Condition Variables and Monitors

CS 4410

Operating Systems

Spring 2017

Cornell University

Lorenzo Alvisi

Anne Bracy

See: Ch 5&6 in OSPP textbook

The slides are the product of many rounds of teaching CS 4410 by Professors Sirer, Bracy, Agarwal, George, and Van Renesse.

CONCURRENT APPLICATIONS

. . .

SYNCHRONIZATION OBJECTS

Locks   Semaphores   Condition Variables   Monitors

ATOMIC INSTRUCTIONS

Interrupt Disable   Atomic R/W Instructions

HARDWARE

Multiple Processors   Hardware Interrupts

Recall: Too Much Milk Solution

Jill

```
PinkNote = 1;
if (BlueNote == 0) {
 if (milk == 0) {
  milk++;
 }
}
PinkNote = 0;
```

Jack

```
BlueNote = 1;
while (PinkNote == 1) {
      ;
}
if (milk == 0) {
   milk++;
 }
}
BlueNote = 0;
```

Pros:

- Safe!
- Live!
- Achieved without *any* special support

Recall: Too Much Milk Solution

Jill

```
PinkNote = 1;
if (BlueNote == 0) {
 if (milk == 0) {
  milk++;
 }
}
PinkNote = 0;
```

Jack

```
BlueNote = 1;
while (PinkNote == 1) {
    ;
}
if (milk == 0) {
   milk++;
  }
}
BlueNote = 0;
```

Cons:

- Complicated: complicated correctness proof
- Inefficient: BUSY-WAITING!!!
- Asymmetric: hard to scale to many threads
- Incorrect(?) : instruction reordering can produce surprising results

CONCURRENT APPLICATIONS

. . .

SYNCHRONIZATION OBJECTS

Locks Semaphores Condition Variables Monitors

ATOMIC INSTRUCTIONS

Interrupt Disable **Atomic R/W Instructions**

HARDWARE

Multiple Processors Hardware Interrupts

CONCURRENT APPLICATIONS

. . .

SYNCHRONIZATION OBJECTS

Locks **Semaphores** Condition Variables Monitors

ATOMIC INSTRUCTIONS

Interrupt Disable Atomic R/W Instructions

HARDWARE

Multiple Processors Hardware Interrupts

Recall: Poem Wall Solution

```
Shared:
int in, out,
poem_t *buf[N];
Semaphore mutex_prod(1), mutex_cons(1);
Semaphore enoughRoom(N), poemThere(0);
```

```
void write_poem(poem_t *p) {
   P(enoughRoom); //space?
   P(mutex_prod);
  buf[in] = p;
  in = (in+1)%N;
   V(mutex_prod);
   V(poemThere); //item!
}
```

```
poem_t *get_poem() {
   P(poemThere); //need item
   P(mutex_cons);
   poem_t *p = buf[out];
   out = (out+1)%N;
   V(mutex_cons);
   V(enoughRoom); // space!
   return p;
}
```

Pros:

- Live & Safe & Correct
- No Busy Waiting! (that we see)
- Scales nicely

Recall: Poem Wall Solution

```
Shared:
int in, out,
poem_t *buf[N];
Semaphore mutex_prod(1), mutex_cons(1);
Semaphore enoughRoom(N), poemThere(0);
```

```
void write_poem(poem_t *p) {
   P(enoughRoom); //space?
   P(mutex_prod);
  buf[in] = p;
  in = (in+1)%N;
   V(mutex_prod);
   V(poemThere); //item!
}
```

```
poem_t *get_poem() {
   P(poemThere); //need item
   P(mutex_cons);
   poem_t *p = buf[out];
   out = (out+1)%N;
   V(mutex_cons);
   V(enoughRoom); // space!
   return p;
}
```

Cons:

- Still seems complicated: is this correct?
- Not so readable
- Easy to introduce bugs

Classic Semaphore Mistakes

```
P(S)
CS
P(S)  ⟵typo
```

I

I stuck on 2nd P(). Subsequent processes freeze up on 1st P().

```
V(S)  ⟵typo
CS
V(S)
```

J

Undermines mutex:

- J doesn't get permission via P()
- "extra" V()s allow other processes into the CS inappropriately

```
P(S)
CS  ⟵omission
```

K

Next call to P() will freeze up. Confusing because the *other* process could be correct but hangs when you use a debugger to look at its state!

```
P(S)
if(x) return;
CS
V(S)
```

L

Conditional code can change code flow in the CS. Caused by code updates (bug fixes, etc.) by someone other than original author of code.

