CPS 310
Synchronization: Digging Deeper

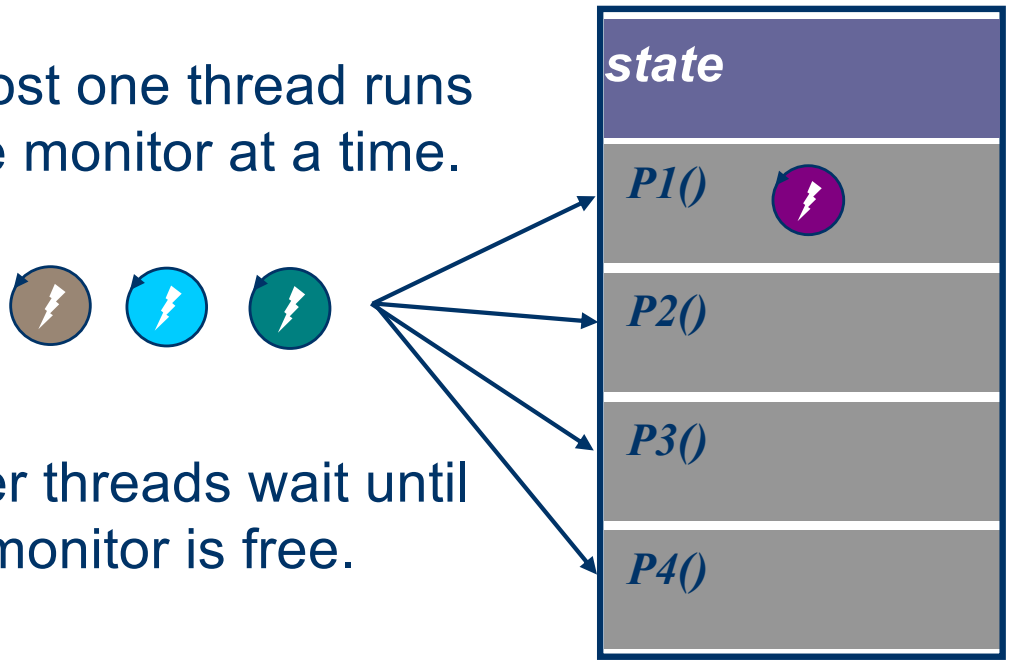
Jeff Chase

Duke University

Roots: monitors

A monitor is a module in which execution is serialized.
A module is a set of procedures with some private state.

At most one thread runs in the monitor at a time.



Other threads wait until the monitor is free.

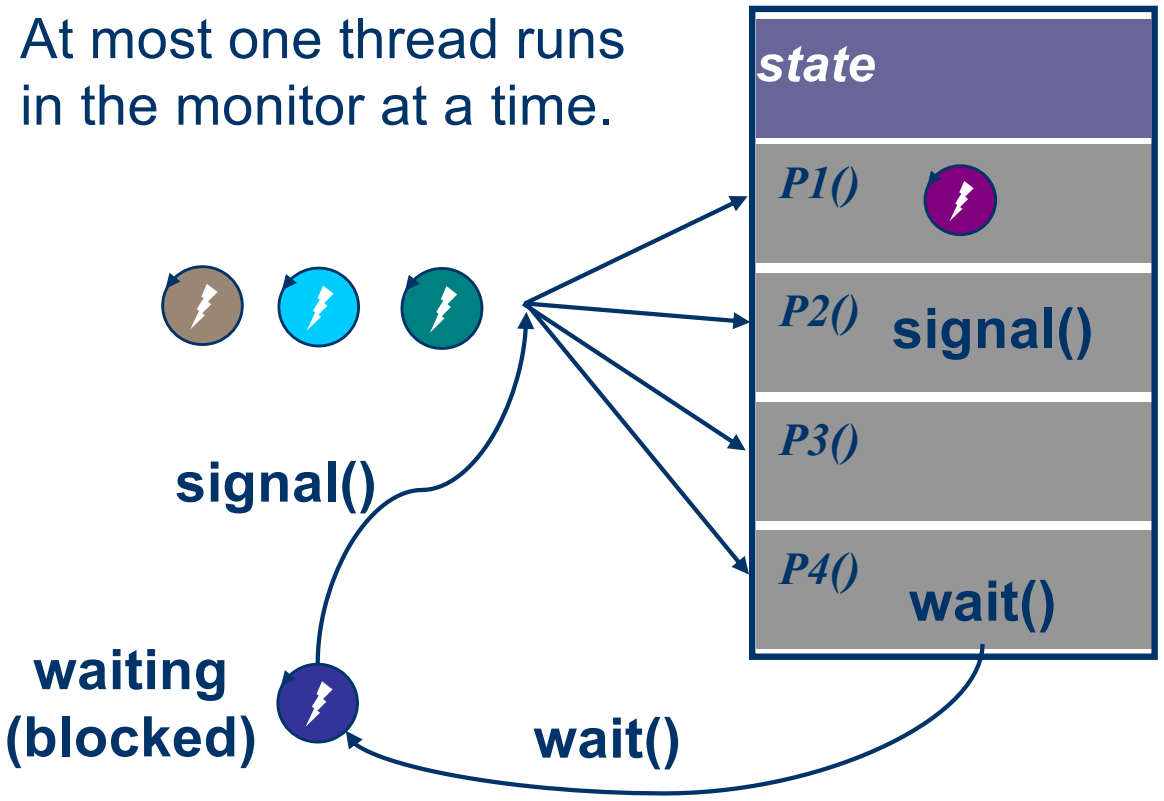
[Brinch Hansen 1973]
[C.A.R. Hoare 1974]

