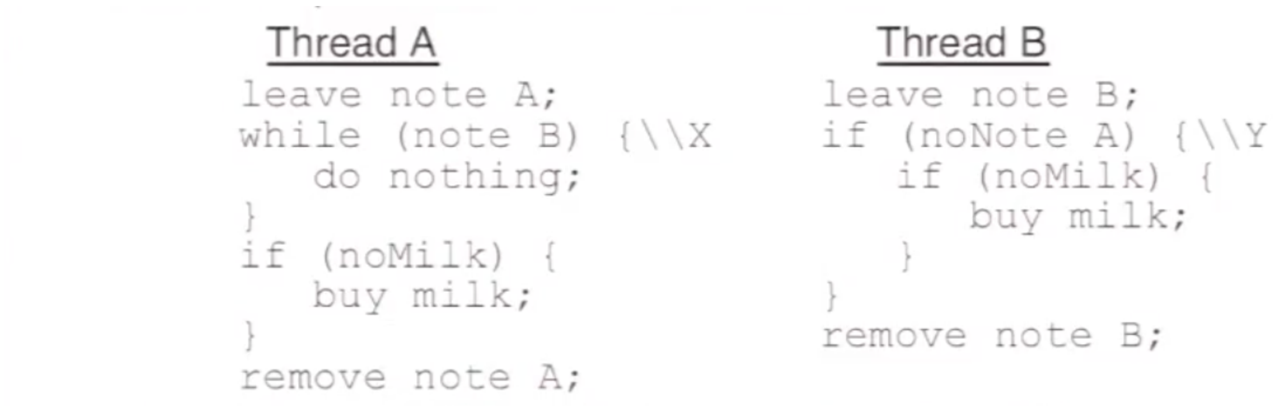
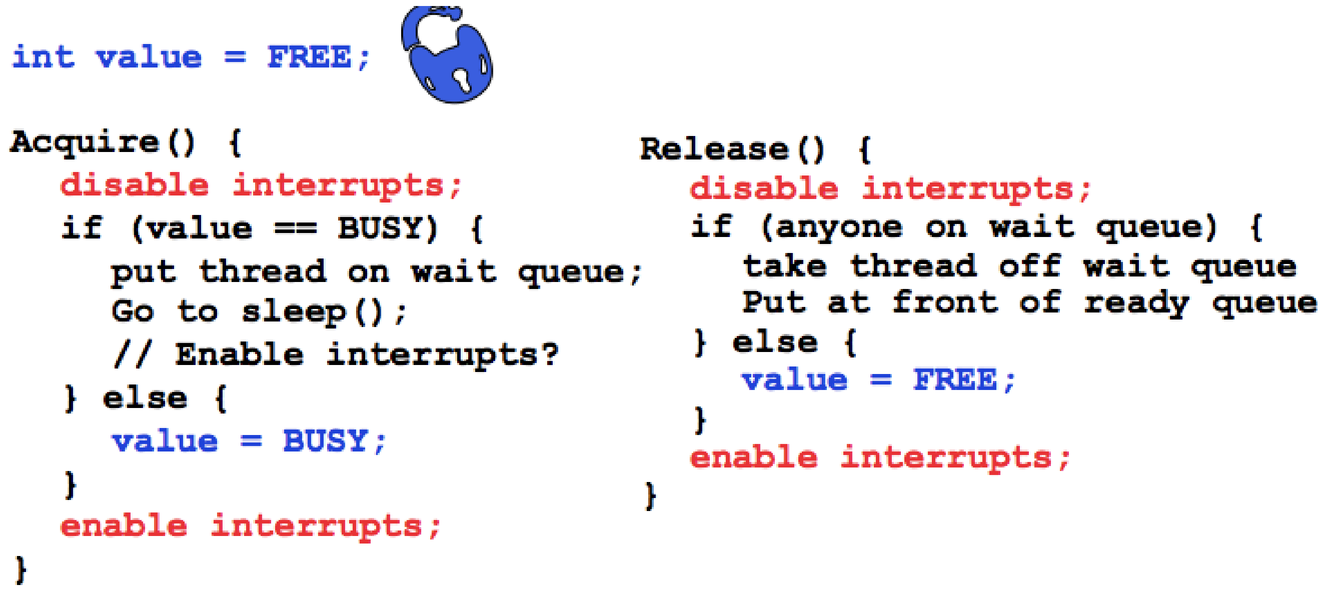
concurrency & synchronization

- the definition of correctness of parallel code is when for all possible legal orders of scheduling and interleaving, the program does the right thing; in all cases.
