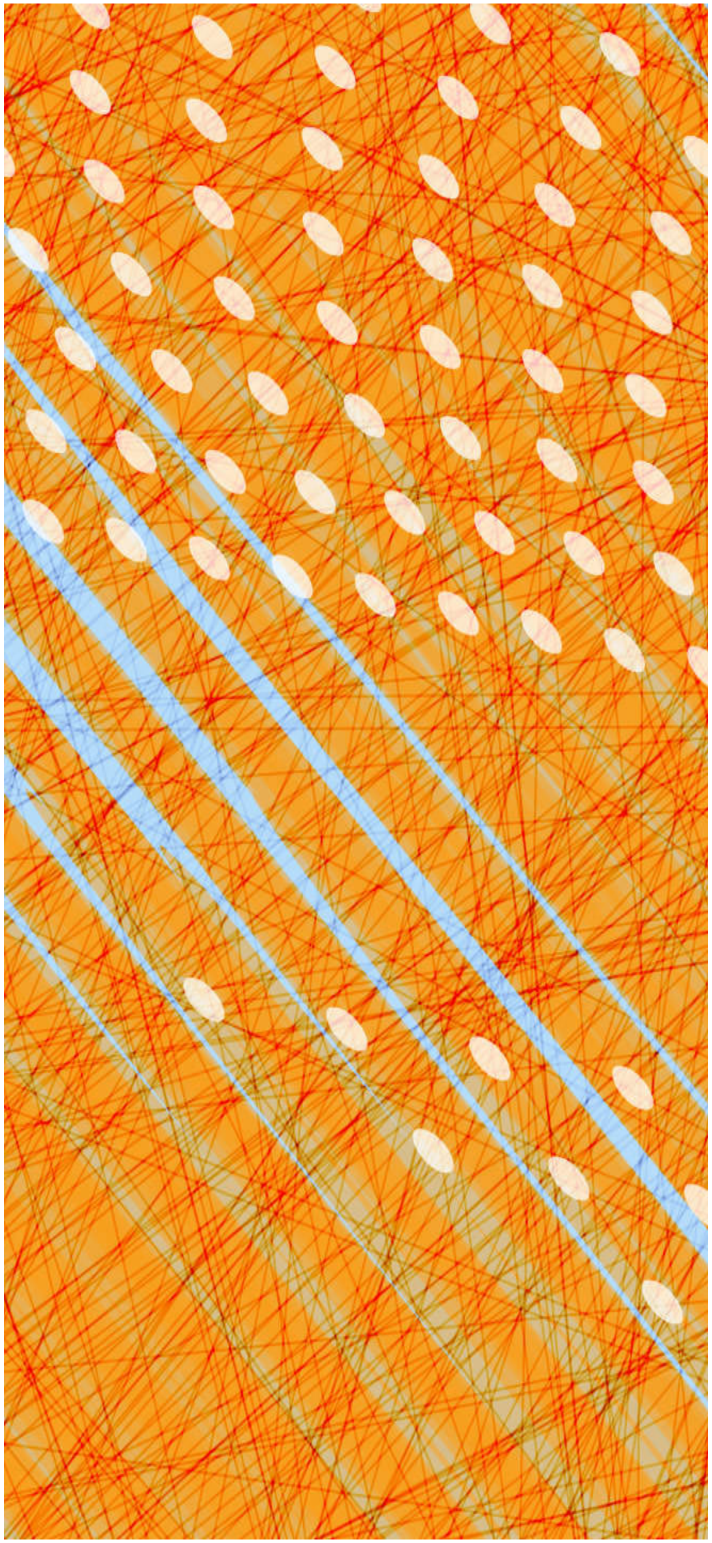
Synchronization

(Chapters 4 & 5)

CS 4410
Operating Systems

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



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

Synchronization Foundations

- Race Conditions
- Critical Sections
- Example: Too Much Milk
- Basic Hardware Primitives
- Building a SpinLock

Recall: Process vs. Thread

Process:

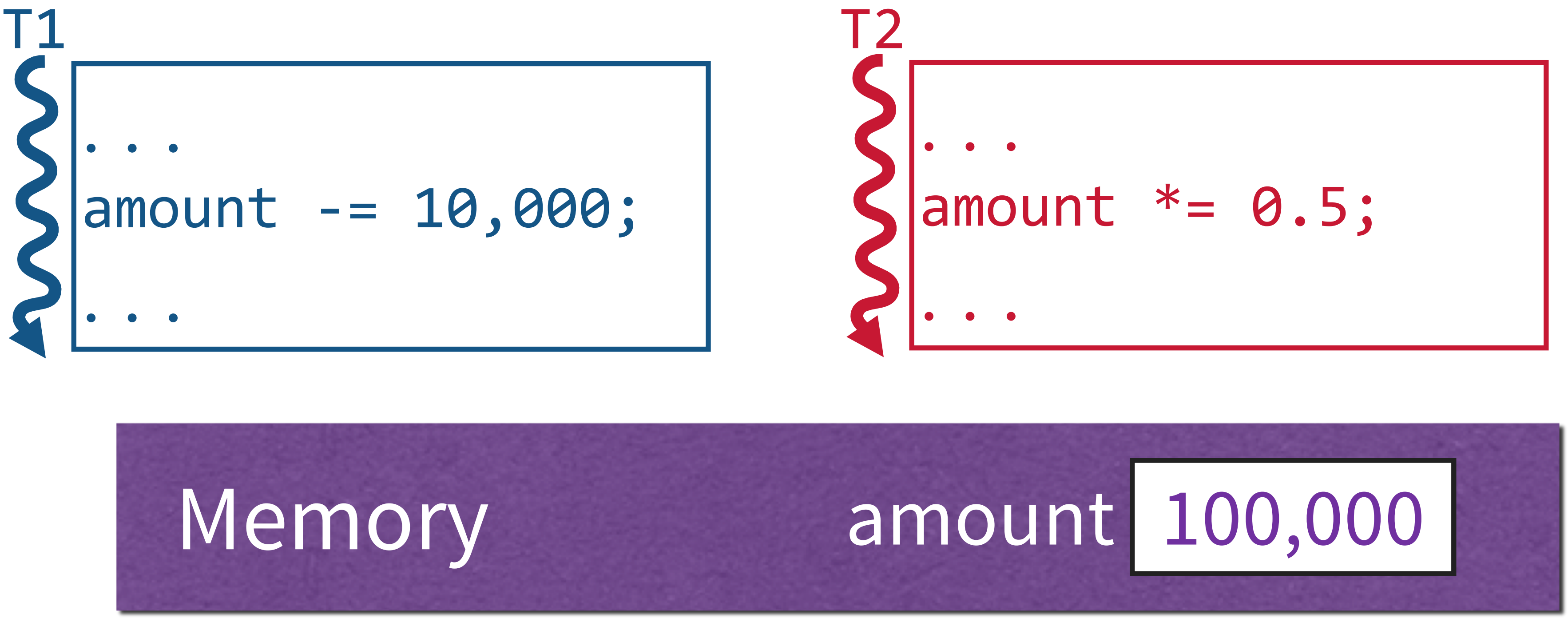
- Privilege Level
- Address Space
- Code, Data, Heap
- Shared I/O resources
- One or more Threads:
    - Stack
    - Registers
    - PC, SP

Shared amongst threads

Two Theads, One Variable

2 threads updating a shared variable **amount**

- One thread wants to decrement amount by $10K
- Other thread wants to decrement amount by 50%



T1

```
. . .
amount -= 10,000;
. . .
```

T2

```
. . .
amount *= 0.5;
. . .
```

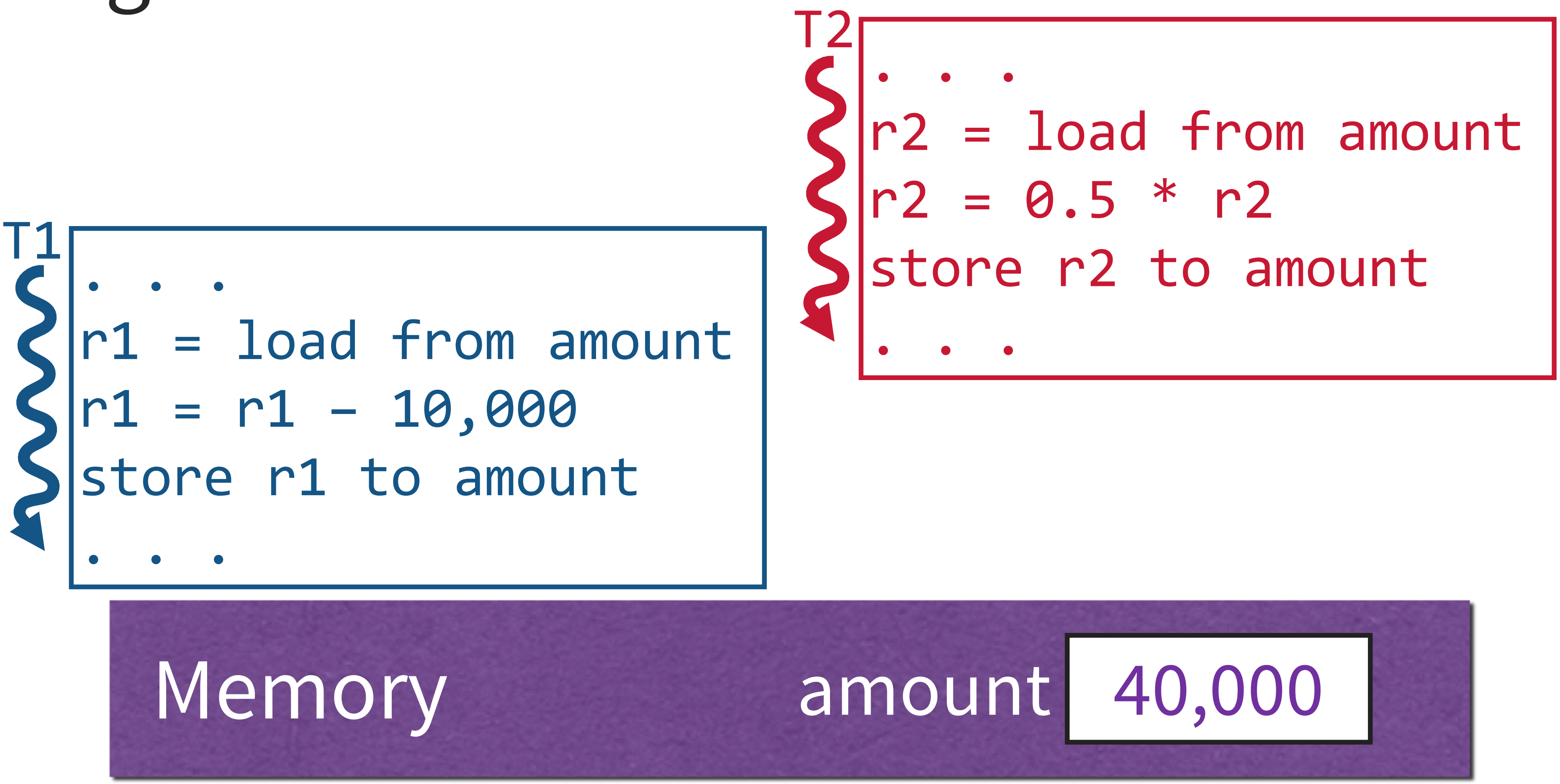
Memory

amount 100,000

What happens when both threads are running?

