CSCI-GA.3033-017

Special Topics:

Multicore Programming

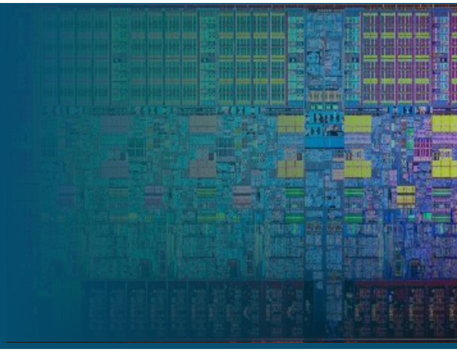
Lecture 6

Coordinating Resources

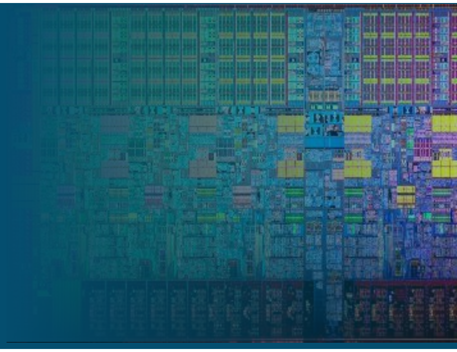
Christopher Mitchell, Ph.D.

cmitchell@cs.nyu.edu || http://z80.me

Homework 1 Review
Types of Parallelism



- Instruction-Level Parallelism [CPU]
  - Pipelining, requires pipelined CPU
- Basic Block Parallelism [CPU] [Compiler]
  - Reordering and parallelizing instructions within a block
  - Parallelizing instructions from multiple blocks
  - Requires register copies and functional unit copies
- Loop Level Parallelism [Compiler]
  - Interleave and parallelize instructions from multiple iterations
- Task Parallelism [Programmer]
  - Threads: related work, often sharing same memory space
- Process Parallelism [Programmer]
  - Distinct work to be completed in parallel
- Machine Parallelism [Programmer]
  - Break work into groups of related processes spread across multiple machines



Homework 2 Review
Thread-Safe Aggregation

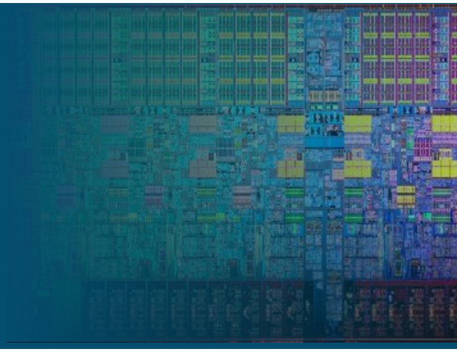
```
static int sum_stat_a = 0;
static int sum_stat_b = 0;
int aggregateStats(int stat_a, int stat_b) {
    sum_stat_a += stat_a;
    sum_stat_b += stat_b;
    return sum_stat_a + sum_stat_b;
}
void init(void) { }
```

1. Use a single pthread mutex to make this function thread-safe. Add global variables and content to the `init()` function as necessary.
2. Modify the original code from to make it thread-safe, but use two mutices this time, one for `sum_stat_a` and one for `sum_stat_b`.

Homework 2 Review
Aggregation (1 Lock)

```
pthread_mutex_t stats_mutex;
static int sum_stat_a = 0;
static int sum_stat_b = 0;
int aggregateStats(int stat_a, int stat_b) {
        pthread_mutex_lock(&stats_mutex);
        sum_stat_a += stat_a;
        sum_stat_b += stat_b;
        int rval = sum_stat_a + sum_stat_b;
        pthread_mutex_unlock(stats_mutex);
        return rval;
}
void init(void) {
        pthread_mutex_init(&stats_mutex, NULL);
}
```

Invariant: The returned value from aggregateStats() must be equal to exactly the sum of statistic A and statistic B at exactly the moment in time when the new additions were aggregated into each sum.

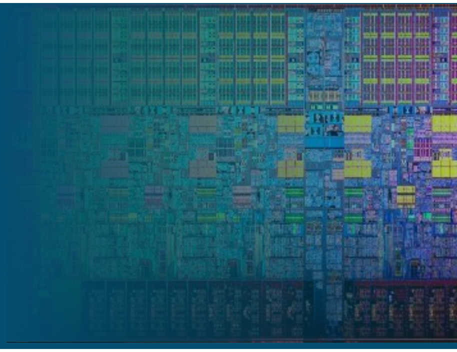


Homework 2 Review
Aggregation (2 Locks)

Invariant: The returned value from aggregateStats() must be equal to exactly the sums of statistic A and statistic B, each taken at the moment in time when the new addition was aggregated into **each** sum. The aggregate sum may therefore represent a value covering two close (but different) time periods.

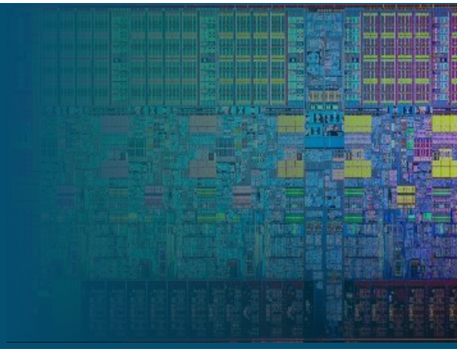
`sum_stat_a`: `stat_a_mutex`
`sum_stat_b`: `stat_b_mutex`

Homework 2 Review
Aggregation (2 Locks)

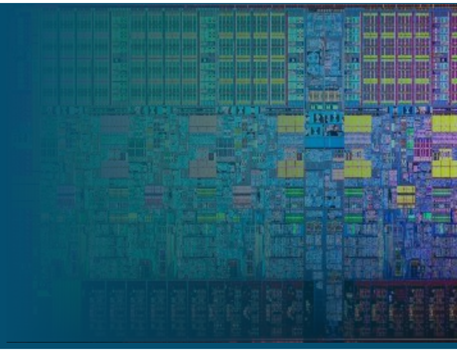


```
pthread_mutex_t stat_a_mutex;
pthread_mutex_t stat_b_mutex;
static int sum_stat_a = 0;
static int sum_stat_b = 0;
int aggregateStats(int stat_a, int stat_b) {
        int rval = 0;
        pthread_mutex_lock(&stat_a_mutex.lock);
        sum_stat_a += stat_a;
        rval += sum_stat_a;
        pthread_mutex_unlock(&stat_a_mutex);
        pthread_mutex_lock(&stat_b_mutex);
        sum_stat_b += stat_b;
        rval += sum_stat_b;
        pthread_mutex_unlock(&stat_b_mutex);
        return rval;
}
void init(void) {
        pthread_mutex_init(&stat_a_mutex, NULL);
        pthread_mutex_init(&stat_b_mutex, NULL);
}
```