- race condition occurs when two threads are accessing the same variable/location in memory. lets say one thread is adding interest to a bank account and another is adding monthly pay check to bank account. if they both read, modify and write in separate instructions, there exists a scheduling order where this gets messed up.
- threaded programs must work for all inter-leavings of thread instruction sequences
- as software gets on more hardware, robotics etc, race conditions can affect safety of people. more often, it can lead to really weird bugs
- race conditions are not only between threads and memory slots. it can also be between two web requests handled from different servers when they are doing a database transaction, if they do read, modify and write on same DB row in a non-atomic way, that's a race condition too → so race condition is really a generic concept
- Atomic Operation: an operation that always runs to completion or not at all → indivisible. atomicity is required for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array
  - when you come across it, think about atomicity of instructions that operate across big chunks of memory like GPU instructions.
- Fine grain sharing:
  - increase concurrency
  - better performance
  - more complex
  - like row level transaction locks in DB
- Coarse grain sharing
  - Simpler to implement
  - Lower performance
  - like table level transaction locks in DB
- with only atomic loads and stores, it is theoretically possible to build a threaded system, without any special "locks". it would look something like this



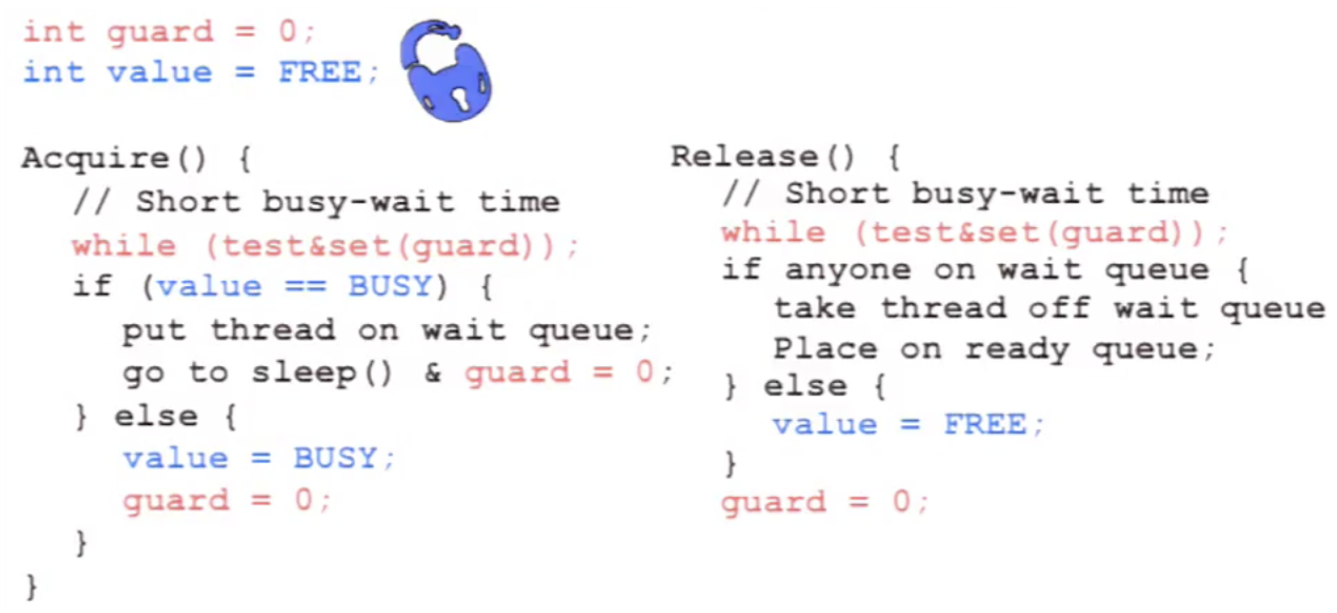
```
Thread A                          Thread B
leave note A;                     leave note B;
while (note B) {\\X               if (noNote A) {\\Y
   do nothing;                       if (noMilk) {
}                                       buy milk;
if (noMilk) {                        }
   buy milk;                      }
}                                 remove note B;
remove note A;
```

- the problem with the above, even though it is correct, is that
  - we have to write different coordination logic for each thread and this thinking needs to scale to multiple threads. thinking about correctness of this grows in complexity too.
  - involves some busy waiting → might be addressed by `yield()` in thread A while B has the lock.
- so to fix the above problems we'd like
  - higher level programming support to make these easier to write and reason about and scales well with number of threads
  - ideally only the thread that others are waiting on gets scheduled → no point scheduling a thread that that's waiting on a lock
- the portions of code within a lock are called "critical sections".
