Notes for CSC 2/454, Nov. 2, 4, and 9, 2020

==========

Concurrency (take 258 to learn more)

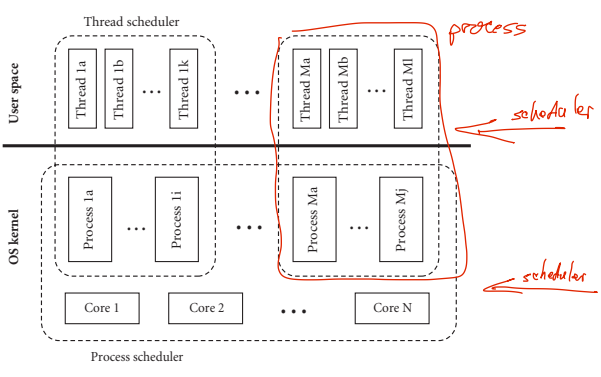
A ***process*** or ***thread*** is a potentially-active execution context.
Classic von Neumann (stored program) model of computing has single thread of control. Parallel programs have more than one.
A process can be thought of as an abstraction of a physical ***processor***.

Processes/threads can come from
- multiple CPUs
- kernel-level multiplexing of single physical machine
- language or library level multiplexing of kernel-level abstraction

They can run
- in true parallel
- unpredictably interleaved
- run-until-block

Most work focuses on the first two cases, which are equally difficult to deal with.

In the most common (but by no means universal :-(use of terminology, each ***processor*** chip has one or more ***cores***, each of which has one or more ***hardware threads***. The operating system multiplexes one or more ***kernel-level threads*** on top of one or hardware threads, and a library package or language run-time system multiplexes one or more ***user-level threads*** on top of one or more kernel-level threads. Kernel-level threads in the same address space are said to constitute a ***process***. (But theoreticians say "process" where systems people say "thread.")



Two main classes of programming notation

1) synchronized access to shared memory
2) message passing between processes that don't share memory

Both approaches can be embedded in a programming language.
Both can be implemented on hardware designed for the other, though shared memory on message-passing hardware tends to be slow.

We'll focus here on shared memory. The book covers message passing on the companion site.

The multicore revolution

Moore's Law doubled the speed of uniprocessors every year and a half for 30 years. That ended around 2004.

Hit the "heat wall": 150W out of 2cm^2 of silicon is as much as you can cool with air. Faster uniprocessors must be liquid cooled or they will melt.

Absent unknown new VLSI technology, the short-term solution has been to move back down the heat curve, build processors with, say, 1/2 the MIPS and 1/50th the heat dissipation, and put a lot of them on one chip. (Heat is roughly proportional to clock rate. Heat curve is nonlinear, though, because of superpipelining, superscalarity, out-of-order execution, speculation, etc.)

To use such a chip, however, programs had to be multithreaded. Concurrency went from exotic high-end systems, to mid-range servers, to deskside machines and game consoles, to laptops, to tablets and phones.

We are currently in the middle of another paradigm shift, as computationally intensive ***kernels*** (not to be confused with the OS kernel) move onto GPUs and other accelerators. The typical cell phone today has half a dozen accelerators.

Languages v. language extensions v. libraries:

Thread creation syntax
- static set
- co-begin: Algol 68, Occam, SR
- ★ ***parallel loops***
- - iterations are independent: SR, Occam, others
- - or iterations are to run (as if) in lock step: Fortran 95 `forall`
- launch-on-elaboration: Ada, SR
- ★ ***fork/join***: Ada, Modula-3, Java, C#, OpenMP
- implicit receipt: DP, Lynx, RPC systems
- early reply: SR, Lynx
- Cf. separated `new()` and `start()` in Java

Most widely used:

	Shared memory	Message passing	Distributed computing
Language	Java, C# C/C++11	Go	Erlang
Extension	OpenMP		Remote procedure call
Library	`pthreads`, Windows threads	MPI	Internet libraries

Race Conditions

A race condition (or just "a ***race***") occurs when program behavior depends on the order in which events occur in different threads.
Race conditions are not all bad; sometimes any of the possible program outcomes are ok (e.g. workers taking things off a task queue). Often, however, we want to avoid race conditions. Suppose processors A and B share memory, and both try to increment variable X at more or less the same time. Very few processors support arithmetic operations on memory (even if the ISA supports provides single instructions for this, they aren't guaranteed to be atomic), so each processor executes

```
LOAD X
INC
STORE X
```

Suppose X is initialized to 0. If both processors execute these instruction sequences concurrently, we could see an increase of either one or two.

Data races v. ***synchronization races***
essentially unannotated v. annotated:
synchronization races are the expected ones, which the programmer tells the implementation to implement correctly

Races of one sort or another are what makes concurrent programming hard
growing consensus that ***data races are bugs***
initialization example

```
// ready == false

p = new foo(args)
ready = true              while (!ready) {}
                          // use *p
```

butterfly "causality" example

```
// x == y == 0
y = 1                     x = 1
a = x                     b = y

a == b == 0 ?
```

must be considered in the implementation of ***nonblocking algorithms*** and synchronization primitives

Modern languages are converging on semantics (***memory models***) that say circularity never occurs in "properly synchronized" (data race free) programs.

Synchronization

Synchronization is the act of ensuring that events in different threads happen in a desired order. Synchronization can be used to eliminate races. In our example we need to make the increment operations ***atomic***. One way to do that (not the only way) is to make them take turns. This is called ***mutual exclusion***: only one thread at a time can execute its ***critical section***. Informally, atomicity requires the ***appearance*** that threads take turns; mutual exclusion really makes them take turns. Most synchronization can be regarded as either atomicity or ***condition synchronization***, which means making sure that a given thread does not proceed until some condition holds (e.g. that a variable contains a given value).

