Problems with Semaphores

- Used for 2 independent purposes
  - Mutual exclusion
  - Condition synchronization
- Hard to get right
  - Small mistake easily leads to deadlock

May want to separate mutual exclusion, condition synchronization

Monitors (Hoare)

- Abstract Data Type
  - consists of vars and procedures, like C++ class
  - 3 key differences from a regular class:
    - **only one thread in monitor at a time (mutual exclusion is automatic)**
    - special type of variable allowed, called "**condition variable**"
      - 4 special ops allowed only on condition variables: wait, signal, broadcast, notempty
    - no public data allowed (must call methods to effect any change)

Wait, Signal, Broadcast

- Given a condition variable “cond”
  - Wait():
    - thread is put on queue for “cond”, goes to sleep
  - Signal():
    - if queue for “cond” not empty, wake up one thread
  - Broadcast():
    - wake up all threads waiting on queue for “cond”

Semantics of Signal

- Signal and Wait (Hoare)
  - signaler immediately gives up control
  - thread that was waiting executes
- Signal and Continue (Java)
  - will be used in this class
  - signaler continues executing
  - thread that was waiting put on ready queue
  - when thread actually gets to run:
    - **state may have changed!** use “while”, not “if”

Monitor Solution to Critical Section

- Just make the critical section a monitor routine!

Readers/Writers Solution using Monitors

- Similar idea to semaphore solution
  - simpler, because don't worry about mutex
- When can't get into database, wait on appropriate condition variable
- When done with database, signal others

Note: can't just put code for "reading database" and code for "writing database" in the monitor (couldn't have >1 reader)

Differences between Monitors and Semaphores

- Monitors enforce mutual exclusion
- P( ) vs Wait
  - P blocks if value is 0, Wait always blocks
- V( ) vs Signal
  - V either wakes up a thread or increments value
  - Signal only has effect if a thread waiting
- Semaphores have “memory”

First Attempt: Implementing Monitors using Semaphores

Shared vars:

sem mutex := 1 (one per monitor)

sem c := 0; int nc := 0 (one per condition var)

Monitor entry: P(mutex)

Wait(mutex):

nc++; V(mutex); P(c); P(mutex)

Signal(mutex):

if (nc > 0) then {nc--; V(c);}

Monitor exit: V(mutex)

Correct Implementation of Monitors using Semaphores (Assume that "tid" is the id of a thread)

Shared vars:

sem mutex := 1; (one per monitor)

int nc := 0; List delayQ (one per condition var)

sem c[NumThreads] := 0; (one entry per thread; one entry per thread per condition works also)

Monitor entry: P(mutex)

Wait(mutex):

nc++; delayQ->Append(tid); V(mutex); P(c[tid]); P(mutex)

Signal(mutex):

if (nc > 0) then {nc--; id = delayQ->Remove( ); V(c[id]);}

Monitor exit: V(mutex);

In-Class Exercise

- Implement a barrier using monitors
  - Hint: use the *notempty* function