Chapter 12: Concurrency

November 15, 2017

Concurrency: logically simultaneous, not necessarily physically so

- Parallel: physical simultaneity

Parallelism

- Quite common to have multiple cores these days; indeed, it's become the rare case for any processor to support only one core.
- As the text notes, instruction-level parallelism (ILP) was of great interest back before the turn of the millenium;
- Vector parallelism refers to the application of one operation on multiple data elements; SIMD is the most common realization of this technique, and Intel/AMD have create a lot of SIMD instructions (and ones to handle quite large registers!)

Levels of parallelism

- "Black box": you don't even know it's there.
- Disjoint tasks: parallel execution over disjoint data
- Synchronicity (but also see RCU-type parallelism)

The cases for and against multithreading

- Multithreading seems reasonable when you have multiple logical processors that can act in concert; certainly some tasks, such as complex i/o driven tasks, seem naturally suited to having separate threads
- However, the great success of the last few years has been the resurrection of event-driven programming, which generally does not use multiple threads, except occasionally as a means to make use of multiple processors. (Your text discusses this alternative rather dismissively in the section title "The Dispatch Loop Alternative".)

Communication and synchronization

- 12.2.1 of your text is written *very* artfully; pay very close attention to the use of "can" as a verb.
- Generally, we have seen communication via either message passing or some sort of shared memory mechanism (though it is possible to mix these two, such as with distributed shared memory models, as your text mentions in the Design and Implementation note on page 587.)
- Synchronization *can* be implemented by, say, spinning (busy-wait), or by blocking.

Languages and libraries

- In higher-level languages, threading can be done at the language level, at the compiler "extension" level, or via a library.
- At the assembly language level, generally you have to ask the operating system to help you (which is what is silently happening in the higher-level languages in the above three methods.)

Languages with significant built-in support for concurrency

- Ada
- Rust
- Clojure
- Erlang
- D
- Go
- Occam

Expressions of parallelism

- Languages that support parallelism generally use some variation on the following themes
  - Co-begin
  - Parallel loops
  - Launch-at-elaboration
  - Fork/join
  - Implicit receipt
  - Early reply

Co-begin

```
co-begin    -- all n statements run concurrently
    ... [stmt1]
    ... [stmt2]
    ... [stmt3]
    ...
    ... [stmtN]
co-end
```

Parallel loops

- Instead of giving series of discrete statements, give the same expression a differing index (SIMD over an index):

```
Parallel.For(0,3,i => { somefunction(i); })
```

Parallel loops

- Or use map/fold over container elements:

```
Parallel.Map( somefunction somelist );
Parallel.Fold( somefunction someaccum somelist );
```

Launch at elaboration

- Ada has this; when a task is elaborated, it starts executing.

Fork/join

- The idea here is a process/thread starts another thread with a "fork", and then at some point does a "join" to reap the finished process.

Implicit receipt

- I haven't seen this is in use under the name "implicit receipt". I think that the author is referring to something like accept(2) (which creates a new file descriptor for the connection) followed by a fork(2) in Unix-land.

Implementation

- In Unix-land, we have
  - fork(2) / wait(2) which creates two distinct processes
  - clone(2) / wait(2) which creates a thread in the same process
- Using these two mechanisms, we can create implementations such as pthreads

Synchronization

- mutexes : a locking mechanism that allow one thread to claim and release an exclusive lock; other threads that ask for the same mutex have to wait for the first thread to release the mutex

Synchronization

- Spin locks: an implementation of mutex that uses busy-waiting. Used quite a bit in the Linux kernel (see, for instance, Kernel Locking, but otherwise I don't know of a common use for it these days.)

Synchronization

- Barriers: force all threads come to a single point before continuing. Not really a metaphor that I have seen a lot of use of, though it is certainly mentioned a lot in the literature.

Semaphores

- A popular and longstanding mechanism. A “post” operation increments the semaphore; if it becomes greater than zero, then any “waiting” processes are woken. A “wait” operation attempts to decrement the semaphore; if the semaphore is currently greater than zero, it is decremented and the operation continues; otherwise, it blocks until it is possible to perform the decrement.

Transactional Memory

- Going lock-free with transactional memory systems: It does seem to have promise, and there are a lot of implementations.
- You declare a code section to be “atomic”, and the system takes responsibility for trying to execute these in parallel. If the system supports rollback, it can even try “speculative” computations.

Hardware support for transactional memory

- So far, Intel has provided some hardware support with "RTM" and "HLE" support; the machine (Ivy Bridge) that I am using to make notes seems to list these both as off:

```
$ cpuid
   [ ... ]
   extended feature flags (7):
      [ ... ]
      HLE hardware lock elision                      = false
      [ ... ]
      RTM: restricted transactional memory           = false
```

Message Passing

- Message passing for the win! Of all of the mechanisms discussed in this chapter, the one with the most success (by far, in my estimation) is that of message passing.