Outline



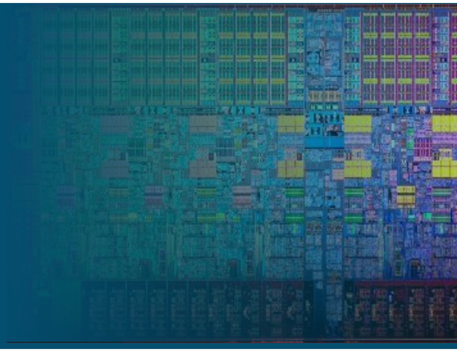
- ~~Homework Review~~
- Coordinating Resources
  - Reasoning about two mutex/semaphore-based schemes
  - Reader-Writer Locks
  - Barriers
- Lab 2 Techniques
  - Socket Refresher
  - Thread Pools



Outline

- ~~Homework Review~~
- Coordinating Resources
  - Reader-Writer Locks
  - Barriers
- Lab 2 Techniques
  - Socket Refresher
  - Thread Pools

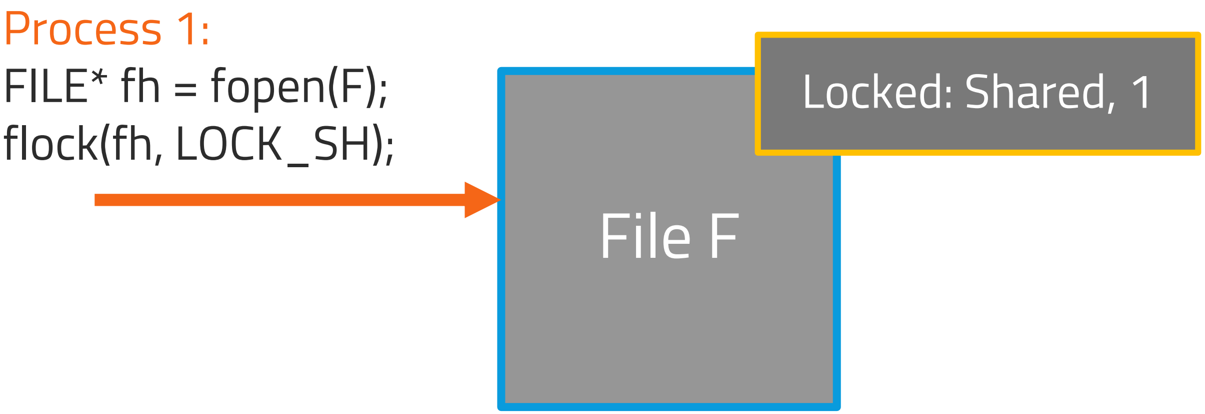
The Reader-Writer Problem



- Consider a resource
  - Shared by several threads
  - Some threads may only want to read
  - Others may want to modify
- Could we coordinate these writers and readers?
- Idea: a reader-writer lock [pair]
  - Each reader acquires a special lock that allows them to share the resource with other readers
  - A writer acquires another kind of lock that gives it exclusive access to the resource
  - The locks work in tandem to guarantee the resource's consistency

POSIX File Reader-Writer Lock

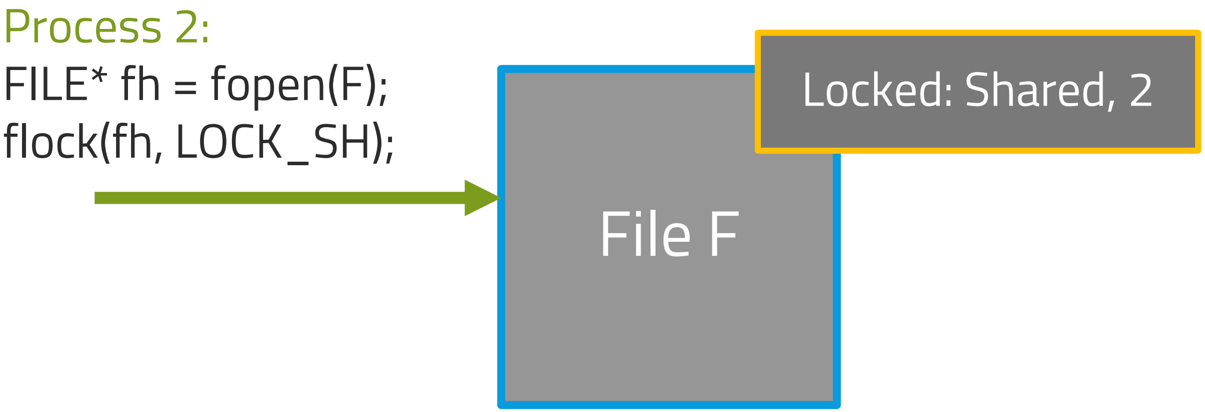
- File locking between processes or threads
- `flock(file_handle, mode)`
  - LOCK_SH: Shared (reader) lock
  - LOCK_EX: Exclusive (writer) lock
  - Bitwise OR with LOCK_NB: Nonblocking



Process 1:
FILE* fh = fopen(F);
flock(fh, LOCK_SH);

POSIX File Reader-Writer Lock

- File locking between processes or threads
- `flock(file_handle, mode)`
  - LOCK_SH: Shared (reader) lock
  - LOCK_EX: Exclusive (writer) lock
  - Bitwise OR with LOCK_NB: Nonblocking

