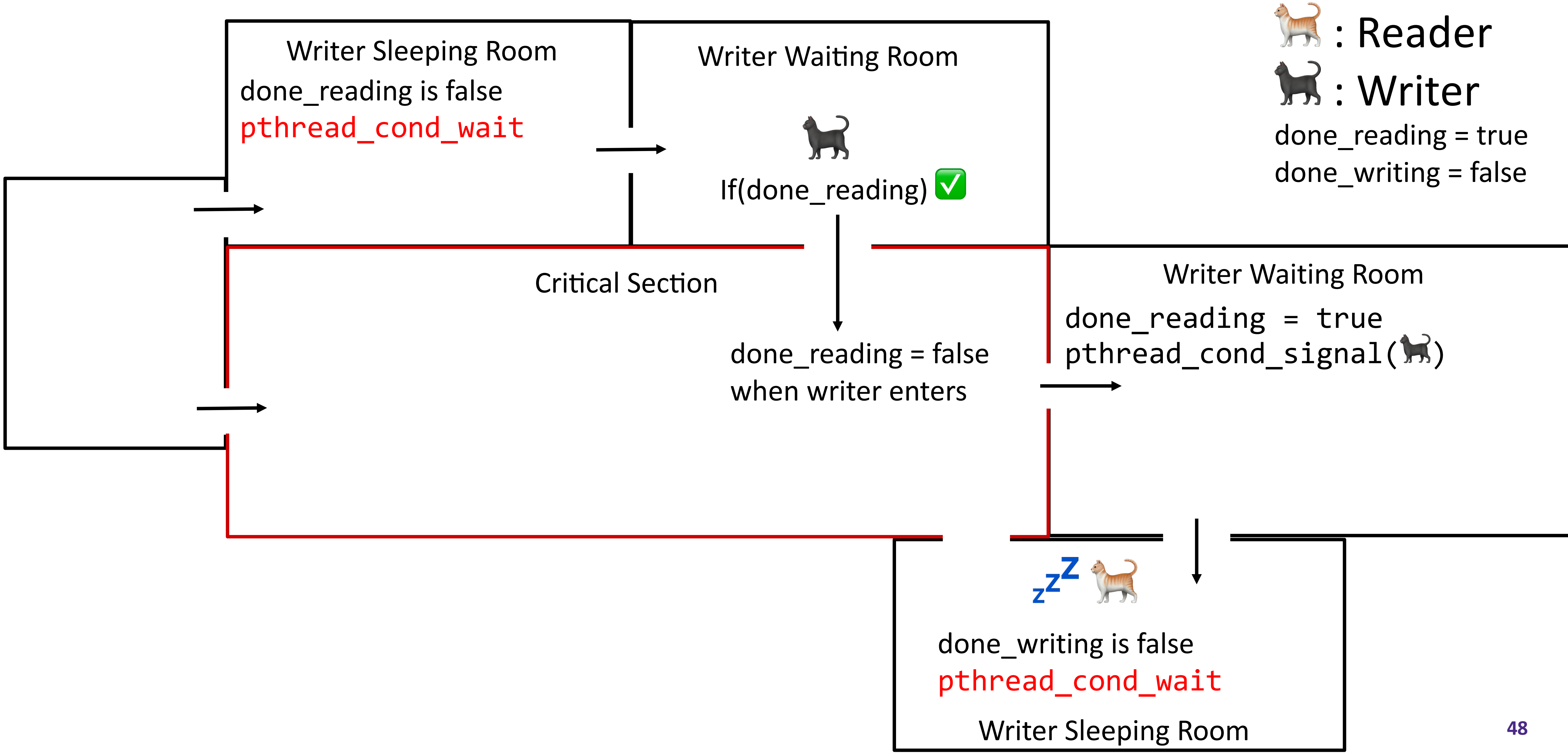
# Condition Variable & Mutex Visualization with our CAT!



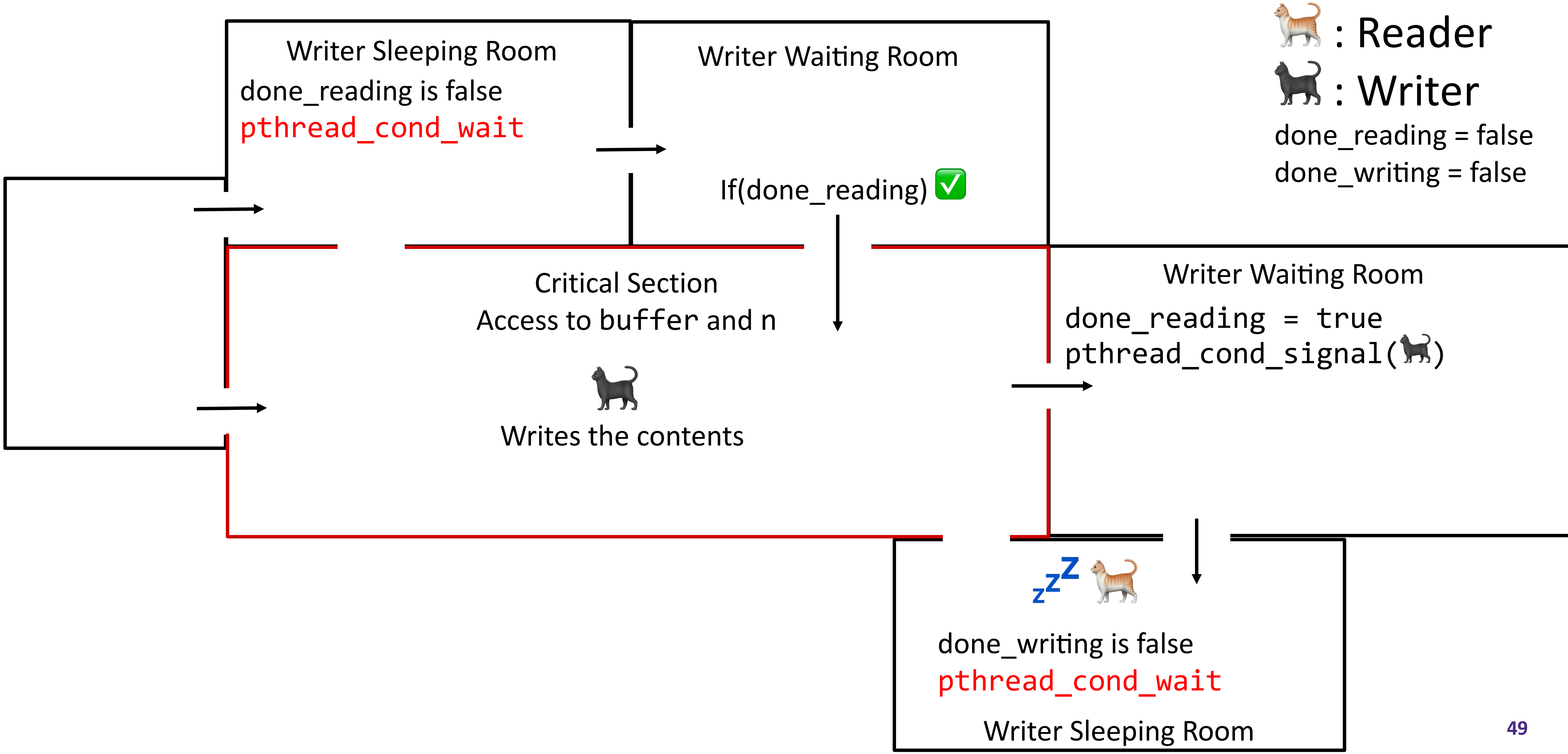
# Condition Variable & Mutex Visualization with our CAT!



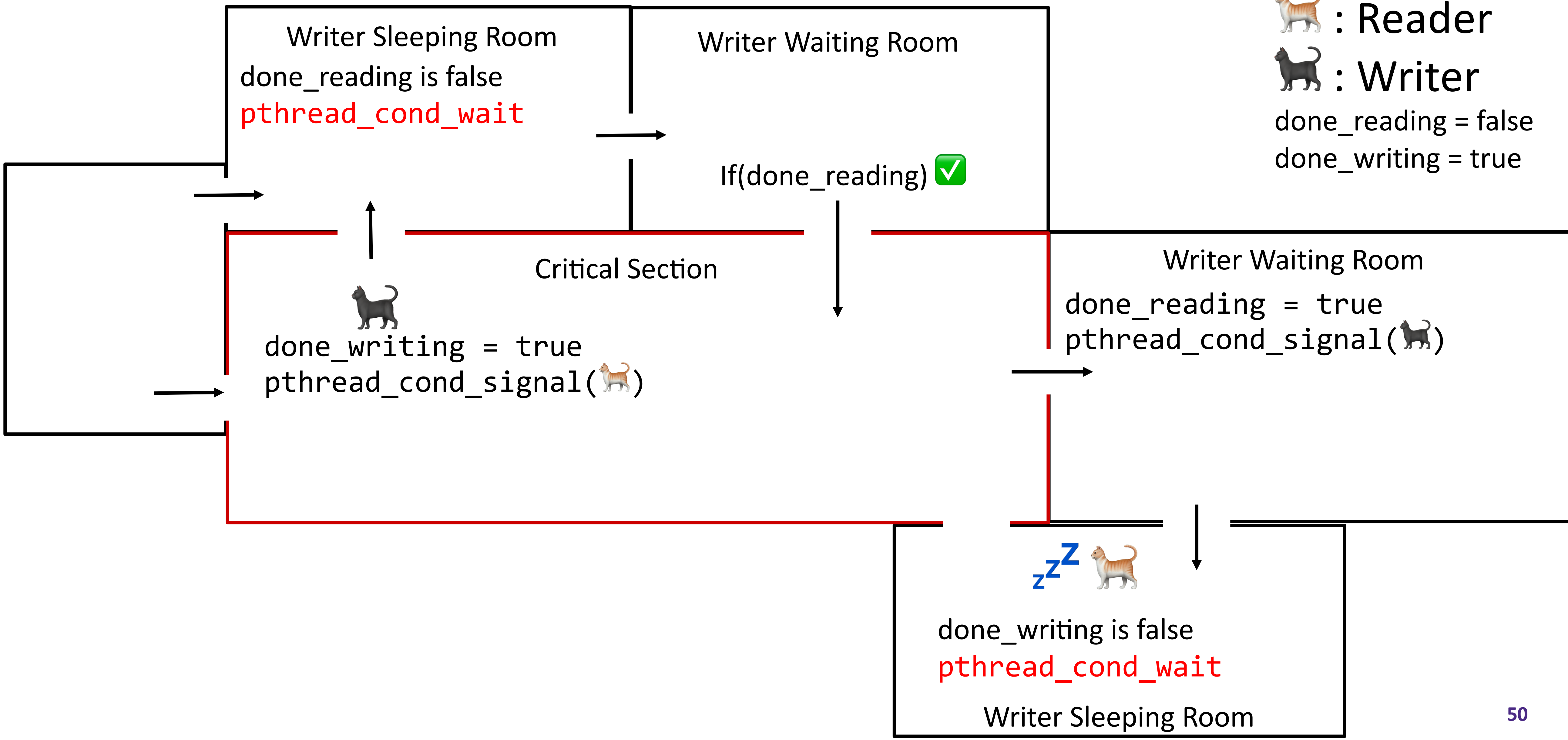
# Condition Variable & Mutex Visualization with our CAT!



