ECE 570/670

David Irwin

Lecture 7

Administrative

- Assignment 1 out
- Assignment 0 due
  - 2 people have yet to sign up with the autograder
- ECE670 Project groups
  - 16 have yet to fill out Google Form
  - Will send reminder email today
  - Will send out appointment calendar soon

Administrative

- New TA
  - Xiaoding (Rebecca) Guan
  - Will post office hours and info soon

Administrative

- Today
  - End-to-End argument
  - Assignment 1
  - Higher-level synchronization

End-to-End Argument

- *In a general-purpose network, application-specific functions ought to reside in end hosts rather than intermediary nodes, provided that they can be implemented "completely and correctly" in the end hosts.*

- **Benefit of adding functions diminishes quickly, especially when hosts must re-implement functions for completeness/correctness**
- Since any function incurs some penalty even when not used, implementing it in the network distributes the penalty across all clients, regardless of whether they use it

End-to-End Argument

- *In a general-purpose network, application-specific functions ought to reside in end hosts rather than intermediary nodes, provided that they can be implemented "completely and correctly" in the end hosts.*

- Benefit of adding functions diminishes quickly, especially when hosts must re-implement functions for completeness/correctness
- **Since any function incurs some penalty even when not used, implementing it in the network distributes the penalty across all clients, regardless of whether they use it**

End-to-End Argument

- **Example: File transfer program**
  - Providing reliable data communication does not reduce the burden of the application to perform checksum/retries
    - MIT example: made too strong assumptions about network reliability
  - Performance Optimization
    - End-to-end recovery cost increases with file size
    - Could increase network reliability for performance
      - Tradeoff b/t overhead and performance benefit
      - Will still need end-to-end check
  - Considerations
    - Low-level optimization applies to all applications
      - E.g., might not work well for real-time communication
    - Lower levels have less information than higher levels

End-to-End Argument

- **Other End-to-End Examples**
  - Delivery guarantees - end-to-end acknowledgements
  - Secure data transmission – end-to-end encryption
  - Duplicate message suppression – application may generate duplicates that the network cannot distinguish; requires end-to-end suppression
  - FIFO message delivery guarantee – messages along different paths cannot guarantee FIFO ordering; have to re-order end-to-end

- **Widely applicable to other areas** – mentions library OSes at the end

Assignment 1

- Assignment 1 is out: two parts
  - 1. Assignment 1 disk scheduler (1d) (due 3/21)
    - How to use concurrency primitives
  - 2. Assignment 1 thread library (1t) (due later)
    - How concurrency primitives actually work by actually building them
  - Both parts treated as separate assignments with equal weight in final grade
- Start early; you will need the entire time

Assignment 1 Overview

- Thread library interface
  - thread_libinit()
  - thread_create()
  - thread_yield()
  - thread_lock()
  - thread_unlock()
  - thread_wait()
  - thread_signal()
  - thread_broadcast()
- Defined in thread.h
  - See description for specific inputs/return values

Assignment 1 (Part 1)

- Given a working thread library (thread.o)
- Disk scheduler
  - Schedules disk I/Os for multiple threads
  - Threads issue I/Os, which queue up at scheduler
    - Queue has a fixed size; must wait if queue full
  - Many threads issue requests; one thread services them
    - Similar to producer/consumer
  - Disk queue not FIFO – instead handles requests in SSTF (Shortest Seek Time First Order)
    - Each request has track number
      - Disk starts at track 0
    - Next request serviced is one that is closest to the current request

Assignment 1 (Part 1)

- Example: “./disk <queue_size> <list of input files>”
  - Assume one input file per thread
  - “./disk 10 disk.in0 disk.in1 disk.in2 disk.in3 disk.in4”

disk.in0	disk.in1	disk.in2	disk.in3	disk.in4
----------	---------	---------	---------	---------
53	914	827	302	631
785	350	567	230	11

Assignment 1 (Part 1)

- After issuing request, requester thread prints:
  - requester 0 track 53
- After servicing request, servicing thread prints
  - service requester 0 track 53

```
requester 0 track 53
requester 1 track 914
requester 2 track 827
service requester 0 track 53
requester 3 track 302
service requester 3 track 302
requester 4 track 631
service requester 4 track 631
requester 0 track 785
service requester 0 track 785
requester 3 track 230
service requester 2 track 827
requester 4 track 11
service requester 1 track 914
requester 2 track 567
service requester 2 track 567
requester 1 track 350
service requester 1 track 350
service requester 3 track 230
service requester 4 track 11
Thread library exiting.
```

Assignment 1 (Part 1)

- Compiling your program
  - **g++ disk.cc thread.o libinterrupt.a -ldl -o disk.**