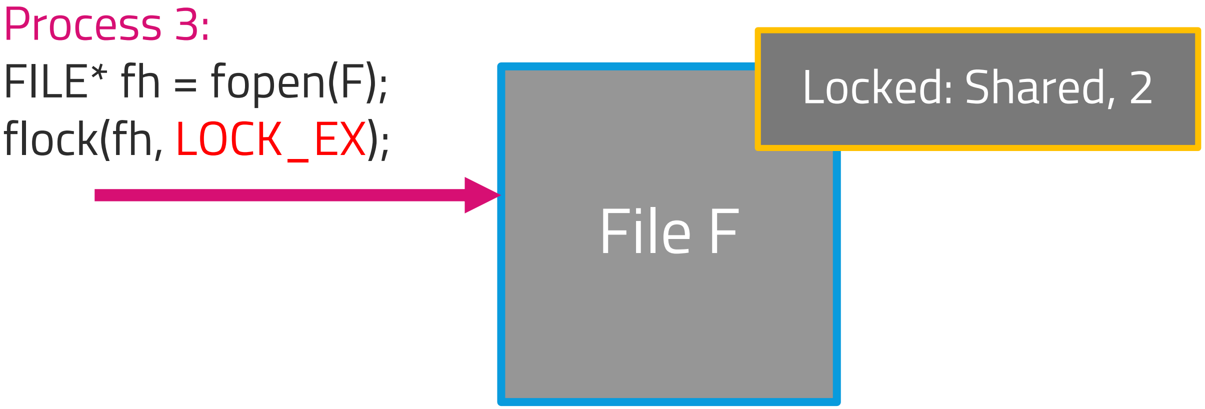


Process 2:

```
FILE* fh = fopen(F);
flock(fh, LOCK_SH);
```

POSIX File Reader-Writer Lock

- File locking between processes or threads
- `flock(file_handle, mode)`
  - LOCK_SH: Shared (reader) lock
  - LOCK_EX: Exclusive (writer) lock
  - Bitwise OR with LOCK_NB: Nonblocking

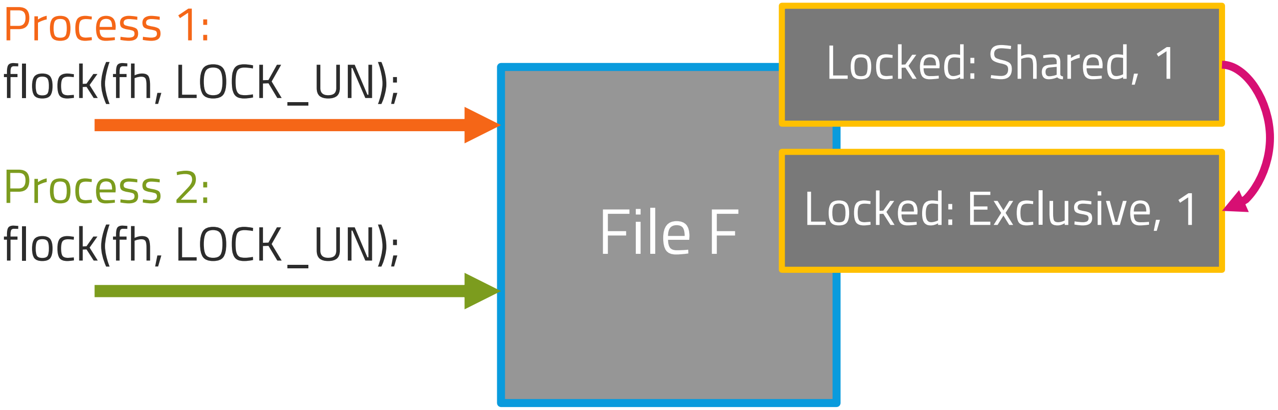


Process 3:

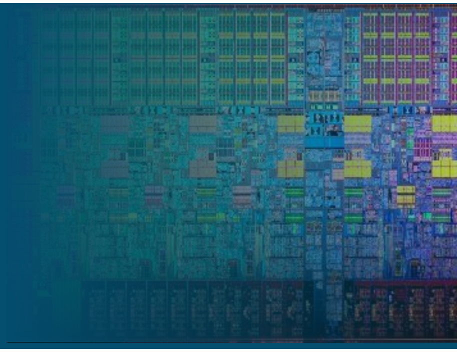
```
FILE* fh = fopen(F);
flock(fh, LOCK_EX);
```

POSIX File Reader-Writer Lock

- File locking between processes or threads
- `flock(file_handle, mode)`
  - LOCK_SH: Shared (reader) lock
  - LOCK_EX: Exclusive (writer) lock
  - Bitwise OR with LOCK_NB: Nonblocking

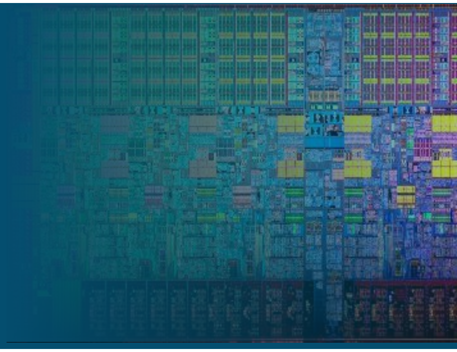


Simple Reader-Writer Lock



- Forgot files: let’s implement a simple reader-writer lock
- Semantics:
  - Allow any number of shared readers
  - Allow a single exclusive writer
  - Fairness? Worry about it later
- Toolset
  - Mutices

Simple Reader-Writer Lock



```
int read_count = 0
mutex mut_read, write_lock
```

```
reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```

```
writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```

Who gets the priority? Readers or writers?

Simple Reader-Writer Lock

Reader arrives before writer

```
int read_count = 1
mutex mut_read, write_lock
```

