

Lecture 05:

Condvars and Semaphores

CS343 – Operating Systems
Branden Ghen a – Fall 2025

Some slides borrowed from:
Stephen Tarzia (Northwestern), and Shivaram Venkataraman (Wisconsin)

Administrivia

- PCLab is out and ready to work on
 - Some of this week's material is relevant
 - But you can totally get started right now
- This lab is pretty straightforward
 - Unless something goes wrong, in which case it can be very hard to debug
 - Start it early enough to have time to fix problems

Today's Goals

- Understand how we can apply locks to gain correctness and maintain performance
 - Counter
- Signaling between threads to enforce ordering
 - Condition Variables
 - Semaphores

Outline

- **Amdahl's Law**
- Applying Locks
- Ordering with Condition Variables
- Semaphores

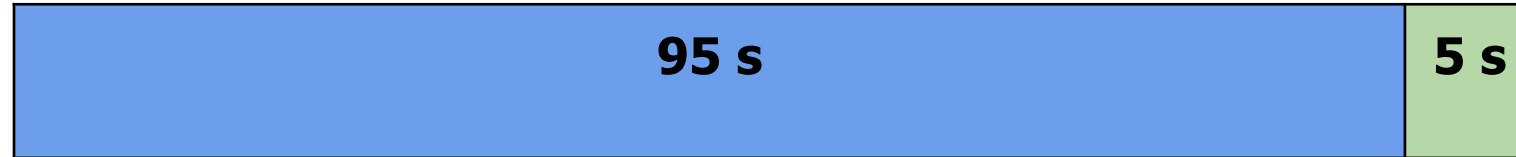
Challenges to concurrency

Concurrency is great! We can do so many things!!

But what's the downside...?

- 1. How much speedup can we get from it?**
2. How hard is it to write parallel programs? (data races)

Speedup Example



Imagine a program that takes 100 seconds to run

- 95 seconds in the blue part
- 5 seconds in the green part

Speedup from improvements



$$\text{Speedup with Improvement} = \frac{\text{Execution time without improvement}}{\text{Execution time with improvement}}$$

$$5 \text{ s} \rightarrow 2.5 \text{ s: Speedup} = 100/97.5 = 1.026$$

$$5 \text{ s} \rightarrow 1 \text{ s: Speedup} = 100/96 = 1.042$$

$$5 \text{ s} \rightarrow 0.001\text{s: Speedup} = 100/95.001 = 1.053$$

The impact of a performance improvement is relative to the importance of the part being improved!

Amdahl's Law

Equivalent to
prior equation

$$\text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

Not improved part $\rightarrow$ Improved part

F = Fraction of execution time speed up

S = Scale of improvement

Example: 2x improvement to 25% of the program

$$\frac{1}{0.75 + \frac{0.25}{2}} = \frac{1}{0.75 + 0.125} = 1.14$$

Parallel speedup example

$$\text{Speedup with improvement} = \frac{1}{(1 - F) + (F/S)}$$

- Consider an improvement which runs 20 times faster but is only usable 15% of the time

$$\text{Speedup with improvement} = \frac{1}{(0.85) + (0.15/20)} = 1.166$$

- What if it's usable 25% of the time?

