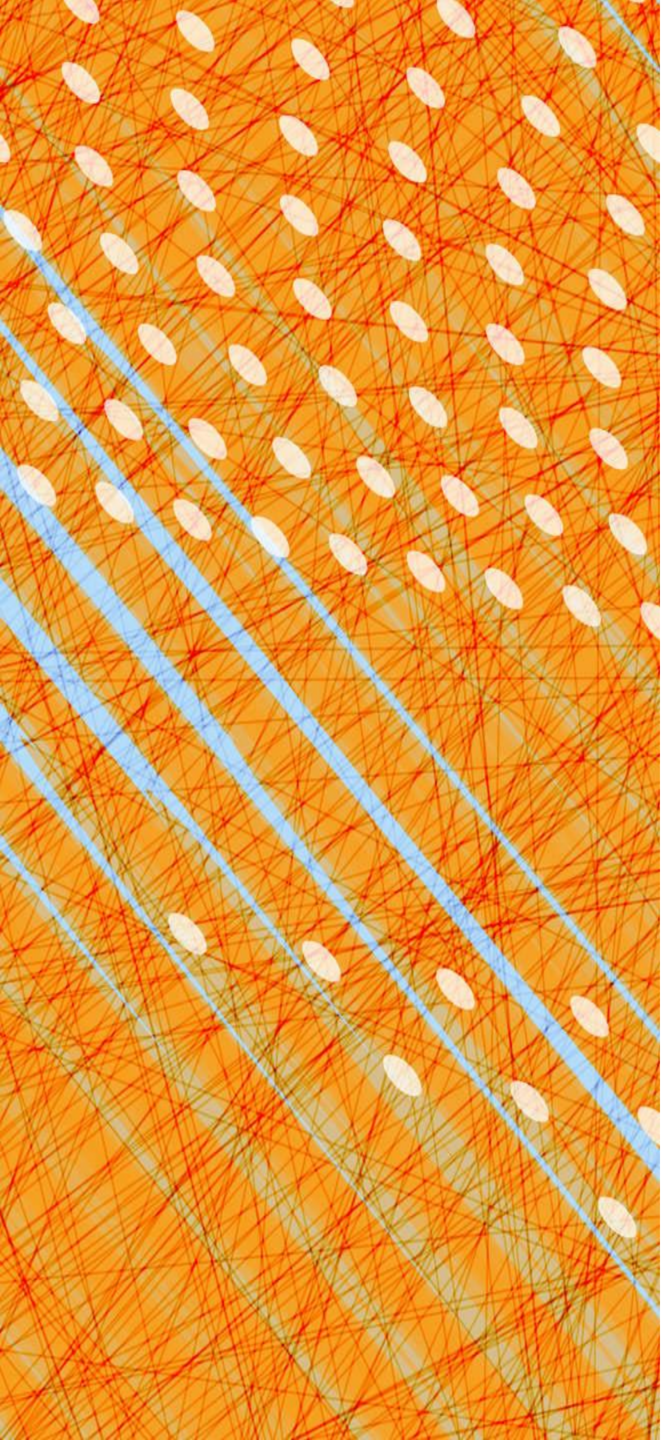
Synchronization

CS 4410
Operating Systems

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]



- Foundations
- Semaphores
- **Monitors & Condition Variables**

Producer-Consumer with locks

```
char buf[SIZE];
int n=0, tail=0, head=0;
lock l;
produce(char ch) {
   l.acquire()
      while(n == SIZE):
      l.release(); l.acquire()
   buf[head] = ch;
   head = (head+1)%SIZE;
   n++;
   l.release();
}
char consume() {
   l.acquire()
   while(n == 0):
      l.release(); l.acquire()
   ch = buf[tail];
   tail = (tail+1)%SIZE;
   n--;
   l.release;
   return ch;
}
```

Thou shalt not busy-wait!

CONCURRENT APPLICATIONS

. . .