Two Theads, One Variable

Might execute like this:



T2

```
. . .
r2 = load from amount
r2 = 0.5 * r2
store r2 to amount
. . .
```

T1

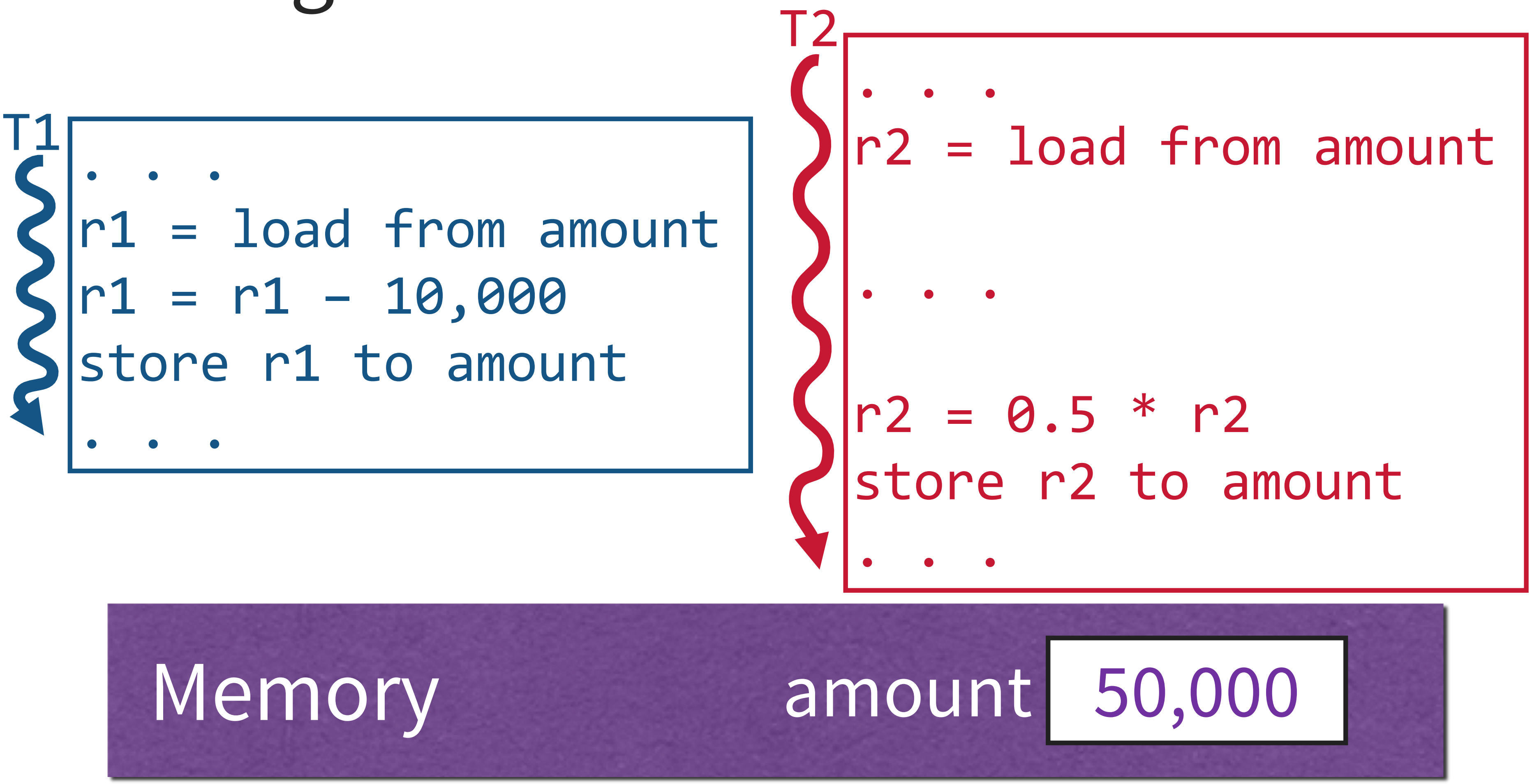
```
. . .
r1 = load from amount
r1 = r1 – 10,000
store r1 to amount
. . .
```

Memory amount 40,000

Or vice versa (T1 then T2 → 45,000)…
either way is fine…

Two Theads, One Variable

Or it might execute like this:



T1

```
. . .
r1 = load from amount
r1 = r1 – 10,000
store r1 to amount
. . .
```

T2

```
. . .
r2 = load from amount

. . .

r2 = 0.5 * r2
store r2 to amount
. . .
```

Lost Update!

Wrong ..and very difficult to debug

Race Conditions

= timing dependent error involving shared state

- Once thread A starts, it needs to "race" to finish
- Whether race condition happens depends on thread schedule
  - Different "schedules" or "interleavings" exist (total order on machine instructions)

All possible interleavings should be safe!

Problems with Sequential Reasoning

1. Program execution depends on the possible interleavings of threads’ access to shared state.

2. Program execution can be nondeterministic.

3. Compilers and processor hardware can reorder instructions.

Race Conditions are Hard to Debug

- Number of possible interleavings is huge
- Some interleavings are good
- Some interleavings are bad:
    - But bad interleavings may rarely happen!
    - Works 100x ≠ no race condition
- Timing dependent: small changes hide bugs

(recall: Therac-25)

Example: Races with Shared Variable

Thread A:

```
while(i < 10)
   i = i + 1;
print “A won!”
```

Thread B:

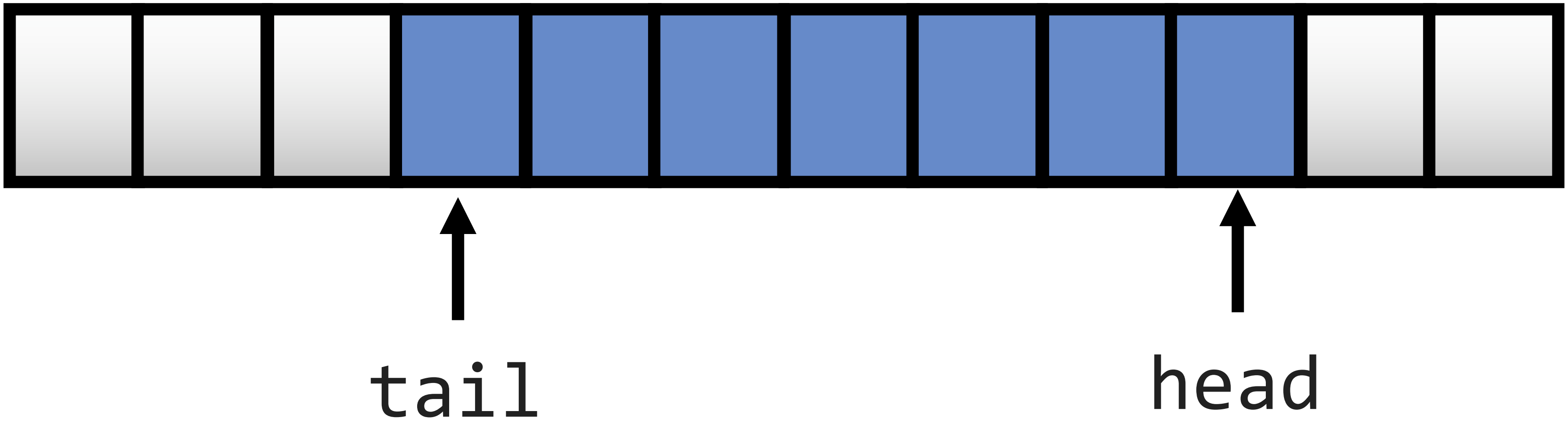
```
while(i > -10)
   i = i - 1;
print “B won!”
```

`i` is shared and initialized to `0`.

Who wins?
Are there any guarantees about this code?
What if both run on different same-speed cores?

Example: Races with Queues

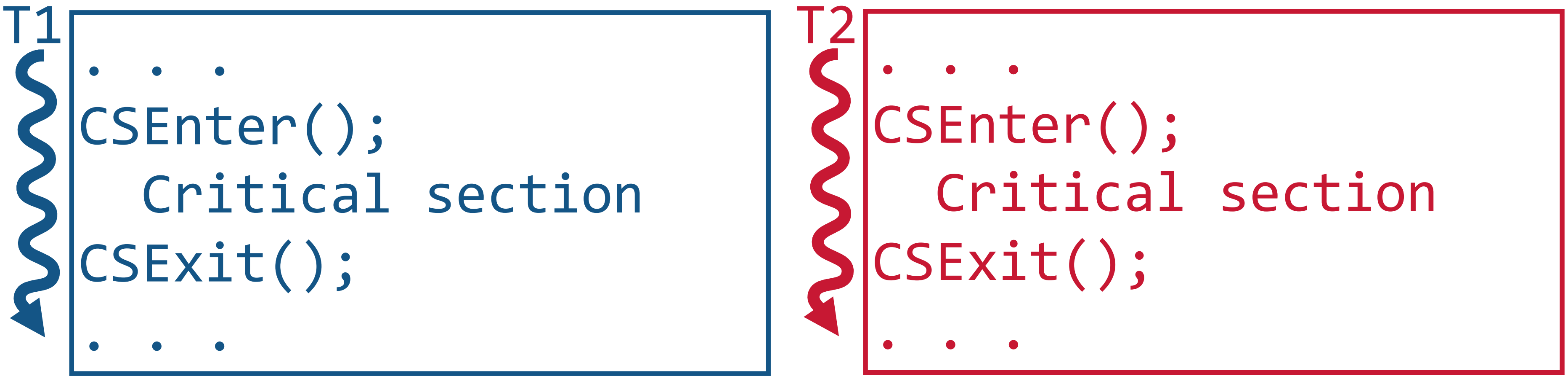
- 2 concurrent enqueue() operations?
- 2 concurrent dequeue() operations?