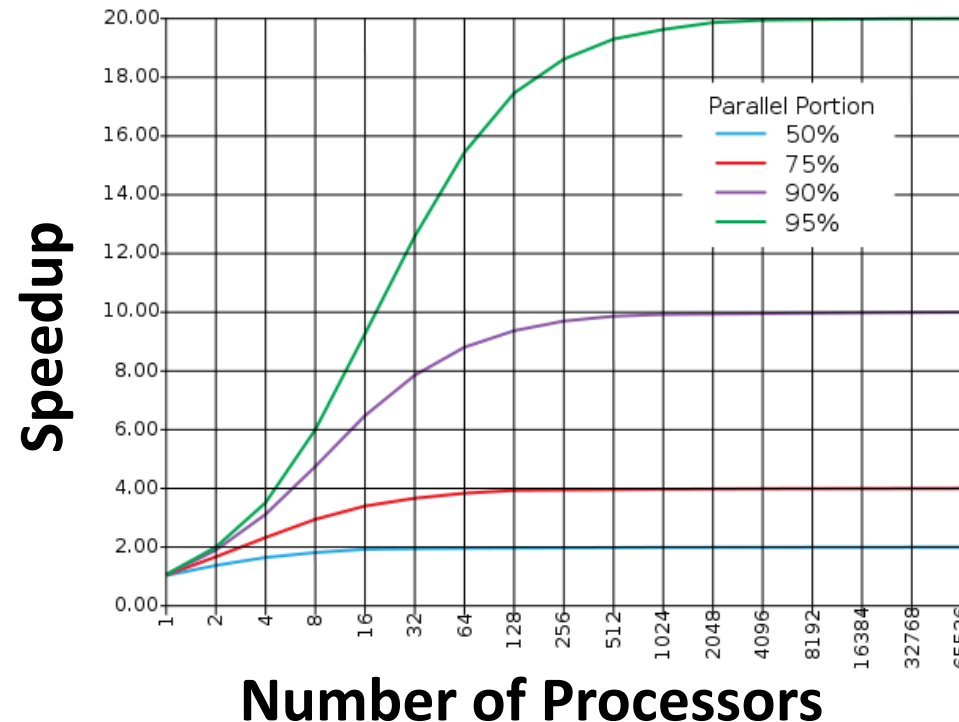
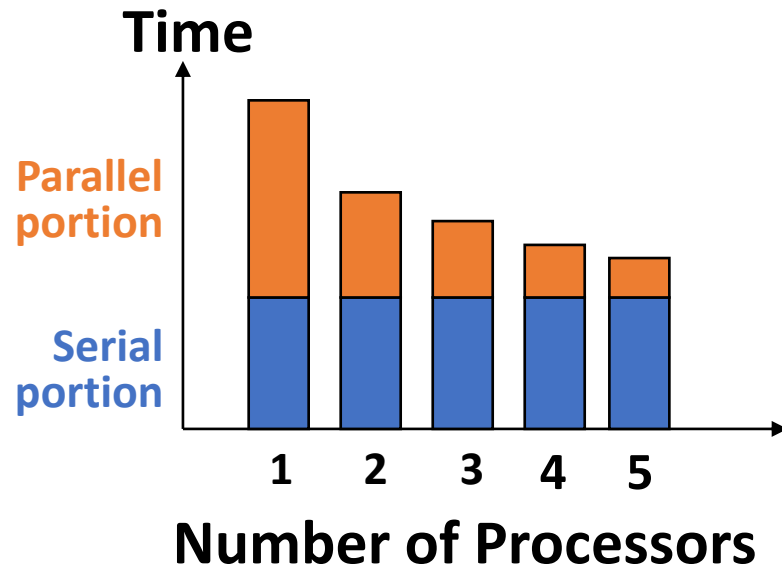
$$\text{Speedup with improvement} = \frac{1}{(0.75) + (0.25/20)} = 1.311$$

**Nowhere near
20x speedup!**



Amdahl's (heartbreaking) Law (in pictures)

- The amount of speedup that can be achieved through parallelism is limited by the non-parallel portion of your program!
 - And every program has at least *some* non-parallel parts



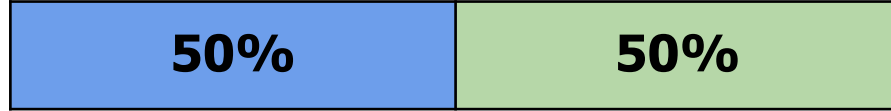
Amdahl's (heartbreaking) Law (in words)

- Amdahl's Law tells us that to achieve linear speedup with more processors:
 - *none* of the original computation can be serial (non-parallelizable)
- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

$$\text{Speedup} = 1/(\text{.001} + \text{.999}/100) = 90.99$$

Break + Question

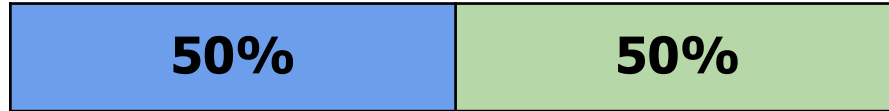
$$\text{Speedup with improvement} = \frac{1}{(1 - F) + (F/S)}$$



- Suppose a program spends 50% of its time in a `square root` routine.
- How much must you speed up `square root` to make the program run 2x faster?

- (A) 10
- (B) 20
- (C) 100
- (D) None of the above

Break + Question



$$\text{Speedup with improvement} = \frac{1}{(1 - F) + (F/S)}$$

- Suppose a program spends 50% of its time in a `square root` routine.
- How much must you speed up `square root` to make the program run 2x faster?

$$\text{Speedup} = 1 / [(1 - F) + (F/S)]$$

$$2 = 1 / [(1 - 0.5) + (0.5/S)]$$

$$S = 0.5 / ((1/2) - 0.5) = \infty$$

(A) 10

(B) 20

(C) 100

(D) None of the above

The square root would need to decrease to nothing before you got 2x speedup

Outline

- Amdahl's Law
- **Applying Locks**
- Ordering with Condition Variables
- Semaphores