- mutual exclusion means ensuring that only one thread executes critical section
- now, it'd be very nice to have an instruction that "checks and sets" a note if none exists and returns whether a note was set. in addition, we'd like the ability to say if a lock isn't available, don't schedule me until it is.
- should we implement this lock as a hardware instruction?
  - we could, but we don't want to make hardware needlessly complicated if we can avoid it.
  - this makes the hardware more complex and expensive and less energy efficient. so we'll avoid it
- what about using interrupts to do this? will this prevent other threads from running that are waiting on locks? yes! how?
  - Interrupts are the way threads are externally preempted. (apart from internal ways to switch out the thread like calling `yield()` or invoking I/O)
  - we will use disabling interrupts as a way to keep running without getting preempted. if we do this in the entire critical section, that's irresponsible and we don't want applications to do that. so instead, we will only make the `acquire()` and `release()` calls within the disabled interrupts function.
  - look carefully at the code below that implements `acquire()` and `release()`.
  - notice how it is really efficient on the compute. once you are queued waiting on a lock you will only get scheduled once the lock is ready for your use.
  - this assumes disabling interrupts in a uniprocessor world where all threads can only be scheduled on the same cpu, but its a similar idea for multiple CPUs.
  - this implementation also scales for many threads, not just two, without changing any code. each thread that needs it simply need call `acquire()`



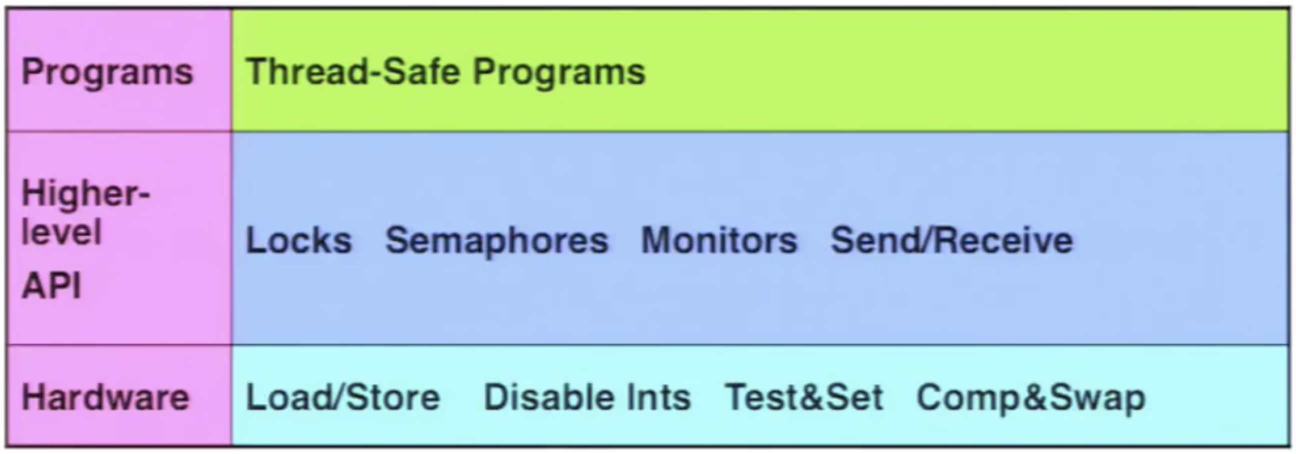
```
int value = FREE;

Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}

Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Put at front of ready queue
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

- note that in `acquire()` if we enable interrupts before putting thread on wait queue (right after the if check), its possible that `release()` of another thread runs and simply sets value to free and exits. next we will put the current thread on the wait queue but nobody will ever release.
- if we enable interrupts after putting thread on wait queue and before `sleep()` then we might get scheduled, go to sleep (with the lock) and never wake up. this is called *deadlock.* deadlock is when a thread has the lock and it'll never release it.
- the big downside of this implementation is that
  - it is all interrupt disabling driven. Disabling interrupts even when we just need to acquire and release locks is bad. we can't let the user disable interrupts like this and the OS must be able to take back control when it wants to.
  - the other big downside is that it is not clear how this works on a multiprocessor setting.
- so can we look at a hardware instruction that will do a "test and set". the problem is if we have one thread do a while loop on the test and set step, it is again not computationally efficient to sit there spinning.