# Condition Variable & Mutex Visualization with our CAT!

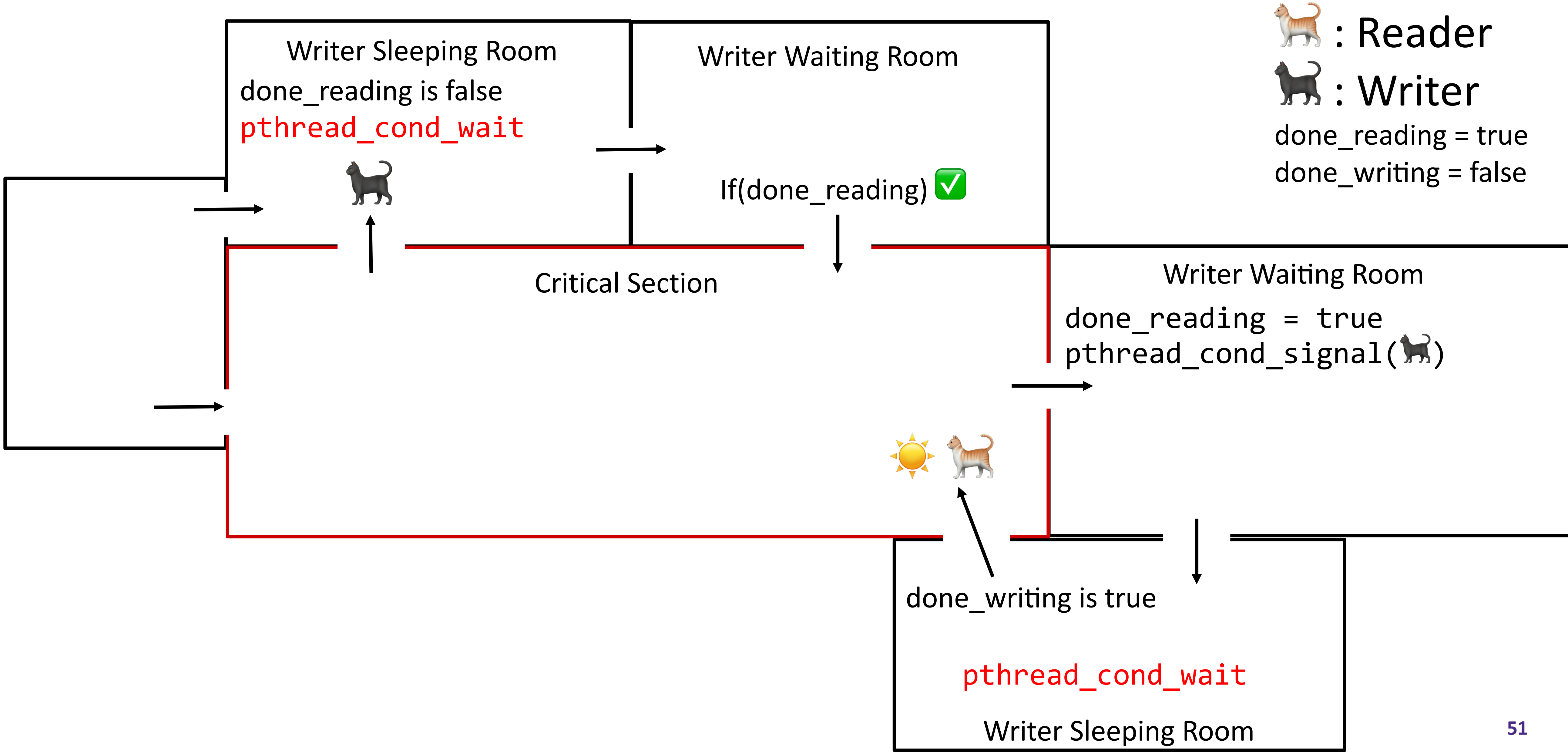


# Condition Variable & Mutex Visualization with our CAT!

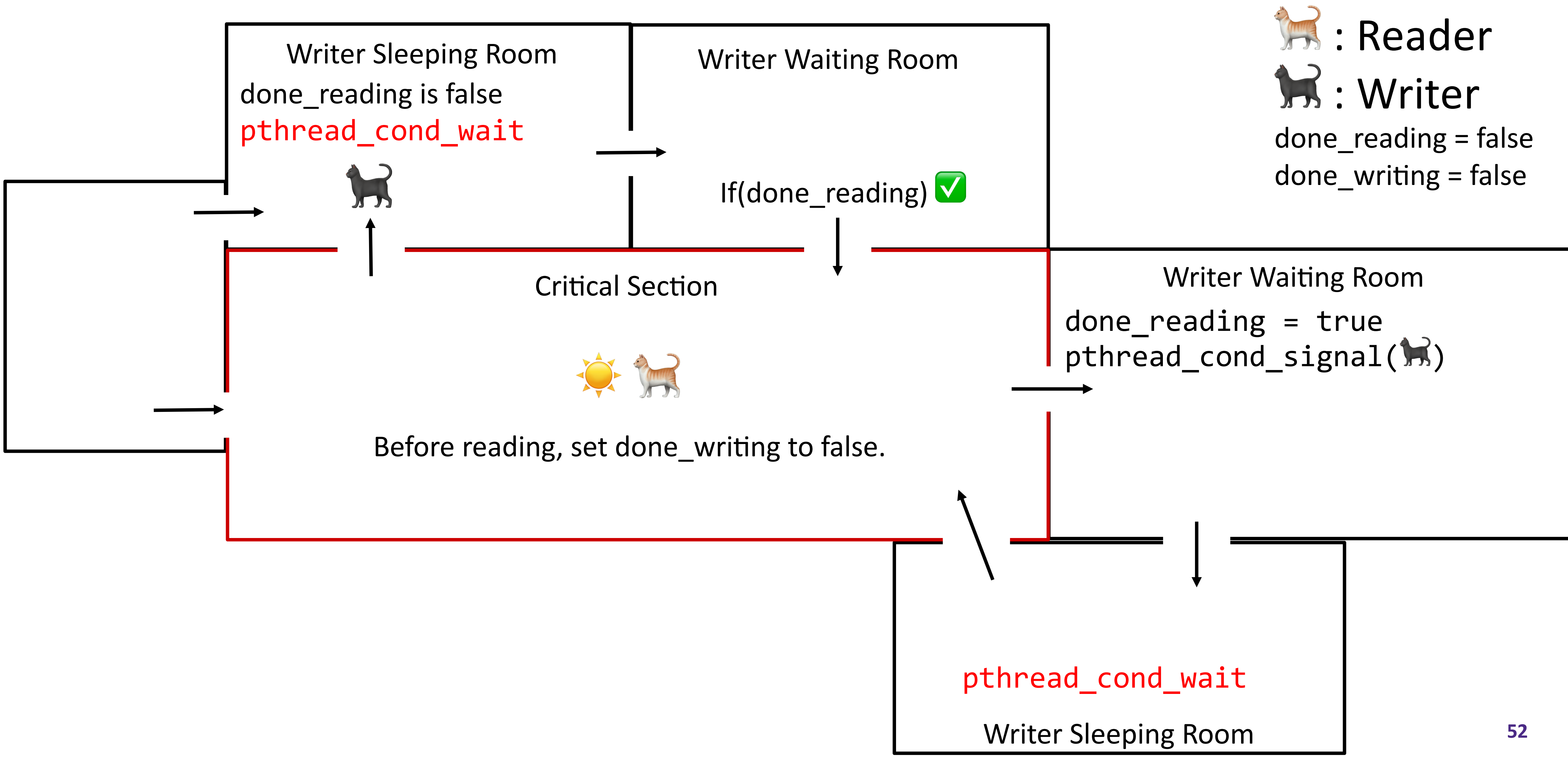



🐈 : Reader
🐈‍⬛ : Writer
done_reading = false
done_writing = true