Review: Locks/Mutexes

- Simple mutual exclusion primitive
- Init(), Acquire()/Lock(), Release()/Unlock()
- Implementations trade complexity, fairness, and performance
 - Spinlocks
 - Ticket locks
 - Yielding locks
 - Queueing locks
- Sophisticated locks are more fair and do not waste processor time “busy waiting”
 - But also have unnecessary context-switch overhead if the lock is only briefly and rarely held

Spinning versus Blocking

- Each approach is better under different circumstances
- **Single core systems**
 - If waiting process is scheduled, then process holding lock is not
 - Waiting process should *a/ways* yield its time
- **Multicore systems**
 - If waiting process is scheduled, then process holding lock could also be
 - Spin or block depends how long until the lock is released
 - If the lock is released quickly, spin wait
 - If the lock is released slowly, block
 - Where quick and slow are relative to context-switch cost

Review: Need to enforce mutual exclusion on critical sections

```
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;
static const int LOOPS = 1e9;

void* mythread(void* arg) {
    printf("%s: begin\n", (char*)arg);
    for (int i=0; i<LOOPS; i++) {
        counter++;
    }
    printf("%s: done\n", (char*)arg);
    return NULL;
}
```

```
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // wait for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d, goal was %d)\n", counter, 2*LOOPS);
    return 0;
}
```

Broken concurrency can actually hurt performance too!

When iterating	Single-threaded counter: 3.850 seconds
one billion times:	Multithreaded no-lock counter: 4.700 seconds (Broken!)

- Why is the no-lock multithreaded version slower?
 - Not 100% certain
 - Likely something to do with hardware memory/cache consistency
 - Processor has to keep reloading value into individual thread's caches from memory

Naively locked counter example

```
static volatile int counter = 0;
static const int LOOPS = 1e9;
static pthread_mutex_t lock;

void* mythread(void* arg) {
    printf("%s: begin\n", (char*)arg);
    for (int i=0; i<LOOPS; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    printf("%s: done\n", (char*)arg);
    return NULL;
}
```

```
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    pthread_mutex_init(&lock, 0);
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // wait for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d, goal
was %d)\n", counter, 2*LOOPS);
    return 0;
}
```

Problem: locking overhead decreases performance

When iterating one billion times:	Single-threaded counter:	3.850 seconds
	Multithreaded no-lock counter:	4.700 seconds (Broken!)
	Naïve-locked counter:	80.000 seconds (Correct...)

- Formerly loop contained 3 instructions (mov, add, mov)
- Now it has
 - Two function calls
 - Multiple instructions inside of those
 - Possibly even interaction with the OS...
 - 3 instructions -> 60 instructions

Simple mutual exclusion: one big lock

- Simple solution “one big lock”
 - Find all the function calls that interact with shared memory
 - Lock at the start of each function call and unlock at the end
- Essentially, no concurrent access
 - Correct but poor performance
 - If you’ve forgotten all of this years from now, “one big lock” will still work

Counter example with big lock technique

code posted with last
lecture on canvas

```
static volatile int counter = 0;
static const int LOOPS = 1e9;
static pthread_mutex_t lock;

void* mythread(void* arg) {
    pthread_mutex_lock(&lock);
    printf("%s: begin\n", (char*)arg);
    for (int i=0; i<LOOPS; i++) {
        counter++;
    }
    printf("%s: done\n", (char*)arg);
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

```
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    pthread_mutex_init(&lock, 0);
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // wait for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d, goal  
was %d)\n", counter, 2*LOOPS);
    return 0;
}
```

Problem: locking decreases performance

When iterating
one billion times:

Single-threaded counter: 3.850 seconds

Multithreaded no-lock counter: 4.700 seconds (Broken!)

Naïve-locked counter: 80.000 seconds

Big lock counter: 3.895 seconds

- Big lock technique basically returned us to single-threaded execution time (and single-threaded implementation)
 - But non-critical section code could still run in parallel

Reducing lock overhead

- We want to enable parallelism, but deal with less lock overhead
 - Need to increase the amount of work done when not locked
 - Goal: lots of parallel work per lock/unlock event
- “Sloppy” updates to global state
 - Keep local state that is operated on
 - Occasionally synchronize global state with current local state
- Counter example
 - Keep a local counter for each thread (not shared memory)
 - Add local counter to global counter periodically

Sloppy counter example

code posted with last
lecture on canvas

```
static volatile int counter = 0;
static const int LOOPS = 1e9;
static pthread_mutex_t lock;

void* mythread(void* arg) {
    int sloppy_count = 0;
    printf("%s: begin\n", (char*)arg);
    for (int i=0; i<LOOPS; i++) {
        sloppy_count++;
        if (i%1000 == 0) {
            pthread_mutex_lock(&lock);
            counter += sloppy_count;
            pthread_mutex_unlock(&lock);
            sloppy_count = 0;
        }
    }
}
```