Java “monitors”? **synchronized** allows finer control over the entry/exit points. Also, each Java object is its own “module”: objects of a class share methods of the class but have private state and a private monitor.

Monitor wait/signal

We need a way for a thread to wait for some condition to become true, e.g., until another thread runs and/or changes the state somehow.

At most one thread runs in the monitor at a time.



A thread may wait (**sleep**) in the monitor, exiting the monitor.

A thread may signal in the monitor.

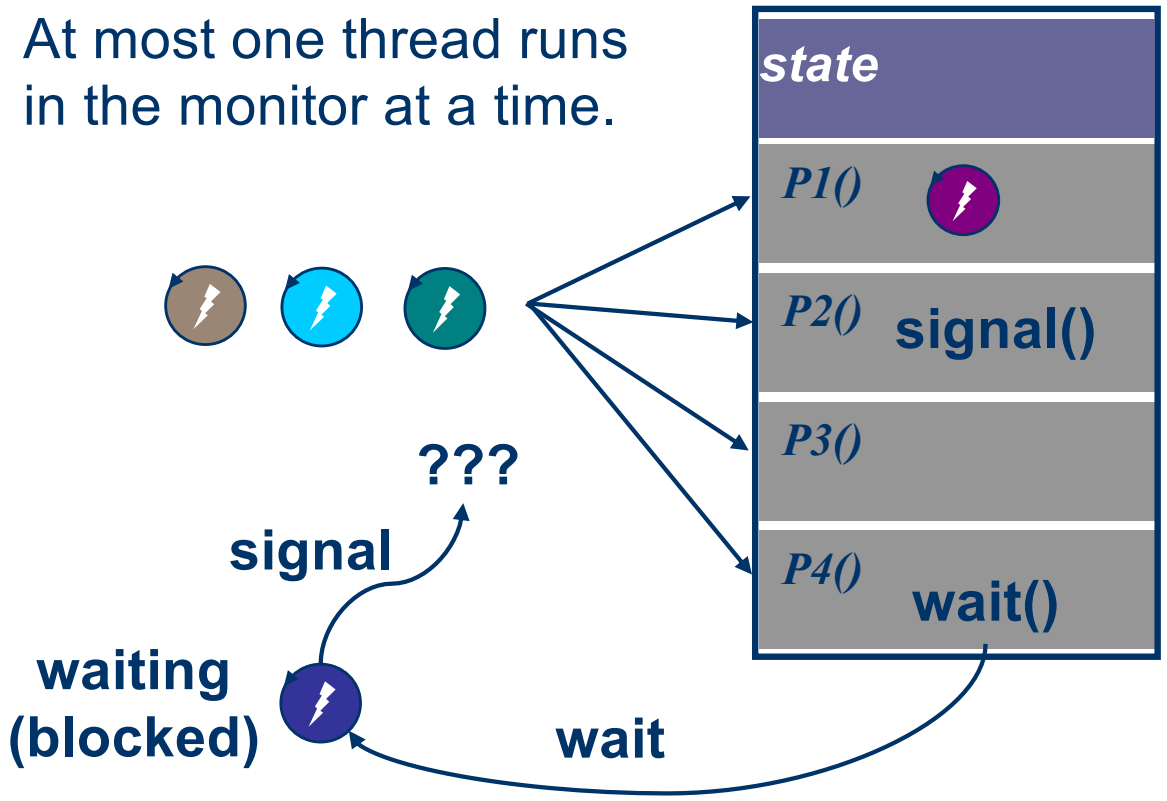
Signal means: wake one waiting thread, if there is one, else do nothing.