SYNCHRONIZATION OBJECTS

Locks Semaphores **Condition Variables** **Monitors**

ATOMIC INSTRUCTIONS

Interrupt Disable Atomic R/W Instructions

HARDWARE

Multiple Processors Hardware Interrupts

Monitors & Condition Variables

- **Definition**
- Simple Monitor Example
- Implementation
- Classic Sync. Problems with Monitors
  - Bounded Buffer Producer-Consumer
  - Readers/Writers Problems
  - Barrier Synchronization
- Semantics & Semaphore Comparisons
- Classic Mistakes with Monitors

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 bounded_buffer
{
    int in=0, out=0, nElem=0;
    int buffer[N];

    consume() {
    }

    produce() {
    }

}
```

can only access shared data via a monitor procedure

only one operation can execute at a time

Producer-Consumer Revisited

Problems:

1. Unprotected shared state (multiple producers/consumers)

Solved via Monitor.

Only 1 thread allowed in at a time.

- *Only one thread can execute monitor procedure at any time*
- *If second thread invokes monitor procedure at that time, it will block and wait for entry to the monitor.*
- *If thread within a monitor blocks, another can enter*

2. Inventory:
   - Consumer could consume when nothing is there!
   - Producer could overwrite not-yet-consumed data!

What about these?

→ Enter Condition Variables

Condition Variables

A mechanism to wait for events

3 operations on **`Condition Variable 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

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

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

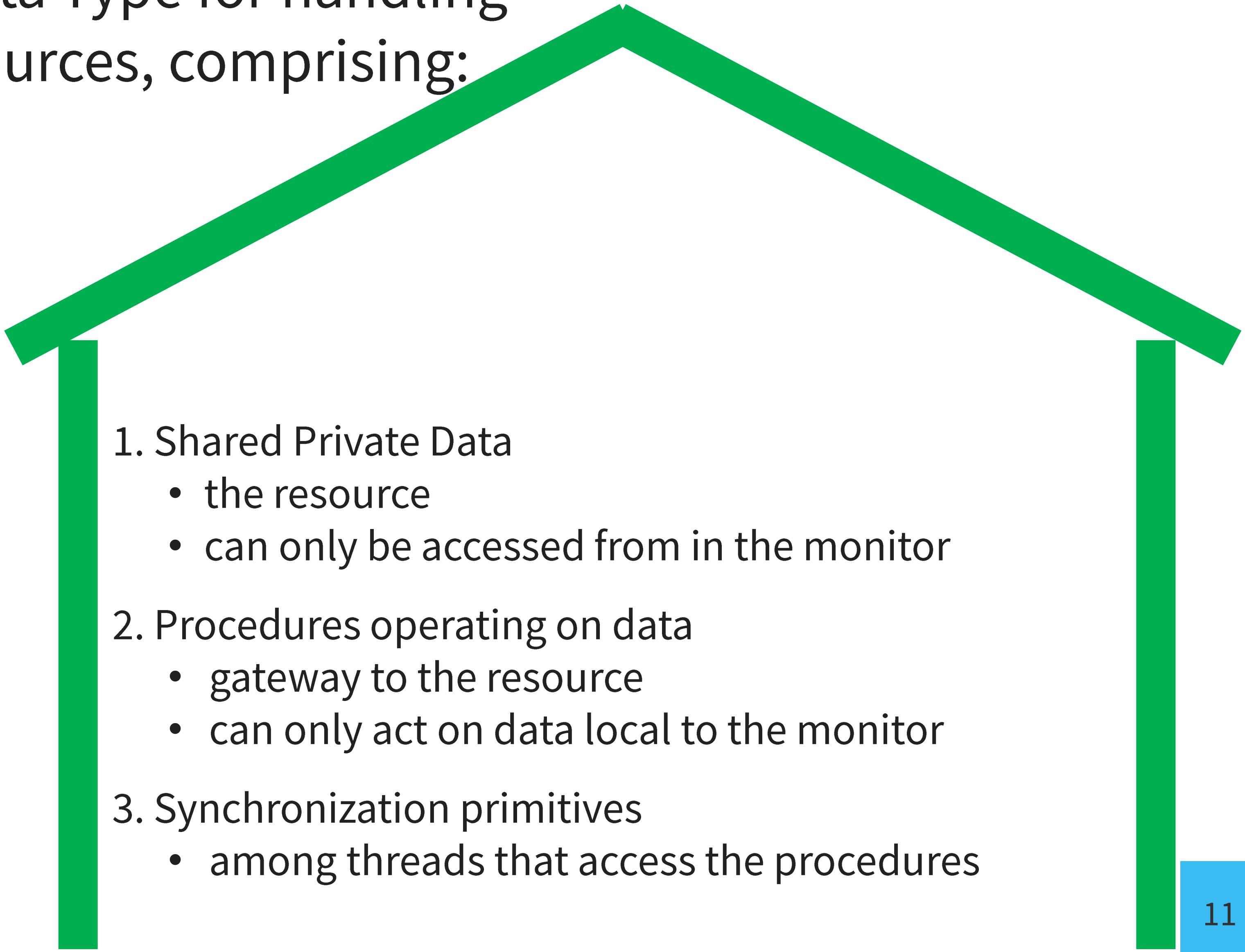
When the condition becomes satisfied:

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

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

Condition Variables Live in the 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]

Types of Wait Queues

Monitors have two kinds of “wait” queues

- **Entry to the monitor:** 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

Kid and Cook Threads

Running