```
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    pthread_mutex_init(&lock, 0);
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // wait for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d, goal  
was %d)\n", counter, 2*LOOPS);
    return 0;
}
```

Offscreen Tail condition: don't forget to update
"counter" again when the for loop is complete!

Problem: locking decreases performance

When iterating
one billion times:

Single-threaded counter:	3.850 seconds
Multi-threaded no-lock counter:	4.700 seconds (Broken!)
Naïve-locked counter:	80.000 seconds

Big lock counter: 3.895 seconds

Sloppy lock (synchronize every 100):	2.150 seconds
Sloppy lock (synchronize every 10000):	1.472 seconds
Sloppy lock (synchronize every 1000000):	1.478 seconds
Sloppy lock (synchronize every 1000000000):	1.500 seconds

- Optimal for this counter example will be synchronizing once, when entirely finished with the local sum

Break + Open Question

- Avoiding data races is challenging
- Synchronization means we're running some code in serial anyways
- **Is concurrency worth it? What kinds of problems work best?**

Break + Open Question

- Avoiding data races is challenging
- Synchronization means we're running some code in serial anyways
- **Is concurrency worth it? What kinds of problems work best?**
 - Problems that do not share data will still be HUGE wins!
 - No (or few) data races. Big concurrency performance gains.
 - Such problems are termed: *embarrassingly parallel*
 - https://en.wikipedia.org/wiki/Embarrassingly_parallel#Examples

Outline

- Amdahl's Law
- Applying Locks
- **Ordering with Condition Variables**
- Semaphores

Requirements for sensible concurrency

- **Mutual exclusion**

- Prevents corruption of data manipulated in critical sections
- Atomic instructions → Locks → Concurrent data structures

- **Ordering** (B runs after A)

- By default, concurrency leads to a lack of control over ordering
- We can use mutex'd variables to control ordering, but it's inefficient:
 - `while(!myTurn) sleep(1);`
- We would like cooperating threads to be able to signal each other.
 - Park/unpark and futex could be used solve this problem
 - But we want a higher-level abstraction

Barriers for all-or-nothing synchronization

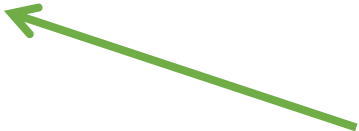
- Barriers create synchronization points in the program
 - **All** threads must reach barrier before **any** thread continues
- `pthread_barrier_init(barrier_t)`
- `pthread_barrier_wait(barrier_t)`
- Example use case: neural network processing
 - Spawn a pool of threads
 - Each thread handles a portion of the input data
 - Collect results from all threads at the end of the layer
 - Distribute results to appropriate threads for next layer

Basic Signaling with Condition Variable (condvar)

- Queue of waiting threads
 - Combine with a **flag** and a **mutex** to synchronize threads
- wait(condvar_t, lock_t)
 - Lock must be held when wait() is called
 - Puts the caller to sleep AND releases lock (atomically)
 - When awoken, reacquires lock before returning
- signal(condvar_t)
 - Wake a single waiting thread (if any are waiting)
 - Do nothing if there are no waiting threads
 - Called while holding the lock
 - (but the newly woken thread won't leave their wait() until they get the lock)

Waiting for a thread to finish

```
pthread_t p1, p2;  
  
// create child threads  
pthread_create(&p1, NULL, mythread, "A");  
pthread_create(&p2, NULL, mythread, "B");  
  
...  
  
// join waits for the child threads to finish  
thr_join(p1, NULL);  
thr_join(p2, NULL);  
  