What could possibly go wrong?

Critical Section

Must be atomic due to shared memory access



T1

```
. . .
CSEnter();
  Critical section
CSExit();
. . .
```

T2

```
. . .
CSEnter();
  Critical section
CSExit();
. . .
```

Goals

Safety: 1 thread in a critical section at time

Liveness: all threads make it into the CS if desired

Fairness: equal chances of getting into CS

… in practice, fairness rarely guaranteed

Too Much Milk:

Safety, Liveness, and Fairness with no hardware support

Too Much Milk Problem

2 roommates, fridge always stocked with milk

- fridge is empty → need to restock it
- *don't want to buy too much milk*

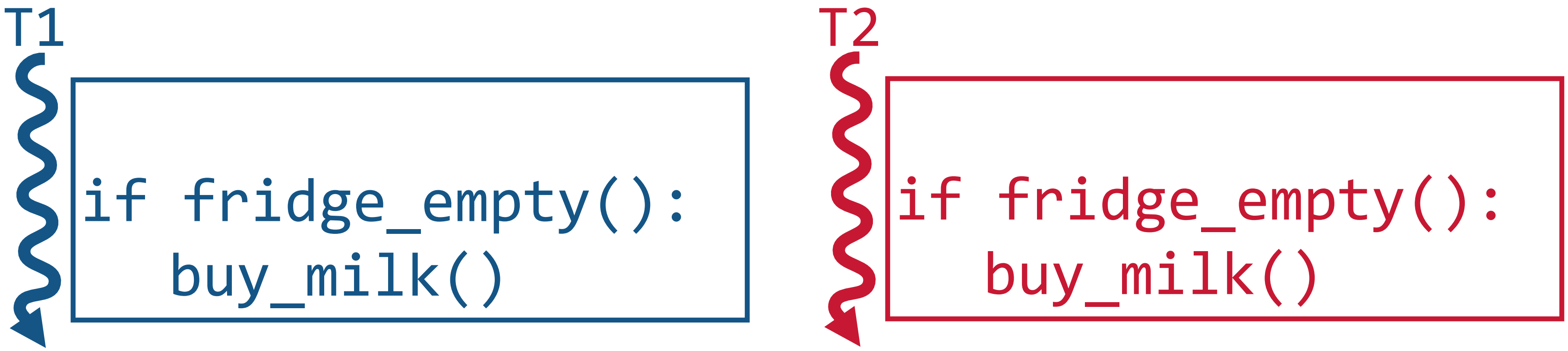
Caveats

- Only communicate by a notepad on the fridge
- Notepad has cells with names, like variables:

`out_to_buy_milk` $\boxed{0}$

TASK: Write the pseudo-code to ensure that at most one roommate goes to buy milk

Solution #1: No Protection



```
if fridge_empty():
    buy_milk()
```

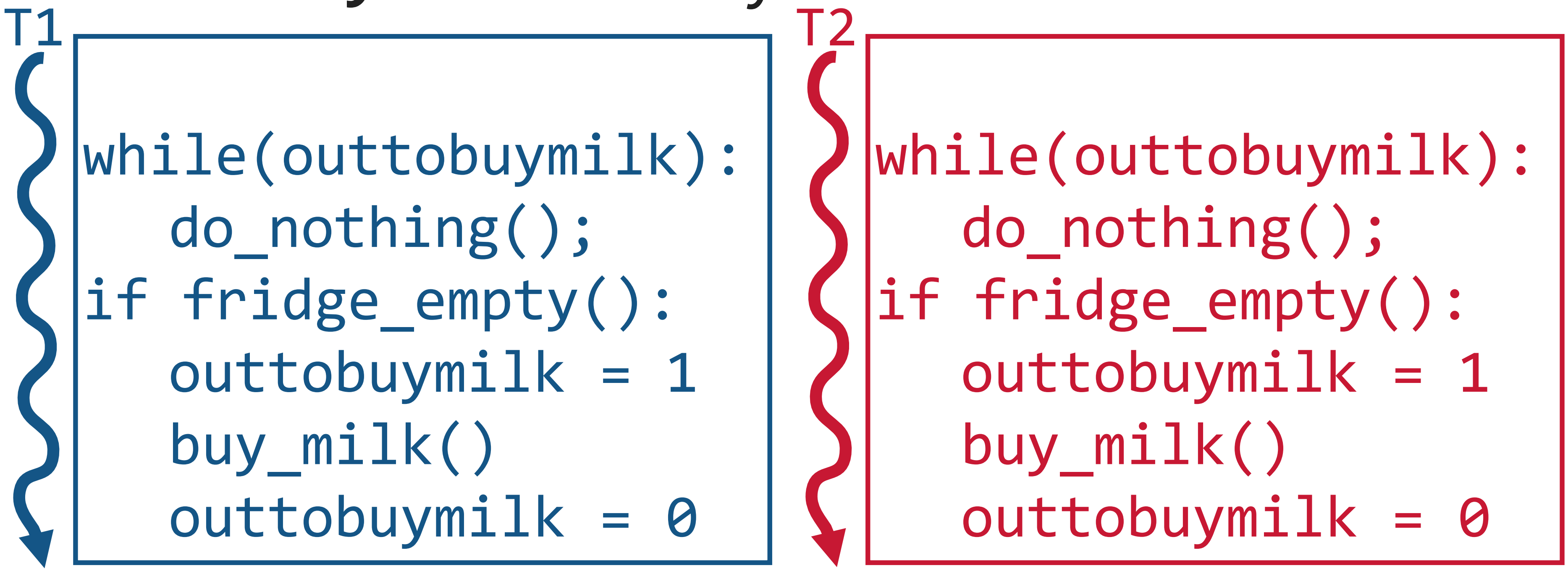
```
if fridge_empty():
    buy_milk()
```

Safety: Only one person (at most) buys milk
Liveness: If milk is needed, someone eventually buys it.
Fairness: Roommates equally likely to go to buy milk.

Safe? Live? Fair?

Solution #2: add a boolean flag

`outtobuymilk` initially false



T1

```
while(outtobuymilk):
   do_nothing();
if fridge_empty():
   outtobuymilk = 1
   buy_milk()
   outtobuymilk = 0
```

T2

```
while(outtobuymilk):
   do_nothing();
if fridge_empty():
   outtobuymilk = 1
   buy_milk()
   outtobuymilk = 0
```

Safety: Only one person (at most) buys milk

Liveness: If milk is needed, someone eventually buys it.

Fairness: Roommates equally likely to go to buy milk.

Safe? Live? Fair?

Solution #3: add two boolean flags!

one for each roommate (initially false):

blues_got_this, ***reds_got_this***

T1

```
blues_got_this = 1
if !reds_got_this and
     fridge_empty():
   buy_milk()
blues_got_this = 0
```

T2

```
reds_got_this = 1
if not blues_got_this
   and fridge_empty():
   buy_milk()
reds_got_this = 0
```

Safety: Only one person (at most) buys milk

Liveness: If milk is needed, someone eventually buys it.

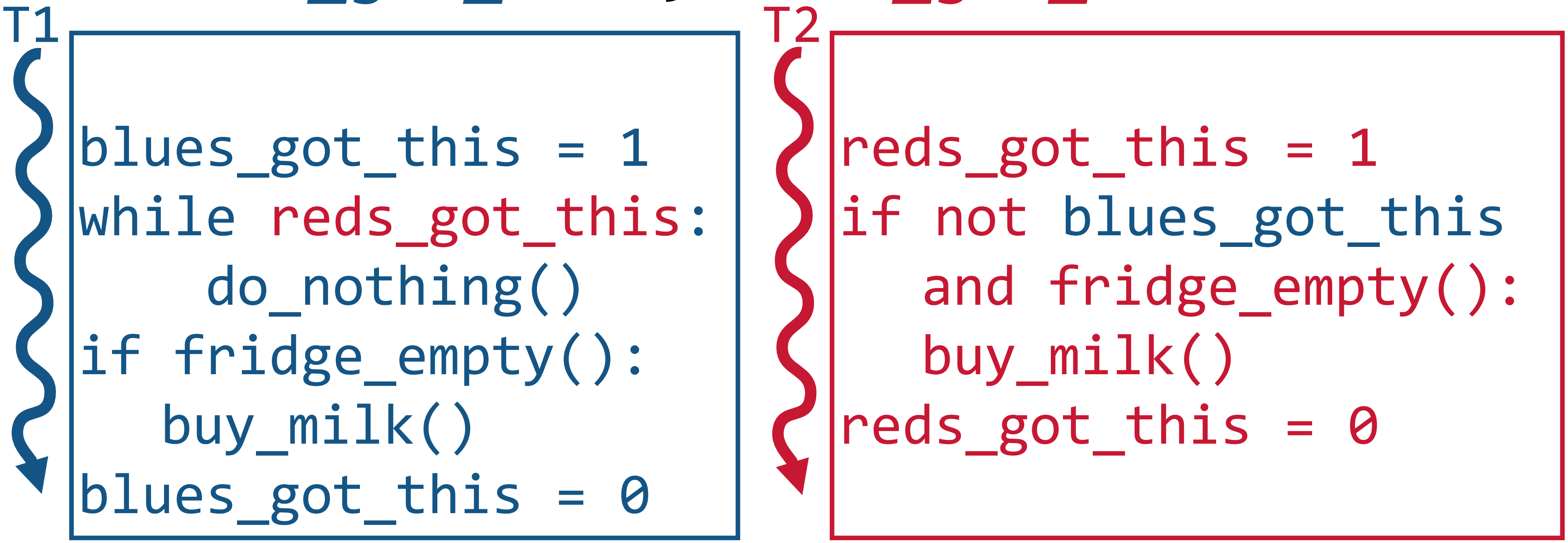
Fairness: Roommates equally likely to go to buy milk.

Safe? Live? Fair?

Solution #4: asymmetric flags!

one for each roommate (initially false):

blues_got_this*, *reds_got_this



T1

```
blues_got_this = 1
while reds_got_this:
    do_nothing()
if fridge_empty():
   buy_milk()
blues_got_this = 0
```

T2

```
reds_got_this = 1
if not blues_got_this
   and fridge_empty():
   buy_milk()
reds_got_this = 0
```

Safe? Live? Fair?