Semaphores Considered Harmful

"During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores."

— Dijkstra "The structure of the 'THE'-Multiprogramming System" Communications of the ACM v. 11 n. 5 May 1968.

Semaphores NOT to the rescue!

Semaphores are "low-level" primitives. Small errors:

- Easily bring system to grinding halt
- Very difficult to debug

Two usage models:

- **Mutual exclusion:** "real" abstraction is a critical section
- **Communication:** threads use semaphores to communicate (*e.g.*, bounded buffer example)

Simplification: Provide concurrency support in compiler

➙Enter Condition Variables & Monitors

CONCURRENT APPLICATIONS

. . .

SYNCHRONIZATION OBJECTS

Locks Semaphores **Condition Variables** Monitors

ATOMIC INSTRUCTIONS

Interrupt Disable Atomic R/W Instructions

HARDWARE

Multiple Processors Hardware Interrupts

Condition Variables

A mechanism to wait for events

3 operations on Condition Variable `Condition x;`

- `x.wait()`: sleep until woken up (could wake up on your own)
- `x.signal()`: wake at least one process waiting on condition (if there is one). No history associated with signal.
- `x.broadcast()`: wake all processes waiting on condition (useful for resource manager)

!! NOT the same thing as UNIX wait and UNIX signal !!

Semaphores vs. Condition Variables

Semaphores

```
Shared:
int in, out, buf[N];
Semaphore mutex_prod(1);
Semaphore enoughRoom(N),
          poemThere(0);
```

```
void write_poem(poem_t *p) {
   P(enoughRoom); // space?
   P(mutex_prod);
  buf[in] = p;
  in = (in+1)%N;
   V(mutex_prod);
   V(poemThere); // poem!
}
```

(ignore the mutexes for now)

Condition Variables

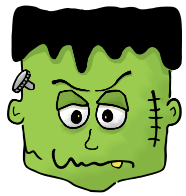
```
Shared:
int in, out, nPoems = 0;
poem_t *buf[N];
Semaphore mutex_prod(1);
Condition enoughRoom,
          poemThere;
```

```
void write_poem(poem_t *p) {
   while (nPoems == N)
     enoughRoom.wait();
   P(mutex_prod);
   buf[in] = p;
   in = (in+1)%N;
   nPoems++;
   V(mutex_prod);
   poemThere.signal();
}
```

CV Observations?

- State (nPoems) is *external*
- Code is self documenting

This example is not complete!



Condition Variables Live in a Monitor

Abstract Data Type for handling shared resources, comprising:

1. Shared Private Data
   - the resource
   - can only be accessed from in the monitor
2. Procedures operating on data
   - gateway to the resource
   - can only act on data local to the monitor
3. Synchronization primitives
   - among threads that access the procedures

[Hoare 1974]

One Thread at a Time in the Monitor!

Monitor Semantics guarantee mutual exclusion

Only one thread can execute monitor procedure at any time (aka "in the monitor")

in the abstract

```
Monitor monitor_name
{
    // shared variable declarations

    procedure P1() {
    }

    procedure P2() {
    }
    .
    .
    procedure PN() {
    }

    initialization_code() {
    }
}
```

for example:

```
Monitor poem_wall
{
    int in=0, out=0, nPoems=0;
    poem_t *buf[N];
    Condition enoughRoom,
              poemThere;

    get_poem() {
    }

    write_poem() {
    }

}
```

can only access shared data, CVs via a monitor procedure

only one operation can execute at a time

Types of Wait Queues

Monitors have two kinds of "wait" queues

- **Entry to the monitor:** has a queue of threads waiting to obtain mutual exclusion & enter

- **Condition variables:** each condition variable has a queue of threads waiting on the associated condition

Using Condition Variables

*You **must** hold the monitor lock to call these operations.*

To wait for some condition:

```
while not some_predicate():
    CV.wait()
```

- Atomically releases monitor lock & yields processor
- as `CV.wait()` returns, lock automatically reacquired

When the condition becomes satisfied:

`CV.broadcast()`: wakes up all threads

`CV.signal()`: wakes up at least one thread

CV semantics: Brinch Hansen vs. Hoare

- The condition variables we have defined obey Brinch Hansen (or Mesa) semantics
  - signaled thread is moved to ready list, but mot guaranteed to run right away
- Hoare proposes an alternative semantics
  - signaling thread is suspended and, atomically, ownership of the lock is passed to one of the waiting threads, whose execution is immediately resumed

What are the implications?