return 0;
```



How to implement
join?

CV for child wait

- Must use mutex to protect "done" flag and condvar
 - Done flag tracks the event
 - Condvar is used for ordering
- Mutex protects both!

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

CV for child wait

- Must use mutex to protect "done" flag and condvar
- **Parent** calls thr_join()
 - wait()'s until done==1

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

CV for child wait

- Must use mutex to protect "done" flag and condvar
- **Parent** calls thr_join()
 - wait()'s until done==1
- **Child** calls thr_exit()
 - sets done to 1
 - calls signal()
 - unlocks mutex

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

Check your understanding: why doesn't this work?

Incorrect Code

Child	1	void thr_exit() {
	2	Pthread_mutex_lock(&m);
	3	Pthread_cond_signal(&c);
	4	Pthread_mutex_unlock(&m);
	5	}
Parent	6	
	7	void thr_join() {
	8	Pthread_mutex_lock(&m);
	9	Pthread_cond_wait(&c, &m);
	10	Pthread_mutex_unlock(&m);
	11	}

Correct Code

5	void thr_exit() {
6	Pthread_mutex_lock(&m);
7	done = 1;
8	Pthread_cond_signal(&c);
9	Pthread_mutex_unlock(&m);
10	}
18	void thr_join() {
19	Pthread_mutex_lock(&m);
20	while (done == 0)
21	Pthread_cond_wait(&c, &m);
22	Pthread_mutex_unlock(&m);
23	}

Consider if an ordering exists that would lead to incorrect behavior

- Lock means that only one critical section will run at a time

Buggy attempts to wait for a child, no flag

Incorrect Code

Child	1	void thr_exit() {
	2	pthread_mutex_lock(&m);
	3	pthread_cond_signal(&c);
	4	pthread_mutex_unlock(&m);
	5	}
Parent	6	
	7	void thr_join() {
	8	pthread_mutex_lock(&m);
	9	pthread_cond_wait(&c, &m);
	10	pthread_mutex_unlock(&m);
	11	}

Correct Code

5	void thr_exit() {
6	pthread_mutex_lock(&m);
7	done = 1;
8	pthread_cond_signal(&c);
9	pthread_mutex_unlock(&m);
10	}
18	void thr_join() {
19	pthread_mutex_lock(&m);
20	while (done == 0)
21	pthread_cond_wait(&c, &m);
22	pthread_mutex_unlock(&m);
23	}

Without *done* variable:

- 1) The child could run first and signal
- 2) Before the parent starts waiting for the child
- 3) Parent waits forever...

Check your understanding: is a lock necessary?

Incorrect Code

```
Child
1 void thr_exit() {
2     done = 1;
3     Pthread_cond_signal(&c);
4 }
5
Parent
6 void thr_join() {
7     if (done == 0)
8         Pthread_cond_wait(&c);
9 }
```

Correct Code

```
5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c);
9     Pthread_mutex_unlock(&m);
10 }
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
```

What could go wrong?

- Without the lock, these lines could be interleaved in any way

Buggy attempts to wait for a child, no mutex

Incorrect Code

```
Child
1 void thr_exit() {
2     done = 1;
3     Pthread_cond_signal(&c);
4 }
5
Parent
6 void thr_join() {
7     if (done == 0)
8         Pthread_cond_wait(&c);
9 }
```

Correct Code

```
5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c);
9     Pthread_mutex_unlock(&m);
10 }
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
```

Without the lock:

- 1) Parent could see `done == 0` and enter the if statement
- 2) Child could then exit, setting `done` to 1 and signaling
- 3) Parent then calls wait (missed the signal) and waits forever

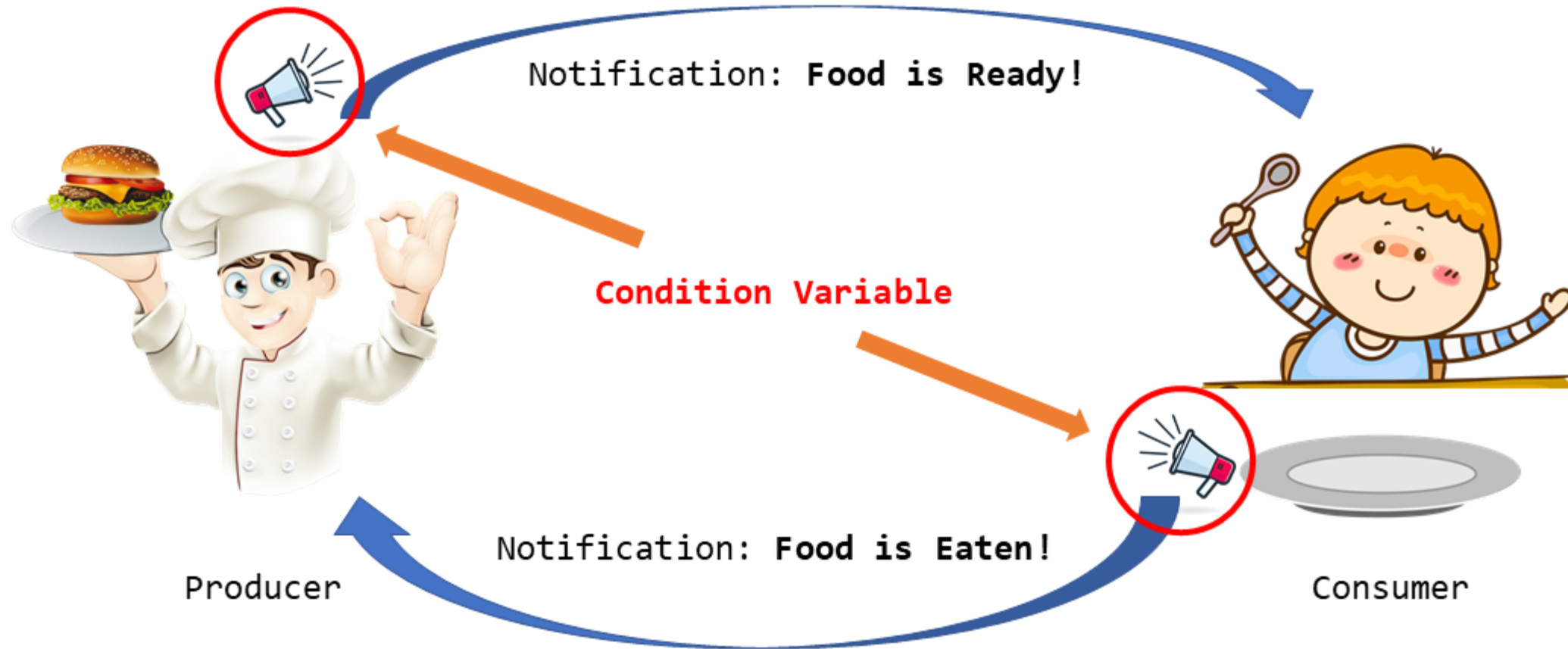
Always use a loop to check the flag variable

- It's possible for the thread to wake up from a wait, but the resource is not available!