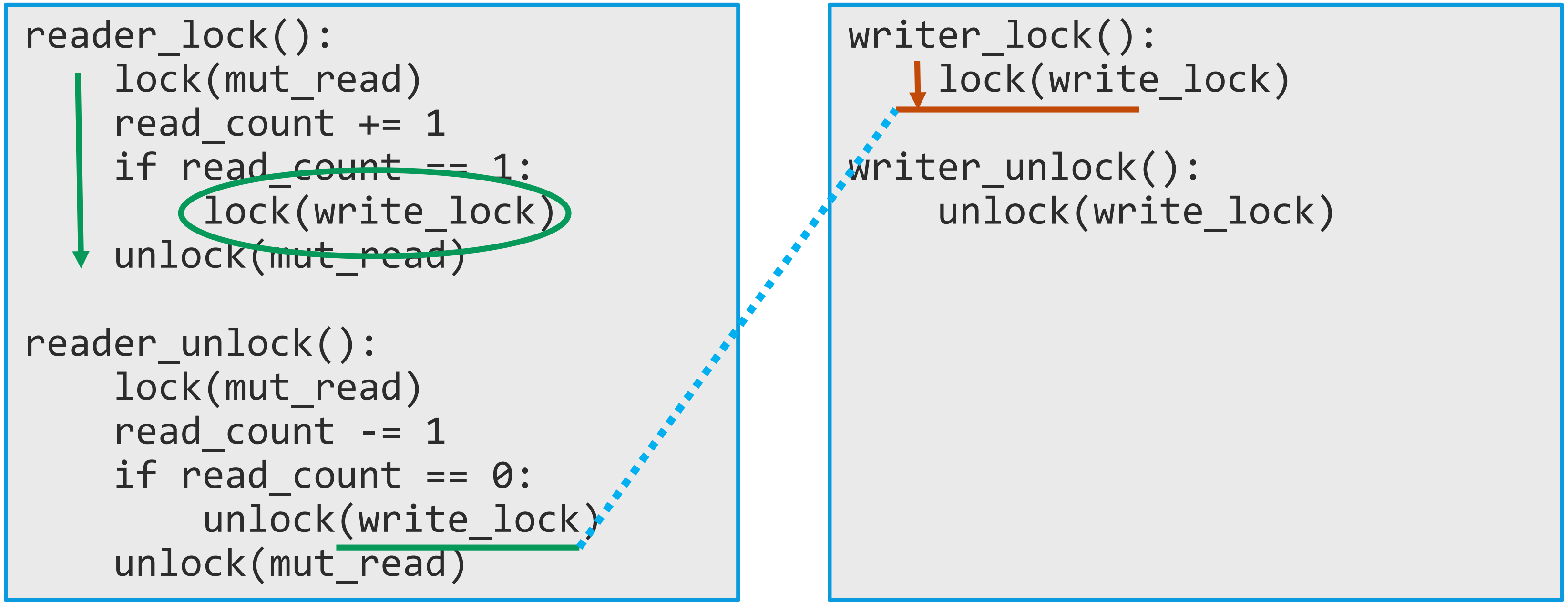
```
reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```

```
writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```

Simple Reader-Writer Lock

Reader arrives before writer

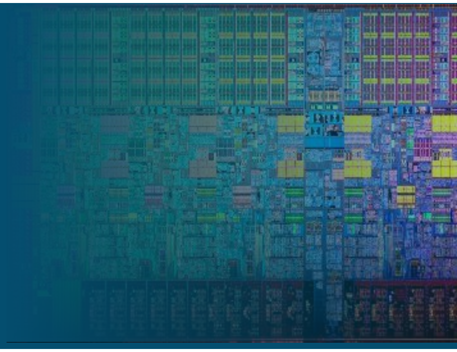
```
int read_count = 1
mutex mut_read, write_lock
```



```
reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```

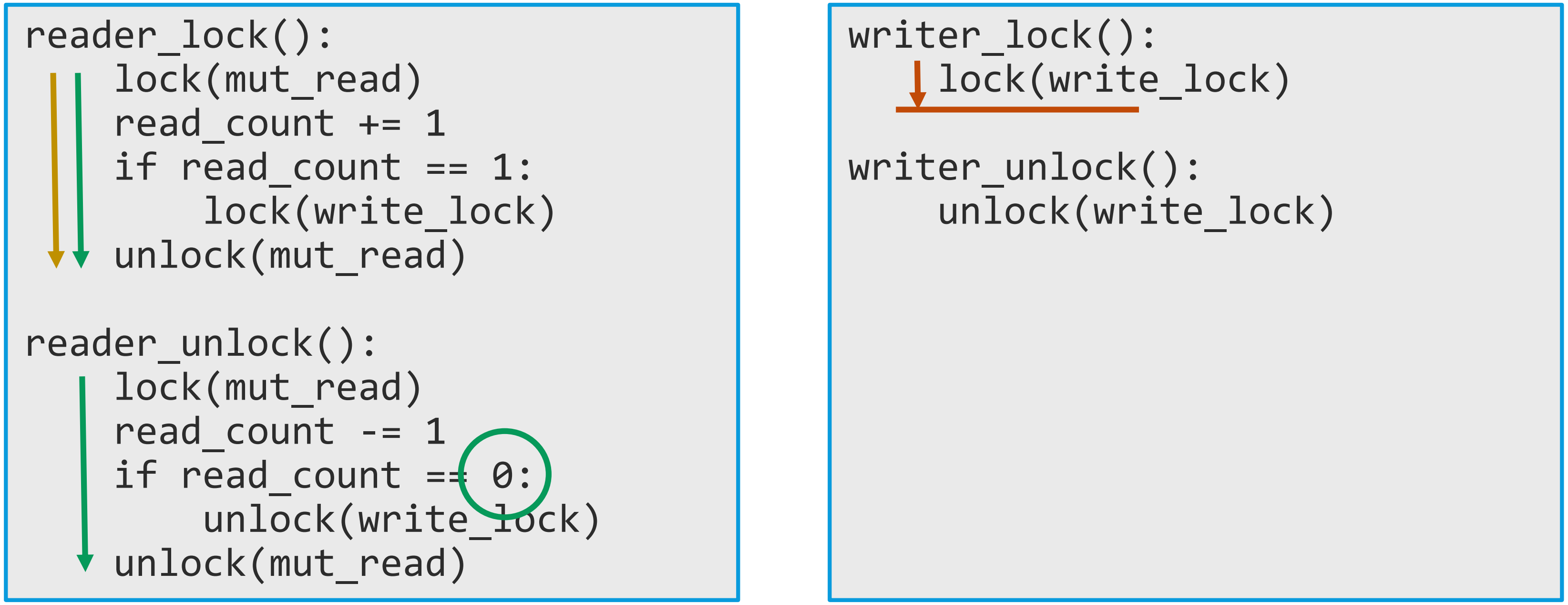
```
writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```