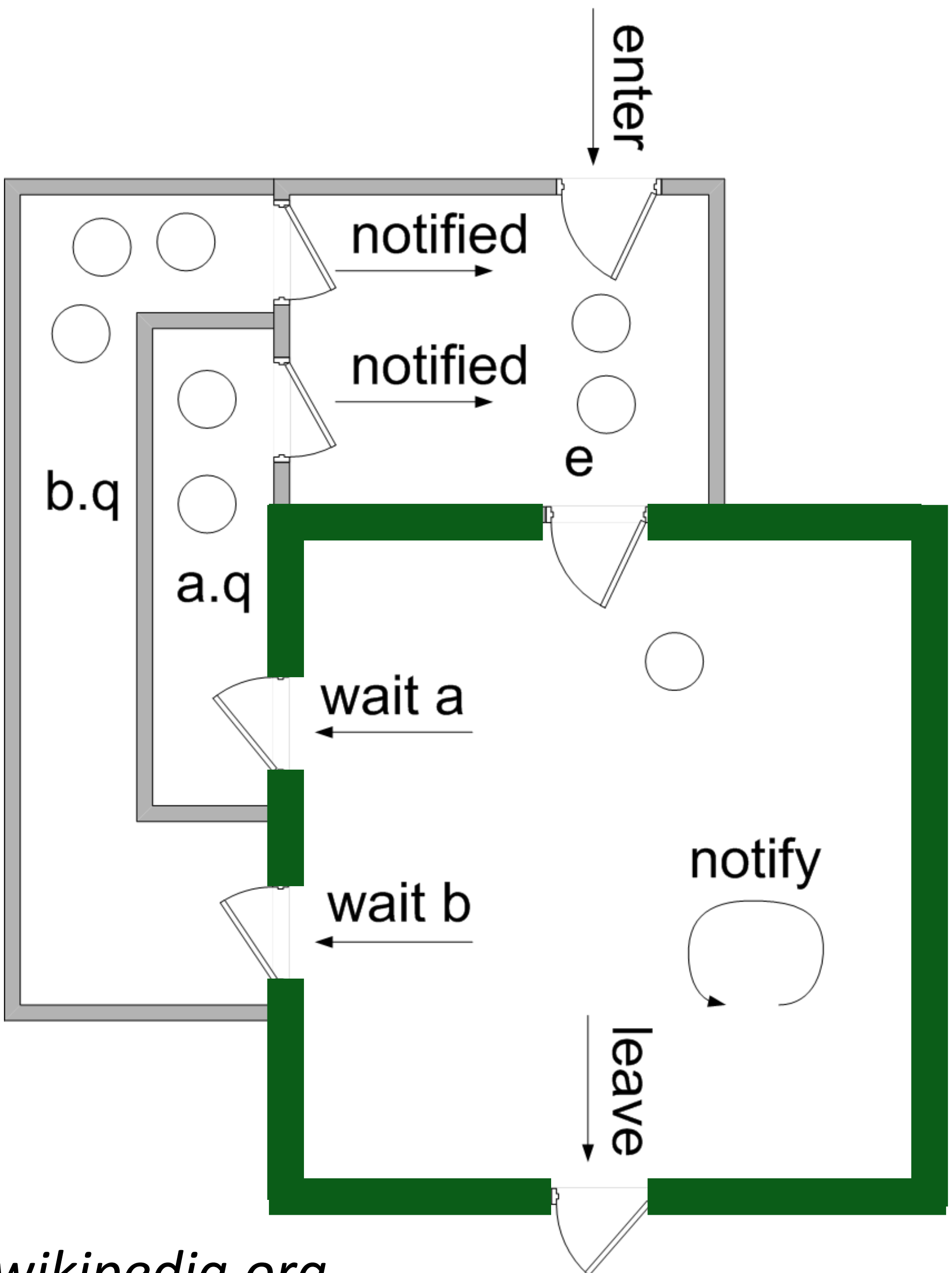
Brinch Hansen/Mesa

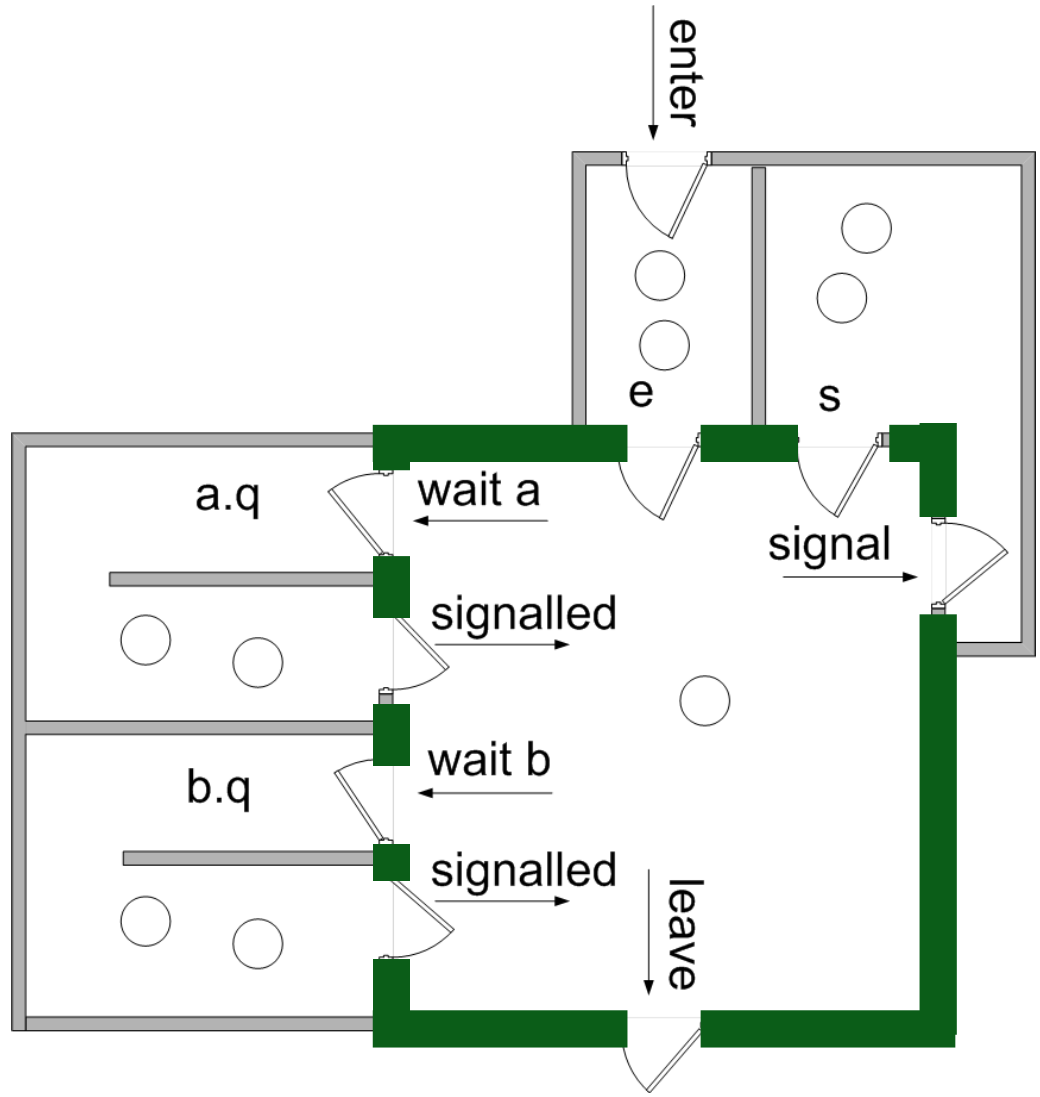
- signal() and broadcast() are hints
  - adding them affects performance, never safety
- Shared state must be checked in a loop (could have changed)
  - robust to spurious wakeups
- Simple implementation
  - no special code for thread scheduling or acquiring lock
- Used in most systems
- Sponsored by a Turing Award
  - Butler Lampson

Hoare

- Signaling is atomic with the resumption of waiting thread
  - shared state cannot change before waiting thread is resumed
- Shared state can be checked using an if statement
- Makes it easier to prove liveness
- Tricky to implement
- Used in most books
- Sponsored by a Turing Award
  - Tony Hoare