```
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
```

- Maybe another thread took the resource first
 - Another thread could run and claim it before the woken thread is scheduled
- Maybe a *spurious wakeup* occurred
 - Often other sources can cause wakeups to occur
 - Signals or Interrupts usually
 - Makes the implementation of condvar simpler, and we need to double-check the flag anyways, so it doesn't matter

Classical concurrency problem: Producer-Consumer



Produce/Consumer Example Details

- We have multiple producers and multiple consumers that communicate with a shared queue (FIFO buffer).
 - Concurrent queue allows work to happen asynchronously.
 - Buffer has finite size (does not dynamically expand)
- Two operations:
 - **Put**, which should block (wait) if the buffer is **full**.
 - **Get**, which should block (wait) if the buffer is **empty**.
- This is more complex than a (linked-list-based) concurrent queue because of the finite size and waiting.
- Example scenario: request queue in a multi-threaded web server.

Managing the buffer

```
1  int buffer[MAX];
2  int fill  = 0;
3  int use   = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill] = value;
8      fill = (fill + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use];
14     use = (use + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

- A simple implementation of a circular buffer that stores data in a fixed-size array.
- *fill* is the index of the tail
- *use* is the index of the head
- *count* is the number of items

This simple implementation assumes:

- Concurrency is managed elsewhere
- It will overwrite data if we try to put more than MAX elements.

Managing the concurrency

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- Always acquire *mutex*
 - Must use same mutex in both functions
- Use *two condvars*

Managing the concurrency

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- Always acquire *mutex*
 - Must use same mutex in both functions
- Use *two condvars*
- Producer waits on **empty** while the buffer is full
 - Producer signals **fill** after put

Managing the concurrency

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- Always acquire *mutex*
 - Must use same mutex in both functions
- Use *two condvars*
- Producer waits on **empty** while the buffer is full
 - Producer signals **fill** after `put`
- Consumer waits on **fill** while the buffer is empty
 - Consumer signals **empty** after `get`

Managing the concurrency

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- Always acquire *mutex*
 - Must use same mutex in both functions
- Use *two condvars*
- Producer waits on **empty** while the buffer is full
 - Producer signals **fill** after `put`
- Consumer waits on **fill** while the buffer is empty
 - Consumer signals **empty** after `get`
- Loops re-check count condition after breaking out of wait, to check that there really is a resource