Simple Reader-Writer Lock: Starvation



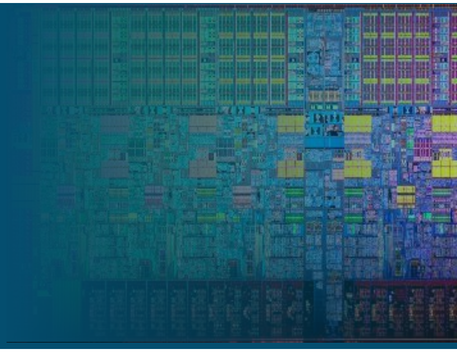
Second reader arrives before first reader finishes

```
int read_count = 2
mutex mut_read, write_lock
```



```
reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```

```
writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```



Reader-Writer Lock v2

Give writers priority over readers.

```
int read_count, write_count
mutex mut_read, mut_write, read_lock, write_lock
```

```
reader_lock():
    lock(read_lock)
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)
    unlock(read_lock)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```

```
writer_lock():
    lock(mut_write)
    write_count += 1
    if write_count == 1:
        lock(read_lock)
    unlock(mut_write)
    lock(write_lock)

writer_unlock():
    lock(mut_write)
    write_count -= 1
    if write_count == 0:
        unlock(read_lock)
    unlock(mut_write)
    unlock(write_lock)
```

Reader-Writer Lock v2

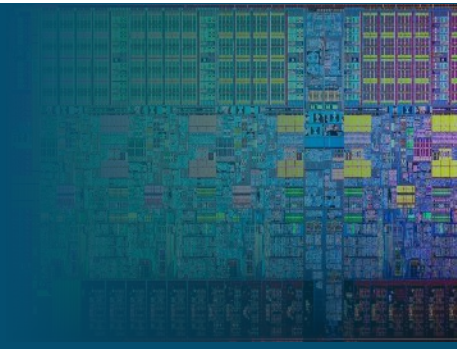
One reader, then one writer, arrives.

```
int read_count = 1, write_count
mutex mut_read, mut_write, read_lock, write_lock
```

```
reader_lock():
    lock(read_lock)
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)
    unlock(read_lock)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```

```
writer_lock():
    lock(mut_write)
    write_count += 1
    if write_count == 1:
        lock(read_lock)
    unlock(mut_write)
    lock(write_lock)

writer_unlock():
    lock(mut_write)
    write_count -= 1
    if write_count == 0:
        unlock(read_lock)
    unlock(mut_write)
    unlock(write_lock)
```

Reader-Writer Lock v2



Second reader arrives.

```
int read_count = 0, write_count = 2
mutex mut_read, mut_write, read_lock, write_lock
```

```
reader_lock():
    lock(read_lock)
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)
    unlock(read_lock)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```

```
writer_lock():
    lock(mut_write)
    write_count += 1
    if write_count == 1:
        lock(read_lock)
    unlock(mut_write)
    lock(write_lock)

writer_unlock():
    lock(mut_write)
    write_count -= 1
    if write_count == 0:
        unlock(read_lock)
    unlock(mut_write)
    unlock(write_lock)
```

Reader-Writer Lock v2

Now writers can starve readers.

```
int read_count = 0, write_count = 2
mutex mut_read, mut_write, read_lock, write_lock
```

```
reader_lock():
    lock(read_lock)
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)
    unlock(read_lock)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```

```
writer_lock():
    lock(mut_write)
    write_count += 1
    if write_count == 1:
        lock(read_lock)
    unlock(mut_write)
    lock(write_lock)

writer_unlock():
    lock(mut_write)
    write_count -= 1
    if write_count == 0:
        unlock(read_lock)
    unlock(mut_write)
    unlock(write_lock)
```

Reader-Writer Lock v3

Tracing a reader, then a writer

```
int a_readers, a_writers, p_readers, p_writers   // Active & pending
mutex mut, cond_var read_cond, write_cond
```

```
reader_lock():
    lock(mut)
    while a_writers + p_writers:
        p_readers += 1
        read_cond.wait(mut)
        p_readers -= 1
    a_readers += 1
    unlock(mut)

reader_unlock():
    lock(mut)
    a_readers -= 1
    if !a_readers && pwriters:
        write_cond.signal()
    unlock(mut)
```

```
writer_lock():
    lock(mut)
    while a_writers + a_readers:
        p_writers += 1
        write_cond.wait(mut)
        p_writers -= 1
    a_writers += 1
    unlock(mut)

writer_unlock():
    lock(mut)
    a_writers -= 1
    if p_writers:
        write_cond.signal()
    else if p_readers > 0:
        read_cond.broadcast()
    unlock(mut)
```

Reader-Writer Lock v3

Tracing a reader, then a writer

```
int a_readers, a_writers, p_readers, p_writers   // Active & pending
mutex mut, cond_var read_cond, write_cond
```