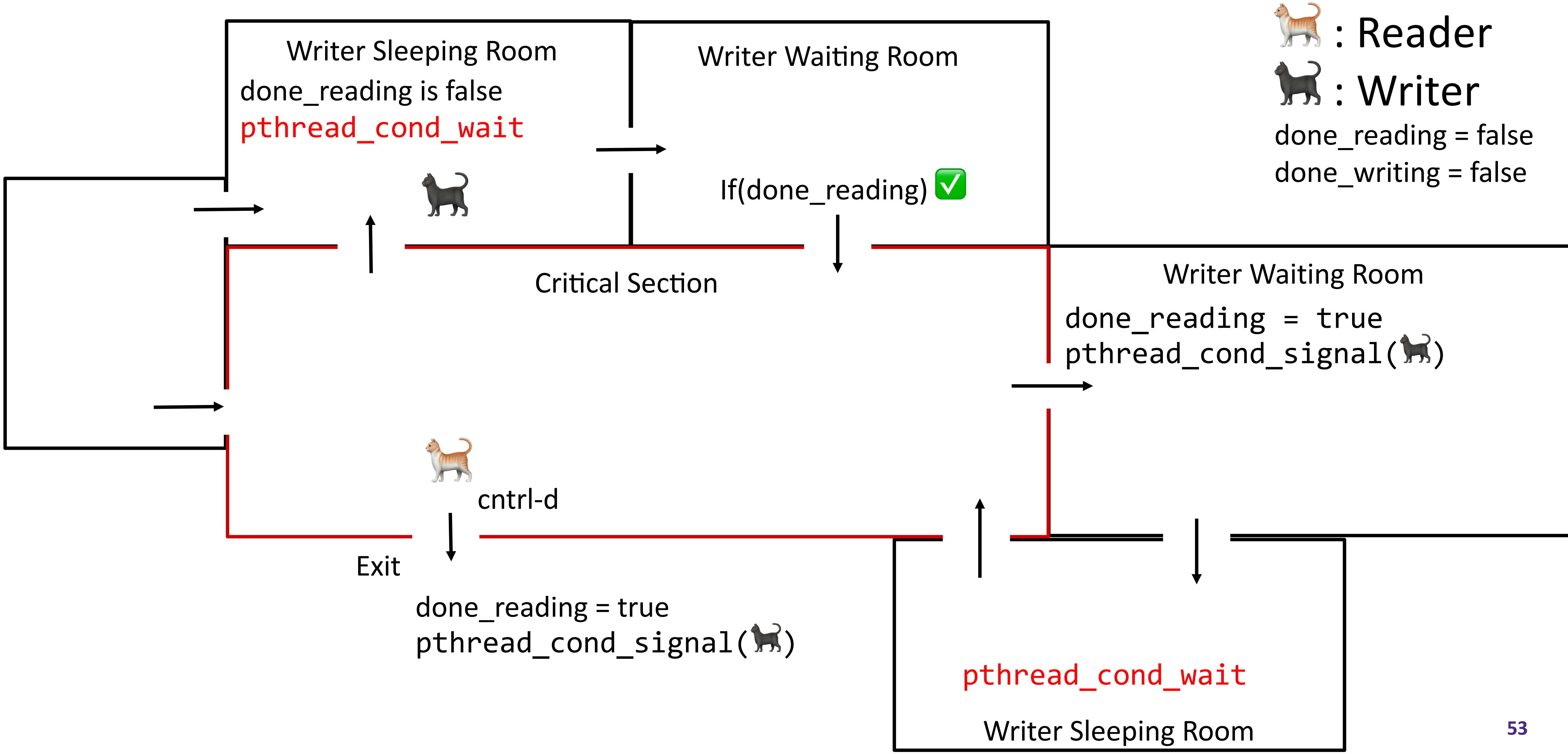
# Condition Variable & Mutex Visualization with our CAT!



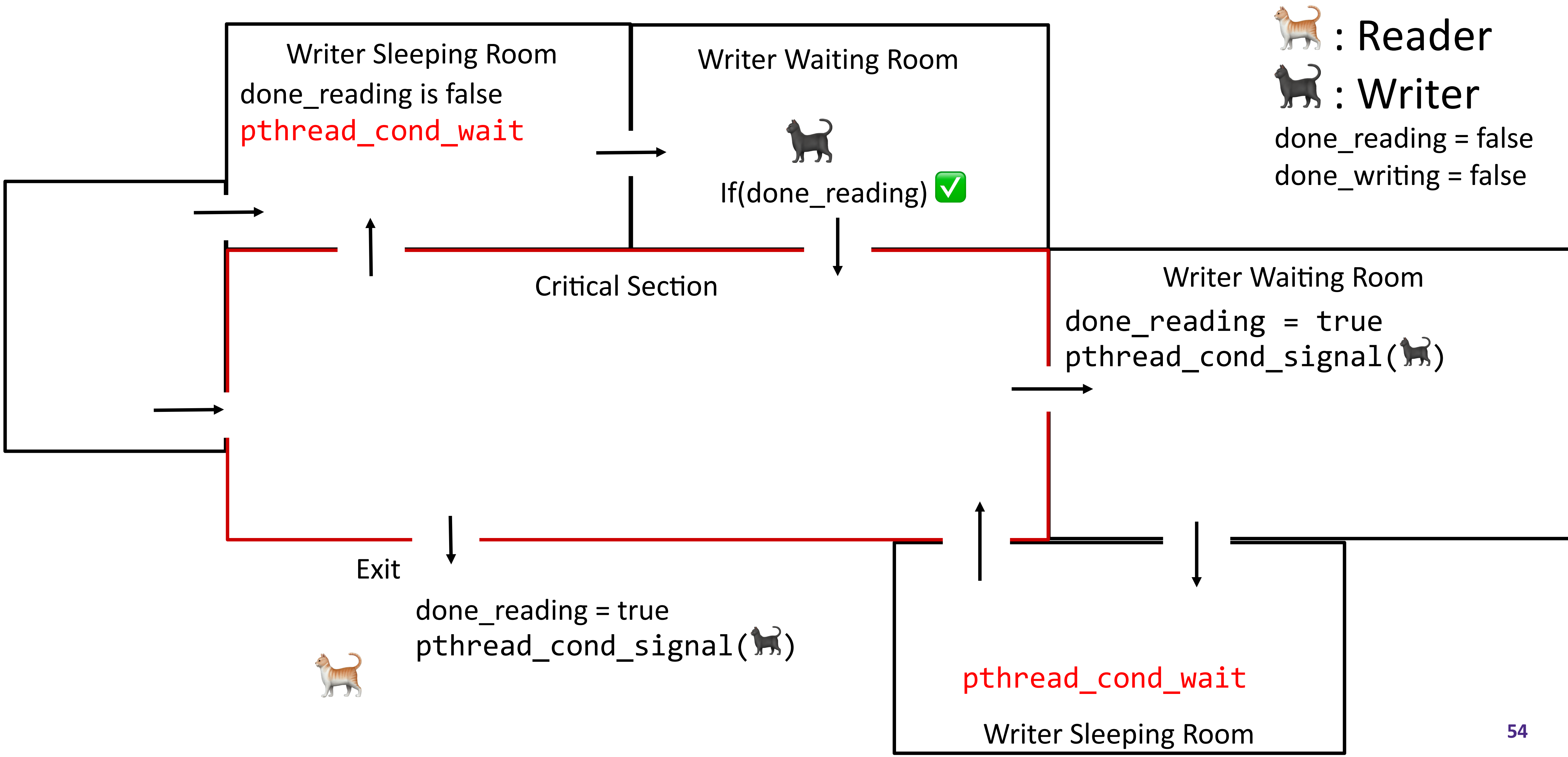
# Condition Variable & Mutex Visualization with our CAT!



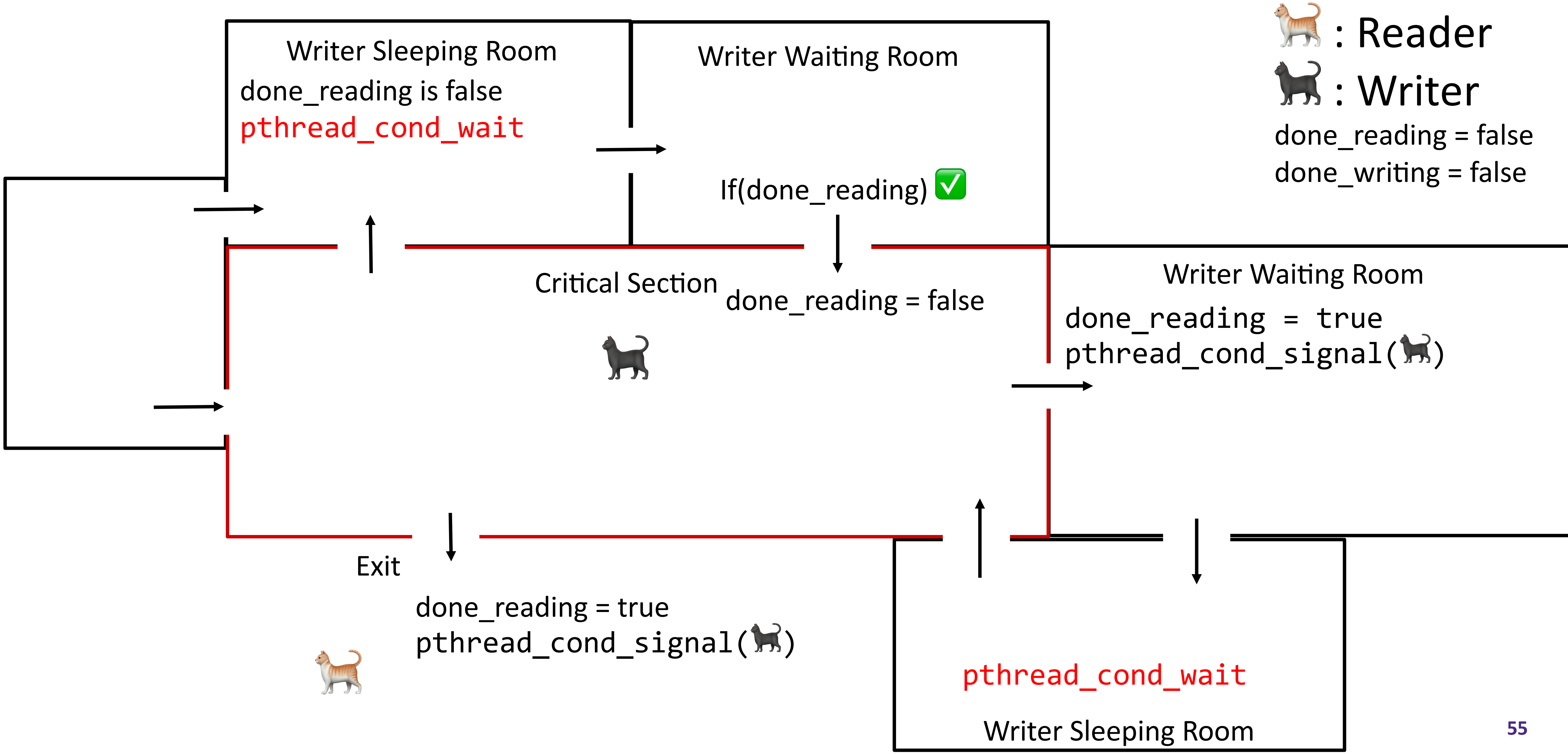
# Condition Variable & Mutex Visualization with our CAT!