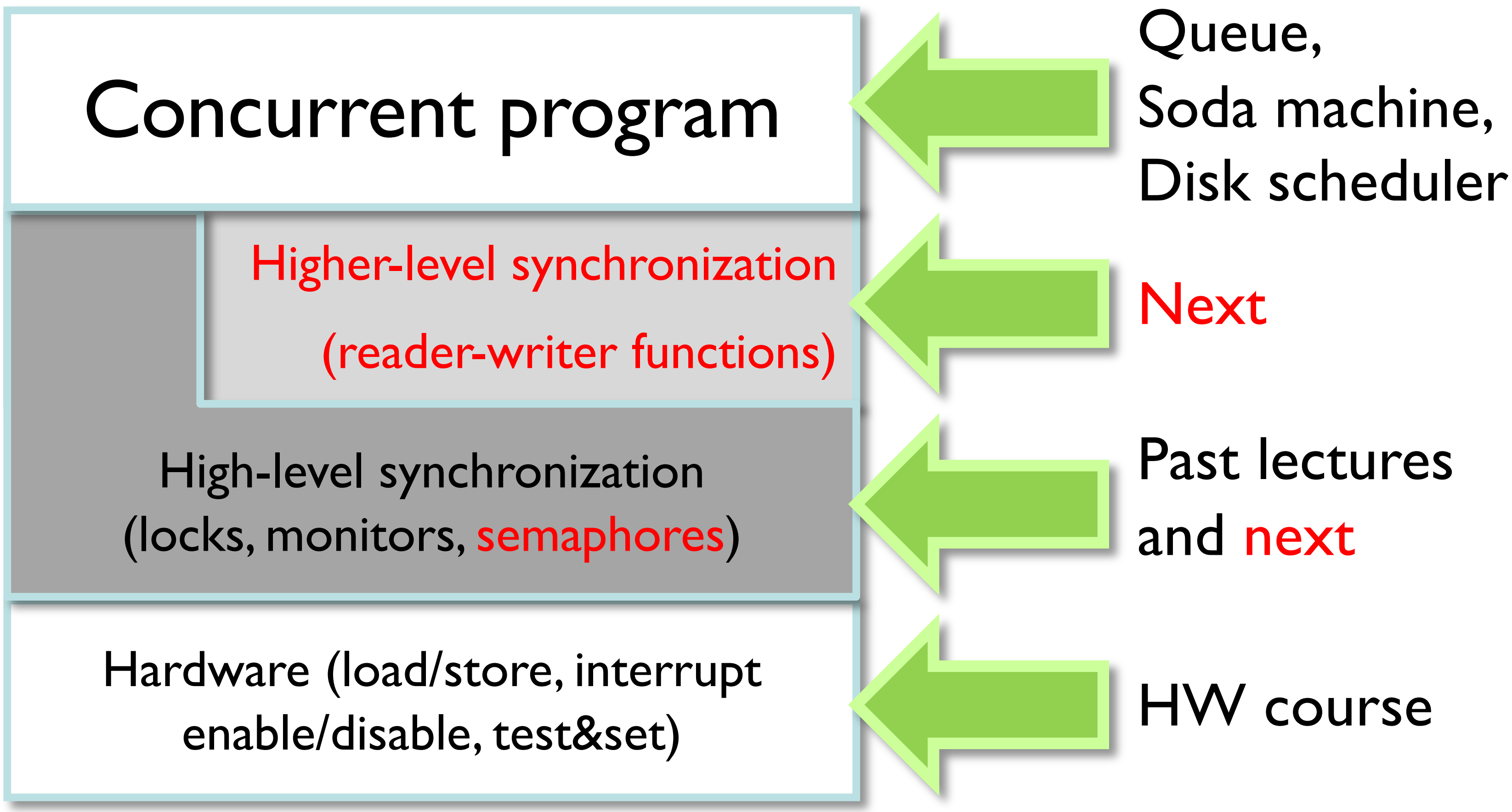
- Thread library includes the ability to:
  - Turn preemptions on and off (start_preemptions())
  - Preemptions may be:
    - Asynchronous (happen at random times every 10ms)
    - Synchronous (happen randomly but deterministically on every run)
      - Change change pattern by changing random seeds
  - **These should be used for testing**

Assignment 1 (Part 2)

- You write the thread library yourself
  - I.e., Implement all of the thread library functions
  - Will use your correct disk scheduler for testing
  - Will discuss more next week

- Get started on the disk scheduler now!
  - It will be challenging and take time
  - **Note: the autograder will not tell you why your program is failing**
    - It is your job to create tests for your program that test various conditions (e.g., boundary values)

Layers of synchronization



Locks thus far

- Lock anytime shared data is read/written
  - Ensures correctness
  - Only one thread can read/write at a time
- Would like more concurrency
- How, without exposing violated invariants?
  - Allow multiple threads to read
  - (as long as none are writing)