– complicated (and this is a simple example!)

– hard to ascertain that it is correct

– asymmetric code is hard to generalize & unfair

Last Solution: Peterson’s Solution

another flag **turn** {blue, red}

T1

```
blues_got_this = 1
turn = red
while (reds_got_this
   and turn==red):
     do_nothing()
if fridge_empty():
   buy_milk()
blues_got_this = 0
```

T2

```
reds_got_this = 1
turn = blue
while (blues_got_this
   and turn==blue):
     do_nothing()
if fridge_empty():
   buy_milk()
reds_got_this = 0
```

Safe? Live? Fair?

– complicated (and this is a simple example!)

– hard to ascertain that it is correct

– hard to generalize

Hardware Solution

- HW primitives to provide mutual exclusion
- A **machine instruction** (part of the ISA!) that:
  - Reads & updates a memory location
  - Is atomic because it is a single instruction!
- Example: Test-And-Set

  1 instruction with the following semantics:

```
ATOMIC int TestAndSet(int *var) {
   int oldVal = *var;
   *var = 1;
   return oldVal;
}
```

  sets the value to 1, returns former value

Buying Milk with TAS

Shared variable: **`int buyingmilk`**, initially 0

T1

```
while(TAS(&buyingmilk))
     do_nothing();
 if fridge_empty():
   buy_milk()
buyingmilk := 0
```

T2

```
while(TAS(&buyingmilk))
     do_nothing();
 if fridge_empty():
   buy_milk()
buyingmilk := 0
```

A little hard on the eyes. Can we do better?

Enter: Locks!

```
acquire(int *lock) {
   while(test_and_set(lock))
    /* do nothing */;
}
```

```
release(int *lock) {
  *lock = 0;
}
```

Buying Milk with Locks

Shared lock: **`int buyingmilk`**, initially 0

T1

```
acquire(&buyingmilk);
 if fridge_empty():
   buy_milk()
release(&buyingmilk);
```

T2

```
acquire(&buyingmilk);
 if fridge_empty():
   buy_milk()
release(&buyingmilk);
```

Now we're getting somewhere!
Is anyone not happy with this?

Thou shalt not busy-wait!

Not just any locks: **Spin**Locks

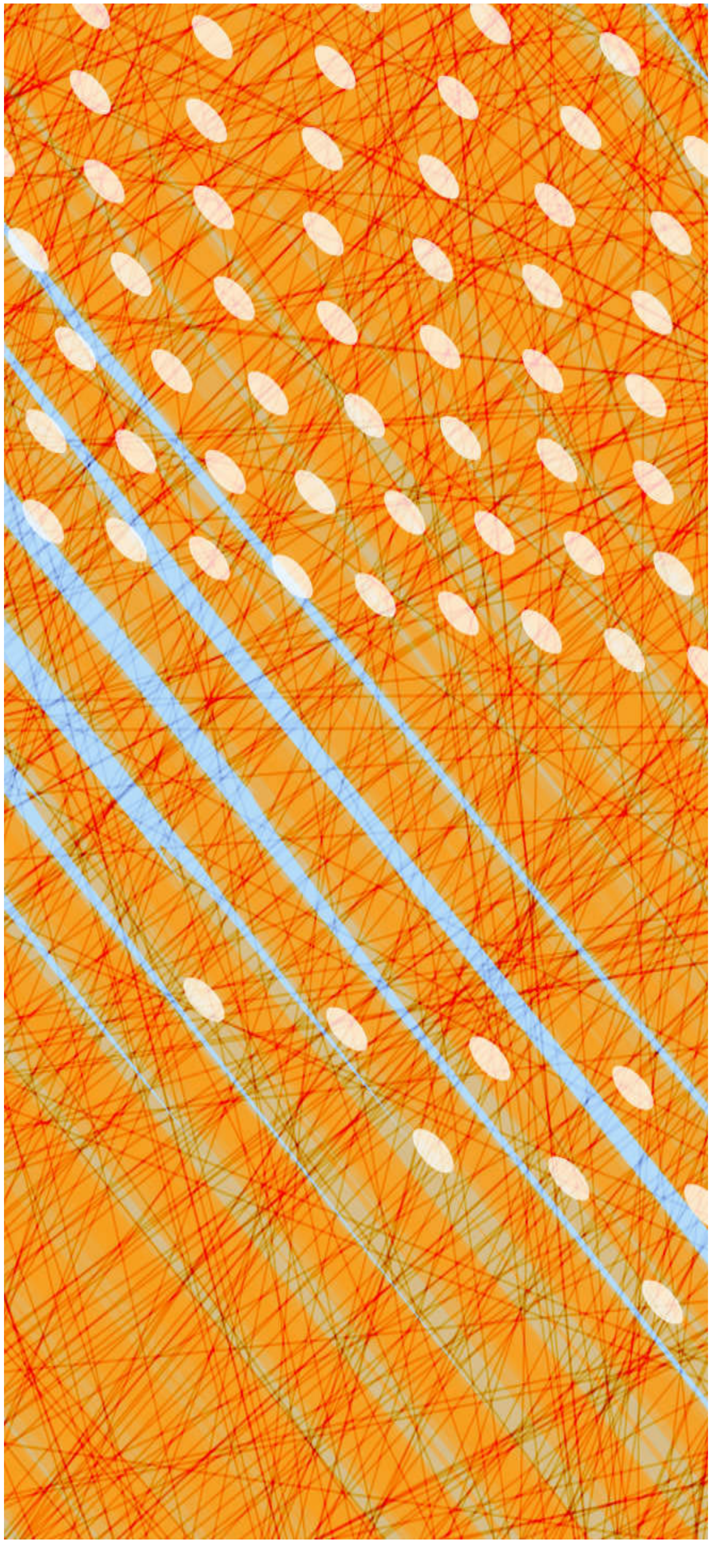
Participants not in critical section must **spin**

→ **wasting CPU cycles**

- Replace the “do nothing” loop with a “yield()”?
- Threads would still be scheduled and descheduled (context switches are expensive)

Need a better primitive:

- allows one thread to pass through
- all others sleep until they can execute again



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

Semaphores

- **Definition**
- Binary Semaphores
- Counting Semaphores
- Classic Sync. Problems (w/Semaphores)
  - Producer-Consumer (w/ a bounded buffer)
  - Readers/Writers Problem
- Classic Mistakes with Semaphores

What is a Semaphore?

[Dijkstra 1962]

Dijkstra introduced in the THE Operating System

Stateful:

- a **value** (incremented/decremented atomically)
- a queue
- a lock

Interface:

- Init(starting value)
- **P (procure)**: decrement, “consume” or “start using”
- **V (vacate)**: increment, “produce” or “stop using”

No operation to read the value!

DUTCH 4410: P = PROBEER (‘TRY'), V = VERHOOG ('INCREMENT', 'INCREASE BY ONE')

Semantics of P and V (Part 1)

P():

- wait until value >0
- when so, decrement VALUE by 1

```
P() {
    while(n <= 0)
        ;
    n -= 1;
}
```

V():

- increment VALUE by 1

```
V() {
    n += 1;
}
```

*These are the **semantics**,*

but how can we make this efficient?

(doesn’t this look like a spinlock?!?)

Semantics of P and V (Complete)

P():

- block (**sit on Q)** til value >0
- when so, decrement VALUE by 1

```
P() {
    while(n <= 0)
        ;
    n -= 1;
}
```

V():

- increment VALUE by 1
- **resume a thread waiting on Q (if any)**

```
V() {
    n += 1;
}
```

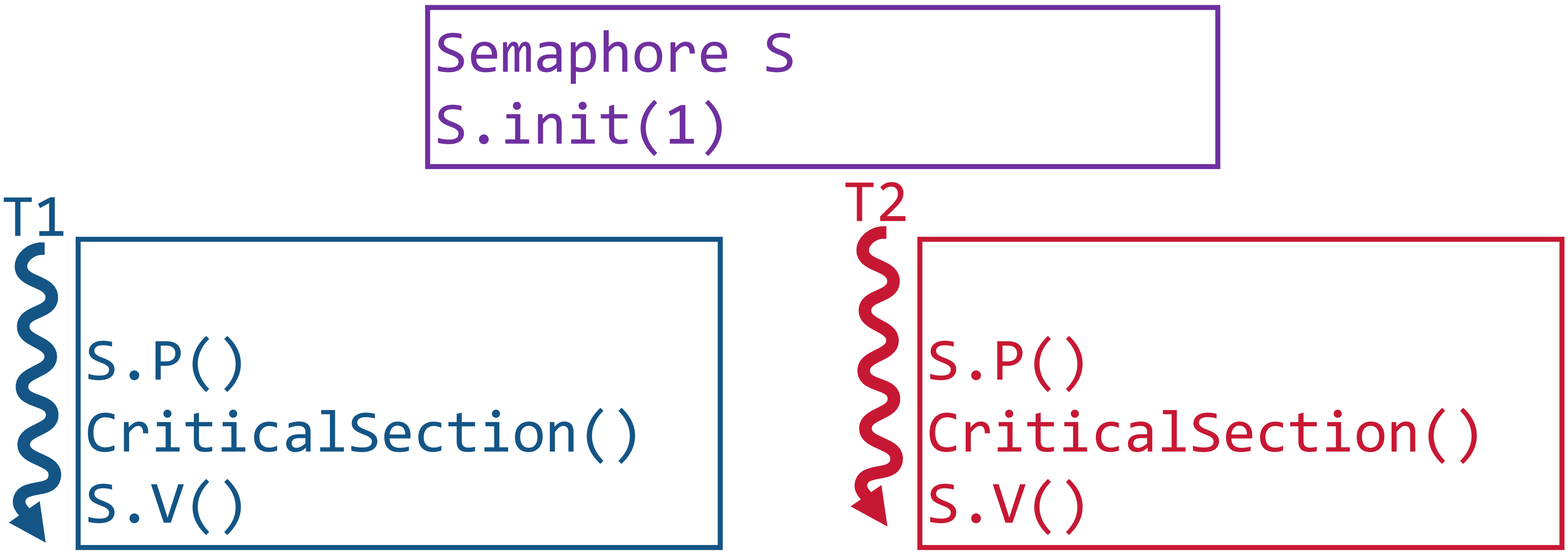
Okay this looks efficient, but how is this safe?