[Other ways to get atomicity:
(1) nonblocking algorithms
(2) transactional memory, which may be implemented in hardware or in a library or language, with either nonblocking algorithms or locks.]

Example: bounded buffer.

```
index: 1..SIZE
buf: array [index] of data
nextfree, nextfull : index

procedure insert(d : data)
   // put something into the buffer, waiting if it's full

procedure remove : data
   // take something out of the buffer, waiting if it's empty
```

A solution requires
(1) the buffer behaves AS IF only one thread manipulates it at a time.
(2) threads wait for non-full or non-empty conditions as appropriate.

(1) is atomicity; (2) is condition synchronization.

You might be tempted to think of mutual exclusion as a form of condition synchronization (the condition being that nobody else is in the critical section), but it isn't. The distinction is basically existential v. universal quantification -- my state v. everybody's state. Mutual exclusion requires multi-thread agreement.

We do ***not*** in general want to over-synchronize. That eliminates parallelism, which we generally want to encourage for performance. Basically, we want to eliminate "bad" race conditions -- the ones that cause the program to give incorrect results.

Synchronization can be based either on ***spinning*** (***busy-waiting***) or ***re-scheduling*** (yielding to a different thread). The latter is built on the former. To get started, you have to have something nontrivial that is atomic in hardware -- something that happens all at once, as an indivisible action.

In most machines, reads and writes of individual memory locations are atomic (note that this is not trivial; memory and/or busses must be designed to arbitrate and serialize concurrent accesses). In early machines, reads and writes of individual memory locations were ***all*** that was atomic.To simplify the implementation of mutual exclusion, hardware designers began in the late 60's to build so-called ***read-modify-write***, or ***fetch-and-φ***, instructions into their machines.

Spin-based condition synchronization with atomic reads and writes is easy. You just cast each condition in the form of "location X contains value Y" and you keep reading X in a loop until you see what you want. Mutual exclusion is harder. Much early research was devoted to figuring out how to build it from simple atomic reads and writes. Dekker is generally credited with finding the first correct solution for two threads in the early 1960s. Dijkstra published a version that works for N threads in 1965. Peterson published a much simpler two-thread solution in 1981, while he was on the faculty here at Rochester. It can be extended to N threads with a log-depth tree.

A busy-wait mutual exclusion mechanism is known as a ***spin lock***. The problem with spin locks is that they waste processor cycles. Synchronization mechanisms are needed that interact with a thread/process scheduler to put a thread to sleep and run something else instead of spinning. Note, however, that spin locks are still valuable for certain things, and are widely used. In particular, it is better to spin than to sleep when the expected spin time is less than the rescheduling overhead.

Semaphores were the first proposed ***scheduler-based*** synchronization mechanism, and remain widely used. ***Conditional critical regions*** and ***monitors*** came later. Monitors have the highest-level semantics, but a few sticky semantic problems. They are also widely used. Synchronization in Java 2 is sort of a hybrid of monitors and CCRs. Java 5 has true monitors. Shared-memory synch in Ada 95 is yet another hybrid.

Spin Locks

Synchronization with only reads and writes is very subtle. I'm not going to go into the details. Dijkstra and Peterson's N-thread locks require O(N) time to acquire, which is bad. All of the locks based on only reads and writes, including Lamport's O(1) lock, require O(N) space, which is bad. Even Lamport's fast lock is only O(1) in the ***absence*** of contention.

Can do better with atomic read-modify-write (fetch-and-phi) instructions.

```
test_and_set
fetch_and_or
fetch_and_and
fetch_and_add
fetch_and_clear_then_add
fetch_and_store (swap)
```

universal:

```
compare_and_swap
load-linked + store-conditional
```

These typically return the old value, prior to changes, from which you can of course deduce the new value.

The simple ***test_and_set lock***:

```
type lock = Boolean := false
procedure acquire(L : ^lock)
   repeat until test_and_set(L) = false
procedure release(L : ^lock)
   L^ := false
```

Problems:
not fair (possible starvation)
LOTS of contention for memory and interconnect bandwidth

Latter problem can be ***partially*** cured, on a cache-coherent machine, by spinning with reads instead of TASes:

```
procedure acquire(L : ^lock)
   // "test-and-test-and-set" lock
   while test_and_set(L) = true
      repeat until L = false
```

This is known as a ***test-and-test_and_set lock***.

There are better solutions to these problems (including my own), but there isn't time to cover them here: take 258!

Busy-wait solution to the bounded buffer problem:

```
index: 0..SIZE-1
buf: array [index] of data
nextfree, nextfull, fullslots : index := 0, 0, 0
mutex : spinlock

procedure insert(d : data)
   loop
      acquire(mutex)
      if fullslots < SIZE
         buf[nextempty] := d
         nextempty++; nextempty %= SIZE
         fullslots++
         release(mutex)
         return
      else
         release(mutex)

procedure remove : data
   loop
      acquire(mutex)
      if fullslots > 0
         data d := buf[nextfull];
         nextfull++; nextfull %= SIZE;
         fullslots--;
         release(mutex)
         return d
      else
         release(mutex)
```

BTW, Fetch-and-phi operations are useful not only for locking, but for ***nonblocking*** data structures as well -- clever algorithms that avoid race conditions without ever locking anything. If a thread is preempted (at any time), other threads can continue to make progress. There exist good nonblocking algorithms for lists, queues, hash tables, search trees, mark-and-sweep garbage collection, and other things. Historically every new nonblocking algorithm has been a publishable result. Transactional memory changes that: some (***not*** all) TM systems are implemented in a nonblocking way under the hood: these provide a universal construction for nonblocking data structures. Operations on traditional nonblocking data structures can then be thought of as optimized hand-written transactions, though it isn't trivial to make these interoperate with general transactions.