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
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

```c
Process 2:
FILE* fh = fopen(F);
flock(fh, LOCK_SH);
```

Locked: Shared, 2
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);

Locked: Shared, 2
POSIX File Reader-Writer Lock

- File locking between processes or threads
- \texttt{flock(file	extunderscore handle, mode)}
  - LOCK\_SH: Shared (reader) lock
  - LOCK\_EX: Exclusive (writer) lock
  - Bitwise OR with LOCK\_NB: Nonblocking

\begin{itemize}
  \item Process 1: \texttt{flock(fh, LOCK\_UN)};
  \item Process 2: \texttt{flock(fh, LOCK\_UN)};
\end{itemize}
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

```c
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

```c
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)
```

```c
writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```
Simple Reader-Writer Lock

Reader arrives before writer

```c
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)
```

Reader arrives before writer

- `int read_count = 1` initializes a counter for read operations.
- `mutex mut_read, write_lock` defines the mutexes for reading and writing.
- `reader_lock()` acquires the `mut_read` mutex, increments the `read_count`, and if `read_count` becomes 1, acquires the `write_lock`.
- `reader_unlock()` releases the `mut_read` mutex, decrements the `read_count`, and if `read_count` becomes 0, releases the `write_lock`.
- `writer_lock()` acquires the `write_lock`.
- `writer_unlock()` releases the `write_lock`.

This simple reader-writer lock ensures that only one reader can access the data at a time, allowing multiple writers to access the data simultaneously.
Simple Reader-Writer Lock: Starvation

Second reader arrives before first reader finishes

```c
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.

```c
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.

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

```c
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)
```

```c
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)
```
Second reader arrives.

```c
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.

```c
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

```c
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 && p_writers:
        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:
        read(cond).broadcast()
    unlock(mut)
```
Reader-Writer Lock v3

Tracing a reader, then a writer

```c
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 && p_writers:
        write_cond.signal()  // New line
    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:
        read_cond.broadcast()
    unlock(mut)
```
Tracing a reader, a writer, and a second writer.

```c
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 && p_writers:
    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:
    read_cond.broadcast()
  unlock(mut)
```
Reader-Writer Lock v3

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

```c
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:
      read_cond.broadcast()
   unlock(mut)
```
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:

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_trywrlock(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

```c
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

```c
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 {
        // Acts as mutex & block on departure
        departure.up();
    }

    departure.down();     // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        // Back to initial conditions
        arrival.up();
    }
}
```
Simple Semaphore-Based Barrier

$n - 1$ arrivals

```c
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

```c
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

```c
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

```c
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

```c
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:
  ```c
  int pthread_barrier_init(pthread_barrier_t* barrier, attributes, unsigned int count);
  ```
- Wait:
  ```c
  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 13, 2017
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 value hashing and storage -> moderate
  3. Implement thread pool -> challenging
  4. GET/POST/DELETE frontend -> provided

• Three weeks to complete
• Due November 13, 2017
Socket (Re-)Primer

• Review: [http://www.linuxhowtos.org/C_C++/socket.htm](http://www.linuxhowtos.org/C_C++/socket.htm)
• Relevant: [http://www.linuxhowtos.org/data/6/server2.c](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

- Saner: [https://www.jmarshall.com/easy/http/#sample](https://www.jmarshall.com/easy/http/#sample)

<table>
<thead>
<tr>
<th>GET /path HTTP/1.1 header header [blank line]</th>
<th>POST /path HTTP/1.1 header Content-Length: XXXX [blank line] contents</th>
<th>DELETE /path HTTP/1.1 header header [blank line]</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1 200 OK Content-Length: XXXX [blank line] contents</td>
<td>HTTP/1.1 200 OK Content-Length: XXXX [blank line] contents</td>
<td>HTTP/1.1 200 OK Content-Length: XXXX [blank line] contents</td>
</tr>
</tbody>
</table>

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