The awakened thread returns from its **wait** and reenters the monitor.

Monitor wait/signal

Design question: when a waiting thread is awakened by **signal**, must it start running immediately? Back in the monitor, where it called **wait**?

At most one thread runs in the monitor at a time.



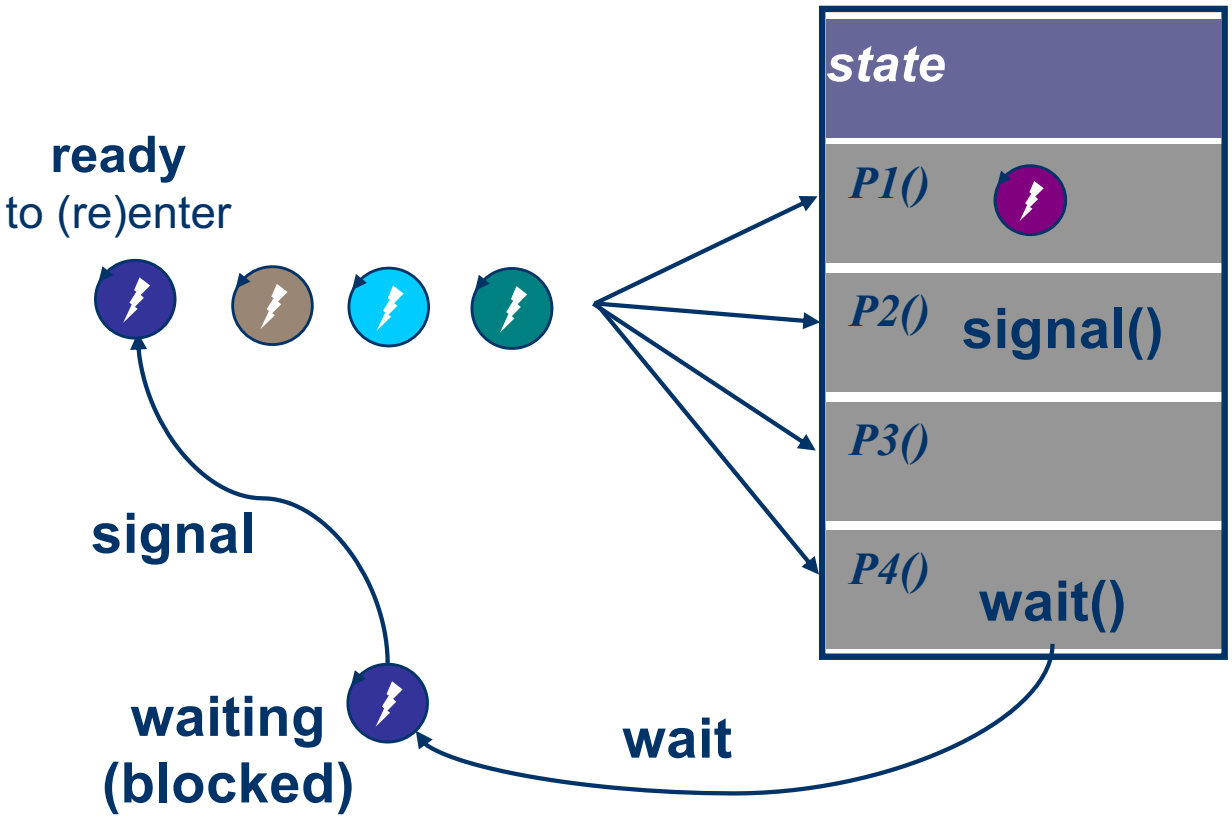
Two choices: **yes** or **no**.