```
kid_main() {

    play_w_legos()
    BK.kid_eat()
   bathe()
   make_robots()
   BK.kid_eat()
    facetime_Edward()
    facetime_grandma()
    BK.kid_eat()
}
```

```
Monitor BurgerKing {
  Lock mlock

  int numburgers = 0
  condition hungrykid

  kid_eat:
   with mlock:
     while (numburgers==0)
          hungrykid.wait()
     numburgers -= 1

  makeburger:
   with mlock:
     ++numburger
     hungrykid.signal()
}
```

```
cook_main() {

  wake()
  shower()
  drive_to_work()
  while(not_5pm)
    BK.makeburger()
  drive_to_home()
  watch_got()
  sleep()
}
```

Monitors & Condition Variables

- Definition
- Simple Monitor Example
- **Implementation**
- Classic Sync. Problems with Monitors
    - Bounded Buffer Producer-Consumer
    - Readers/Writers Problems
    - Barrier Synchronization
- Semantics & Semaphore Comparisons
- Classic Mistakes with Monitors

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

Monitors easier & safer than semaphores

- Compiler can check
- Lock acquire and release are implicit and cannot be forgotten

Monitors in Python

```
class BK:
  def __init__(self):
    self.lock = Lock()
    self.hungrykid = Condition(self.lock)
    self.nBurgers= 0

  def kid_eat(self):
     with self.lock:
        while self.nBurgers == 0:
           self.hungrykid.wait()
        self.nBurgers = self.nBurgers - 1

  def make_burger(self):
    with self.lock:
       self.nBurgers = self.nBurgers + 1
       self.hungrykid.notify()
```

wait
- releases lock when called
- re-acquires lock when it returns

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

Producer-Consumer

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

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

```
Monitor Producer_Consumer {
  char buf[SIZE];
  int n=0, tail=0, head=0;
  condition not_empty, not_full;
  produce(char ch) {
      while(n == SIZE):
          wait(not_full);
      buf[head] = ch;
      head = (head+1)%SIZE;
      n++;
     notify(not_empty);
   }
  char consume() {
      while(n == 0):
          wait(not_empty);
      ch = buf[tail];
      tail = (tail+1)%SIZE;
      n--;
      notify(not_full);
      return ch;
  }
}
```

Readers and Writers

```
Monitor ReadersNWriters {

 int waitingWriters=0, waitingReaders=0, nReaders=0, nWriters=0;
 Condition canRead, canWrite;

BeginWrite()
  with monitor.lock:
    ++waitingWriters
    while (nWriters >0 or nReaders >0)
      canWrite.wait();
    --waitingWriters
    nWriters = 1;