(that's what the lock is for – both P&V need to TAS the lock)

Binary Semaphore

Semaphore value is either 0 or 1

- Used for **mutual exclusion**
  (semaphore as a more efficient lock)
- Initially 1 in that case



```
Semaphore S
S.init(1)
```

T1

```
S.P()
CriticalSection()
S.V()
```

T2

```
S.P()
CriticalSection()
S.V()
```

Example: A simple mutex

```
Semaphore S
S.init(1)
```

```
S.P()
CriticalSection()
S.V()
```

```
V() {
    n += 1;
}
```

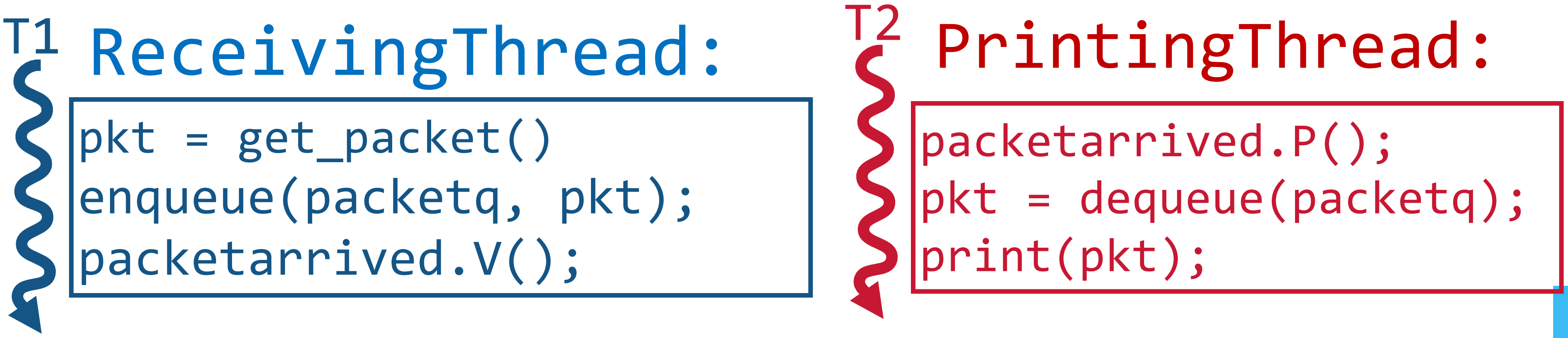
```
P() {
    while(n <= 0)
       ;
    n -= 1;
}
```

Counting Semaphores

Sema count can be any integer

- Used for signaling or counting resources
- Typically:
  - one thread performs P() to await an event
  - another thread performs V() to alert waiting thread that event has occurred

```
Semaphore packetarrived
packetarrived.init(0)
```



ReceivingThread:

```
pkt = get_packet()
enqueue(packetq, pkt);
packetarrived.V();
```

PrintingThread:

```
packetarrived.P();
pkt = dequeue(packetq);
print(pkt);
```

Semaphore's count:

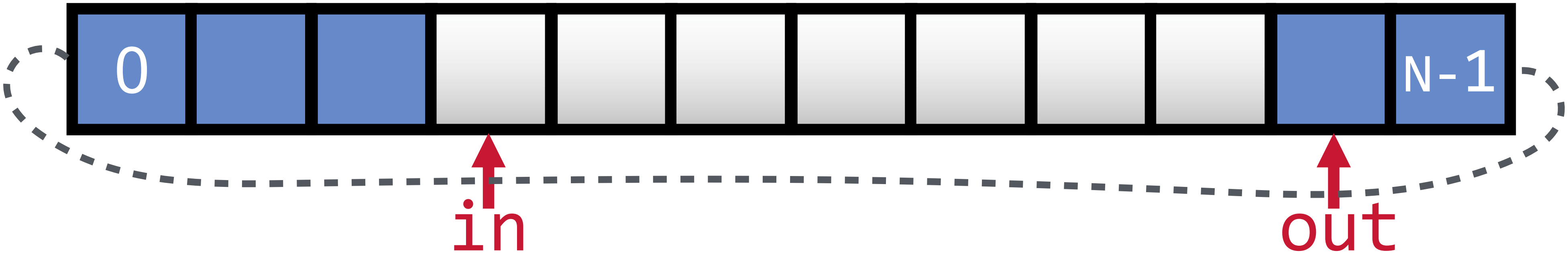
- must be initialized!
- keeps state
  - reflects the sequence of past operations
  - `>0` reflects number of future P operations that will succeed

Not possible to:

- read the count
- grab multiple semaphores at same time
- decrement/increment by more than `1`!

Producer-Consumer Problem

2+ threads communicate:
some threads **produce** data that others **consume**



Bounded buffer: size —**N entries**—

Producer process writes data to buffer

- Writes to **`in`** and moves rightwards

Consumer process reads data from buffer

- Reads from **`out`** and moves rightwards

Producer-Consumer Applications

- Pre-processor produces source file for compiler’s parser
- Data from bar-code reader consumed by device driver
- File data: computer → printer spooler → line printer device driver
- Web server produces data consumed by client’s web browser
- “pipe” (|) in Unix `>cat file | sort | more`

Starter Code: No Protection

```
Shared:
int buf[N];
int in, out;
```

```
// add item to buffer
void produce(int item) {
  buf[in] = item;
  in = (in+1)%N;
}
```

```
// remove item
int consume() {
  int item = buf[out];
  out = (out+1)%N;
  return item;
}
```

Problems:

1. Unprotected shared state (multiple producers/consumers)
2. Inventory:
   - Consumer could consume when nothing is there!
   - Producer could overwrite not-yet-consumed data!

Part 1: Guard Shared Resources

```
Shared:
int buf[N];
int in, out;
Semaphore mutex_in(1), mutex_out(1);
```

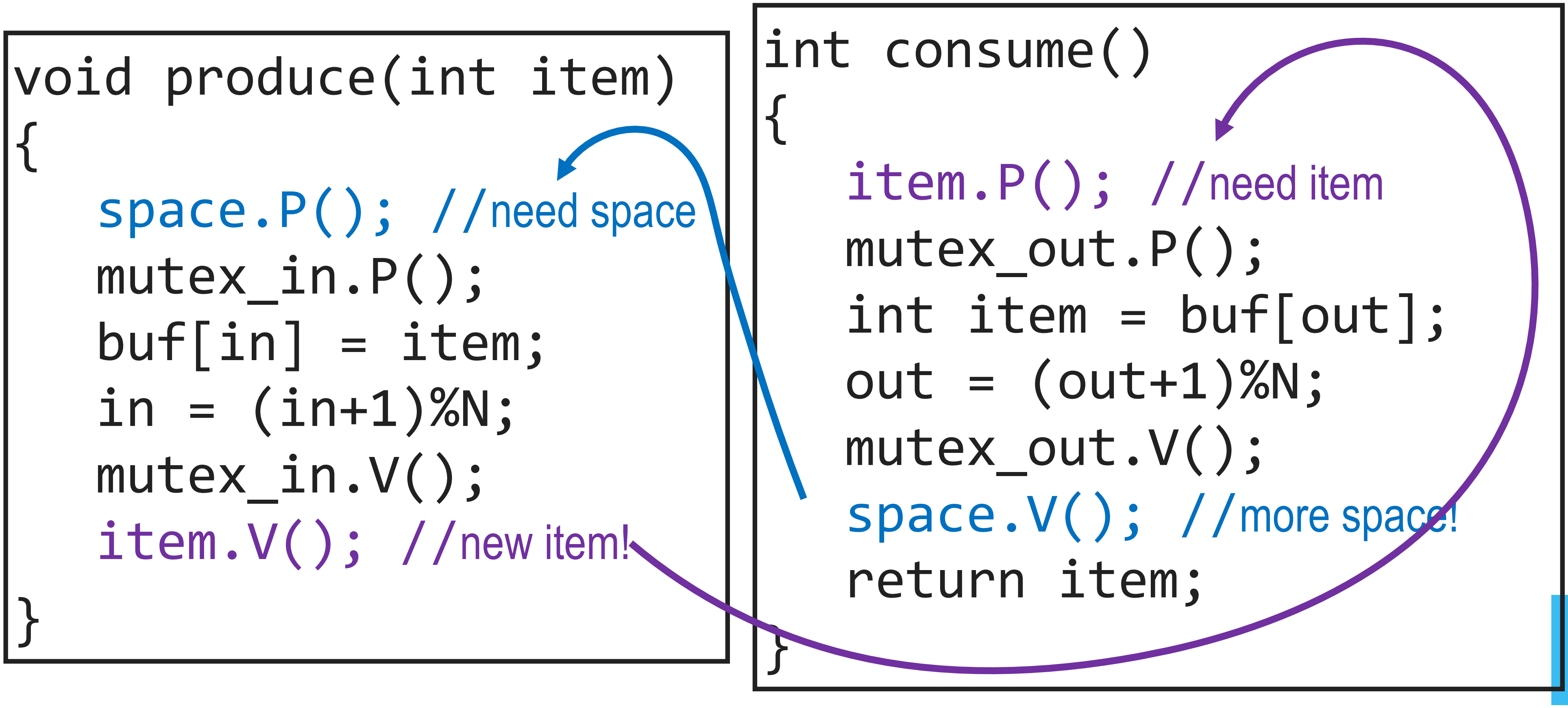
```
// add item to buffer
void produce(int item)
{
  mutex_in.P();
  buf[in] = item;
  in = (in+1)%N;
  mutex_in.V();

}
```

now atomic

```
// remove item
int consume()
{
  mutex_out.P();
  int item = buf[out];
  out = (out+1)%N;
  mutex_out.V();
  return item;
}
```

Part 2: Manage the Inventory

```
Shared:
int buf[N];
int in, out;
Semaphore mutex_in(1), mutex_out(1);
Semaphore space(N), item(0);
```



```
void produce(int item)
{
   space.P(); //need space
   mutex_in.P();
   buf[in] = item;
   in = (in+1)%N;
   mutex_in.V();
   item.V(); //new item!
}
```

```
int consume()
{
   item.P(); //need item
   mutex_out.P();
   int item = buf[out];
   out = (out+1)%N;
   mutex_out.V();
   space.V(); //more space!
   return item;
}
```

Sanity checks

1. **Is there a V for every P?**
2. **Mutex initialized to 1?**
3. **Mutex P&V in same thread?**

```
Shared:
int buf[N];
int in, out;
Semaphore mutex_in(1), mutex_out(1);
Semaphore space(N), item(0);
```

```
void produce(int item)
{
   space.P(); //need space
   mutex_in.P();
   buf[in] = item;
   in = (in+1)%N;
   mutex_in.V();
   item.V(); //new item!
}
```

```
int consume()
{
   item.P(); //need item
   mutex_out.P();
   int item = buf[out];
   out = (out+1)%N;
   mutex_out.V();
   space.V(); //more space!
   return item;
}
```

Producer-consumer: How did we do?