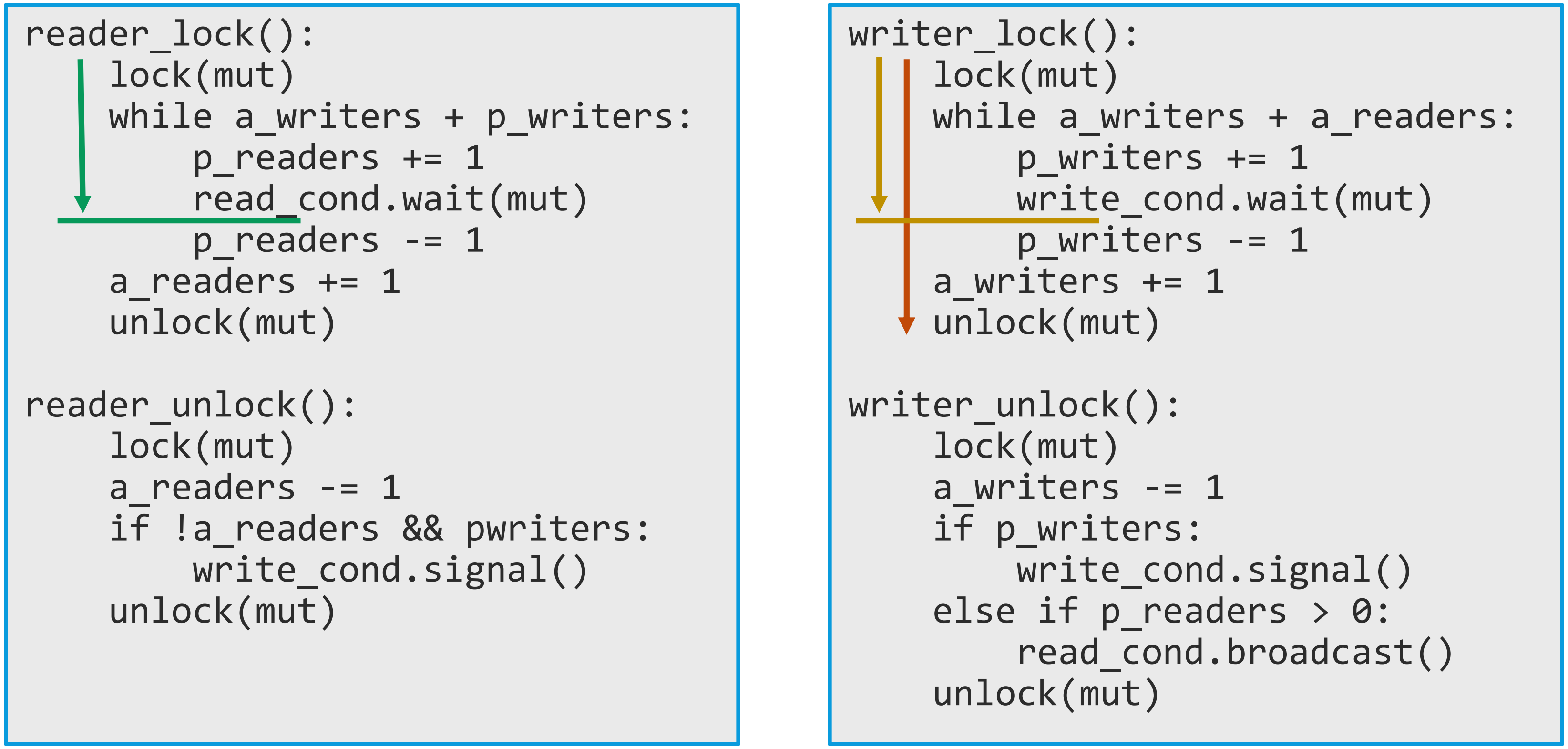
```
reader_lock():
    lock(mut)
    while a_writers + p_writers:
        p_readers += 1
        read_cond.wait(mut)
        p_readers -= 1
    a_readers += 1
    unlock(mut)

reader_unlock():
    lock(mut)
    a_readers -= 1
    if !a_readers && pwriters:
        write_cond.signal()
    unlock(mut)
```

```
writer_lock():
    lock(mut)
    while a_writers + a_readers:
        p_writers += 1
        write_cond.wait(mut)
        p_writers -= 1
    a_writers += 1
    unlock(mut)

writer_unlock():
    lock(mut)
    a_writers -= 1
    if p_writers:
        write_cond.signal()
    else if p_readers > 0:
        read_cond.broadcast()
    unlock(mut)
```

Reader-Writer Lock v3

Tracing a reader, a writer, and a second writer.

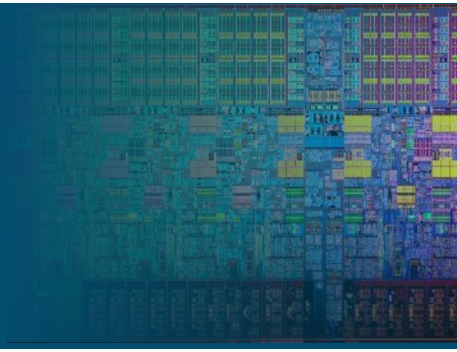
```
int a_readers, a_writers, p_readers, p_writers   // Active & pending
mutex mut, cond_var read_cond, write_cond
```



```
reader_lock():
    lock(mut)
    while a_writers + p_writers:
        p_readers += 1
        read_cond.wait(mut)
        p_readers -= 1
    a_readers += 1
    unlock(mut)

reader_unlock():
    lock(mut)
    a_readers -= 1
    if !a_readers && pwriters:
        write_cond.signal()
    unlock(mut)
```

```
writer_lock():
    lock(mut)
    while a_writers + a_readers:
        p_writers += 1
        write_cond.wait(mut)
        p_writers -= 1
    a_writers += 1
    unlock(mut)

writer_unlock():
    lock(mut)
    a_writers -= 1
    if p_writers:
        write_cond.signal()
    else if p_readers > 0:
        read_cond.broadcast()
    unlock(mut)
```

Reader-Writer Lock v3



Tracing a reader, a writer, and a second writer.

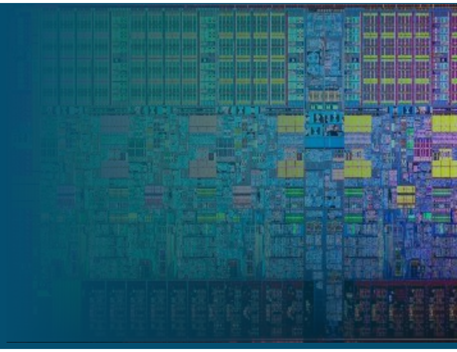
```
int a_readers, a_writers, p_readers, p_writers   // Active & pending
mutex mut, cond_var read_cond, write_cond
```

```
reader_lock():
    lock(mut)
    while a_writers + p_writers:
        p_readers += 1
        read_cond.wait(mut)
        p_readers -= 1
    a_readers += 1
    unlock(mut)

reader_unlock():
    lock(mut)
    a_readers -= 1
    if !a_readers && pwriters:
        write_cond.signal()
    unlock(mut)
```

```
writer_lock():
    lock(mut)
    while a_writers + a_readers:
        p_writers += 1
        write_cond.wait(mut)
        p_writers -= 1
    a_writers += 1
    unlock(mut)

writer_unlock():
    lock(mut)
    a_writers -= 1
    if p_writers:
        write_cond.signal()
    else if p_readers > 0:
        read_cond.broadcast()
    unlock(mut)
```