EndWrite()
  with monitor.lock:
    nWriters = 0
    if WaitingWriters > 0
      canWrite.signal();
    else if waitingReaders > 0
      canRead.broadcast();

void BeginRead()
  with monitor.lock:
    ++waitingReaders
    while (nWriters>0 or waitingWriters>0)
      canRead.wait();
    --waitingReaders
    ++nReaders


void EndRead()
  with monitor.lock:
    --nReaders;
    if (nReaders==0 and waitingWriters>0)
      canWrite.signal();
}
```

Understanding the Solution

A writer can enter if:

- no other active writer

&&

- no active readers

A reader can enter if:

- no active writer

&&

- no waiting writers

When a writer finishes:

check for waiting writers

Y → lets one enter

N → let all readers enter

Last reader finishes:

- it lets 1 writer in (if any)

Fair?

- If a writer is active or waiting, readers queue up
- If a reader (or another writer) is active, writers queue up

… gives preference to writers, which is often what you want

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 and nArrived > 0:
          allCheckedIn.wait()
     else:
       allCheckedIn.broadcast()
        nArrived = 0
```

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”

Monitors & Condition Variables

- Definition
- Simple Monitor Example
- Implementation
- Classic Sync. Problems with Monitors
  - Bounded Buffer Producer-Consumer
  - Readers/Writers Problems
  - Barrier Synchronization
- **Semantics & Semaphore Comparisons**
- Classic Mistakes with Monitors

CV semantics: Hansen vs. Hoare

The condition variables we have defined obey Brinch Hansen (or Mesa) semantics

- signaled thread is moved to ready list, but not 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

Kid and Cook Threads *Revisited*

Hoare vs. Mesa semantics

- What happens if there are lots of kids?

```
ki_main() {

   play_w_legos()
   BK.kid_eat()
  bathe()
  make_robots()
  BK.kid_eat()
   facetime_Edward()
   facetime_grandma()
   BK.kid_eat()
}
```

```
Monitor BurgerKing {
  Lock mlock

  int numburgers = 0
  condition hungrykid

  kid_eat:
   with mlock:
     while (numburgers==0)
        hungrykid.wait()
     numburgers -= 1

  makeburger:
   with mlock:
     ++numburger
     hungrykid.signal()
}
```

```
cook_main() {

  wake()
  shower()
  drive_to_work()
  while(not_5pm)
    BK.makeburger()
  drive_to_home()
  watch_got()
  sleep()
}
```