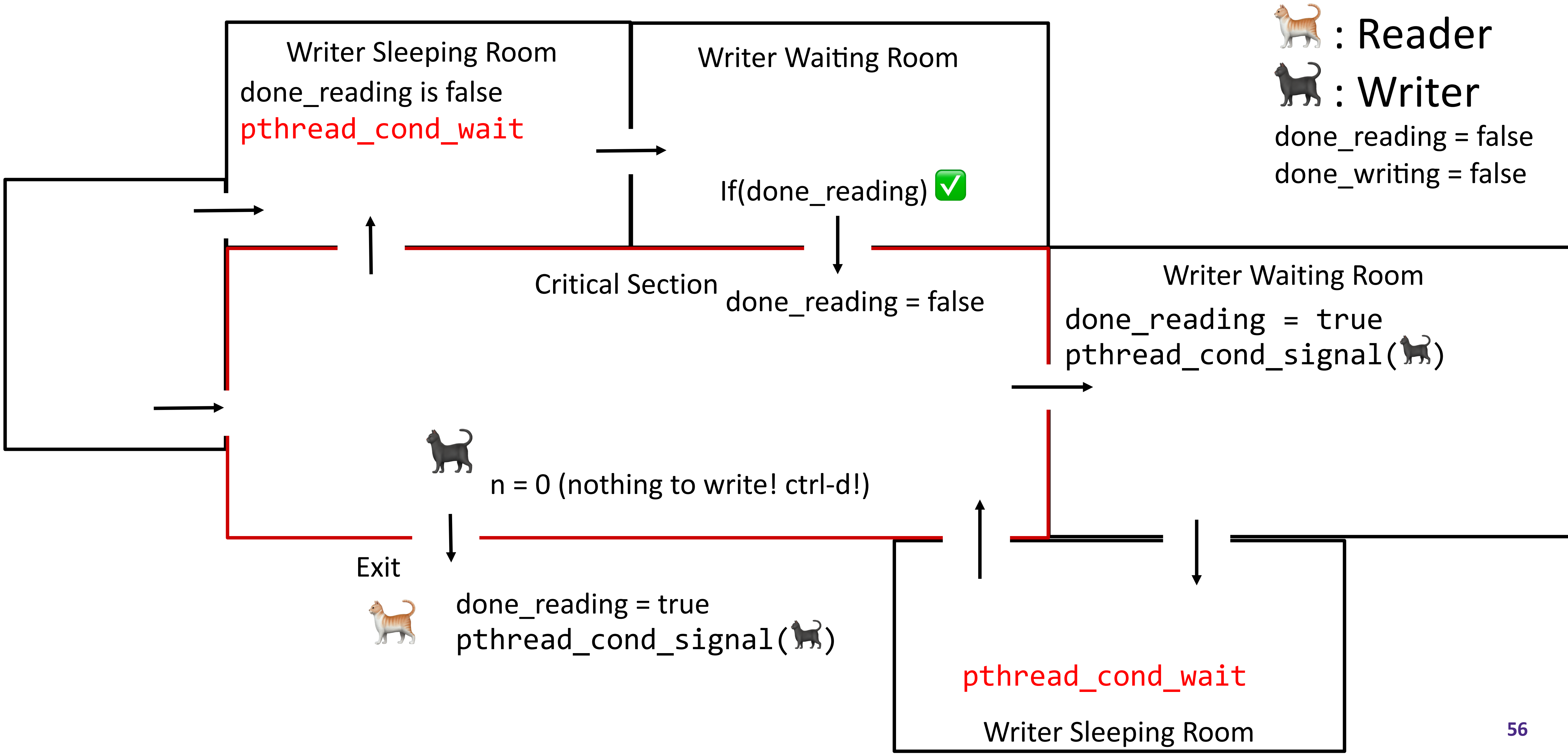
# Condition Variable & Mutex Visualization with our CAT!



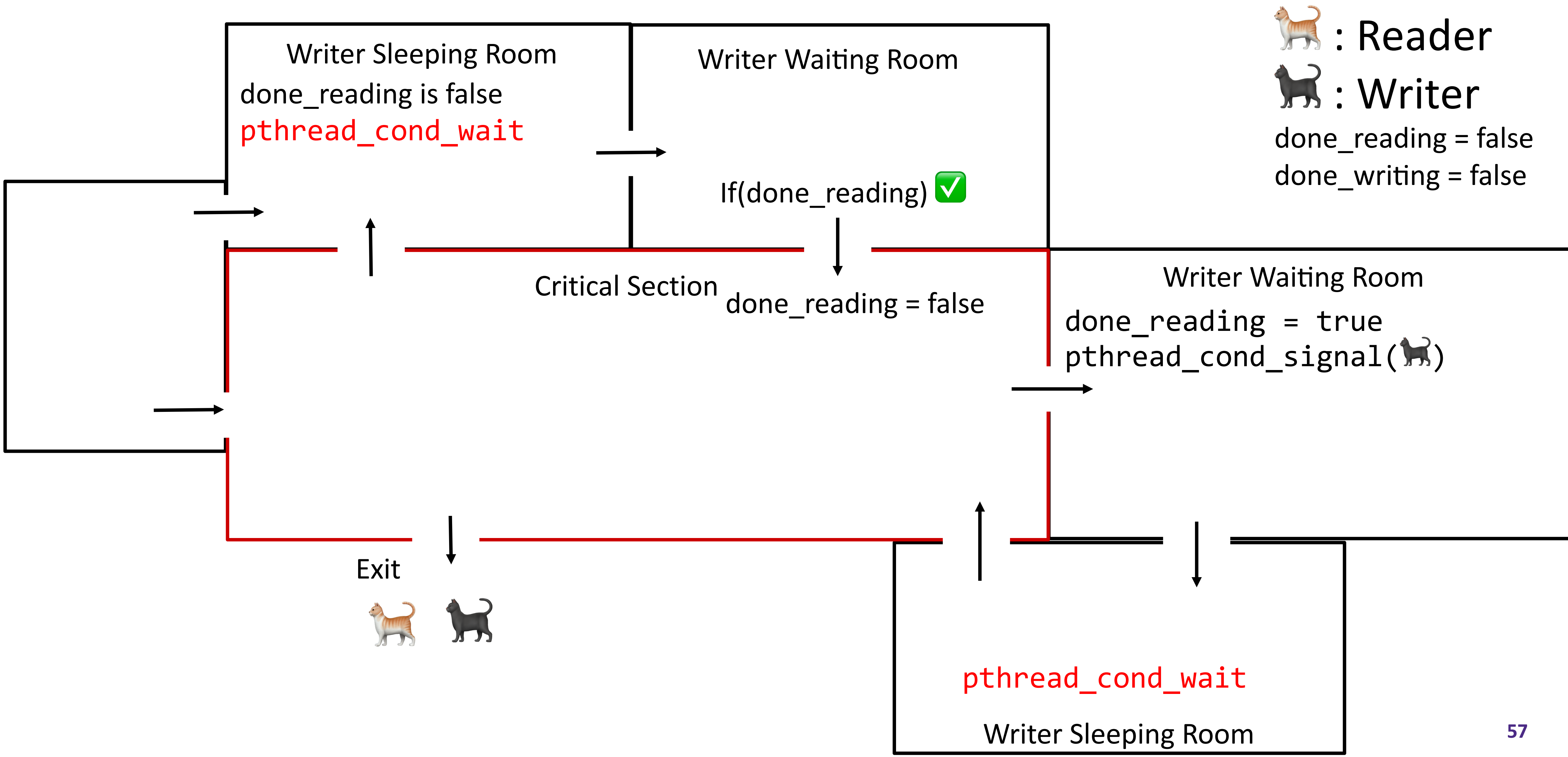
# Condition Variable & Mutex Visualization with our CAT!



# Condition Variable & Mutex Visualization with our CAT!



# Condition Variable & Mutex Visualization with our CAT!



# Demo: cat_cv.c

- Let’s check out the cat_cv implementation and see how it differs with utilization!

# Poll Everywhere

pollev.com/cis5480

- ***Are these while loops necessary?***

```c
while (true){
    n = read(STDIN_FILENO, buffer, BUF_SIZE);
    if (n == 0){
        break;
    }

    pthread_mutex_lock(&has_read);
    done_reading = true;
    pthread_mutex_unlock(&has_read);

    while(done_reading == true) pthread_cond_signal(&writer);

    pthread_mutex_lock(&has_written);
    while(!done_writing){
        pthread_cond_wait(&reader, &has_written);
    }
    done_writing = false;
    pthread_mutex_unlock(&has_written);
}
```

reader

```c
while(true){

    pthread_mutex_lock(&has_read);
    while(!done_reading){
        pthread_cond_wait(&writer, &has_read);
    }
    done_reading = false;
    pthread_mutex_unlock(&has_read);

    if (n > 0){
        write(STDOUT_FILENO, buffer, n);
    } else{
        break;
    }

    pthread_mutex_lock(&has_written);
    done_writing = true;
    pthread_mutex_unlock(&has_written);

    while(done_writing == true) pthread_cond_signal(&reader);
}
```

writer

# A Quick Introduction; Semaphores

- Condition Variable, Mutex, & Bool
  - All of these together, helped us create a rudimentary Semaphore.
  - Semaphores allow threads to either
    - Continue (The light is green)
    - Stop (The light is red)
- We made two Semaphores(0)
  - Where 0 means that ***no threads*** can pass.
  - Until the other threads set the bool to (1), then one thread passes.
  - That thread then takes that permission slip, and thus resets the semaphore to (0).
- Our Example uses a Boolean, but for multiple threads, you can use a signed integer.
  - Try to see if you can implement one yourself as practice!

# Lecture Outline

- Multithreaded Cat
  - Producer v Consumer
- Locks Only
- Locks + Boolean Flag
- Posix Condition Variables
- Amdahl's Law
  - How much can threads really help?
  - Live Mandelbrot Example

# Amdahl's Law

- For most algorithms, there are parts that parallelize well and parts that don't. This causes adding threads to have diminishing returns
  - (even ignoring the overhead costs of creating & scheduling threads)

- Consider we have some parallel algorithm $T_1 = 1$
  - The 1 subscript indicates this is run on 1 thread
  - we define the work for the entire algorithm as 1

- We define S as being the percentage that can be parallelized
  - $T_1 = S + (1 – S)$ *// (1-S) is the sequential part*

# Amdahl's Law

- For running on one thread:
  - $T_1 = (1 - S) + S$
- If we have P threads and perfect linear speedup on the parallelizable part, we get
  - $T_P = (1\text{-}S) + \frac{S}{P}$
- Speed up multiplier for P threads from sequential is:
  - $\frac{T_1}{T_p} = \frac{1}{1-S+\frac{S}{P}}$

# Amdahl's Law

- Let's say that we have 100000 threads (P = 100000) and our algorithm is only 2/3 parallel? (s = 0.6666..)
  - $\frac{T_1}{T_p} = \frac{1}{1-0.6666+\frac{0.6666}{100000}} = 2.9999\ times\ faster\ than\ sequential$

- What if it is 90% parallel? (S = 0.9):
  - $\frac{T_1}{T_p} = \frac{1}{1-0.9+\frac{0.9}{100000}} = 9.99\ times\ faster\ than\ sequential$

- What if it is 99% parallel? (S = 0.99):
  - $\frac{T_1}{T_p} = \frac{1}{1-0.99+\frac{0.99}{100000}} = 99.99\ times\ faster\ than\ sequential$

# Limitation: Hardware Threads

- These algorithms are limited by hardware.
- Number of Hardware Threads: The number of threads can genuinely run in parallel on hardware

- We may be able to create a huge number of threads, but only run a few (e.g. 4) in parallel at a time.