If **yes**, what happens to the thread that called **signal** within the monitor? Does it just hang there? They can't both be in the monitor.

If **no**, can't other threads get into the monitor first and change the state, causing the condition to become false again?

Mesa semantics

Design question: when a waiting thread is awakened by **signal**, must it start running immediately? Back in the monitor, where it called **wait**?



Mesa semantics: **no.**

An awakened waiter gets back in line. The **signal** caller keeps the monitor.

So, can't other threads get into the monitor first and change the state, causing the condition to become false again?

Yes. So the waiter must recheck the condition: **"Loop before you leap".**

Experience with Processes and Monitors in Mesa[1]

Butler W. Lampson
Xerox Palo Alto Research Center

David D. Redell
Xerox Business Systems

Abstract

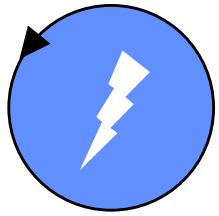
The use of monitors for describing concurrency has been much discussed in the literature. When monitors are used in real systems of any size, however, a number of problems arise which have not been adequately dealt with: the semantics of nested monitor calls; the various ways of defining the meaning of WAIT; priority scheduling; handling of timeouts, aborts and other exceptional conditions; interactions with process creation and destruction; monitoring large numbers of small objects. These problems are addressed by the facilities described here for concurrent programming in Mesa. Experience with several substantial applications gives us some confidence in the validity of our solutions.

1980

Thread project APIs

Threads

```
thread_create(func, arg);
thread_yield();
```



Mesa monitors

Locks/Mutexes

```
thread_lock(lockID);
thread_unlock(lockID);
```

Condition Variables

```
thread_wait(lockID, cvID);
thread_signal(lockID, cvID);
thread_broadcast(lockID, cvID);
```

All functions return an error code: 0 is success, else -1.

Terminology and syntax

- The abstractions for concurrency control in this class are now sort-of universal at the OS/API level.
- Monitors (mutex+CV) with Mesa semantics appear in:
  - Java (e.g., on Android) and JVM languages (e.g., Scala)
  - POSIX threads or Pthreads (used on Linux and MacOS/iOS)
  - Windows, C#/.NET, and other Microsoft systems
- Terminology and APIs vary a bit.
  - mutex == lock == Java “synchronized”
  - monitor == mutex + optional condition variable(s) or CV(s)
  - signal() == notify(), broadcast() == notifyAll()
  - Windows/C# calls notify “pulse”
- The slides use interchangeable terms interchangeably.