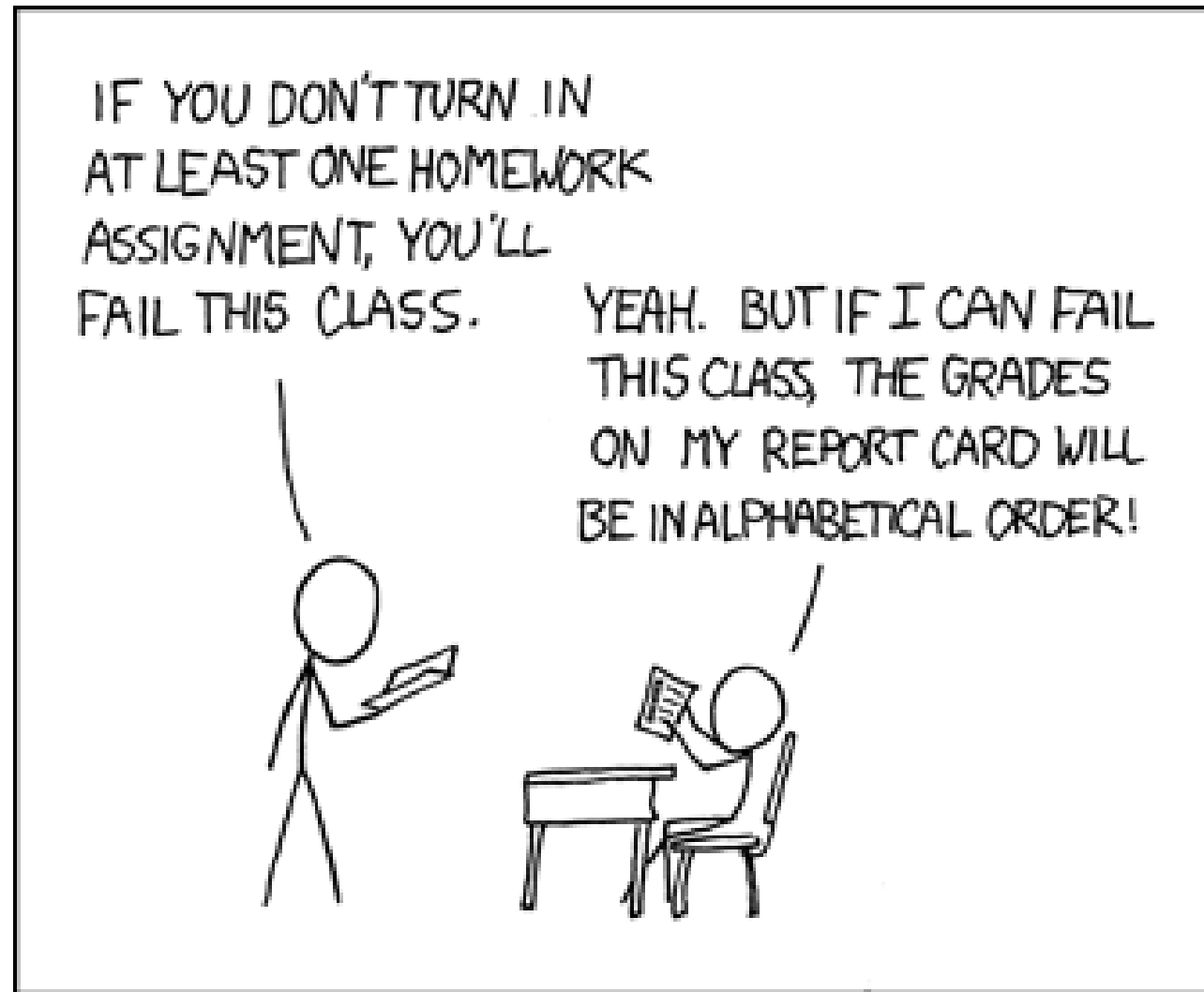
Broadcast makes more complex conditions possible

- Recall that *signal* wakes one waiting thread (FIFO)
 - But there are times when threads are not all equivalent
 - The signal may not be serviceable by any of the threads
- For example, consider memory allocation/free requests
 - An allocation can only be serviced by free of $\geq$ size
- **pthread_cond_broadcast** wakes all threads
 - This approach may be inefficient, but it may be necessary to ensure progress

Condition Variable: rules of thumb

- Shared state determines if condition is true or not
 - Check the state in a while loop before waiting on condvar
- Use a mutex to protect:
 - The shared state on which condition is based, and
 - Operations on the condvar itself
- Use different condvars for different conditions
 - Sometimes, `cond_broadcast()` helps if you can't find an elegant solution using `cond_signal()`

Break + xkcd (not relevant, just funny)



Outline

- Amdahl's Law
- Applying Locks
- Ordering with Condition Variables
- **Semaphores**

Generalizing Synchronization

- Condvars have no state or lock, just a waiting queue
 - The rest is handled by the programmer
- Semaphores are a combination of condvars and locks
 - Includes internal (locked) state
 - Sometimes this makes them more complicated, sometimes simpler

Semaphores (by Edsger Dijkstra, 1965)

- Keeps an internal integer value that determines what happens to a calling thread
- Init(val)
 - Set the initial internal value
 - Value cannot otherwise be directly modified
- Up/Signal/Post/V() (from Dutch *verhogen* "increase")
 - Increase the value. If there is a waiting thread, wake one.
- Down/Wait/Test/P() (from Dutch *proberen* "to try")
 - Decrease the value. Wait if the value is negative.



Dijkstra invented
Dijkstra's Algorithm!