Problem definition

- Shared state that many threads want to access
  - Many more reads than writes
  - Peoplesoft, Ebay, any large database

1. When no writers, allow multiple readers
2. One writer at a time, when no readers

- Goal: build reader-writer locks using monitors

Reader-writer interface

- `readerStart` (called when thread begins reading)
- `readerFinish` (called when thread is finished reading)
- `writerStart` (called when thread begins writing)
- `writerFinish` (called when thread is finished writing)
- If no threads between `writerStart`/`writerFinish`
  - Many threads between `readerStart`/`readerFinish`
- Only 1 thread between `writerStart`/`writerFinish`

Solving reader-writer locks

1. **What are the variables/shared state?**
   - Number of readers (numReaders)
   - Number of writers (numWriters)
2. **Locks?**
   - 1 to protect all shared state (rwLock)
3. **Mutual exclusion?**
   - Only one writing thread at a time
4. **Ordering constraints?**
   - readerStart must wait if there are writers
   - writerStart must wait if there are readers or writers
   - Due to overlap, use one CV (ReaderOrWriterFinishCV)

Reader-writer code

```
writerStart () {
  lock (RWLock)
  while (numWriters > 0 ||
         numReaders > 0) {
    wait (RWlock, rowfCV)
  }
  numWriters++;
  unlock (RWLock)
}
```

```
readerStart () {
  lock (RWLock)
  while (numWriters > 0) {
    wait (RWlock, rowfCV)
  }
  numReaders++
  unlock (RWLock)
}
```

```
writerFinish () {
  lock (RWLock)
  numWriters--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

```
readerFinish () {
  lock (RWLock)
  numReaders--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

Reader-writer code

```
writerStart () {
  lock (RWLock)
  while (numWriters > 0 ||
         numReaders > 0) {
    wait (RWlock, rowfCV)
  }
  numWriters++;
  unlock (RWLock)
}
```

```
readerStart () {
  lock (RWLock)
  while (numWriters > 0) {
    wait (RWlock, rowfCV)
  }
  numReaders++
  unlock (RWLock)
}
```

```
writerFinish () {
  lock (RWLock)
  numWriters--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

```
readerFinish () {
  lock (RWLock)
  broadcast (rowfCV)
  numReaders--
  unlock (RWLock)
}
```

?

Sure, both are protected by RWLock.

Reader-writer code

```
writerStart () {
  lock (RWLock)
  while (numWriters > 0 ||
         numReaders > 0) {
    wait (RWlock, rowfCV)
  }
  numWriters++;
  unlock (RWLock)
}
```

```
readerStart () {
  lock (RWLock)
  while (numWriters > 0) {
    wait (RWlock, rowfCV)
  }
  numReaders++
  unlock (RWLock)
}
```

```
writerFinish () {
  lock (RWLock)
  numWriters--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

```
readerFinish () {
  lock (RWLock)
  numReaders--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

If writer leaves with waiting readers and writers, who wins?

Reader-writer code

```
writerStart () {
  lock (RWLock)
  while (numWriters > 0 ||
         numReaders > 0) {
    wait (RWlock, rowfCV)
  }
  numWriters++;
  unlock (RWLock)
}
```

```
readerStart () {
  lock (RWLock)
  while (numWriters > 0) {
    wait (RWlock, rowfCV)
  }
  numReaders++
  unlock (RWLock)
}
```

```
writerFinish () {
  lock (RWLock)
  numWriters--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

```
readerFinish () {
  lock (RWLock)
  numReaders--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

How long might a writer have to wait?

(Don't want to starve the writers!)

Priority

- We want to give waiting writer threads a higher priority than waiting reader threads
- This will also help prevent writer starvation
- How can this be accomplished?
  - More complexity
  - Keep track of waiting and active writers separately
  - Only allow new reader access if there are no waiting or active writers
  - This might starve readers, but we are assuming that there are far more readers than writers!

Prioritizing waiting writers

```
writerStart () {
  lock (RWLock)
  while (actWriters > 0 ||
         numReaders > 0) {
    waitWriters++
    wait (RWlock, rowfCV)
    waitWriters--
  }
  actWriters++;
  unlock (RWLock)
}
```

```
writerFinish () {
  lock (RWLock)
  actWriters--
  broadcast (rowfCV)
  unlock (RWLock)
}
```

```
readerStart () {
  lock (RWLock)
  while ((actWriters +
          waitWriters) > 0){
    wait (RWlock, rowfCV)
  }
  numReaders++
  unlock (RWLock)
}
```

```
readerFinish () {
  lock (RWLock)
  numReaders--
  broadcast (rowfCV)
  unlock (RWLock)
}
```