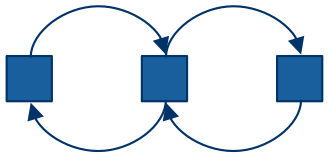
Using locks/mutexes: invariants

T

Get from list

```
lock(mx);
element = list.pop();
unlock(mx);
```

Shared list

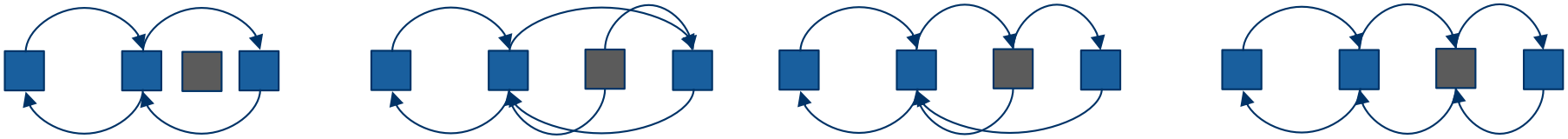


T2

Put to list

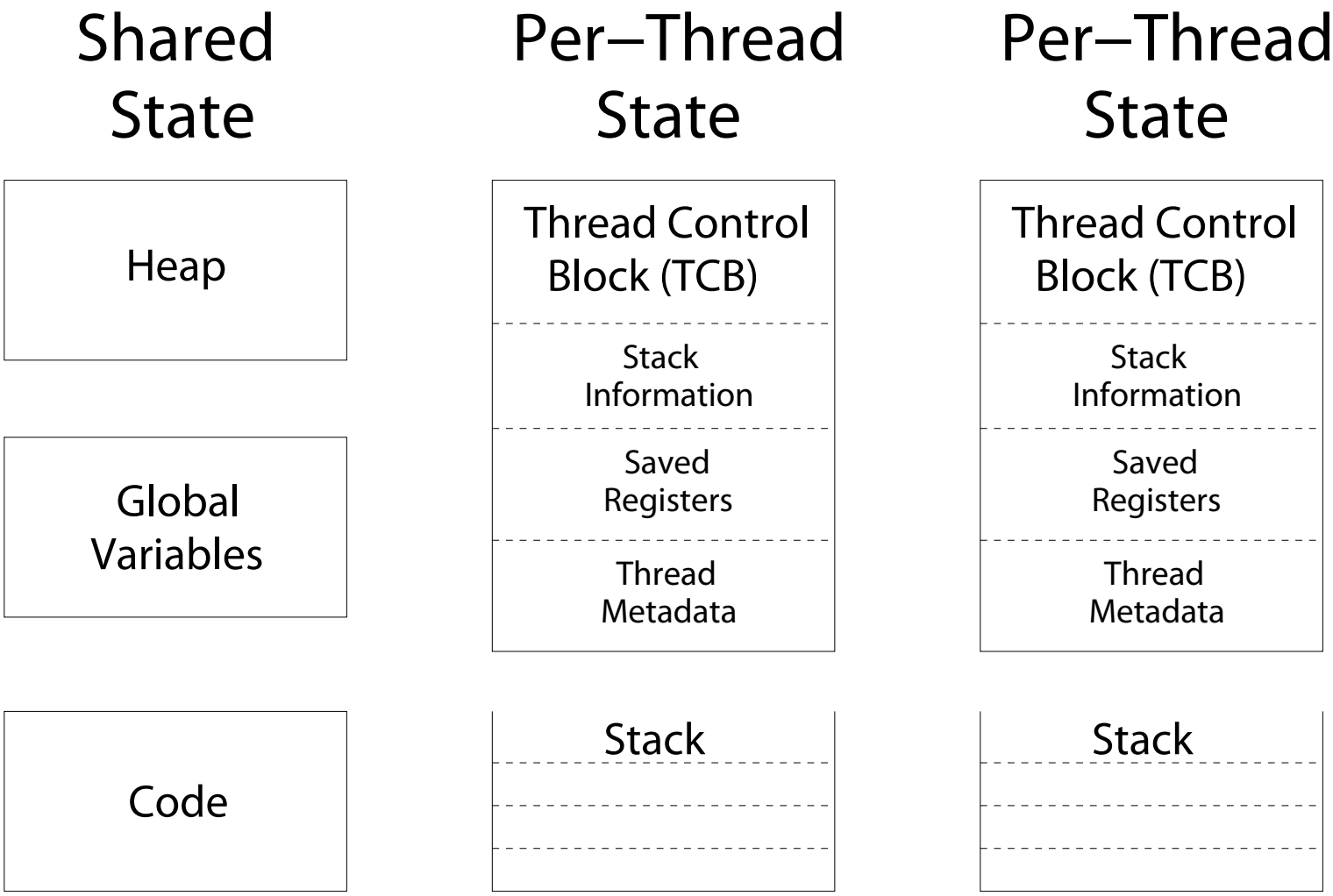
```
lock(mx);
list.push(element);
unlock(mx);
```

- Every data structure has **invariant conditions**: assumptions that must be true for the code to work.
  - E.g., e.back.next == e for a doubly linked list.
- Code methods temporarily violate the invariants, then restore them before completing. E.g, insertion to a list:

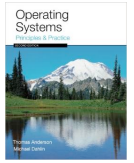


Lock it down! Until invariants are restored.

Shared vs. Per-Thread State



Lock it down! When accessing **shared** state.

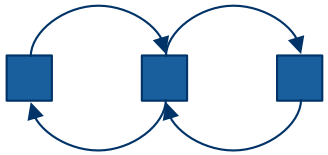


The missed wakeup problem

T

Get from list

```
lock(mx);
while (empty(list))
        unlock; sleep(); lock;
element = list.pop();
unlock(mx);
```



Pseudocode OK.
But this won't work!