- Can see this information in with `lscpu` in bash
  - A computer can have some number of CPU sockets
  - Each CPU can have one or more cores
  - Each Core can run 1 or more threads

# If Time: Fun Example – Mandelbrot Sets

- Singly Threaded v Multithreaded (12 threads)

# That’s all!

- As practice, see if you can implement the writer/reader cat from scratch going from Spining Mutex Locks to Condition Variables!
  - This is great practice since it’s not straightforward to synchronize two threads.

# Poll Everywhere

discuss

- ***Exam style question: Is there a way for the writer to never write anything?***

```c
void *cat_read(void *arg){
     while (true){
         pthread_mutex_lock(&buf_lock);
         n = read(STDIN_FILENO, buffer, BUF_SIZE);
         if (n == 0){ // Ctrl-D
             pthread_mutex_unlock(&buf_lock);
             break;
         }
          pthread_mutex_unlock(&buf_lock);
      }
      pthread_mutex_lock(&done_lock);
      done = true;
      pthread_mutex_unlock(&done_lock);
      return NULL;
}
```

```c
void *cat_write(void *arg){
    while(true){
         pthread_mutex_lock(&done_lock);
         if (done) break;
         pthread_mutex_unlock(&done_lock);
         pthread_mutex_lock(&buf_lock);
         if (n > 0){
             write(STDOUT_FILENO, buffer, n);
         }
         pthread_mutex_unlock(&buf_lock);
    }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

- ***Yes. If the reader is always pre-empted by the writer when it has the buf_lock, then the writer will never be able to enter the write condition because it can never acquire the lock! (The writer will starve.)***

# Poll Everywhere

discuss

```
void *cat_read(void *arg){
     while (true){
         pthread_mutex_lock(&buf_lock);
         n = read(STDIN_FILENO, buffer, BUF_SIZE);
         if (n == 0){ // Ctrl-D
             pthread_mutex_unlock(&buf_lock);
             break;
         }
          pthread_mutex_unlock(&buf_lock);

          pthread_mutex_lock(&has_read);
          done_reading = true;
          pthread_mutex_unlock(&has_read);

      }
      pthread_mutex_lock(&done_lock);
      done = true;
      pthread_mutex_unlock(&done_lock);
      return NULL;
}
```

- ***What is missing here?***
  - **Although the writer now waits for the reader, we also need the reader to wait for the writer.**
  - **If not, the reader could proceed to the next line before the writer even has a chance to read from it.**
  - **Just like in the writer, we have a similar piece of code right here.**

# Poll Everywhere

discuss

```
void *cat_read(void *arg){
    while (true){
        pthread_mutex_lock(&buf_lock);
        n = read(STDIN_FILENO, buffer, BUF_SIZE);
        if (n == 0){ // Ctrl-D
            pthread_mutex_unlock(&buf_lock);
            break;
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_read);
        done_reading = true;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&has_written);
        while(!done_writing){
            pthread_mutex_unlock(&has_written);
            pthread_mutex_lock(&has_written);
        }
        done_writing = false;
        pthread_mutex_unlock(&has_written);

    }
    // done flag set down here.
}
```

- ***What is missing here?***
  - **Although the writer now waits for the reader, we also need the reader to wait for the writer.**
  - **If not, the reader could proceed to the next line before the writer even has a chance to read from it.**
  - **Just like in the writer, we have a similar piece of code right here.**
  - **And we also consume the permission to proceed, `done_writing`, setting it to false.**

# Poll Everywhere

pollev.com/cis5480

```
void *cat_write(void *arg){
    while(true){
         pthread_mutex_lock(&done_lock);
         if (done) break;
         pthread_mutex_unlock(&done_lock);

         pthread_mutex_lock(&has_read);
         while(!done_reading){
              pthread_mutex_unlock(&has_read);
              pthread_mutex_lock(&has_read);
         }
         done_reading = false;
         pthread_mutex_unlock(&has_read);

         pthread_mutex_lock(&buf_lock);
         if (n > 0){
             write(STDOUT_FILENO, buffer, n);
         }
         pthread_mutex_unlock(&buf_lock);

         }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

- ***What is missing here?***
  - **Now the writer needs to indicate to the reader that it is done writing.**

# Poll Everywhere

**pollev.com/cis5480**

```c
void *cat_write(void *arg){
    while(true){
        pthread_mutex_lock(&done_lock);
        if (done) break;
        pthread_mutex_unlock(&done_lock);

        pthread_mutex_lock(&has_read);
        while(!done_reading){
            pthread_mutex_unlock(&has_read);
            pthread_mutex_lock(&has_read);
        }
        done_reading = false;
        pthread_mutex_unlock(&has_read);

        pthread_mutex_lock(&buf_lock);
        if (n > 0){
            write(STDOUT_FILENO, buffer, n);
        }
        pthread_mutex_unlock(&buf_lock);

        pthread_mutex_lock(&has_written);
        done_writing = true;
        pthread_mutex_unlock(&has_written);
    }
    pthread_mutex_unlock(&done_lock);
    return NULL;
}
```

- ***What is missing here?***
  - **Now the writer needs to indicate to the reader that it is done writing.**

# Poll Everywhere

pollev.com/cis5480

- ***Are these while loops necessary?*** ***Not in this example.***

```c
while (true){
    n = read(STDIN_FILENO, buffer, BUF_SIZE);
    if (n == 0){
        break;
    }

    pthread_mutex_lock(&has_read);
    done_reading = true;
    pthread_mutex_unlock(&has_read);

    while(done_reading == true) pthread_cond_signal(&writer);

    pthread_mutex_lock(&has_written);
    while(!done_writing){
        pthread_cond_wait(&reader, &has_written);
    }
    done_writing = false;
    pthread_mutex_unlock(&has_written);
}
```

```c
while(true){

    pthread_mutex_lock(&has_read);
    while(!done_reading){
        pthread_cond_wait(&writer, &has_read);
    }
    done_reading = false;
    pthread_mutex_unlock(&has_read);

    if (n > 0){
        write(STDOUT_FILENO, buffer, n);
    } else{
        break;
    }

    pthread_mutex_lock(&has_written);
    done_writing = true;
    pthread_mutex_unlock(&has_written);

    while(done_writing == true) pthread_cond_signal(&reader);
}
```

While loops like these are used when you are worried that a thread will reach a wait, before it can be signaled, meaning it will miss the signal!

# Poll Everywhere

pollev.com/cis5480

## Are these while loops necessary? Not in this example.

```
while (true){
    n = read(STDIN_FILENO, buffer, BUF_SIZE);
    if (n == 0){
        break;
    }

    pthread_mutex_lock(&has_read);
    done_reading = true;
    pthread_mutex_unlock(&has_read);

    while(done_reading == true) pthread_cond_signal(&writer);

    pthread_mutex_lock(&has_written);
    while(!done_writing){
        pthread_cond_wait(&reader, &has_written);
    }
    done_writing = false;
    pthread_mutex_unlock(&has_written);
}
```

```
while(true){

    pthread_mutex_lock(&has_read);
    while(!done_reading){
        pthread_cond_wait(&writer, &has_read);
    }
    done_reading = false;
    pthread_mutex_unlock(&has_read);

    if (n > 0){
        write(STDOUT_FILENO, buffer, n);
    } else{
        break;
    }

    pthread_mutex_lock(&has_written);
    done_writing = true;
    pthread_mutex_unlock(&has_written);

    while(done_writing == true) pthread_cond_signal(&reader);
}
```

If the reader is done_reading, then it will set it to true! If the writer has acquired the lock, then it must have already been suspended before the reader could set the variable. Thus, the *signal* will be received! If the reader sets it before the writer sleeps, then the writer will not even call `_cond_wait`.

# Extra Things that you won't be tested on.

# Parallel Algorithms

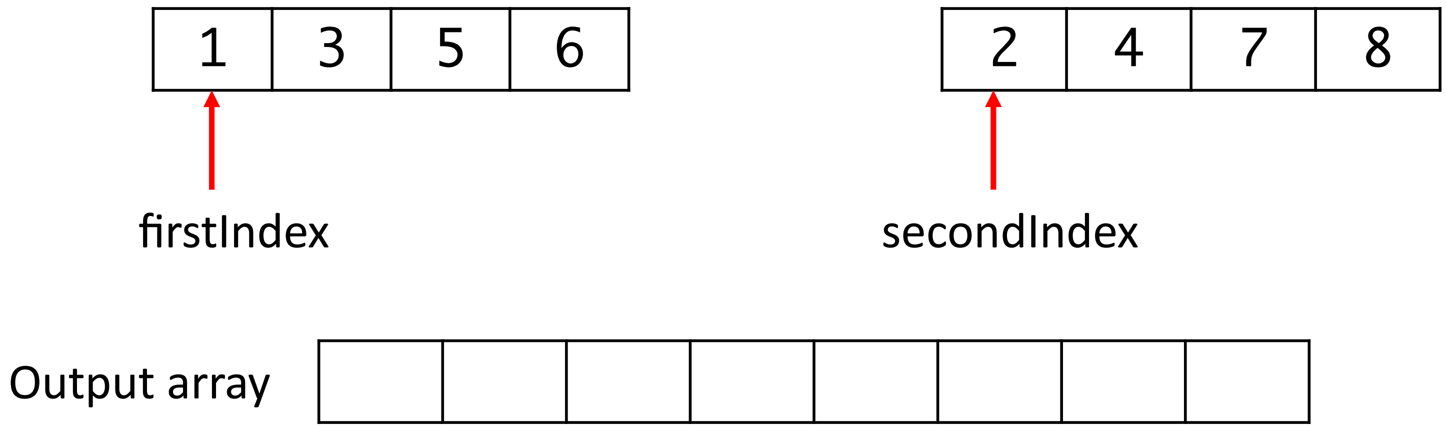
- One interesting applications of threads is for faster algorithms

