Discussion Week 3

TA: Kyle Dewey

Overview

- Concurrency overview
- Synchronization primitives
  - Semaphores
  - Locks
  - Conditions
- Project #1

Concurrency

- Looks easy
- Really hard to get right
  - Really hard
  - No seriously, borderline impossible

Race Condition

- Different results are possible based on different process/thread orderings
- Ordering may be correct 99.999% of the time

Deadlock

- Two processes/threads wait for each other to do something
- While they wait, they do not do whatever it is they are waiting for
- Potential outcome of a race condition

(Sort of) Real Deadlock Example

Critical Region

- A point in code where the ordering matters
- Almost always this is some state that is shared between processes/threads

Client

```
connect to server:port1
connect to server:port2
 do something with both
```

Server

```
accept from port1
accept from port2
do something with both
```

Fixing the Problem

- Do not share state
- Only share read-only state
- **Carefully** regulate write access to shared state

Regulation

- A critical region can be manipulated by only one thread at a time
- Need a way to enforce that at most one thread at any time point is in such a region

Solving in Java

- Java provides the `synchronized` keyword for blocks
- Only one thread at a time may access a block marked with the `synchronized` keyword

```
int x = 0;
public synchronized  void
set( int y ) {x = y;}
public int get() {return x;}
```

Who cares about Java?

- Many concurrency primitives work **exactly** like this, just with a little more work
- One call upon entrance to critical region, another upon exit
- The entrance and exit are implicit through blocks with Java

Semaphores

- Simply a shared integer
- One call decrements, another increments
- By convention, 0 is locked, and values > 0 are unlocked
  - Values < 0 mean?

Semaphores

- Increment/decrement are **atomic** - they are uninterruptible
- The highest possible number it can hold is equal to the max number of callers to the region it protects

Usage Example

Fix the Notebook Problem

NACHOS Semaphore Methods

- `P()`: wait until the value is > 0, then decrement
- `V()`: increment the value, waking up any waiting threads

NACHOS Semaphore Implementation

Spinlock

- Alternative to blocking
- A.K.A. busy waiting
- “Spin” in a tight loop
- More efficient for short critical regions

Everything In Between

- May spinlock under certain conditions
- May schedule differently if in a locked state
- Implementation can do whatever it wants

Project 1 Task 1

- Experiment according to instructions
- Explain the execution of multithreaded code
- Add semaphores and contrast the difference

Project 1 Task 2

- Implement locks - essentially semaphores with a maximum of one caller at a time
- Given all the semaphore code to look at
- Hint hint it is a special case of a semaphore

Lock Methods

- `Acquire()`: calling thread waits until lock is available, then grabs the lock
- `Release()`: calling threads gives up the lock

Lock vs. Semaphore

- Locks permit at most one thread in a region, not n
- Locks make sure that only the thread that grabs the lock can release the lock

Lock Example

Project 1 Task 3

- Implement conditions
- Requires a correct `Lock` implementation

Conditions

- Allow a group of threads to synchronize on a given condition
- Until the condition is true, they wait

Condition Methods

- `Wait( lock )`: release the given lock, wait until signaled, and acquire the lock
- `Signal( lock )`: wake up any single thread waiting on the condition
- `Broadcast( lock )`: wake up all threads waiting on the condition

Condition Semantics

- The lock should be owned by the calling thread
  - Only reason why the reference implementation's `Signal()` and `Broadcast()` needs the lock
- `Signal()` and `Broadcast()` require that the lock is currently held

Condition Example - Broadcast

Condition Example - Signal

Project 1 Task 4

- Identify and describe a race condition in a given section of code
- Fix the race condition using semaphores
- Fix it another way using locks and/or conditions

Identifying Race Conditions

- NACHOS is more or less deterministic
- Some of the hardest errors to find

Project Tips

- Start early
- Use the given implementation as a guide
  - Overcomplicated
  - Buggy
  - Ugly
- The Print() method is a lifesaver

FAQ

"What's the difference?"

- Not much
- Possible to implement some in terms of others
- Some may be more natural in different contexts

“Are these even working?”

- If everything is done correctly, the output remains the same for first task
- NACHOS thread scheduler is simple
- No interrupts
- All threads are part of the same program

“Why bother?”

- Change any of the aforementioned things, and it will matter big time
- Later projects will need this for correctness
- Gentle introduction to concurrency and synchronization primitives

“Conditions make no sense!”

- Name most people are used to: monitors
- http://www.java-samples.com/showtutorial.php?tutorialid=306 has an excellent example of usage (Java standpoint. Examples were adapted from this.)