Also Semaphores and the
*entire field of Concurrent
Programming*

[https://en.wikipedia.org/
wiki/Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra)

Semaphores vs Condition Variables

- Semaphores
 - ***Up/Post***: increase value and wake one waiting thread
 - ***Down/Wait***: decrease value and wait if it's negative
- Compared to condvars, semaphores add an integer value that controls when waiting is necessary
- Purpose: semaphores track resource availability/reservations
 - *Value* counts the quantity of a shared resource currently available
 - *Up* makes a resource available, *down* reserves a resource
- Condition Variables
 - ***Signal***: wake one waiting thread
 - ***Wait***: wait

Check your understanding: build a mutex

- How would we build a mutex out of a semaphore?

```
typedef struct {  
    sem_t sem;  
} lock_t;  
init(lock_t* lock){  
  
}  
acquire(lock_t* lock) {  
  
}  
release(lock_t* lock) {  
  
}
```

```
sem_init(sem_t*, int initial)  
sem_wait(sem_t*): Decrement, wait until  
                  value >= 0  
sem_post(sem_t*): Increment value then  
                  wake a single waiter
```

Check your understanding: build a mutex

- How would we build a mutex out of a semaphore?

```
typedef struct {  
    sem_t sem;  
} lock_t;  
init(lock_t* lock){  
    sem_init(&(lock->sem), 1);  
}  
acquire(lock_t* lock) {  
    sem_wait(&(lock->sem));  
}  
release(lock_t* lock) {  
    sem_post(&(lock->sem));  
}
```

sem_init(sem_t*, int initial)
sem_wait(sem_t*): Decrement, wait until
value ≥ 0
sem_post(sem_t*): Increment value then
wake a single waiter

Explanation of semaphore mutex implementation

```
typedef struct {  
    sem_t sem;  
} lock_t;  
init(lock_t* lock){  
    sem_init(&(lock->sem), 1);  
}  
acquire(lock_t* lock) {  
    sem_wait(&(lock->sem));  
}  
release(lock_t* lock) {  
    sem_post(&(lock->sem));  
}
```

- The semaphore value represents the number of resources available
 - For a lock, there is 1 available initially
- Acquiring the lock might give it to you immediately
 - Or it might wait
 - Multiple threads could be waiting
- Releasing the lock only occurs after acquiring and resets it to 1

Semaphores reduce effort for numerical conditions

	Condition Variable	Semaphore
Child	<pre>5 void thr_exit() { 6 Pthread_mutex_lock(&m); 7 done = 1; 8 Pthread_cond_signal(&c); 9 Pthread_mutex_unlock(&m); 10 }</pre>	<pre>void thr_exit() { sem_post(&s); }</pre>
Parent	<pre>18 void thr_join() { 19 Pthread_mutex_lock(&m); 20 while (done == 0) 21 Pthread_cond_wait(&c, &m); 22 Pthread_mutex_unlock(&m); 23 }</pre>	<pre>void thr_join() { sem_wait(&s); } // somewhere before all of this sem_init(&s, 0);</pre>

- Want parent to wait immediately so initialize to 0
- If child thread finishes first, semaphore increments to 1
- Resource: number of threads completed

Readers-Writers Problem

- Some resources don't need strict mutual exclusion, especially if they have many *read-only* accesses. (eg., a linked list)
- Any number of readers can be active simultaneously, but
- Writes must be mutually exclusive AND cannot happen during read
- API:
 - `acquire_read_lock()`, `release_read_lock()`
 - `acquire_write_lock()`, `release_write_lock()`

Reader-writer Lock

- “lock” semaphore used as a mutex

```
1  typedef struct _rwlock_t {
2      sem_t lock;          // binary semaphore (basic lock)
3      sem_t writelock;    // used to allow ONE writer or MANY readers
4      int    readers;     // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8      rw->readers = 0;
9      sem_init(&rw->lock, 0, 1);
10     sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); // first reader acquires writelock
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); // last reader releases writelock
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

Reader-writer Lock

- “writelock” must be held during read to block writes or during write to block reads.
- During reads
 - Number of active readers is counted.
 - First/last reader handles acquiring/releasing writelock.

```
1  typedef struct _rwlock_t {
2      sem_t lock;          // binary semaphore (basic lock)
3      sem_t writelock;     // used to allow ONE writer or MANY readers
4      int  readers;        // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8      rw->readers = 0;
9      sem_init(&rw->lock, 0, 1);
10     sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); // first reader acquires writelock
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); // last reader releases writelock
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

Classical concurrency problems

- Note that this particular solution could starve writers
 - There might always be readers in the critical section
- Full solution to readers-writers problem with progress guarantee
 - https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem
- Generally: try to map your problem to one of these solved problems
 - Producers/Consumers or Readers/Writers
 - There are MANY solutions to these problems available online

Outline

- Amdahl's Law
- Applying Locks
- Ordering with Condition Variables
- Semaphores