- Common Example: Merge sort

# Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

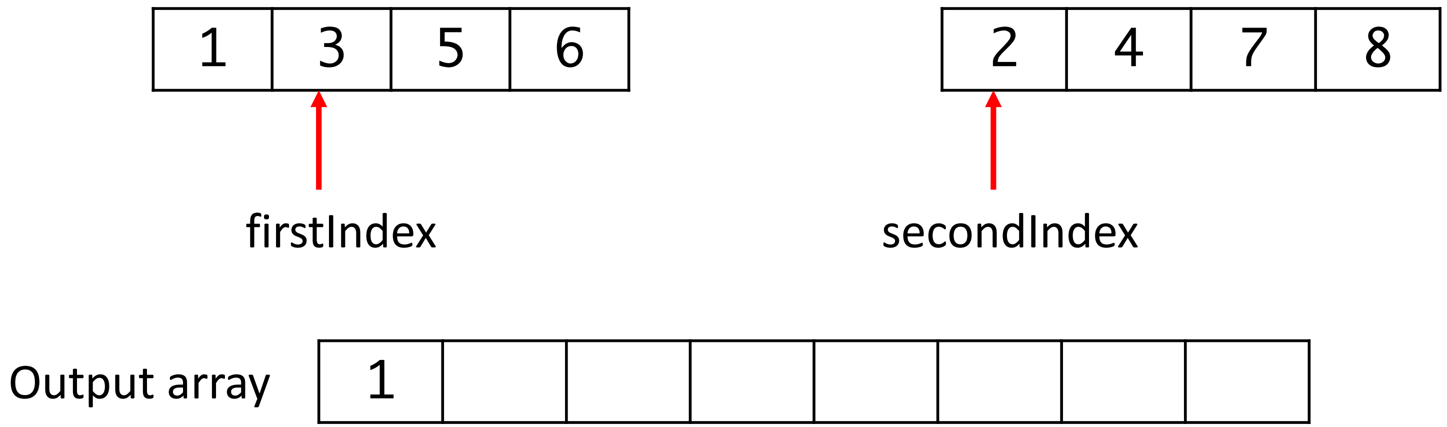
  - Consider the two sorted arrays:



# Merge Sort: Core Ideas

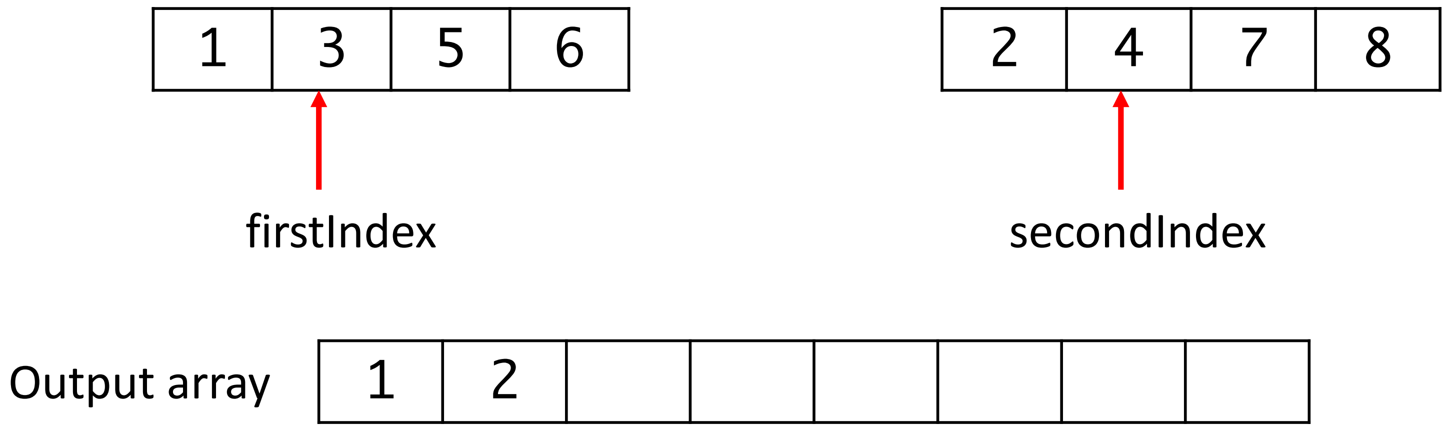
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



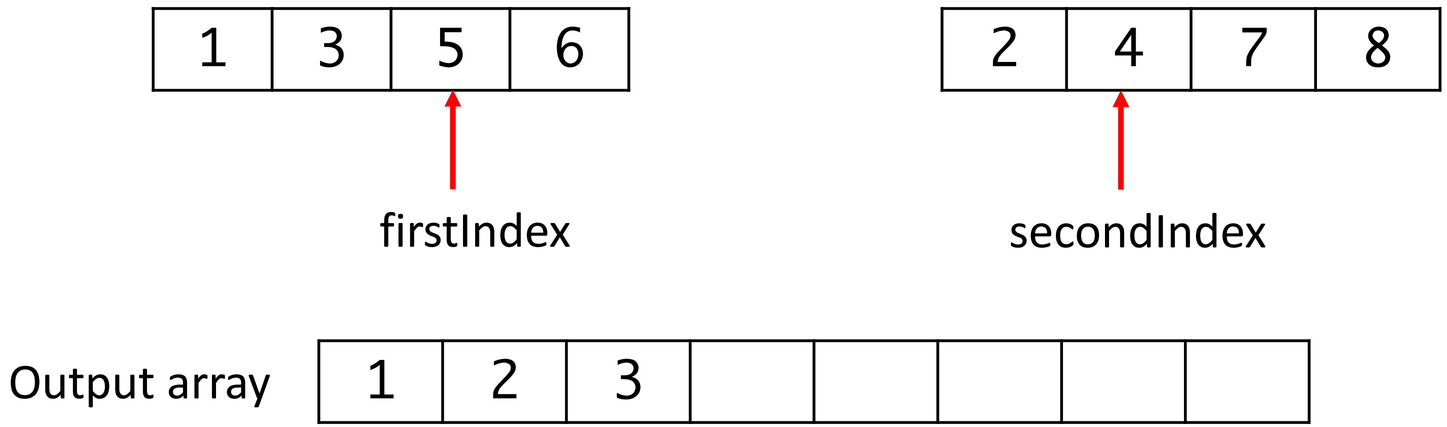
# Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



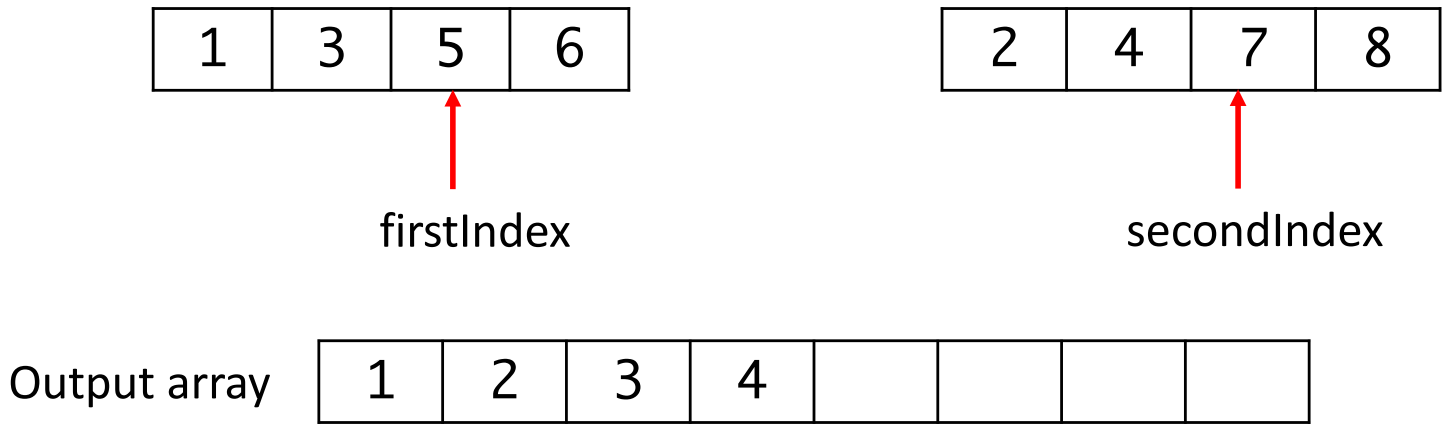
# Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



# Merge Sort: Core Ideas

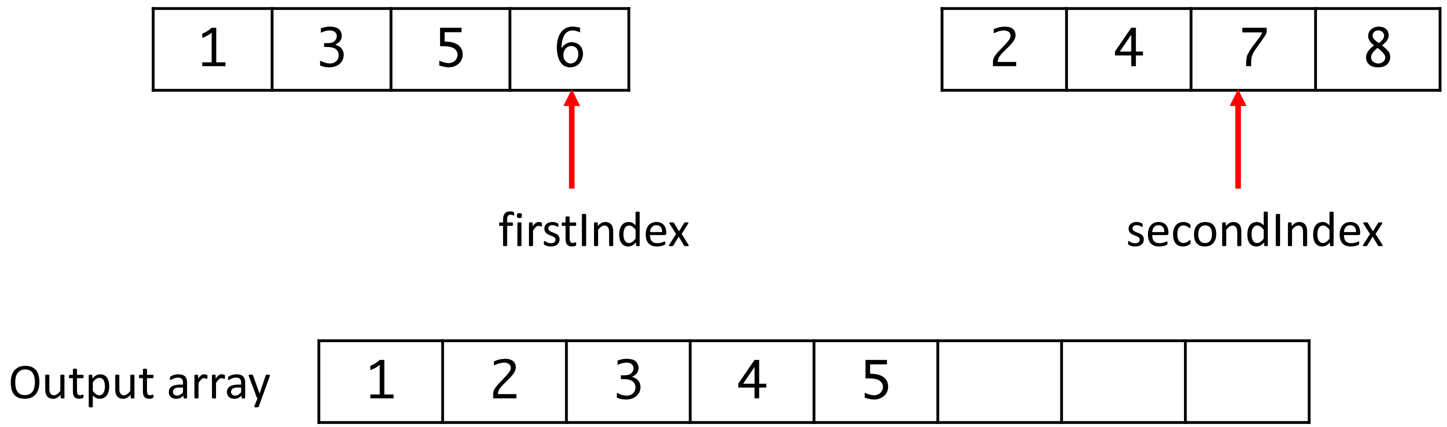
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
    - Consider the two sorted arrays:



# Merge Sort: Core Ideas

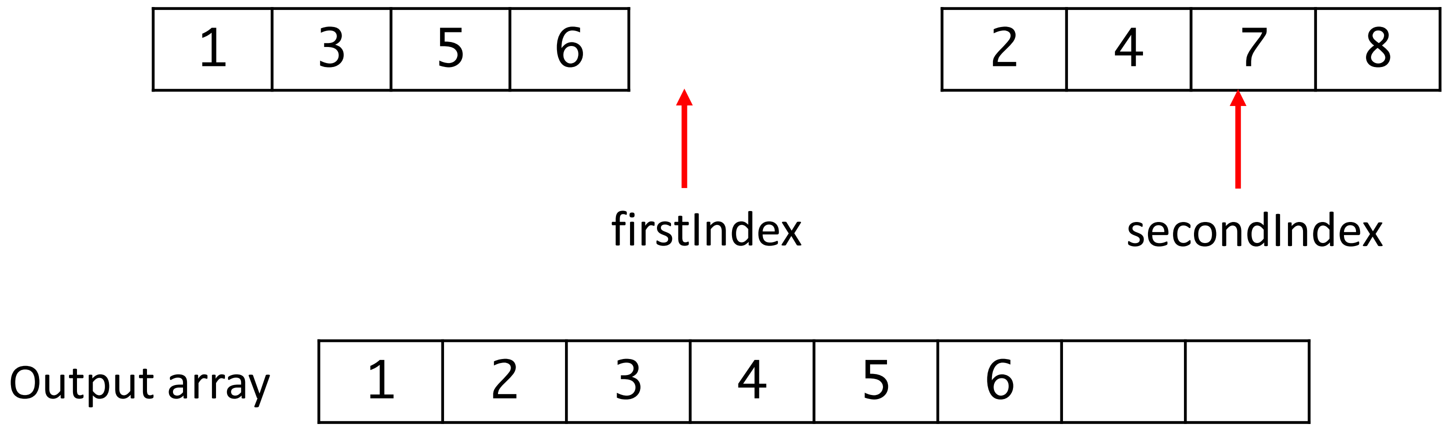
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



# Merge Sort: Core Ideas

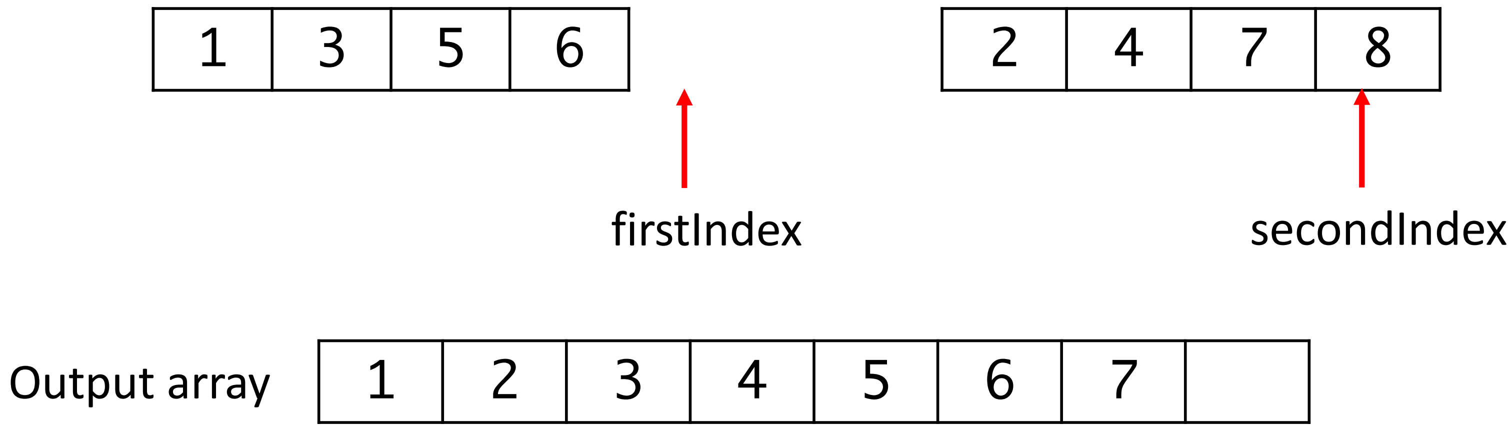
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



# Merge Sort: Core Ideas

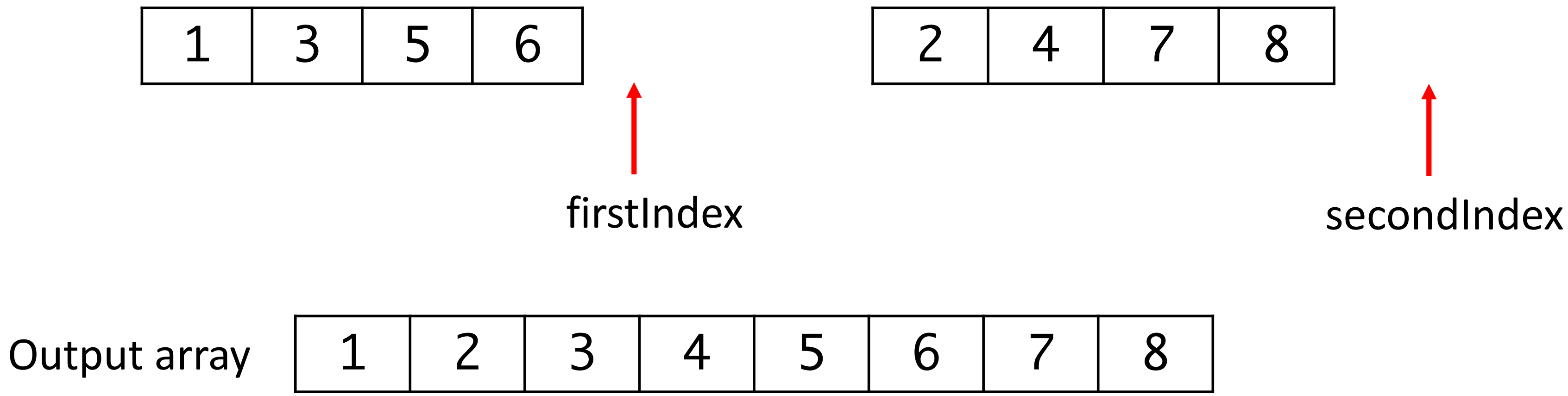
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



# Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



# Merge Sort: High Level Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 20 | 10 | 15 | 54 | 55 | 11 | 78 | 14 |

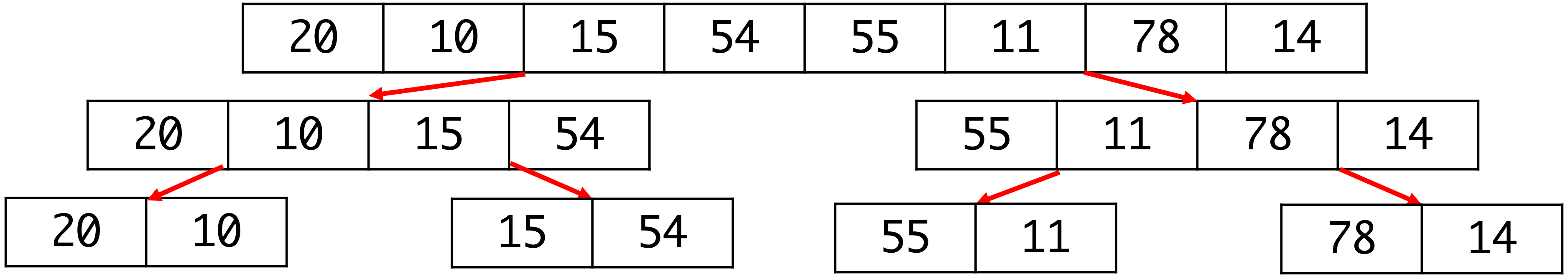
# Merge Sort: High Level Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 20 | 10 | 15 | 54 | 55 | 11 | 78 | 14 |

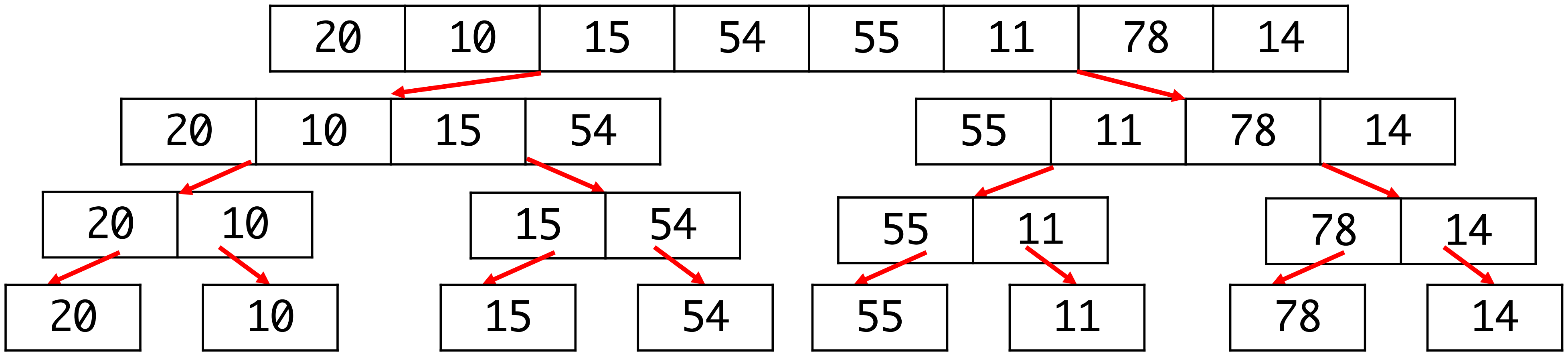
| | | | |
|---|---|---|---|
| 20 | 10 | 15 | 54 |

