Concurrency (take 258 to learn more)

A **process** or **thread** is a potentially-active execution context. The classic von Neumann (stored program) model of computing has a single thread; parallel programs have more than one. A process/thread 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, or run-until-block; most work focuses on the first two cases, which are equally difficult to deal with.

In common (but by no means universal) 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 the kernel-level threads. Kernel threads in the same address space 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 2cm² 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 was 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 desktop 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.

Programmers can be provided w/ concurrency via languages, language extensions, or 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:

<table>
<thead>
<tr>
<th>Shared memory</th>
<th>Message passing</th>
<th>Distributed computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Go</td>
<td>Erlang</td>
</tr>
<tr>
<td>Java, C#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extension</td>
<td>OpenMP</td>
<td>Remote procedure call</td>
</tr>
<tr>
<td>Library</td>
<td>pthreads,</td>
<td>MPI</td>
</tr>
<tr>
<td></td>
<td>Windows threads</td>
<td>Internet libraries</td>
</tr>
</tbody>
</table>

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.
widespread consensus that data races are bugs
Races are particularly pernicious if unintentional or unannotated (so unknown to the compiler), because hardware rarely guarantees visible consistent order of accesses across cores.

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 ?

Races must be considered—and annotated—in the implementation of nonblocking algorithms (covered in 258) 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 art of ensuring that events in different threads happen in a desired order. Synchronization can be used to eliminate races. In our increment example we need to make each thread’s operation atomic. One way to do that (not
the only way) is to make the threads 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
nextempty, 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.

On 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 synchronization 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`:

```pascal
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:

```pascal
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:

```plaintext
index: 0..SIZE-1
buf: array [index] of data
nextempty, 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.

Schedulers

Give us the ability to “put a thread/process to sleep” and run something else on its kernel thread/processor.

- Start with coroutines
- make uniprocessor run-until-block threads
- add preemption
- add multiple processors

Coroutines

As in Simula and Modula-2. Covered in section 8.6 in the book.

Multiple execution contexts, only one of which is active.

```plaintext
transfer(other):
    save all callee-saves registers on stack  // including ra & fp
    *current := sp
    current := other
    sp := *current
    pop all callee-saves registers       // including ra, but not sp
    return                               // into different coroutine!
```

other and current are pointers to context blocks.
Each contains sp; may contain other stuff as well
(priority, I/O status, accounting info, etc.)

No need to change pc; always changes at the same place.
Create new coroutine in a state that looks like it's blocked in transfer.
(Or maybe let it execute and then "detach". That's basically early reply.)
Run-until block threads on a single process
Need to get rid of explicit argument to transfer.
ready_list data structure: threads that are runnable but not running.

reschedule:
    t : cb := dequeue(ready_list)
    transfer(t)

To do this safely, we need to save current somewhere. Two options.

(1) Suppose we're just relinquishing the processor for the sake of
    fairness (as in MacOS 9 or Windows 3.1):
    yield:
        enqueue(ready_list, current)
    reschedule

(2) Now suppose we're implementing synchronization:
    sleep_on(q):
        enqueue(q, current)
    reschedule

Some other thread/process will move us to the ready list when we can
continue.

Preemption

Use timer interrupts (in OS) or signals (in library package) to trigger
involuntary yields.

Requires that we protect the scheduler data structures:

yield:
    disable_signals()
    enqueue(ready_list, current)
    reschedule
    re-enable_signals()

Note that reschedule takes us to a different thread, possibly in code
other than yield. Invariant: every call to reschedule must be made with
signals disabled, and must re-enable them upon its return.
disable_signals()
if not <desired condition>
    sleep_on <condition queue>
re-enable_signals()

Multiprocessors

Disabling signals doesn't suffice:

yield:
    disable_signals()
    acquire(scheduler_lock)    // spin lock
    enqueue(ready_list, current)
    reschedule
    release(scheduler_lock)
re-enable_signals()

disable_signals()
acquire(scheduler_lock)    // spin lock
if not <desired condition>
    sleep_on <condition queue>
release(scheduler_lock)
re-enable_signals()

------------------------------------
Scheduler-Based Synchronization

semaphores

So-called binary semaphores are scheduler-based mutual exclusion locks. The acquire operation is named P; the release operation is named V; these stand for words in Dutch. (Mnemonically, I think of P as standing for "pause", though it doesn't.) Binary semaphores are called "binary" because we can think of them as a counter that is always 0 or 1, and that indicates the number of threads that could perform acquire operations without blocking.

We can extend this to general semaphores, with non-binary counters. These are useful for certain algorithms, though they don't add any
additional power (you can implement them trivially with binary semaphores).

In either case, a semaphore is a special sort of counter. It has an initial value, and it keeps track of the excess (if any) of past V operations over past P operations. A P operation is delayed (the thread is de-scheduled) until \( \#P - \#V \leq C \), the initial value of the semaphore.

Here is one possible implementation:

```plaintext
type semaphore = record
    N : integer // initialized to something non-negative
    Q : queue of threads

procedure P(ref S : semaphore):
    disable_signals()
    acquire(scheduler_lock)
    if S.N > 0
        S.N := 1
    else
        sleep_on(S.Q)
    release(scheduler_lock)
    re-enable_signals()

procedure V(ref S : semaphore):
    disable_signals()
    acquire(scheduler_lock)
    if S.Q is nonempty
        enqueue(ready_list, dequeue(S.Q))
    else
        S.N := 1
    release(scheduler_lock)
    re-enable_signals()
```

What can we do with semaphores? Here is a bounded buffer:

```plaintext
shared buf : array [1..SIZE] of data
shared next_full, next_empty : integer := 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0
```
procedure insert(d : data) :
P(empty_slots)
P(mutex)
buf[next_empty] := d
next_empty := next_empty mod SIZE + 1
V(mutex)
V(full_slots)

function remove returns data :
P(full_slots)
P(mutex)
d : data := buf[next_full]
next_full := next_full mod SIZE + 1
V(mutex)
V(empty_slots)
return d

It is generally assumed that semaphores are fair, in the sense that threads complete P operations in the same order they start them.

Problems with semaphores:

(1) They're pretty low-level. When using them for mutual exclusion, for example (the most common usage), it's easy to forget a P or a V, especially when they don't occur in strictly matched pairs (because you do a V inside an if statement, for example, as in the use of the spin lock in the implementation of P).

(2) Their use is scattered all over the place. If you want to change how threads synchronize access to a data structure, you have to find all the places in the code where they touch that structure, which is difficult and error-prone.

These problems are addressed by monitors and other language mechanisms.

========================================
Language-level Synchronization

Scheduler-based locks in many languages (C/C++, Rust, Scala, Ruby, ...)

Monitors
Attempt to address the two weaknesses of semaphores previously discussed.
Suggested by Dijkstra, developed more thoroughly by Brinch Hansen, and formalized nicely by Hoare (a real cooperative effort!) in the early 1970s. Several parallel programming languages have incorporated monitors as their fundamental synchronization mechanism. None, to my knowledge, incorporates the precise semantics of Hoare's formalization.

A monitor is a shared object with operations, internal state, and a number of condition queues. Only one operation of a given monitor may be active at a given point in time. A thread that calls a busy monitor is delayed until the monitor is free. On behalf of its calling thread, any operation may suspend itself by waiting on a condition. An operation may also signal a condition, in which case one of the waiting threads is resumed, usually the one that waited first.

![Monitor diagram]

The precise semantics of mutual exclusion in monitors are the subject of considerable dispute. Hoare's original proposal remains the clearest and most carefully described. It specifies two bookkeeping queues for each monitor: an entry queue, and an urgent queue. When a thread executes a signal operation from within a monitor, it waits in the monitor's urgent queue and the first thread on the appropriate condition queue obtains control of the monitor. When a thread leaves a monitor it unblocks the first thread on the urgent queue or, if the urgent queue is empty, it unblocks the first thread on the entry queue instead.

The two main semantic controversies:

(1) Should a signal-er keep going, rather than moving to the urgent queue and letting the wait-er in? That reduces context switches but requires that we treat signals as "hints" instead of "absolutes".
The idiom
    if not condition wait
becomes
    while not condition wait

(2) The "nested monitor problem": A calls M1.e1, which calls M2.e2, which
waits. Should A release exclusion on M1? If it does, the world may change
before it returns, and it may not even be able to resume, if some other
thread enters and locks M1. If A doesn't release M1, however, B may not be
able to pass through M1 to reach M2 to signal A. The most elegant solution
I know of (but which I don't think anybody implements) was suggested by
Wettstein: to make signals hints, release all monitors, re-acquire them all
outermost first on wakeup, and require that invariants hold when making
nested calls. Java does not release outer monitors.

Building a correct monitor requires that one think about the monitor invariant.
(Everybody remember loop invariants?) The monitor invariant is a predicate that
captures the notion "the state of the monitor is consistent." It needs to be true
initially, and at monitor exit. It also needs to be true at every wait statement. In
Hoare's formulation, needs to be true at every signal operation as well, since some
other thread may immediately run.

