Java Monitors

From
“Alfonse, Wait Here For My Signal!”,
Stephen J. Hartley, SIGCSE 3/99 New
Orleans, LA, USA

```java
class Monitor extends ... {
    private ... // data fields (state variables)
    public synchronized type method1(...) {
        ...
        notifyAll(); // if any wait conditions altered
        while (!condition) try { wait(); }
            catch (InterruptedException e) {} 
        ...
        notifyAll(); // if any wait conditions altered
    }
    ...
}
```
Can’t Control Lock Acquisition

- The thread blocked the longest on a monitor synchronized method call is not guaranteed to be next in the monitor when the monitor lock is released
- The thread blocked the longest on a monitor wait() call is not guaranteed to be the one chosen to reenter the monitor when a notify() is done

Barging is possible

- The signaling discipline is signal-and-continue so barging is possible:
  - a thread waiting for the monitor lock so it can call a monitor synchronized method might get the lock before another already waiting thread reacquires the lock.
- Usually best to put condition tests in a while loop
Can’t distinguish different waiters

• Each monitor object has a single nameless anonymous condition variable.
  – we cannot signal one of several threads waiting on a specific condition with a notify()
  – It is safer to use notifyAll() to wake up all waiting threads
  – However, this can be very inefficient and can greatly increase overhead

Notify Early & Often

• A notifyAll() needs to be done by a thread before a wait() if any state variables were altered by the thread after entering the monitor that might affect other thread waiting conditions
• This also applies before leaving the monitor (returning from the method).
Handling InterruptedException

- Ignoring InterruptedException with an empty catch block is okay as long as we are using notifyAll() instead of notify().
  - Also not okay in using if...wait()
- May be better to have the containing method throw the exception back to the method's caller so the latter knows an interrupt occurred

A Counting Semaphore

- Can be used to allow no more than \( n \) threads into a critical section
- Has two operations
  - P (down)
    - If semaphore’s value is 0 block, else decrement value
  - V (up)
    - If one or more threads are blocked inside P then unblock one, else increment the semaphore’s value
Desired Properties

- Safety: No more Ps complete than is possible
- Liveness: As many Ps complete as possible
- No barging: No sneaking in and stealing the value from a notified thread
- Single signals: The V operation is implemented with notify() rather than notifyAll()
- Passed back exceptions: If a thread blocked inside P in wait is interrupted, the exception is thrown back to the calling thread

1st Attempt

- A thread calling P might get in the monitor before a waiting thread reacquires lock
- Both P calls will then succeed when only one should, a safety violation
- Possible fix:
  - Change the if in front of the wait to a while
2nd Attempt

- Several threads blocked inside P and some other thread does a V and notify
- One of the waiting threads is moved from the wait set to the (re)acquire monitor lock set
- Several other threads barge in and call V before the notified thread reenters the monitor
- No notifies are done

2nd Attempt

- Also have a barging problem that can cause starvation
- A thread calling P can barge ahead of a notified thread, causing the notified thread to wait again
- Fix: allow semaphore value to go negative, in which case its absolute value is the number of threads blocked inside P in wait
3rd Attempt

- Suppose the semaphore value is -1 due to one thread blocked in wait 0 inside P. Then suppose that thread is interrupted.
- The value is left at -1 even though no threads are blocked in P.
- The next V will increment the value to 0 whereas it should now be 1.
- Fix: count the wait set separately, rather than letting the semaphore value go negative.

3rd Attempt

- Suppose several threads are blocked inside wait and then one of them is notified and then interrupted before it reacquires the monitor lock.
- The notify gets “lost” in that one of the other waiting threads should now proceed.
- Fix: catch the exception when a thread is interrupted out of wait and regenerate the notify.
4th Attempt

- Suppose a thread blocked inside wait is interrupted without being notified
- The notify it does in its catch block will move some other waiting thread, if there is one, out of the wait set into the lock (re)acquire set
- When that thread gets back into the semaphore monitor, its P operation will complete successfully, an error
- Fix: add a notification flag so a thread can distinguish between the notify in V and the notify in the catch block

5th Attempt

- Suppose multiple threads are blocked inside wait in P and some other thread calls V
- Several additional V calls might barge into the monitor before a notified thread reacquires the monitor lock, so we can lose notifies
- Fix: make the notification flag an integer counter to avoid lost notifies
6th Attempt

- One of several remaining problems:
- A thread calls P and waits because the semaphore value is 0. Then, some thread calls V. Before the notified thread reacquires the monitor lock, it is interrupted (value and notifyCount are 1)
- Some thread calls P and it completes successfully. value == 0, notifyCount == 1.
- Then several threads call P and wait.
- Next, one of the waiting threads is interrupted, causing its wait call to throw an exception. Due to the catch block, the thread reenters the monitor, calls notify and rethrows the exception.

6th Attempt

- Now, one of the remaining waiting threads reenters the monitor
- Because notifyCount == 1, the thread thinks V was called and its P completes erroneously
- Fix: decrement notifyCount if it exceeds the waitCount whenever a P is not required to call wait
7th Attempt

- It works. Wasn’t that easy?