| | | | |
|---|---|---|---|
| 55 | 11 | 78 | 14 |

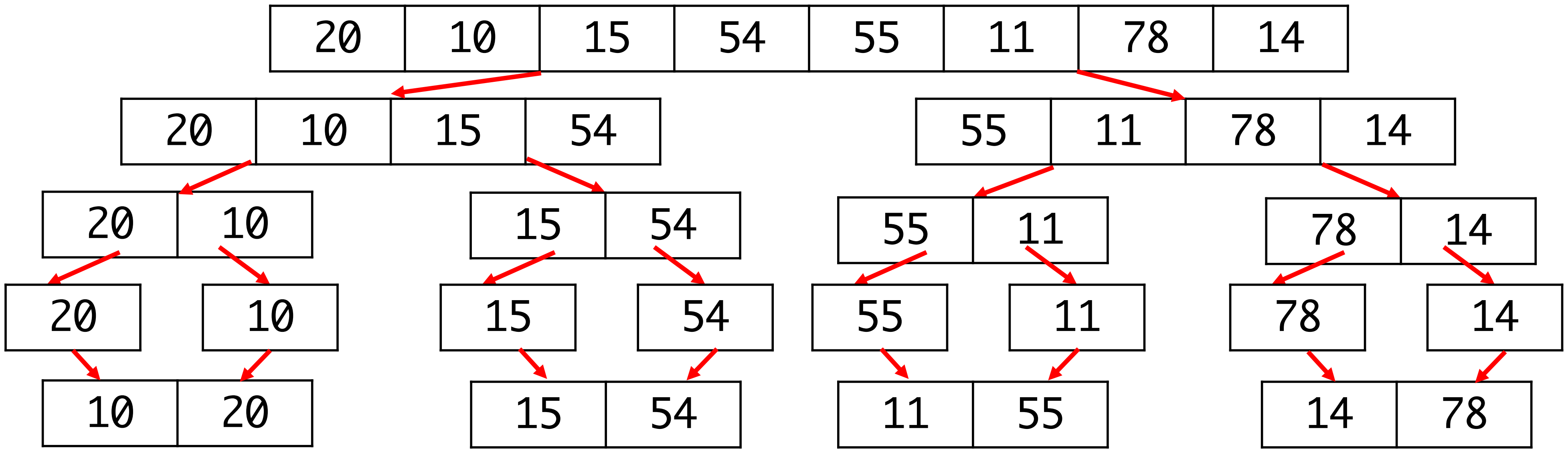
# Merge Sort: High Level Example



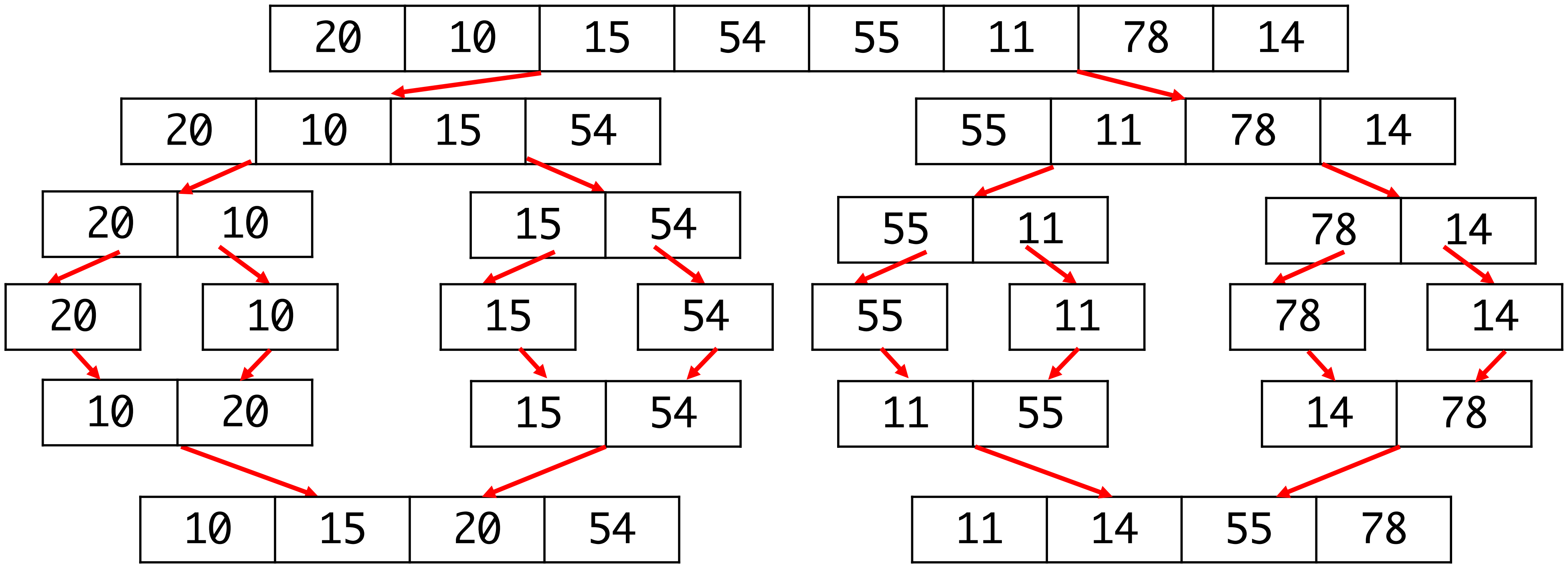
# Merge Sort: High Level Example



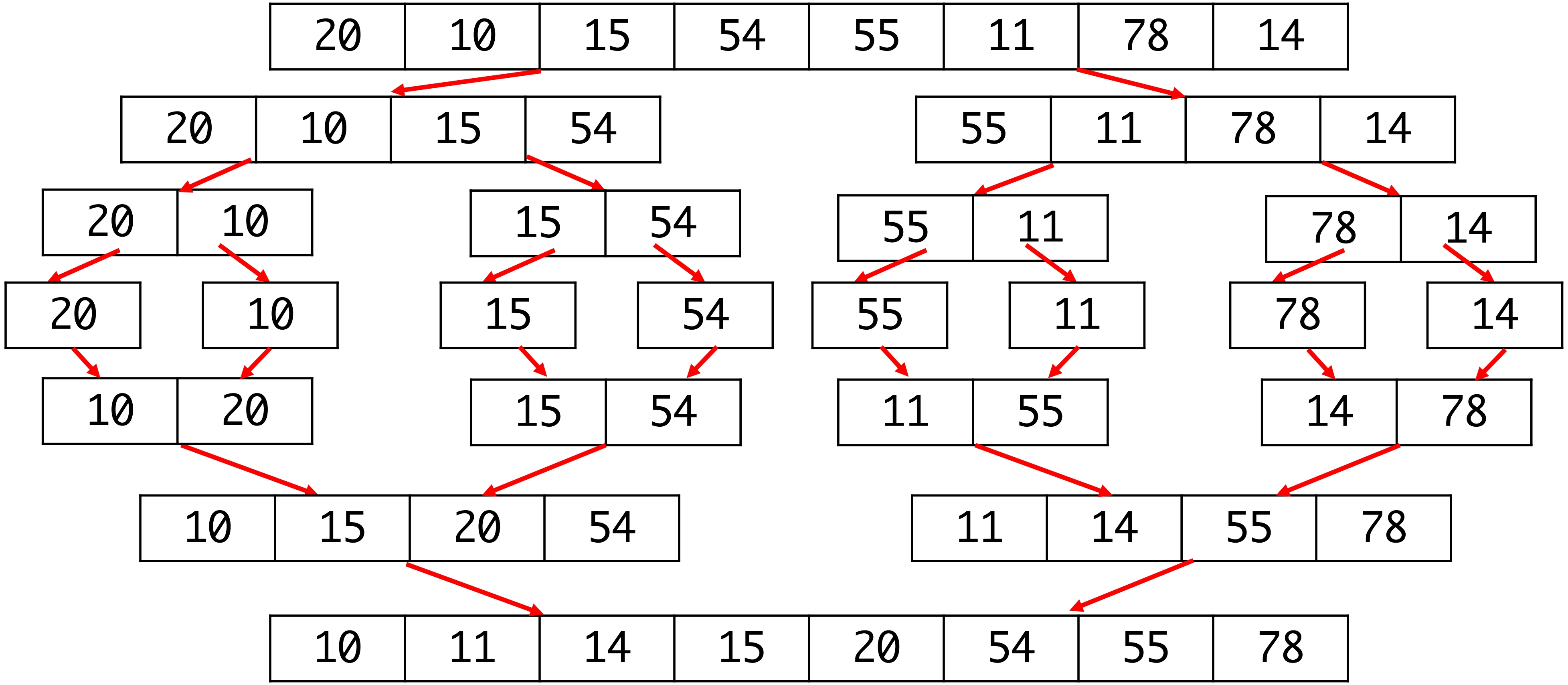
# Merge Sort: High Level Example



# Merge Sort: High Level Example



# Merge Sort: High Level Example



# Merge Sort Algorithmic Analysis

- Algorithmic analysis of merge sort gets us to O(n * log(n)) runtime.

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;
  merge_sort(arr, lo, mid);  // sort the bottom half
  merge_sort(arr, mid, hi);  // sort the upper half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

- We recurse $\log_2(N)$ times, each recursive "layer" does O(N) work

# Merge Sort Algorithmic Analysis

- We can use threads to speed this up:

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;

  // sort bottom half in parallel
  pthread_create(merge_sort(arr, lo, mid));
  merge_sort(arr, mid, hi);  // sort the upper half

  pthread_join(); // join the thread that did bottom half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

  - Now we are sorting both halves of the array in parallel!

# Poll Everywhere

pollev.com/cis5480

- We can use threads to speed this up:

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;

  // sort bottom half in parallel
  pthread_create(merge_sort(arr, lo, mid));
  merge_sort(arr, mid, hi);  // sort the upper half

  pthread_join(); // join the thread that did bottom half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

  - Now we are sorting both halves of the array in parallel!
  - **How long does this take to run?**
  - **How much work is being done?**

# Parallel Algos:

**Will not test you on this**

- We can define **T(n)** to be the running time of our algorithm
- We can split up our work between two parts, the part done sequentially, and the part done in parallel
  - T(n) = sequential_part + parallel_part
  - T(n) = O(n) *merging* + T(n/2) *sort half the array*
    - This is a recursive definition
- If we start recurring…
  - T(n) = O(n) + O(n/2) + T(n/4)
  - T(n) = O(n) + O(n/2) + O(n/4) + T(n/8)

# Parallel Algos:

**Will not test you on this**

- If we start recurring…
  - T(n) = O(n) + O(n/2) + T(n/4)
  - T(n) = O(n) + O(n/2) + O(n/4) + T(n/8)
  - …
  - Eventually we stop, there is a limit to the length of the array.
    And we can say an array of size 1 is already sorted, so T(1) = O(1)

- This approximates to T(n) = ~2 * O(n) = O(n)
  - This parallel merge sort is O(n), but there are further optimizations that can be done to reach ~O(log(n))

- There is a lot more to parallel algo analysis than just this, I am just giving you a sneak peek