Which is Mesa/Hansen? Which is Hoare?





wikipedia.org

Language Support

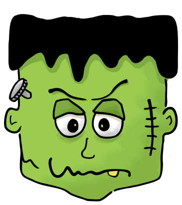
Can be embedded in programming language:

- Compiler adds synchronization code, enforced at runtime
- Mesa/Cedar from Xerox PARC
- Java: synchronized, wait, notify, notifyall
- C#: lock, wait (with timeouts) , pulse, pulseall
- Python: acquire, release, wait, notify, notifyAll

Complete Poem Wall

```
Shared:
int in, out, buf[N];
int nPoems = 0;
Semaphore mutex_prod(1);
Condition enoughRoom,
          poemThere;
```

```
void write_poem(int item) {
  while (nPoems == N)
    enoughRoom.wait();
  P(mutex_prod);
  buf[in] = item;
  in = (in+1)%N;
  nPoems++;
  V(mutex_prod);
  poemThere.signal();
}
```



This example is not complete!

mutexes achieved via monitor lock

```
Monitor poem_wall {
  lock mlock;
  poem_t *buf[N];
  int in=0, out=0, nPoems=0;
  condition nuffRoom,poemsThere;

  void write_poem(poem_t *p) {
     mlock.acquire();
     while(nPoems == N)
         nuffRoom.wait();
     buf[in] = p;
     in = (in+1)%N;
     nPoems++;
     signal(not_empty);
     mlock.release();
  }
}
```

100% Monitor

Complete Poem Wall: a closer look

What if no thread is waiting when signal() called?

Then signal is a nop.
Very different from calling V() on a semaphore – semaphores remember how many times V() was called!

```
Monitor poem_wall {
   lock mlock;
   poem_t *buf[N];
   int in=0, out=0, nPoems=0;
   condition nuffRoom,poemsThere;

   void write_poem(poem_t *p) {
      mlock.acquire();
      while(nPoems == N)
         nuffRoom.wait();
      buf[in] = p;
      in = (in+1)%N;
      nPoems++;
      signal(not_empty);
      mlock.release();
    }
}
```

Condition Variables vs. Semaphores

Access to monitor is controlled by a lock. To call wait or signal, thread must be in monitor (= have lock).

Wait vs. P:

- Semaphore P() blocks thread only if value < 1
- wait always blocks & gives up the monitor lock

Signal vs. V: causes waiting thread to wake up

- V() increments ➙ future threads don't wait on P()
- No waiting thread ➙ signal = nop
- Condition variables have no history!

Monitors easier and safer than semaphores

- Lock acquire/release are implicit, cannot be forgotten
- Condition for which threads are waiting explicitly in code

Classic Mistakes with Monitors

#1: Naked Waits

```
while not some_predicate():
    CV.wait()
```

What is wrong with this?

```
random_fn1()
CV.wait()
random_fn2()
```

How about this?

```
with self.lock:
    a=False
    while not a:
        self.cv.wait()
        a=True
```

Classic Mistakes with Monitors

#2: If vs. While

What is wrong with this?

```
if not some_predicate():
    CV.wait()
```

Classic Mistakes with Monitors

#3: Split Predicates What is wrong with this?

```
with lock:
  while not condA:
      condA_cv.wait()
  while not condB:
      condB_cv.wait()
```

Better:

```
with lock:
  while not condA or not condB:
    if not condA:
      condA_cv.wait()
    if not condB:
      condB_cv.wait()
```

Monitors in Python

Where does the actual reading take place?

```
class RWlock:
  def __init__(self):
    self.lock = Lock()
    self.canRead = Condition(self.lock)
    self.canWrite = Condition(self.lock)
    self.nReaders = 0
    self.nWriters = 0
    self.nWaitingReaders = 0
    self.nWaitingWriters = 0

  def begin_read(self):
    with self.lock:
      self.nWaitingReaders += 1
      while self.nWriters > 0 or self.nWaitingWriters > 0:
        self.canRead.wait()
      self.nWaitingReaders -= 1
      self.nActiveReaders += 1

  def end_read(self):
    with self.lock:
      self.nReaders -= 1
      if self.nReaders == 0 and self.nWaitingWriters > 0:
          self.canWrite.notify()
```

Remember that **wait**
- releases the lock when called
- re-acquires the lock when it returns

signal() → notify()
broadcast) → notifyAll()