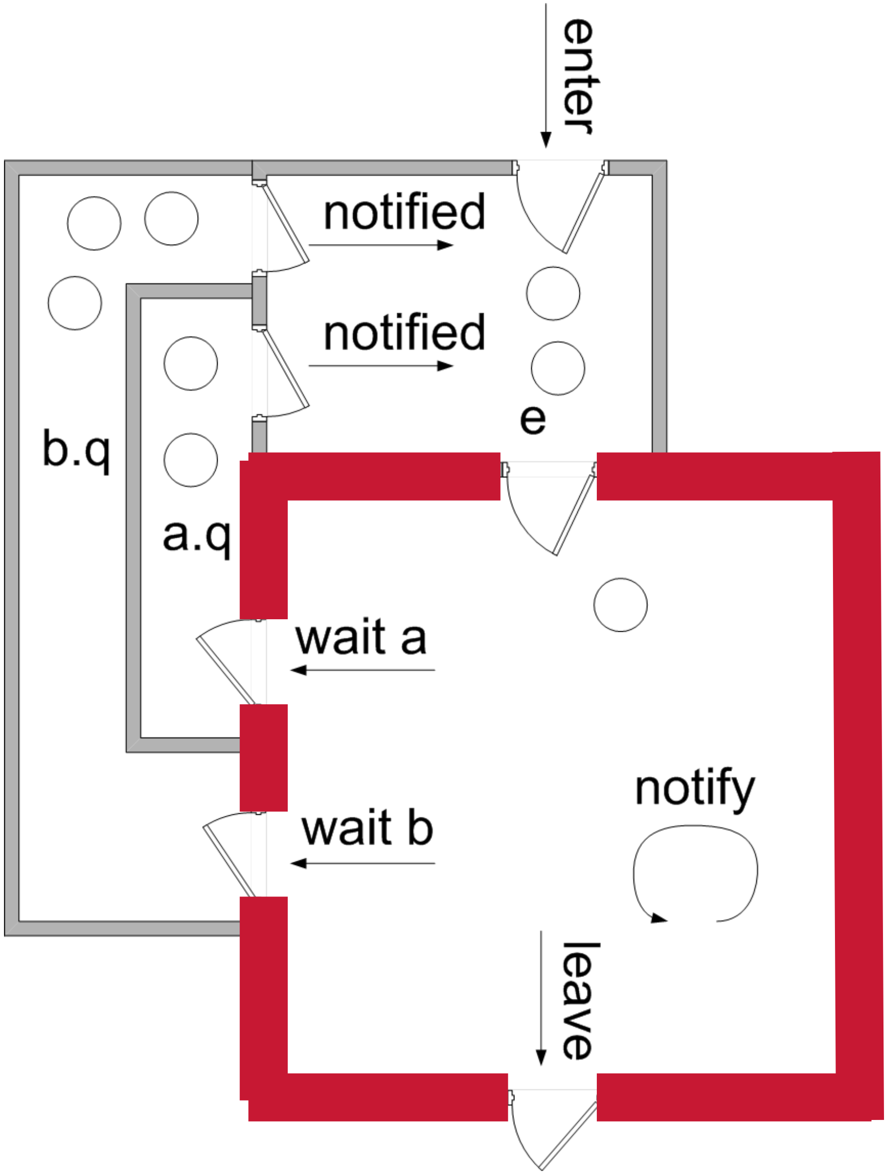
Hoare vs. Mesa/Hansen Semantics

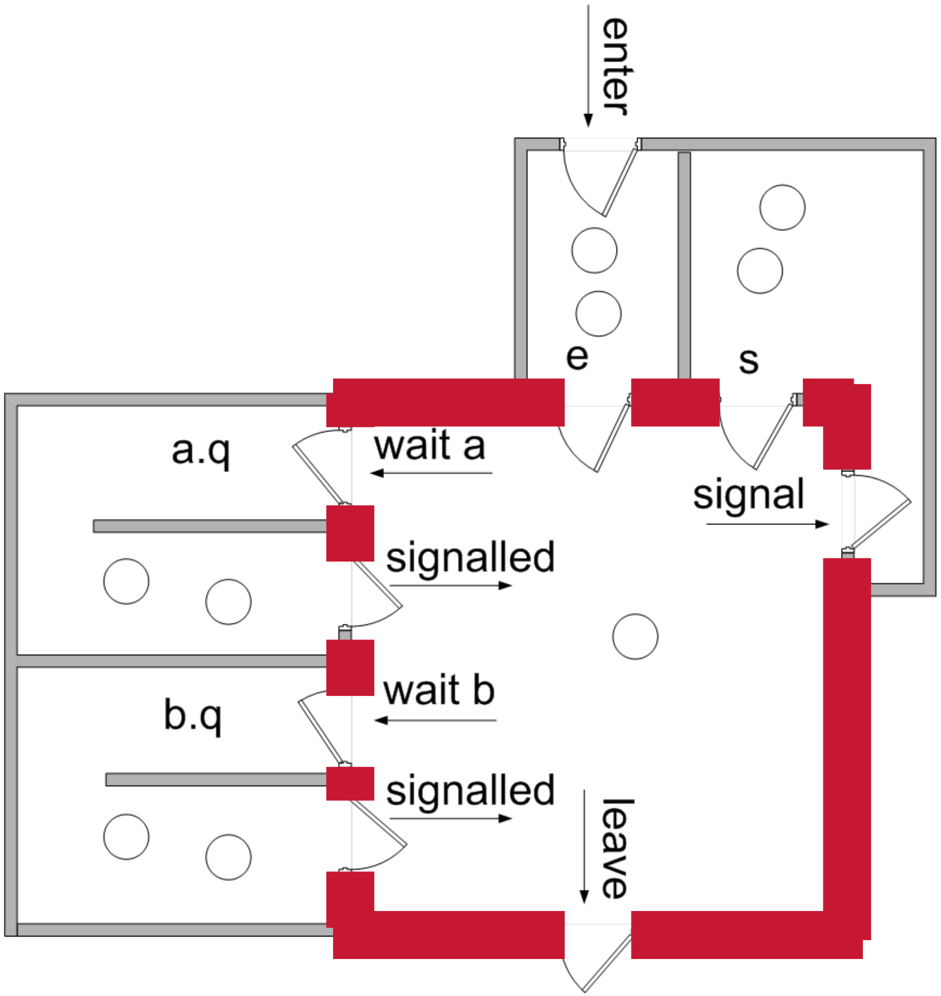
Hoare Semantics: monitor lock transferred directly from signaling thread to woken up thread

