CS 162
Discussion Section
Week 3

Who am I?

Mosharaf Chowdhury

http://www.mosharaf.com

Cs162-ta@cory.eecs.berkeley.edu

Office Hours: @ 651 Soda 4-5PM W && 9-10AM F

Research

Datacenter Networks

Cloud Computing

Project 1

- Can be found in the course website
  - Under the heading "Projects and Nachos"

- Stock Nachos has an incomplete thread system. Your job is to
  - complete it, and
  - use it to solve several synchronization problems

Project 1 Grading

- Design docs [40 points]
  - First draft [10 points]
  - Design review [10 points]
  - Final design doc [20 points]

- Code [60 points]

Design Document

- Overview of the project as a whole along with its parts

- Header must contain the following info
  - Project Name and #
  - Group Members Name and ID
  - Section #
  - TA Name

Design Document Structure

Each part of the project should be explained using the following structure

- Overview
- Correctness Constraints
- Declarations
- Descriptions
- Testing Plan

Design Docum

- First draft [9th Feb]
  - Initial ideas
  - At most **10 pages**
- Final draft [22nd Feb]
  - At most **15 pages**

- Include diagram showing interactions between system components

> Talking much about oneself can also be a means to conceal oneself
> -- *Friedrich Nietzsche*

> Talking too much about your design is a means to conceal your ideas. **Overdo it to lose 20%.**
> -- *CS162 Teaching Staff*

Project 1 Dea

1. Signup for a timeslot in your section.
2. ***If anyone is absent, everyone loses 20% on the whole project***

- Initial design: 9th Feb
- Design reviews: Week of 13th Feb
- Code: 21st Feb
- Group evaluations, test cases, and final design docs: 22nd Feb

Synchronization.

Say what?!

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread excludes the other while doing its task
- Critical Section: piece of code that only one thread can execute at once
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Examples of Read-Modify-Write

-
  ```
  test&set (&address) {     /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
  ```

-
  ```
  swap (&address, register) { /* x86 */
      temp = M[address];
      M[address] = register;
      register = temp;
  }
  ```

-
  ```
  compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
  ```

Implementing Locks with test&set

- Simple

Busy-waiting

```
int val          0; // free
Acquire() {
   while (test&set(value)); // while busy
}
Release() {
   value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

test&set without busy-waiting? => Nope

- Only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;

Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can’t we do it just before or just after the sleep?

Life without locks?

Semaphores

- A Semaphore has a non-negative integer value (S) and supports the following two operations
  - P(S) = Down(S) = Wait(S)
  - V(S) = Up (S) = Signal(S)

- Note that P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch

Classical definition of Wait and Signal

Busy-waiting

```
Wait(S) {
      while (S <= 0) { }
      S = S - 1;
}

Signal(S) {
      S = S + 1;
}
```

Blocking implementation of Semaphore

```
Wait(S) {
    S.val = S.val – 1;
    if (S.val < 0) {
        S.list.add(calling_thread);
        sleep();
    }
}

Signal(S) {
    S.val = S.val + 1;
    if (S.val <= 0) {
        T = S.list.removeHead();
        wakeup(T);
    }
}
```

```
Initialize(S, X) {
    S.val = X
}
```

Mutex

- Used to control access to shared data
  - Only one thread can execute inside a Mutex
  - Others are blocked until the Mutex is unlocked
- Can be implemented using Semaphore
  - Just initialize your Semaphore to 1

Condition Variables (CV)

- Used to wait for specific events; e.g.,
  - When free memory is too low; wake up the garbage collector
  - New packet arrived from the network; push it to appropriate handlers

- Each CV has a single associated Mutex
  - Condition of the CV depends on data protected by the Mutex

Condition Variables Semantics

- `Wait`
  - Atomically unlocks the Mutex and blocks the thread

- `Signal`
  - Thread is awaken *inside* `Wait`
  - Tries to Lock the Mutex
  - When it (finally) succeeds, returns from `Wait`

CV Example

```
Mutex io_mutex;
Condition non_empty;

Consumer:
Lock (io_mutex) {
       while (port.empty())
              Wait(io_mutex, non_empty);
       process_data(port.first_in());
}

Producer:
Lock (io_mutex) {
       port.add_data();
       Signal(non_empty);
}
```