Monitors in "4410 Python" : `__init__`

```
class RWlock:
  def __init__(self):
    self.lock = Lock()
    self.canRead = Condition(self.lock)
    self.canWrite = Condition(self.lock)
    self.nReaders = 0
    self.nWriters = 0
    self.nWaitingReaders = 0
    self.nWaitingWriters = 0
```

```
from rvr import MP, MPthread

class MonitorExample(MP):
 def __init__(self):
    MP.__init__(self,None)
    self.lock = Lock("monitor lock")
    self.canRead = self.Lock.Condition("can read")
    self.canWrite = self.Lock.Condition("can write")
    self.nReaders = self.Shared("num readers", 0)
    self.nWriters = self.Shared("num writers", 0)
    self.nWaitingReaders = self.Shared("n waiting readers", 0)
    self.nWaitingWriters = self.Shared("n waiting writers", 0)
```

Monitors in "4410 Python" : begin_read

```
def begin_read(self):
   with self.lock:
     self.nWaitingReaders += 1
     while self.nWriters > 0 or self.nWaitingWriters > 0:
        self.canRead.wait()
     self.nWaitingReaders -= 1
     self.nActiveReaders += 1
```

```
def begin_read(self):
   with self.lock:
     self.nWaitingReaders.inc()
     while self.nWriters.read() > 0 or self.nWaitingWriters.read() > 0:
        self.canRead.wait()
     self.nWaitingReaders.dec()
     self.nActiveReaders.write(self.nActiveReaders.read() + 1)
```

Why do we do this?

- helpful feedback from auto-grader
- helpful feedback from debugger

Look in the A2/doc directory for details and example code.

Barrier Synchronization

- Important synchronization primitive in high-performance parallel programs
- nThreads threads divvy up work, run rounds of computations separated by barriers.
- could fork & wait but
  - thread startup costs
  - waste of a warm cache

```
Create n threads & a barrier.

Each thread does round1()
barrier.checkin()

Each thread does round2()
barrier.checkin()
```

Checkin with 1 condition variable

```
self.allCheckedIn = Condition(self.lock)

def checkin():
  with self.lock:
     nArrived++
     if nArrived < nThreads:
       while nArrived < nThreads:
          allCheckedIn.wait()
    else:
       allCheckedIn.broadcast()
```

What's wrong with this?

Checkin with 2 condition variables

```
self.allCheckedIn = Condition(self.lock)
self.allLeaving = Condition(self.lock)

def checkin():
   nArrived++
   if nArrived < nThreads:               // not everyone has checked in
      while nArrived < nThreads:
         allCheckedIn.wait()              // wait for everyone to check in
   else:
      nLeaving = 0                // this thread is the last to arrive
      allCheckedIn.broadcast()    // tell everyone we're all here!

   nLeaving++
   if nLeaving < nThreads:               // not everyone has left yet
      while nLeaving < nThreads:
         allLeaving.wait()                // wait for everyone to leave
   else:
      nArrived = 0                // this thread is the last to leave
      allLeaving.broadcast()      // tell everyone we're outta here!
```

- Implementing barriers is not easy.
- Solution here uses a "double-turnstile"

The Six Commandments

1. **Thou shalt always do things the same way**
   - habit allows you to focus on core problem
   - easier to review, maintain and debug your code
2. **Thou shalt always synchronize with locks and condition variables**
   - either CV & locks or semaphores
   - CV and locks make code clearer
3. **Thou shalt always acquire the lock at the beginning of a method and release at the end**
   - make a chunk of code that requires a lock its own procedure

The Six Commandments

4. **Always hold a lock when operating on a condition variable**
    - condition variables are useless without shared state
    - shared state should only be accessed using a lock

5. **Always wait in a while() loop**
    - while works every time if does
    - makes signals hints
    - protects against spurious wakeups

6. **(Almost) never sleep()**
    - use sleep() only if an action should occur at a specific real time
    - never wait on sleep()

Conclusion: Race Conditions are a big ~~pain~~ deal!

Several ways to handle them

- Each has its own pros and cons

Programming language support simplifies writing multithreaded applications

- Python condition variables
- Java and C# support at most one condition variable per object, so are slightly more limited

Some program analysis tools automate checking

- make sure code is using synchronization correctly
- Hard part is to defining “correct”