Pros:

- Live & Safe & Correct
- No Busy Waiting! *(is this true?)*
- Scales nicely

Cons:

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

Readers-Writers Problem

[Courtois+ 1971]

Models access to a database: shared data that some threads **read** and other threads **write**

At any time, want to allow:

- multiple concurrent readers *—OR—(exclusive)*
- only a single writer

Example: making an airline reservation

- Browse flights: web site acts as a **reader**
- Reserve a seat: web site has to **write** into database (to make the reservation)

Readers-Writers Specifications

N threads share **1** object in memory

- Some write: **1** writer active at a time
- Some read: **n** readers active simultaneously

Insight: generalizes the critical section concept

Implementation Questions:

1. Writer is active. Combo of readers/writers arrive.

   Who should get in next?

2. Writer is waiting. Endless of # of readers come.

   Fair for them to become active?

For now: back-and-forth turn-taking:

- If a reader is waiting, readers get in next
- If a writer is waiting, one writer gets in next

Readers-Writers Solution

```
Shared:
int rcount;
Semaphore count_mutex(1);
Semaphore rw_lock(1);
```

```
void write()
    rw_lock.P();
    . . .
    /*perform write */
    . . .
     rw_lock.V();

}
```

```
int read()
{
    count_mutex.P();
    rcount++;
    if (rcount == 1)
      rw_lock.P();
    count_mutex.V();
    . . .
    /* perform read */
    . . .
    count_mutex.P();
    rcount--;
    if (rcount == 0)
       rw_lock.V();
     count_mutex.V();
}
```

Readers-Writers: Understanding the Solution

If there is a writer:

- First reader blocks on **`rw_lock`**
- Other readers block on **`mutex`**

Once a reader is active, all readers get to go through

- Which reader gets in first?

The last reader to exit signals a writer

- If no writer, then readers can continue

If readers and writers waiting on rw_lock & writer exits

- Who gets to go in first?

Readers-Writers: Assessing the Solution

When readers active no writer can enter ✔

- Writers wait @ `rw_lock`.P()

When writer is active nobody can enter ✔

- Any other reader or writer will wait (where?)

Back-and-forth isn't so fair:

- Any number of readers can enter in a row
- Readers can "starve" writers

Fair back-and-forth semaphore solution is tricky!

- Try it! (don't spend too much time…)

Semaphores

- Definition
- Binary Semaphores
- Counting Semaphores
- Classic Sync. Problems (w/Semaphores)
  - Producer-Consumer (w/ a bounded buffer)
  - Readers/Writers Problem
- **Classic Mistakes with Semaphores**

Classic Semaphore Mistakes

```
P(S)
CS
P(S)
```

I

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

```
V(S)
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
```

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 … 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!

These are “low-level” primitives. Small errors:

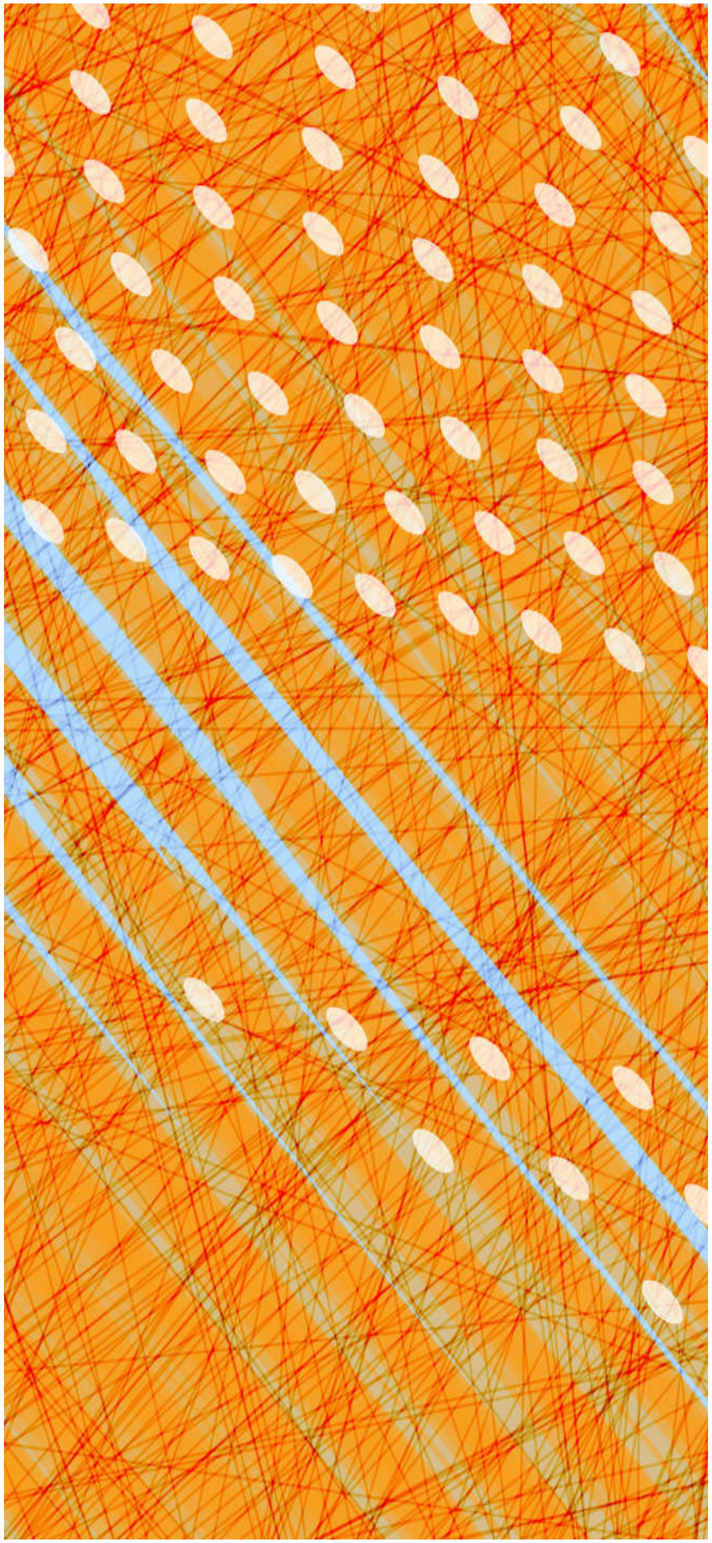
- Easily bring system to grinding halt
- Very difficult to debug

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 Monitors



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

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() {
    }
}
```

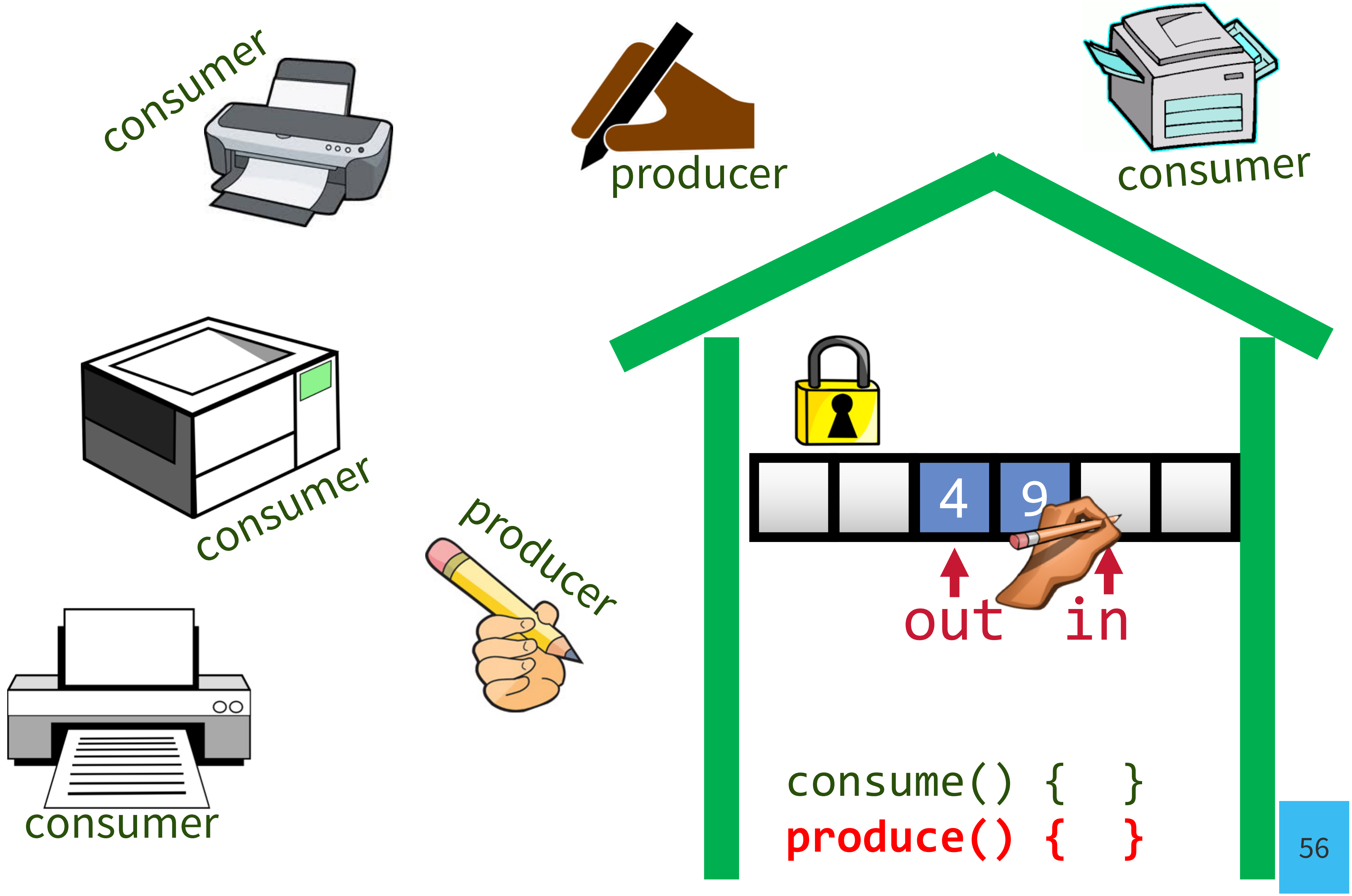
for example:

```
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

One Thread at a Time in the Monitor!



```
consume() {  }
produce() {  }
```

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 **`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 & 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

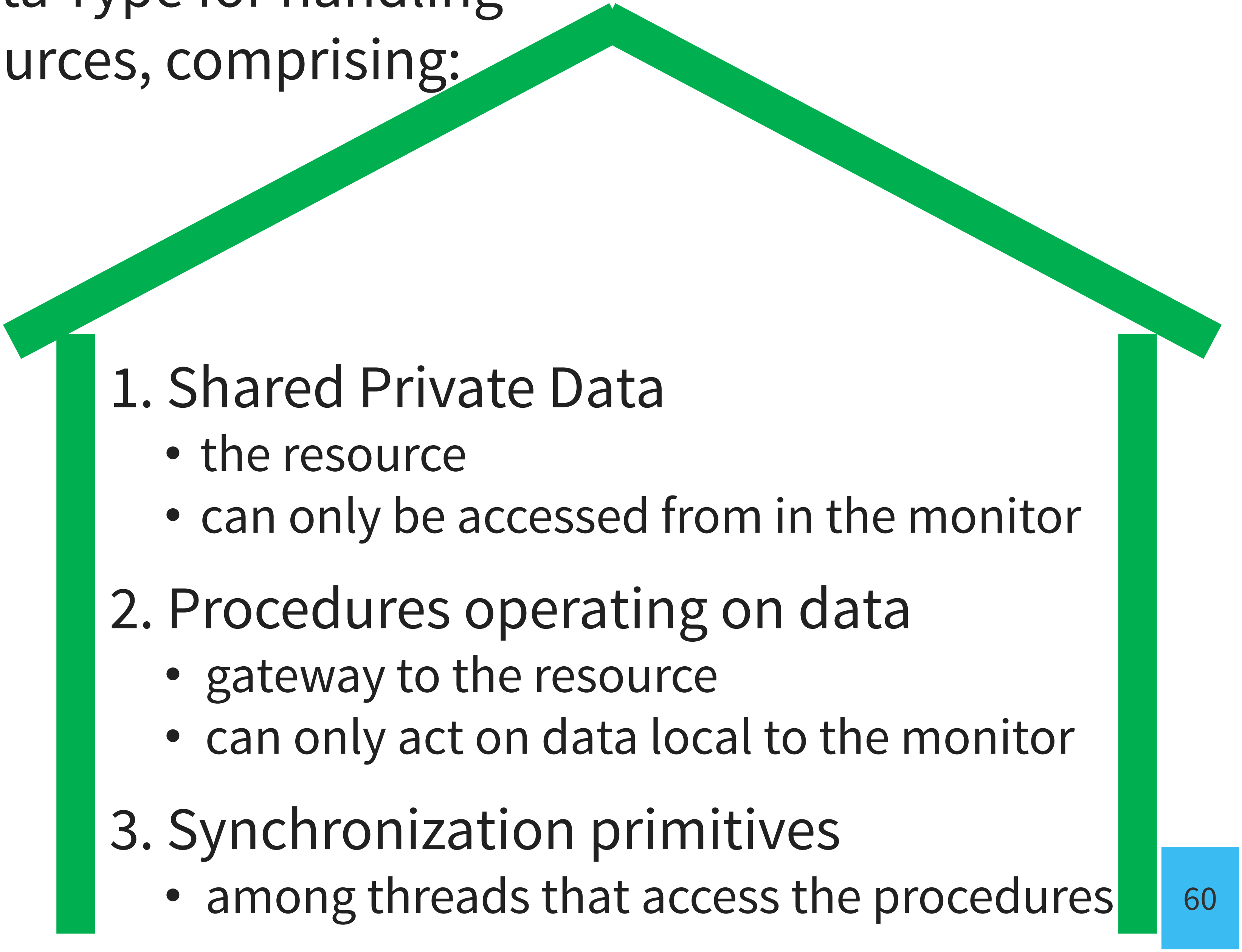
When the condition becomes satisfied:

`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_Karthik()
    facetime_oma()
    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()`

Monitors in “4410 Python” : __init__

Python

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

4410 Python

```
from rvr import MP, MPthread

class BurgerKingMonitor(MP):
  def __init__(self):
    MP.__init__(self,None)
    self.lock = Lock(“monitor lock”)
    self.hungrykid = self.lock.Condition(“hungry kid”)
    self.nBurgers = self.Shared(“num burgers”, 0)
```

Monitors in “4410 Python” : kid_eat

Python

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

4410 Python

```python
def kid_eat(self):
  with self.lock:
    while (self.nBurgers.read() == 0):
      self.hugryKid.wait()
    self.nBurgers.dec()
```

We do this for helpful feedback:

- from auto-grader
- from debugger

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

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

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 waiting readers

A reader can enter if:

- no active writer

  &&

- no waiting writers

When a writer finishes:

check for waiting readers

Y → lets all enter

N → if writer waiting, lets 1 enter

Last reader finishes:

- it lets 1 writer in (if any)

Fair?

Tries to be fair:

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

… mostly fair, although once it lets a reader in, it lets ALL waiting readers in all at once, even if some showed up “after” other waiting writers

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?

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?

```
kid_main() {

    play_w_legos()
    BK.kid_eat()
    bathe()
    make_robots()
    BK.kid_eat()
    facetime_Karthik()
    facetime_oma()
    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()
}
```