- \+ clean semantics, easy to reason about
- – not desirable to force signaling thread to give monitor lock immediately to woken up thread
- – confounds scheduling with synchronization, penalizes threads

Mesa/Hansen Semantics: puts a woken up thread on the monitor entry queue, but does not immediately run that thread, or transfer the monitor lock

Which is Mesa/Hansen? Which is Hoare?





wikipedia.org

What are the implications?

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 resumed

Shared state can be checked using an if statement

Easier to prove liveness

Tricky to implement

Used in most books

Sponsored by a Turing Award
(Tony Hoare)

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 than semaphores

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

Pros of Condition Variables

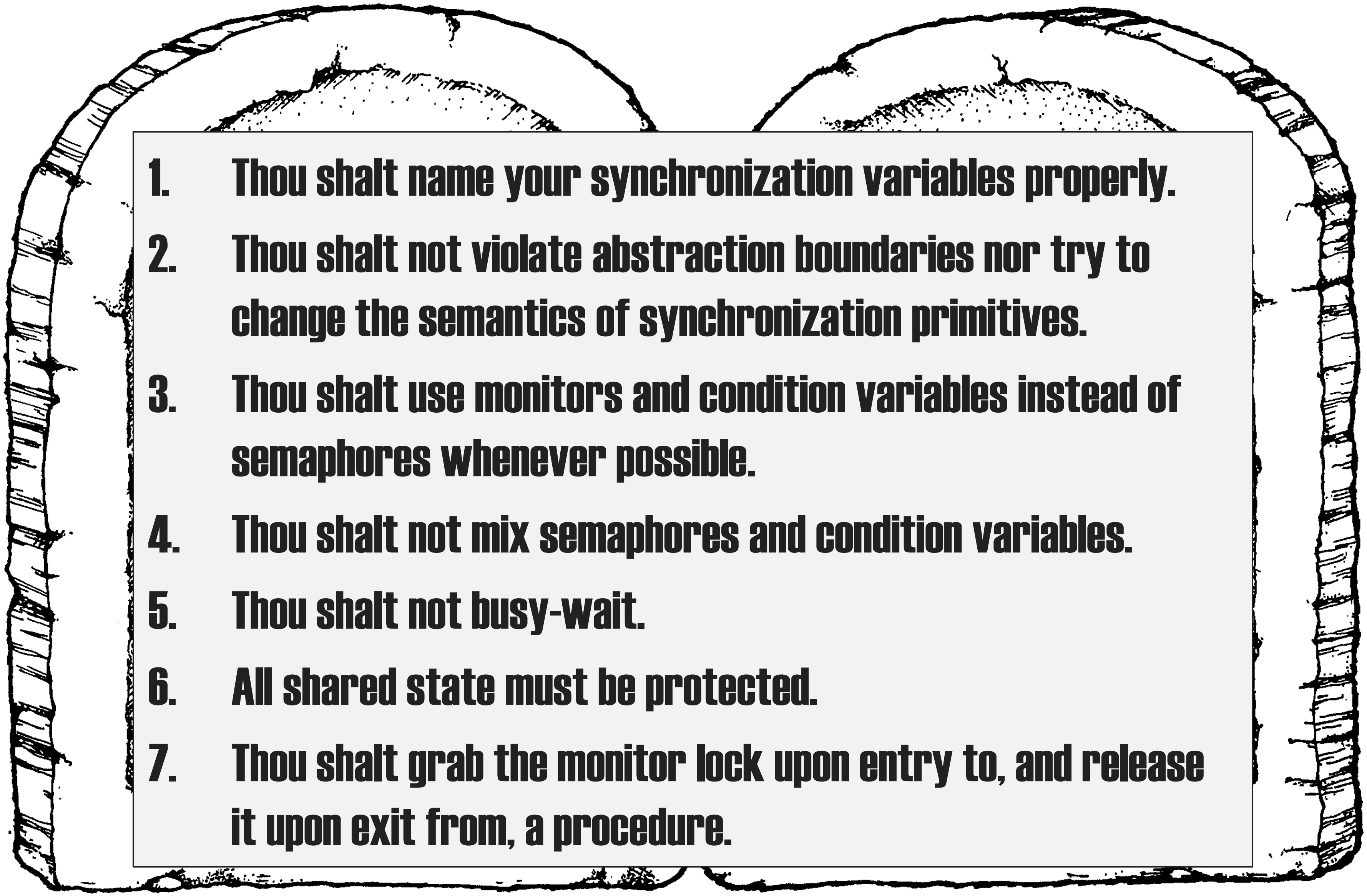
Condition variables force the actual conditions that a thread is waiting for to be made explicit in the code

