LECTURE 1.7

MONITORS AND MESSAGES

COP4600

Dr. Matthew Gerber

2/15/2016

MONITORS

Monitors (5.8)

- You already have some notion of an *object*: an abstract data type that contains:
  - Public and private data
  - Public and private *methods* – i.e., functions and procedures that operate upon the object
- A *monitor* is an object with two extra properties. The first:
  - *Only one function of a monitor can execute at the same time*

Implementing Monitors, Part I

```
class acts_like_a_monitor
  mutex m = true

  method method1(parameters)
    wait(m)
    // Do everything else
    signal(m)
  end // method1

  method method2(parameters)
    wait(m)
    // Do everything else
    signal(m)
  end // method2

end // acts_like_a_monitor
```

Condition Variables

- Monitors also provide *condition variables* internally
- A condition variable is a lot like a queued mutex, but has two important differences in behavior
- The first:
  - A condition *cannot* be “free” in the sense a mutex can
  - Waiting on a condition *always* waits
  - Signaling a condition can *only* free a process *already* waiting on it
- The second:
  - A thread that is waiting on a condition variable *does not* occupy the monitor

Producer-Consumer with Monitors

```
monitor pcq {
  data buffer[BUFSIZE]
  int in = 0, out = 0
  condition full, empty

  offer(data d) {
    if (in + 1) % BUFSIZE == out {
      wait(full)
    } // full
    buffer[in] = d
    in = (in + 1) % BUFSIZE
    if in == (out + 1) % BUFSIZE {
      signal(empty)
    } // no longer empty
  } // offer

  data request {
    if in == out {
      wait(empty)
    } // empty
    request = buffer[out]
    out = (out + 1) % BUFSIZE
    if out == (in + 2) % BUFSIZE {
      signal(full)
    } // no longer full
  } // request
} // pcq
```

```
producer_thread {
  data d
  while true {
    d = produce_item()
    pcq.offer(d)
  }
}

consumer_thread {
  data d
  while true {
    d = pcq.request()
    consume_item(d)
  }
}
```

Monitors in Practical Use

- This doesn't seem like we've gained anything – **but**
- That's just because of how trivial a synchronization problem this raw producer-consumer example is
- We've put all the synchronized behavior in *one place*: the monitor
  - No longer synchronizing in the "main" code of the threads at all
  - Threads' code contains logic particular to each thread
  - Monitor's code contains all the logic that lets the threads interact
- …in producer/consumer, the threads don't have much logic to them to begin with

Condition Variables and Signaling

- One problem with this:
  - *What do we do when we signal?*
  - If we allow the queued process to resume immediately, both it and the signaling process now occupy the monitor
- Three choices:
  - **Signal and Wait:** The signaling process is locked until the resumed process leaves the monitor
  - **Signal and Continue:** The signaling process continues, and the resumed process waits until it can enter the monitor normally
  - **Concurrent Pascal Method:** Signaling *leaves the monitor* (don't signal until you're done with your critical section)

MONITORS IN LANGUAGES

Monitors in Java

- In Java, *every* object is a monitor
  - (…okay, more accurately, every object *has* a monitor *available*)
- To declare a method as working like a method of a monitor, just declare it as a `synchronized` method

```
    public synchronized void something() {
        …
    }
```

- You can also declare `synchronized` *blocks*
  - *(Think twice before you do this)*
- `wait()` and `notify()` implement a single condition variable per object for "free"
  - For anything more complex than that, you'll need to use `Lock` objects and their associated `Condition` interface

Monitors in C#

- In C#, there actually are no monitors
  - (Okay, you can use `[MethodImpl(MethodImplOptions.Synchronized)]` on a method if you *really* want to)
  - (It's actually a bad idea because of how C#'s low-level synchronization functionality works)
- C# provides a type of lock variable *called* a `Monitor`, associated with a specific object
- You can use `Monitors` to *simulate* the actions of monitors yourself
  - `Enter()` and `Exit()` are the key methods
  - They work exactly the way you think they do
- Similarly to Java, C#'s `Monitor` gives you one condition variable for free with the `Wait()` and `Pulse()` methods; if you want more than that, you'll have to use the `WaitHandle` class

INTER-PROCESS COMMUNICATION

More About Message Passing

- All message passing involves *sending* and *receiving*.
- There are three types of addressing…

- Symmetric

```
send(P, message)       message = receive(P)
```

- Asymmetric

```
send(P, message)       receive(var message, var sender)
```

- Mailbox *(but how do the rules work?)*

```
send(M, message)       message = receive(M)
```

More About Message Passing (3.4.2)

- Sending and receiving can be either *blocking* or *non-blocking* – including several queued variations
  - *Blocking sends* force a process to wait until there is space in the send queue
  - *Non-blocking sends* fail (not the same thing as producing an error) if there isn't space in the send queue
  - *Blocking receives* force a process to wait until data are available
  - *Non-blocking receives* fail if no data are available
- Producer-consumer with blocking message passing available is trivial. Do you see why?
  - C# actually *has* a container class called `BlockingCollection`!

Pipes (3.6.1)

- The *pipe* model implements message passing using the file metaphor
  - Varying degrees of complexity and flexibility
  - *Ordinary* pipes can only communicate between parent and child processes
  - *Named* pipes, once created, can communicate between any number of processes
  - They sit in the file system, and can be opened, closed, read and written by any process with permissions to do so

Sockets (3.6.3)

- *Sockets* are metaphors used for pipes that go between machines on a network
  - Properly, the socket is the *endpoint* of the pipe
  - Each active network connection therefore has *two* sockets
- Rather than having a filename, a socket has a network address and port
- Sockets are the fundamental metaphor used by TCP/IP networking
  - …and are based on pipes
  - …which are based on message passing

NEXT TIME: DEADLOCKS