- so instead we will have a lock on the lock, also called a guard (a second order lock). we need to acquire the guard first to add ourself on the wait queue for the lock. once we're on the queue, we will go to sleep until we are woken up again. yes acquiring the guard could involve spinning but this takes bounded time because we're only competing against threads who just want to get on the queue and then they'll release the guard. if we didn't have a second order guard, we would sit and spin while the thread with the lock does actual work (which could take a long time)
- so this now looks like



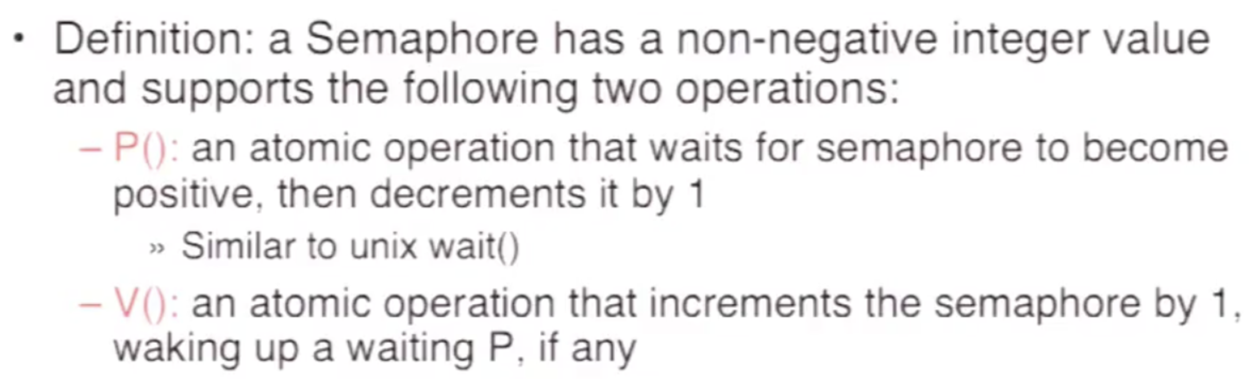
```
int guard = 0;
int value = FREE;

Acquire() {                              Release() {
   // Short busy-wait time                  // Short busy-wait time
   while (test&set(guard));                 while (test&set(guard));
   if (value == BUSY) {                     if anyone on wait queue {
      put thread on wait queue;                take thread off wait queue
      go to sleep() & guard = 0;               Place on ready queue;
   } else {                                 } else {
      value = BUSY;                            value = FREE;
      guard = 0;                            }
   }                                        guard = 0;
}
```

- Here is the low level hardware primitives vs higher level software abstractions



Programs	Thread-Safe Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- semaphores



- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Similar to unix wait()
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

- a semaphore can be easily implemented with a lock
- a monitor is a an object (an instance of a class) whose methods are all mutual exclusion. that means only method can execute at a time.
- he shows an implementation of a producer-consumer queue using a semaphore for `fullSlots`, one for `emptySlots` and a regular lock for updating the queue. the point of the semaphores is to put either the producer or consumer to sleep while waiting on the other one.

- thread’s `join`, `broadcast` (also called `notifyAll`) and `wait` & `signal` are all higher level abstractions built on top of locks
- To see why its better to use `notifyAll()` than to use `notify()`, unless you have proved otherwise, see the most upvoted (not the accepted) answer here.
- A condition variable implements `cond.wait()` and `cond.signal()` and `cond.broadcast()`. Optionally, `wait()` can take in a lock to release if the thing that will signal needs it to proceed.
- As a side note, working on large teams has lot of lessons from concurrency and parallelization. you want to design your teams so as to have clear interfaces and minimal communication between them. give them clear ownership so that they are each clear on what they own and can make independent progress without a ton of communication (avoid synchronization)
- also good parallelization allows for least wasted developer time spinning (empty while loops)
- having said that, communication is HUGE. slack, regular sync ups, design docs are all geared towards one goal → solid communication.
- the thing to keep in mind with exceptions is that you need to think of how to release a lock if a thread that has a lock has an exception while holding the lock.
- deadlock detection algorithm
  - make a graph of resources and threads. an edge from thread to resource indicates thread owns the resource. an edge from resource to thread indicates thread is waiting on the resource. as you simulate thru different states of the program (using sample input?), for all possible scheduling orders, is there a cycle that starts at a thread, goes thru another resource, another thread and another resource and back to the original thread?
- a couple of strategies to prevent deadlocks:
  - have a lot of resources (like file handles) so that two processes one of which has file A and another has file B don’t get stuck waiting on the resource that the other has.
  - no sharing resources, might not always be practical
  - make every thread explicitly declare what they will need ahead of time