Choose priority here

Final Reader-Writer Lock Miscellania

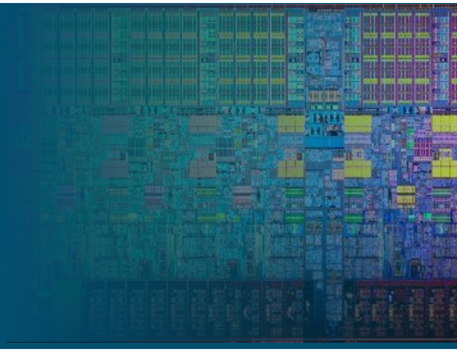


- Every time we see a structure taking many readers, R/W seem the thing to do.
- However…
  - Even in the reader-only case, there could be contention on the reader counter mutex.
  - Maintaining fairness can cause contention
- Recent work:
  - “Scalable Reader-Writer Locks”, from Lev et al. 2009

Pthread Reader-Writer Lock

- Type: `pthread_rwlock_t`
- Initialization: `int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);`
- Lock for read:
  - Blocking:
    `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);`
  - Nonblocking:
    `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);`
- Lock for write
  - Blocking:
    `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`
  - Nonblocking:
    `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`

Outline

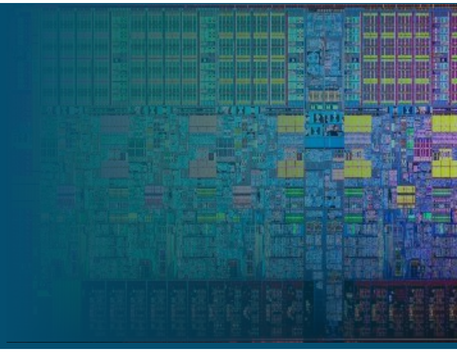


- ~~Homework Review~~
- Coordinating Resources
  - Reader-Writer Locks
  - Barriers
- Lab 2 Techniques
  - Socket Refresher
  - Thread Pools

Barrier

- Synchronize group of threads at single point
  - Each thread waits until all threads arrive
  - Each thread continues
- Solution
  - Mutex or semaphore to count arrivals
  - Mutex or semaphore to hold threads until count is equal to number of threads

Simple Semaphore-Based Barrier



```
semaphore arrival = 1, departure = 0;
int counter = 0, int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```

Must be known a priori

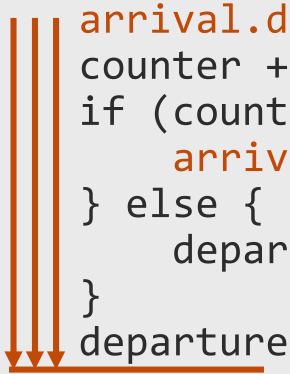
Simple Semaphore-Based Barrier

First arrival

```
semaphore arrival = 1, departure = 0;
int counter = 1, int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```

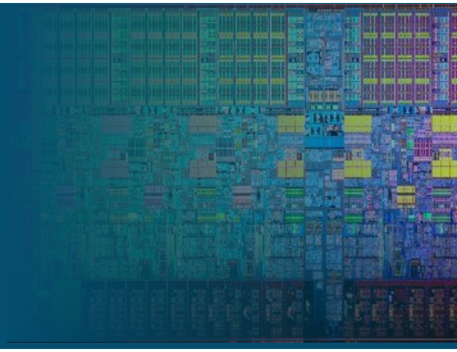
Simple Semaphore-Based Barrier

n - 1 arrivals

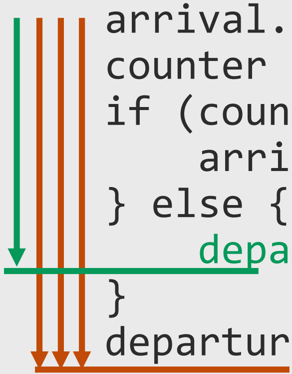


```
semaphore arrival = 1, departure = 0;
int counter = n - 1, int n = num_threads;

void await(void) {
    arrival.down();     // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```

Simple Semaphore-Based Barrier



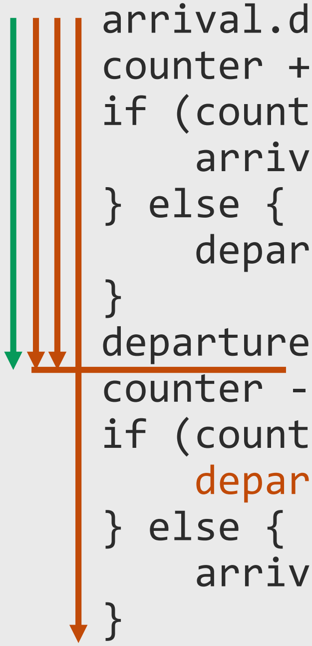
n arrivals



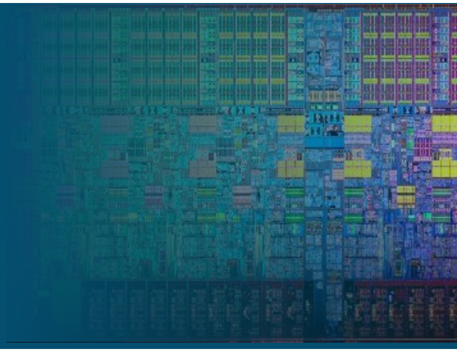
```
semaphore arrival = 0, departure = 1;
int counter = n , int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```

Simple Semaphore-Based Barrier

n arrivals, 1 departure

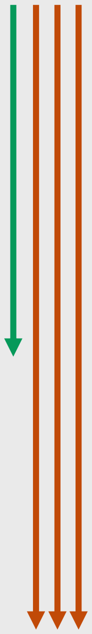


```
semaphore arrival = 0, departure = 1;
int counter = n - 1, int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```



Simple Semaphore-Based Barrier

n arrivals, n - 1 departures

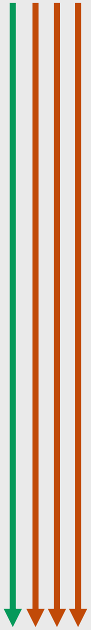


```
semaphore arrival = 0, departure = 1;
int counter = 1, int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```