Hoare's definition of monitors in terms of semaphores makes clear that semaphores
can do anything monitors can. The inverse is also true; it is trivial to build a
semaphores from monitors (if you don't see how, you should figure it out :-)

----------------------------------------
Concurrency in Java

Explicit threads (in Java from the beginning)

class Foo extends Thread {
    public Foo (...)
    {
        // constructor; does not start thread running
    }
    public void run()
    {
        // this is where the thread starts running
    }
}
Foo f = new Foo(...);  // returns when constructor is done
f.start();
   // puts new thread on the ready list,
   //   set up to execute its run routine
...
f.join();
   // optional; waits for f to finish (return from its run method)

start() is implemented by Thread. It calls run().
In classes derived from Thread you should always override run, and you
should make threads begin execution by calling start().
Never override start(). Never call run().

Executors (introduced in Java 5). Allow caching of threads to avoid
start-up / shut-down costs. Also abstract out the physical parallelism,
so you can have N tasks run by M ≤ N threads under the hood.

class Foo implements Runnable {
   ...
   // constructor and run() method same as before
}
...
ExecutorService pool = Executors.newCachedThreadPool();
...
pool.execute(new Foo( constructor_args ));
...
// indicate that pool will never get any additional tasks
pool.shutdown();
// wait for all workers to complete
try {
   pool.awaitTermination(60, TimeUnit.SECONDS);
} catch(InterruptedException e) {}{}

Runnables are object closures. They're useful for other things besides concurrency --
basically anything you want to package up for future execution. There's also a
Callable that produces a value that can be picked up later.

Can also use newFixedThreadPool(numThreads) or newSingleThreadExecutor()
These are all factory methods that create and manage Executor objects.
Java 2 synchronization (really should be generic; leaving that out for simplicity):

class BB {
    final private int SIZE = 10;
    private Object[] buf = new Object[SIZE];

    private int nextEmpty = 0;
    private int nextFull = 0;
    private int fullSlots = 0;

    synchronized public void insert(Object d) throws InterruptedException {
        while (fullSlots == SIZE) {
            wait();
        }
        buf[nextEmpty] = d;
        nextEmpty = (nextEmpty + 1) % SIZE;
        ++fullSlots;
        notifyAll(); // explain why!
    }

    synchronized public Object remove() throws InterruptedException {
        while (fullSlots == 0) {
            wait();
        }
        Object d = buf[nextFull];
        nextFull = (nextFull + 1) % SIZE;
        --fullSlots;
        notifyAll(); // explain why!
        return d;
    }
}

Several operations (e.g. wait, join, sleep) can throw InterruptedException. Any time you call one of these you have to be either (a) inside a try block with a handler (catch clause) for InterruptedException, or (b) inside a method whose header indicates "throws InterruptedException". Kind of a nuisance.

What is InterruptedException for? It's the way to get a waiting thread to wake up. If t is of class Thread, t.interrupt() will cause t to
experience an InterruptedException the next time it calls a blocking operation.
Note that there is no acceptable way to cause an exception in a non-blocked thread –
all such mechanisms have been deprecated.

Notes:

1) You can also use a synchronized statement (alternative to
   synchronized method):

   synchronized(my_obj) {
       // critical section
   }

2) There is only a single queue associated with each object. If you
   have threads that may wait for more than one reason, you need to
   worry about the "wrong kind" of thread waking up:

   if (!condition) {
       wait();
   }
   while (!condition) {
       notify();
       wait();
   }

   This can be expensive. In some cases you can get around it by waiting
   in multiple sub-objects, but this doesn't work in general because a
   waiting thread relinquishes only the inner lock, not any outer ones
   (leading to deadlock if the releasing thread has to get into the same
   outer objects).

3) A single thread can acquire the lock on a single object multiple times
   (doesn't exclude itself). If it waits, it releases the lock "the
   appropriate number of times." When it awakes, it will have
   re-acquired the lock "the appropriate # of times," and must leave
   that many synchronized methods or statements (or wait again) before
   anybody else can get in.

4) my_obj.notifyAll() will wake up all threads waiting on my_obj.
5) The lock on an object is associated with the data in the object only by convention. Java guarantees that if one thread releases a lock and a second then acquires it, the second sees all previous writes to all data by the first.

BB solution in Java 2 without using notifyAll is quite a bit harder:

