A bit more about notify() and notifyAll()

- Review behavior of wait(), notify() and notifyAll()
- Discuss appropriate when to use notifyAll() instead of notify()

What does wait() do?

- Put the current thread on the wait queue associated with the object whose lock is held by the current synchronized block or method
- Release the lock
- The thread is now asleep and will not run again until notified (or interrupted)
- When the thread is awoken, re-acquire (contend for) the lock
- Return from wait()

What do notify() and notifyAll() do?

- notify() – makes one thread (or none) currently asleep in the wait queue of the object ready to run.
- notifyAll() – makes all threads currently asleep in the wait queue of the object ready to run.
- Neither notify() nor notifyAll() gives the lock to the awoken thread – it remains held by the thread that calls notify() or notifyAll()
- The awoken thread(s) must contend for the lock with any other threads that want it, whether just awoken or not

notifyAll() is necessary more often than you might think

```
synchronized put(o) {
   while (occupied) { wait(); }
   mb = o;
   *
   occupied = true;
}
synchronized Object get() {
   while (!occupied) { wait(); }
   *
   occupied = false;
   return mb;
}
```

- (Wrong) Theorem: number of put'ter threads wait'ing and number of get'ter threads waiting are not both > 0.
- (Wrong) Corollary: notify() is sufficient at *s

What goes wrong?

- Suppose 4 put'ter threads and 4 get'ter threads and they executed as follows
- put, put, put, put: one after the other
  - One has completed occupied is true and 3 are wait'ing
- get, get, get, get, all but the first queued on the synchronization lock
  - The first get sets occupied to false, and does a notify which wakes up one of the put'ters. It is now enqueued behind the other 3 get'ters for the lock
  - The remaining get'ters acquire the lock, see occupied false, and wait
  - We now have 2 put'ters waiting and 3 get'ters waiting
  - The previously wakened put'ter finally sets occupied true, and calls notify() which wakes up a put'ter
  - That put'ter sees occupied true, and waits. All the remaining threads are now waiting – deadlock!

What's the problem?

- A thread was awakened when it could not proceed
- The effect is as-if the notify was lost (see pp. 302-303)
- The problem is that Java's notify is too blunt an instrument – it does not allow distinguishing threads that are waiting for different states – in this case for the mbox to be empty and for the mbox to be full
- Therefore: notifyAll() is the preferred approach unless all threads are waiting for the same property on the state
- Also therefore: wait() goes in a loop, not an if!
- Beginning with Java 5.0 the ReentrantLock class (Chapter 13) offers multiple "condition" queues to solve this problem but it is somewhat harder to use than synchronized methods and wait()

Introducing the Erlang Language

- Why Erlang?
  - Exemplifies the “you can have code and concurrency that is free from side effects…there is no other way” perspective
  - Real language used to build real systems
  - A pretty good, practical book exists to use as a text
  - Multi-core applications
  - Fault-tolerant applications

Erlang is a *Functional* Language

- Lots of definitions but some implications
  - Computation is primarily done by evaluating functions that take immutable values as inputs and produce immutable values as outputs
  - Programs are written by defining functions (in terms of other functions)
  - Functions are side-effect free
- Other functional languages: Haskell, Standard ML, LISP (in some variants)
- You can *program functionally* in imperative languages (e.g. C, Java, etc.) but some make it easier than others
- Automatic storage management (GC) is almost always a feature of functional languages (but not vice-versa)

Erlang values

- Numbers
  - Integers with arbitrary precision (dd*)
  - Floating point (d*d.dd*)
- Tuples
  - {value [, value]*} ; 1-tuple allowed and different from the contained value: 1 != {1}
  - Tuple models data structures of known fixed size
- Atoms
  - Alphanumeric sequences starting with lower-case letter (may contain but not start with _ and @)
  - Any sequence enclosed in single quotes
  - The value of an atom is itself
- Lists
  - Sequence of values of unknown size
  - Compare to python tuples and lists – Erlang lists are not mutable!

An Erlang idiom using tuples

struct person { char *first; char *last }; -- C

{person, {first, “carl”}, {last, “hauser”}} – this is just a value; note the use of atoms as tags in the tuples

Erlang is a dynamically, strongly typed language. Using tags helps people keep things straight about programs. Compare {“carl”, “hauser”}

“strongly typed” means you can’t use a value incorrectly by passing it to an operator that interprets it as something else thus computing a wrong answer. The computation fails at that point.

Lists

- List values can be written directly – [value, value, …, value] or created one element at a time using the *list constructor* [value, …, value | list]
- The first element of a list is called the *head*
- Everything else is the *tail*

Strings

- Erlang strings are really just lists of integers
- 5> "abc".
  "abc"
  6> [1|"abc"].
  [1,97,98,99]
- Shell has a special rule for printing lists that contain only printable characters
- The integer for a character can be written $c
- 7> $a.
  97
- This feels like a *hack* to me

Erlang variables

- Start with upper-case letter (enforced convention for human beings sake)
  - X is a variable; x is an atom
- Are *write-once*
  - A new variable is called *unbound*
  - After assignment a variable is said to be *bound* and cannot be further changed (limits the possibility for side effects!)
    - "assignment" is technically the wrong term – more properly called *binding*

An aside on vocabulary

- Learn to distinguish
  - bind – to tie together
    - "X = 3 in Erlang binds the variable X to the value 3"
  - bound – past tense of bind
    - "Each variable starts out in unbound state; once bound to a value it cannot be re-bound to a different value."
  - bounded – contained, limited,
    - "the value is bounded above by 3"

Erlang approach to binding: assertion of equality

Idea: a form such as X = {1, 3} is an assertion of equality between X and {1, 3}. If something equals something else then they can't later be not equal. Hence the bind-once rule.

```
9> X = {1,3}.
{1,3}
10> x.
x
11> X.
{1,3}
12> X=X.
{1,3}
13> X={1,3}.
{1,3}
14> X={1,4}.
** exception error: no match of right hand side value {1,4}
```

Binding using pattern matching

- You may recall this idea from 355 if you talked about ML there
- What if the LHS is not a single variable but something that looks like a value but with variables embedded at some places

```
17> {Y,Z}=X.  % {Y,Z} here is called a pattern and we see that the = operator is
    capable of much more than just assignment.
{1,3}
18> Y.
1
19> Z.
3
```

- You may have also noticed this in Python

```
>>> a = [1,2,3]
>>> [d,e,f] = a
>>> d
1
>>> e
2
>>> f
3
```

More pattern matching - lists

```
22> L1 = [0,1,2,3,4].
[0,1,2,3,4]
23> [H|T] = L1.
[0,1,2,3,4]
24> H.
0
25> T.
[1,2,3,4]
```

Guide to the textbook for Sequential Erlang

- Chapters 1&2 introduction and overview of Erlang Concurrency – skim
- Chapters 3&4 core of sequential Erlang – worth a fairly close reading
- Chapter 5: Records and maps – we won't be using this
- Chapter 6: Read for basic familiarity
- Chapter 7: binaries and bits – skip
- Chapter 8&9: a good reference but don't need to read
- Chapter 10: become familiar and refer back to it while programming

By Wednesday

- Install Erlang on your computer
  - Version doesn't matter critically. The maps described in Chapter 5 become available only with R17, but we won't be using them
- Chapter 3, exercises 2 & 3 would be good practice
- Chapter 4, exercise 5, 6 & 7 would be good practice