- comparison preceding the “wait()” call concisely specifies what the thread is waiting for

Condition variables themselves have no state → monitor must explicitly keep the state that is important for synchronization

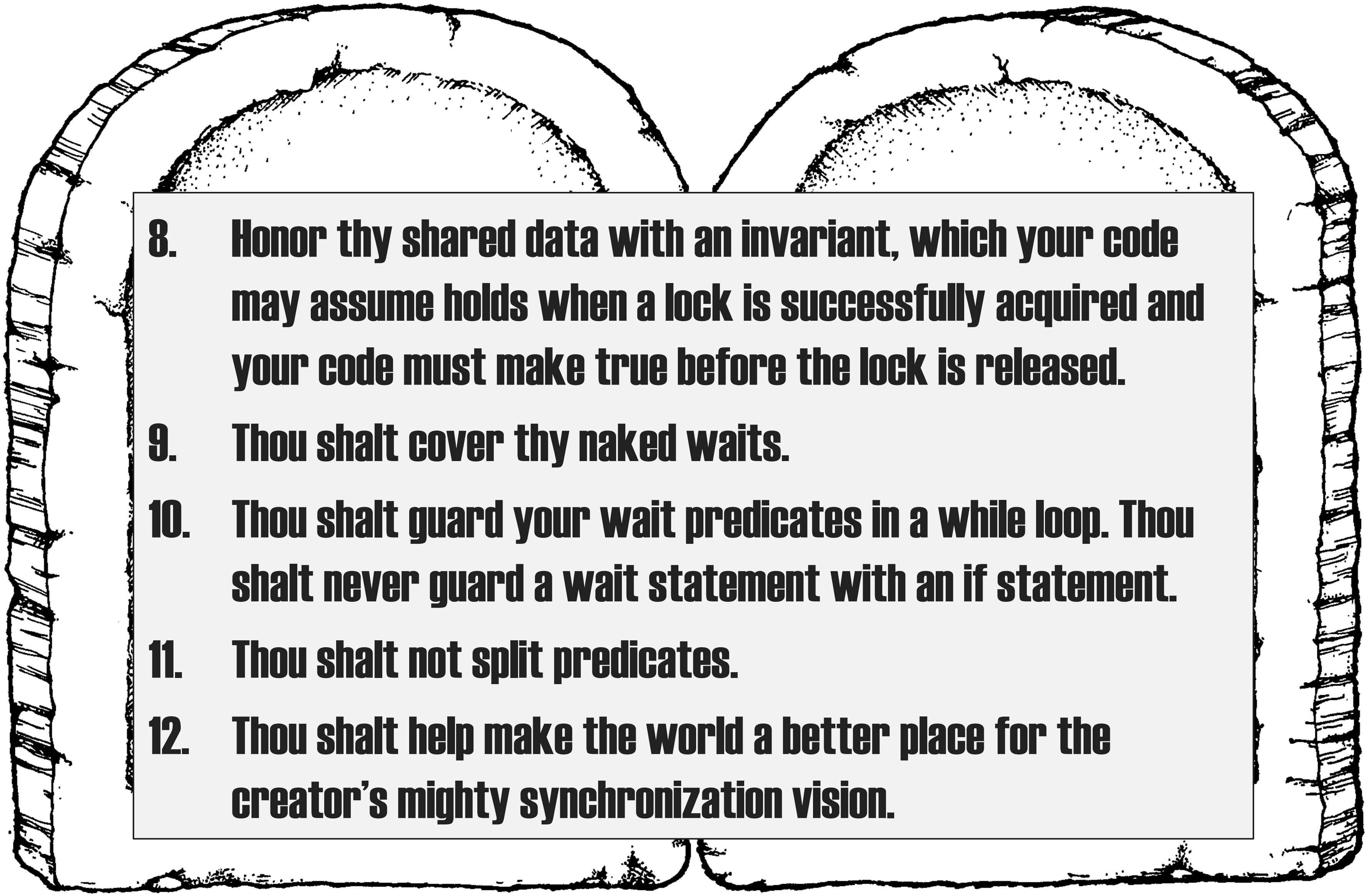
- This is a good thing!