```java
class BB {
    final private int SIZE = 10;
    private Object[] buf = new Object[SIZE];

    private Object producerMutex = new Object();
    // waited upon only by producers; protects the following:
    private int nextEmpty = 0;
    private int emptySlots = SIZE;

    private Object consumerMutex = new Object();
    // waited upon only by consumers; protects the following:
    private int nextFull = 0;
    private int fullSlots = 0;

    public void insert(Object d) throws InterruptedException {
        synchronized (producerMutex) {
            while (emptySlots == 0) {
                producerMutex.wait();
            }
            --emptySlots;
            buf[nextEmpty] = d;
            nextEmpty = (nextEmpty + 1) % SIZE;
        }
        synchronized (consumerMutex) {
            ++fullSlots;
            consumerMutex.notify();
        }
    }

    public Object remove() throws InterruptedException {
        Object d;
        synchronized (consumerMutex) {
            while (fullSlots < 0) {
                consumerMutex.wait();
            }
            --fullSlots;
            d = buf[nextFull];
        }
    }
}
```
nextFull = (nextFull + 1) % SIZE;
}
synchronized (producerMutex) {
    ++emptySlots;
    producerMutex.notify();
}
return d;
}
}

Solution using Java 5 locks is efficient and arguably more algorithmically elegant, but syntactically more cluttered due to library-based synchronization:

class BB {
    final private int SIZE = 10;
    private Object[] buf = new Object[SIZE];

    private int nextEmpty = 0;
    private int nextFull = 0;
    private int fullSlots = 0;

    Lock l = new ReentrantLock();
    final Condition emptySlot = l.newCondition();
    final Condition fullSlot = l.newCondition();

    public void insert(Object d) throws InterruptedException {
        l.lock();
        try {
            while (fullSlots == SIZE) {
                emptySlot.await();
            }
            buf[nextEmpty] = d;
            nextEmpty = (nextEmpty + 1) % SIZE;
            ++fullSlots;
            fullSlot.signal();
        } finally {
            l.unlock();
        }
    }

    public Object remove() throws InterruptedException {
        l.lock();
        try {
            while (fullSlots == 0) {
                fullSlot.await();
            }
            Object d = buf[nextFull];
            buf[nextFull] = null;
            ++nextFull;
            fullSlot.signal();
        } finally {
            l.unlock();
        }
        return d;
    }
}

Object d = buf[nextFull];
nextFull = (nextFull + 1) % SIZE;
--fullSlots;
emptySlot.signal();
return d;
} finally {
l.unlock();
}