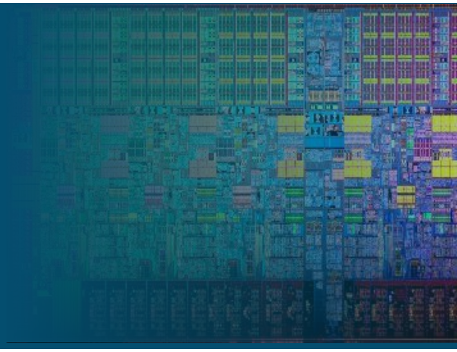
Simple Semaphore-Based Barrier

n arrivals

```
semaphore arrival = 1, departure = 0;
int counter = n - 1, int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```



Pthread Barrier



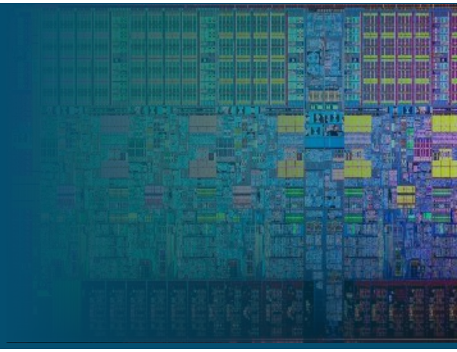
- Surprise! Pthread has a barrier primitive
- Type: `pthread_barrier_t`
- Initialization:

```
int pthread_barrier_init(pthread_barrier_t* barrier,
attributes, unsigned int count);
```

- Wait:

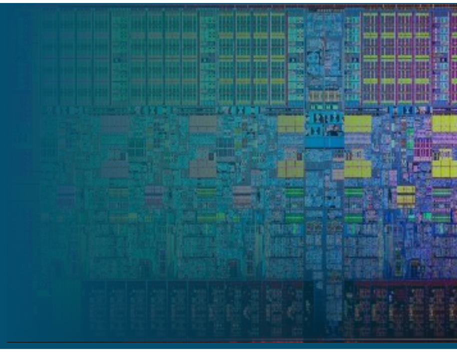
```
int pthread_barrier_wait(pthread_barrier_t* barrier);
```

Outline



- ~~Homework Review~~
- Coordinating Resources
  - Reader-Writer Locks
  - Barriers
- Lab 2 Techniques
  - Socket Refresher
  - Thread Pools

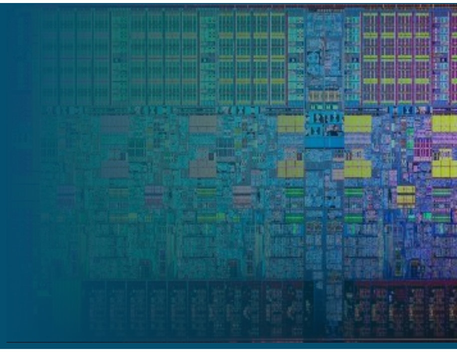
Lab 2



- Make our concurrent key-value store more useful: a multi-threaded key-value store server

1. Implement reader-writer lock(s)
2. Implement thread pool
3. Implement GET/POST/DELETE frontend

- Three weeks to complete
- Due November 3, 2016

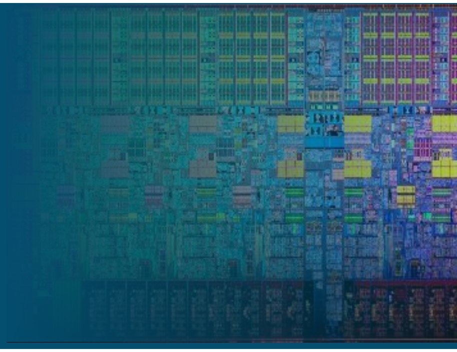


Lab 2

- Make our concurrent key-value store more useful: a multi-threaded key-value store server

1. Implement reader-writer lock(s) -> easy (pthreads!)
2. Implement thread pool -> challenging
3. Implement GET/POST/DELETE frontend -> moderate

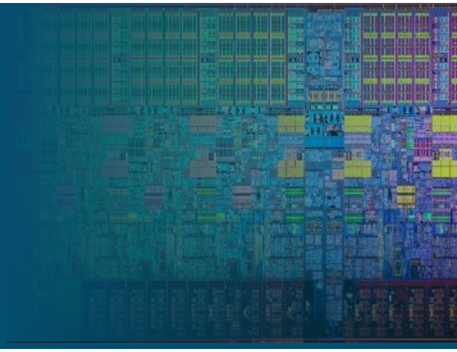
- Three weeks to complete
- Due November 3, 2016



Socket (Re-)Primer

- Review: http://www.linuxhowtos.org/C_C++/socket.htm
- Relevant: http://www.linuxhowtos.org/data/6/server2.c
  - Please don’t copy it, but good reference
- Concepts:
  - Socket connection (TCP: connectionful)
  - Passive (`listen()`ing/`accept()`ing) side
  - Active (`connect()`ing) side
- Server:
  - listen()
  - Repeatedly accept() -> use fd -> close fd

GET/POST/DELETE

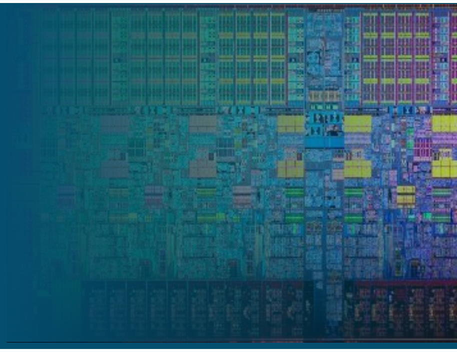


- HTTP 1.1: https://tools.ietf.org/html/rfc2616 (ouch)
- Saner: https://www.jmarshall.com/easy/http/#sample

GET /path HTTP/1.1 header header *[blank line]*	POST /path HTTP/1.1 header header *[blank line]* contents	DELETE /path HTTP/1.1 header header *[blank line]*
HTTP/1.1 200 OK Content-length: XXXX *[blank line]* contents	HTTP/1.1 200 OK Content-length: XXXX *[blank line]* contents	HTTP/1.1 200 OK Content-length: XXXX *[blank line]* contents

Note: newline is \r\n; see https://www.w3.org/Protocols/rfc2616/rfc2616-sec2.html#sec2.2

Thread Pool



- Thread work can be small pieces
  - Creating and destroying threads is expensive
  - Reduce overhead: reuse threads

1. Create group of N threads
2. Use thread-safe queue to identify “idle” threads
3. Atomically remove and invoke an idle thread when new work arrives
4. Atomically add self back to queue when work is done