12 Commandments of Synchronization



1. Thou shalt name your synchronization variables properly.
2. Thou shalt not violate abstraction boundaries nor try to change the semantics of synchronization primitives.
3. Thou shalt use monitors and condition variables instead of semaphores whenever possible.
4. Thou shalt not mix semaphores and condition variables.
5. Thou shalt not busy-wait.
6. All shared state must be protected.
7. Thou shalt grab the monitor lock upon entry to, and release it upon exit from, a procedure.

12 Commandments of Synchronization



8. Honor thy shared data with an invariant, which your code may assume holds when a lock is successfully acquired and your code must make true before the lock is released.
9. Thou shalt cover thy naked waits.
10. Thou shalt guard your wait predicates in a while loop. Thou shalt never guard a wait statement with an if statement.
11. Thou shalt not split predicates.
12. Thou shalt help make the world a better place for the creator's mighty synchronization vision.

#8: Honor Thy Shared State with an Invariant

Shared state: buf, n, tail, head

What invariants do we need?

- 0 ≤ n ≤ SIZE
- 0 ≤ head < SIZE
- 0 ≤ tail < SIZE
- 0 ≤ (head – tail) ≤ SIZE

How do we ensure these invariants hold before releasing the lock?

```
Monitor Producer_Consumer {
  char buf[SIZE];
  int n=0, tail=0, head=0;
  condition not_empty, not_full;
  synchronized produce(char ch) {
      while(n == SIZE):
         wait(not_full);
      buf[head] = ch;
      head = (head+1)%SIZE;
      n++;
     notify(not_empty);
   }
   synchronized char consume() {
      while(n == 0):
         wait(not_empty);
      ch = buf[tail];
      tail = (tail+1)%SIZE;
      n--;
      notify(not_full);
      return ch;
  }
}
```

#9: Cover Thy Naked Waits

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

What's 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
```

#10: Guard your wait in a while loop

What is wrong with this?

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

#11: Thou shalt not split predicates

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

What is wrong with this?

Better:

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

A few more guidelines

- Use consistent structure
- Always hold lock when using a condition variable
- Never spin in 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 defining “correct”