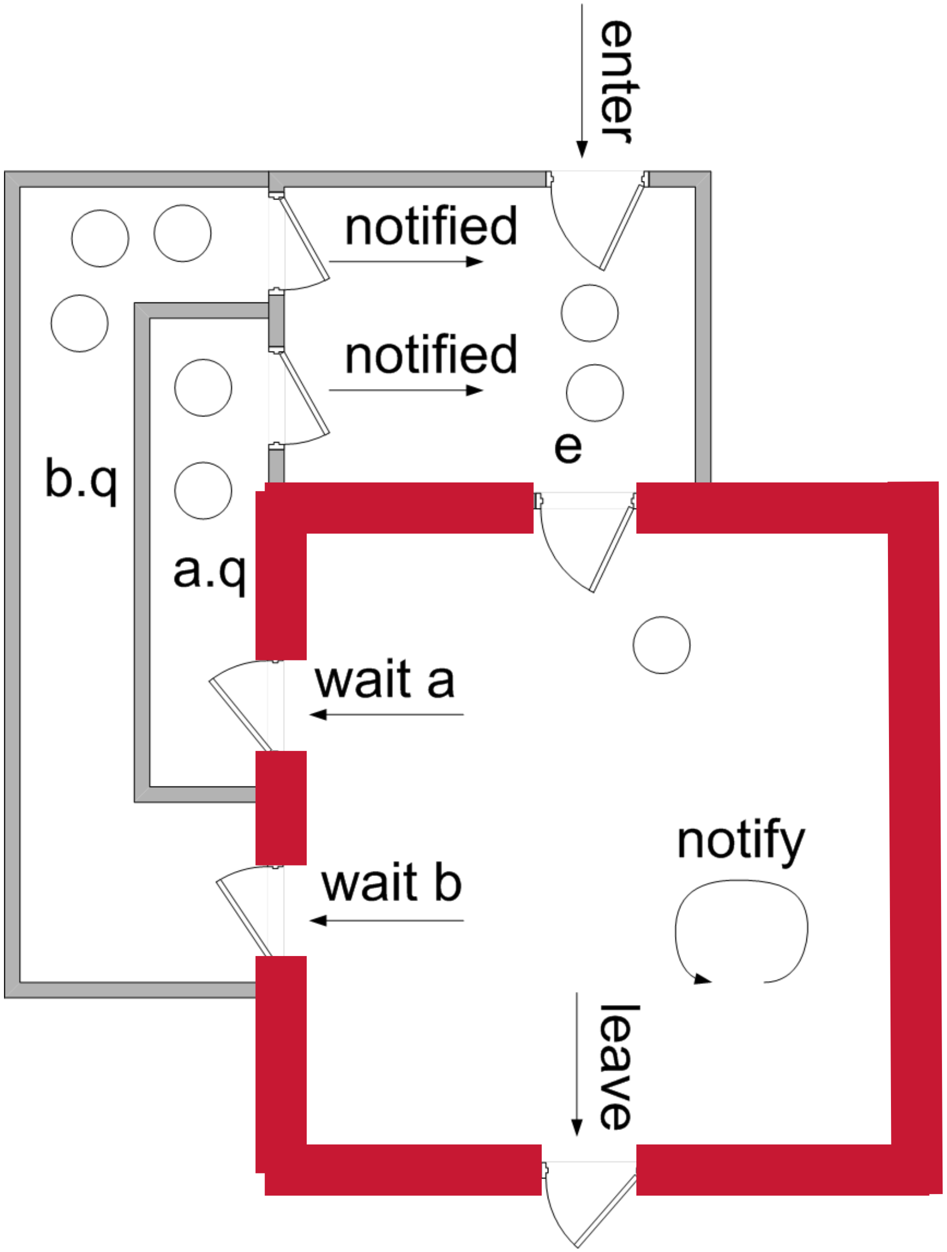
Hoare vs. Mesa/Hansen Semantics

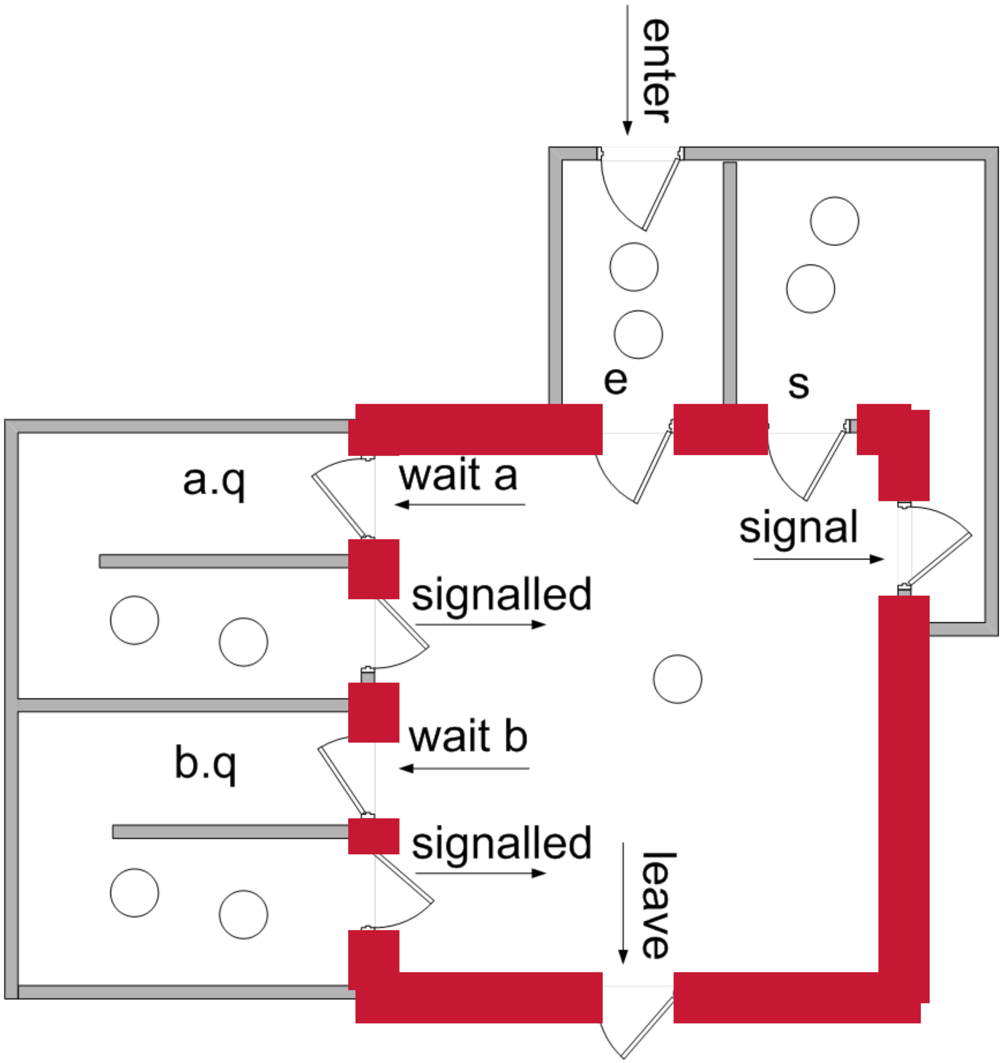
Hoare Semantics: monitor lock transferred directly from signaling thread to woken up thread

+ 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 and safer than semaphores

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

Pros of Condition Variables

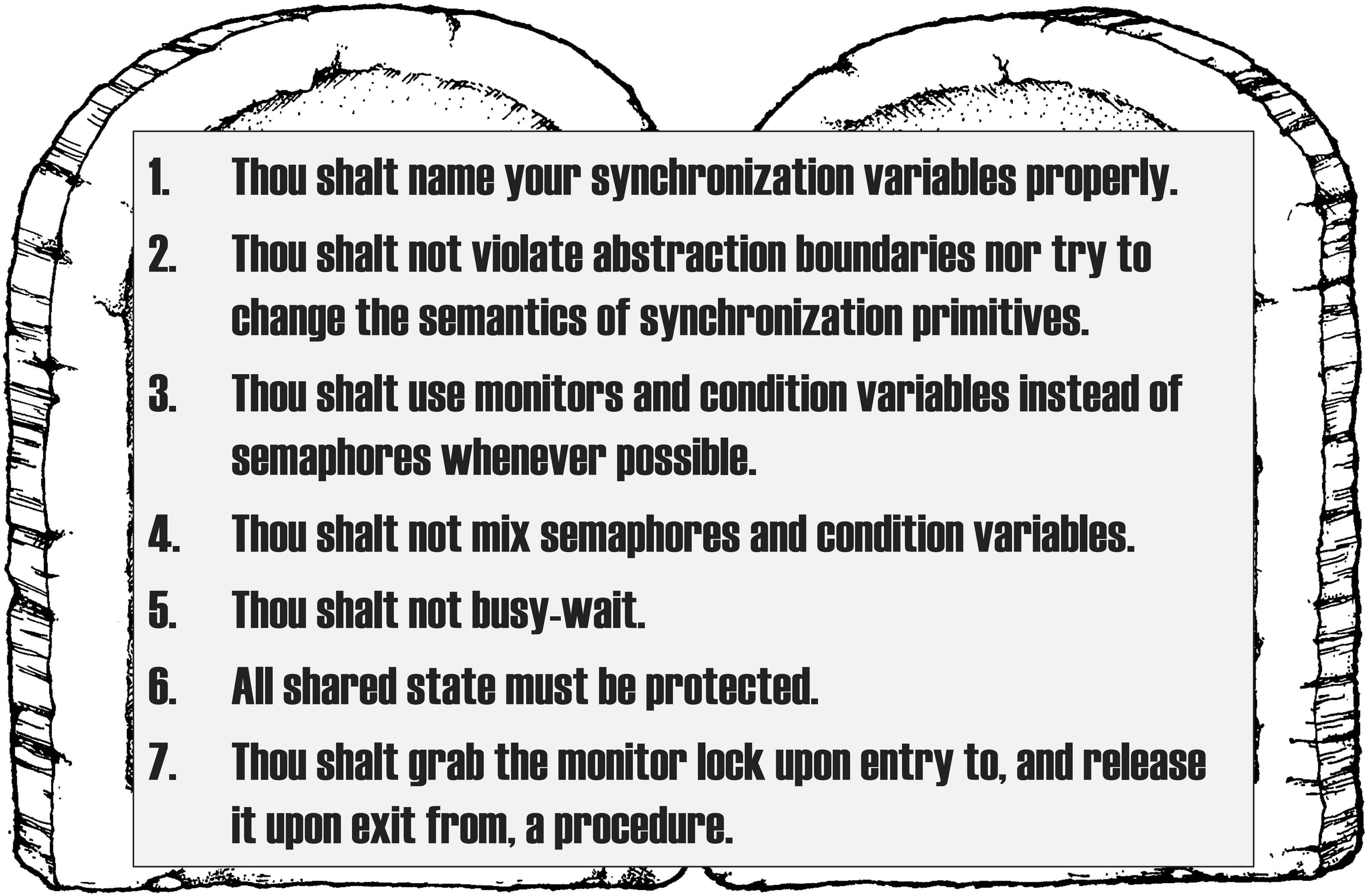
Condition variables force the actual conditions that a thread is waiting for to be made explicit in the code

- 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

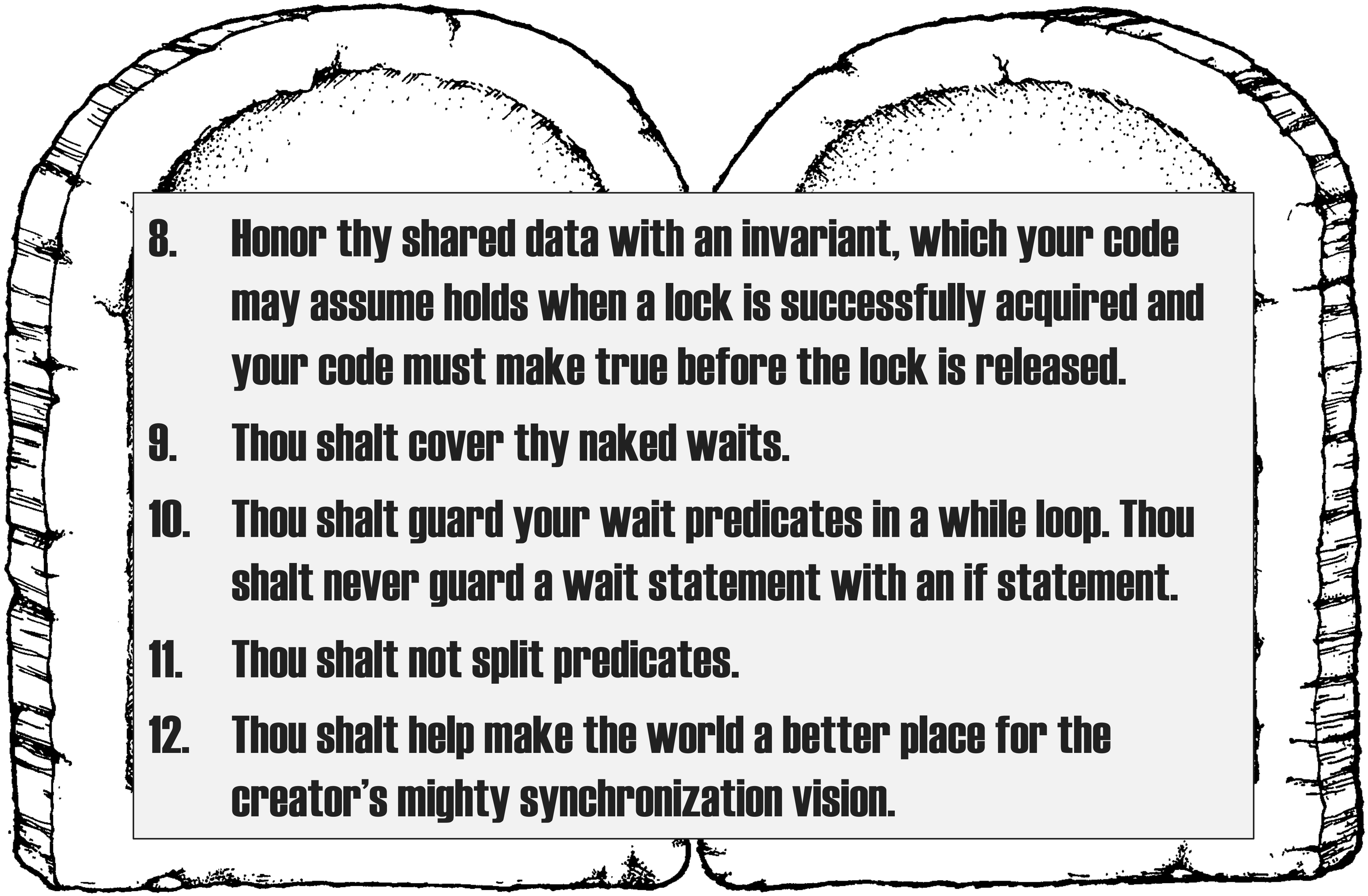
- This is a good thing!

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.

#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”