

More Project Info

- After this section, you should know enough to be able to design solutions for the project.
- CVS Groups have been created
 - o Unless you got an email from me about errors, should be able to import your nachos directory into cvs.
- Schedule: Looking Ahead. . .
 - o Wednesday Feb 18th: Design Doc due
 - Submitted online
 - UPDATE:
 - **My previous 3 pages or so assertion was on the short side. To do this right, you'll likely have more.**
 - PLEASE no more than needed!
 - o Wednesday Feb 18th: Design review signups will go online, on my section homepage.
 - Design reviews will be held the following Monday and Tuesday.
 - Each design review will be 20 minutes.
 - You will present your solution to me on the board.
 - I will answer questions and make sure you're on the right path.
 - These are graded!
 - I will return your design document to you after your review.
 - o Monday Feb 23rd – Tuesday Feb 24th: Design reviews
 - Location TBD
 - o Thursday, March 4th: Project 1 code due
 - Submitted online
 - o Friday, March 5th: Project 1 final design documents due
 - Submitted online
 - These will consist of your REVISED design document and answers to questions in the project handout.

Design Document Expectations

- How you will solve each of the project sections
 - o Psuedo-code ONLY!!! NO JAVA
 - o What state you'll need to maintain
 - o Description
- How you will test each project section
 - o Details of tests you plan to run.
 - o Design your tests thoroughly!

No Quiz!

- Will spend last few minutes taking photos for a photo roster of the section.
- Check out my quizzes on scheduling and synchronization from last semester.

Questions from Lecture??

Clarification: Threads vs. Processes

- Process: UNIX OS unit of scheduling
 - o Each process has its own address space.
- User-level Threads: Java, pthreads, etc. . . .
 - o Share a process address space
 - o OS doesn't know what's going on
- **QUESTION:** Is this good or bad?
- **ANSWER:** OS can schedule a process whose threads have nothing to do.
- Kernel Threads: threads that the OS DOES know about and control.
 - o Still share an address space
 - o We haven't talked much about these—check your book for details.

Locks, Semaphores, Monitors, and Condition Variables

- Locks
 - o Lock the critical section, wait if can't get lock.
 - o On release, wait up anyone waiting to acquire.
- Semaphores
 - o Like a generalized lock, with a counter
 - You can't get the value out of the counter.
 - o Two functions:
 - P() – Decrement if semaphore is positive
 - Sleep otherwise
 - V() – Increment, waking up anyone waiting
- **QUESTION:** What do 'P' and 'V' really mean?
- **ANSWER:** 'proberen' means 'try' or 'attempt' in Dutch, and 'verhogen' means 'increase' or 'raise'
 - o Mention opposite from down trick!
 - o Semaphores are great for resource control
 - Example: Producer/Consumer problem
- Condition variables
 - o Let us sleep within a critical section by atomically releasing lock when we sleep.
 - Obviously, must use with a lock

- Three functions:
 - Wait() – Release lock, sleep, must re-acquire on wake.
 - Signal() – Wake up waiter
 - Broadcast() – Wake up ALL waiters
- **QUESTION:** Do condition variables keep state? Do semaphores?
- **ANSWER:** By themselves, semaphores keep state: i.e. if you increment a semaphore, that means a thread can continue in the future even if no thread is waiting. A signal when no thread is waiting is lost.
- Monitor
 - High-level synchronization primitive.
 - Consists of a lock and zero or more condition variables
 - Encapsulates shared data or method and regulates access.
 - Mesa vs. Hoare:
 - Mesa: signaler keeps lock, waking thread must wait on acquire
 - Hoare: signaler releases lock, waking thread acquires and runs.
- These things can be built out of each other
- **QUESTION:** How to use semaphores to implement a lock?
- **ANSWER:** Make a Lock class using a binary semaphore, and map acquire() and release() accordingly. Start semaphore at 1.
- **QUESTION:** How to use semaphores to implement condition variables?
- **ANSWER:**
 - Keep a counter of how many threads are trying to get this condition variable. Have a semaphore that starts at 0
 - Wait: update counter, call semaphore->P()
 - Signal: increment semaphore only if there is a thread waiting.
 - Why? (otherwise signal doesn't behave correctly)
 - Broadcast: increment semaphore for each thread waiting.
- **QUESTION:** How to use lock to implement semaphores?
 - Brainteaser for your week. ☺

Intro to Scheduling

- Definition: Preemptive vs. Nonpreemptive
- Some basic types of scheduling:
 - Round Robin (pre-e)
 - Give each thread an equal slice of time
 - Pros: fair, easy
 - Cons: average waiting time can get bad, io-bound processes suffer
 - Shortest Job First (non pre-e)
 - Always run the shortest job first

- **QUESTION:** Can we actually implement SJF?
- **ANSWER:** How can we predict the future—i.e. how do we know how long a job is going to run?
 - Pros: optimal waiting time!
 - Cons: um, impossible to implement. . .
- First Come, First Served (non pre-e)
 - Run threads in FIFO order
 - Pros: easy, gets jobs done in order of submission
 - Cons: can have terrible response times
- Priority Scheduling (pre-e)
 - Always run the thread with the highest priority!
 - Pros: gives good control over what runs first
 - Cons: unfair

Priority Inversion

- What happens when a high-priority thread is waiting on a lock held by a low-priority thread, and another high-priority thread comes in?
 - Oops. Deadlock.
- One way of fixing: Priority donation.
 - Have waiting high-priority thread donate priority to low-priority thread!
- **QUESTION:** Any problems here?
- **ANSWER:** What if the low-priority thread is waiting on another thread? Need to recurse up the dependency list! (think about this when you design your solution)

Photo Session!