T2

Put to list

```
lock(mx);
list.push(element);
if (…)
        T.wakeup();
unlock(mx);
```

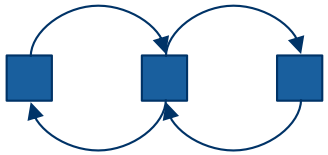
- We need sleep/wakeup primitives that are **integrated with locking**. Why? Sleep is inherently racy!
- If T sleeps due to state of shared structure (e.g., empty list), it **must** unlock first, else T2 can't get in to "fix it".
- But if T unlocks, then T2 could finish before T even gets to sleep! T can **miss the wakeup** and sleep "forever".
- This is called the **missed wakeup** or **wake up waiter** problem.

Monitors fix the missed wakeup problem

T

Get from list

```
lock(mx);
while (empty(list))
      wait(mx, cv);
element = list.pop();
unlock(mx);
```



Monitor wait() unlocks for duration of the wait, **atomically**. It's magic!

T2

Put to list

```
lock(mx);
list.push(element);
signal(cv);
unlock(mx);
```

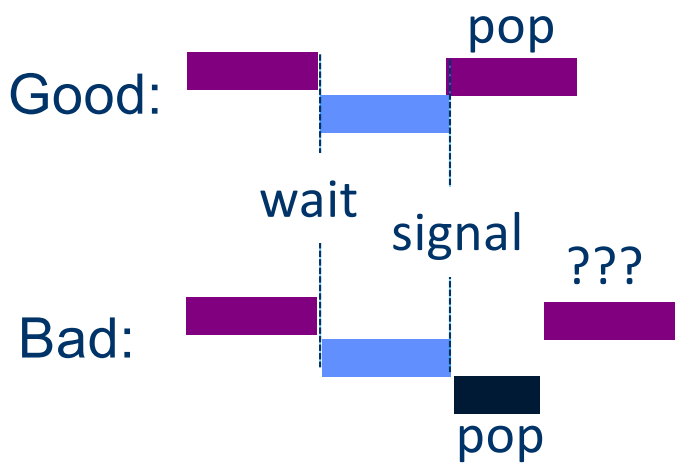
- Monitors are designed to solve the missed wakeup problem.
- The integrated lock protects the data structure and condition covered by the CV.
- Another synchronization construct called **semaphores** also fixes the problem in a different way.
- More on all this later.

Monitors make no promises

T

Get from list

```
lock(mx);
if (empty(list))
      wait(mx, cv);
element = list.pop();
unlock(mx);
```



T2

Put to list

```
lock(mx);
list.push(element);
signal(cv);
unlock(mx);
```

- The monitor **contract** is **nondeterministic**.
- User cannot know which waiter a mutex or CV chooses next.
- The monitor implementation is free to choose. E.g., the thread lab is All FIFO All the Time.
- But a user/programmer should never assume any particular choice.
- **Loop before you leap!**

The 12 Commandments of Synchronization

Emin Gün Sirer
Computer Science Department, Cornell University

October 4, 2011

Abstract

In the beginning, there was hardware. Now the hardware was formless and empty, darkness was over the surface of silicon.

And then the creator said "let there be operating systems," and there were OSes. The creator saw that OSes were good.

And the creator said "let there be processes, and threads." OSes were teeming

Commandment 9. *Thou shalt cover thy naked waits.*

Commandment 10. *Thou shalt guard your wait predicates in a while loop. Thou shalt never guard a wait statement with an if statement.*

The spurious wakeup problem

T

Get from list

```
lock(mx);
while (empty(list) ||
       list.head.type != what_I_want)
       wait(mx, cv);
element = list.pop();
unlock(mx);
```

T2

Put to list

```
element.type = not_what_T_wants;
lock(mx);
list.push(element);
signal(cv);
unlock(mx);
```

- Suppose elements on the list are typed.
- T and some T1 (not shown) wait for different types.
- T2 pushes an element of T1's type, but T gets the signal (spurious).
- T ignores element and goes back to wait, and T1 waits "forever".
- Spurious wakeups occur when a CV is used for multiple conditions.
- **How to ensure that the "right" thread wakes up?**

Wake ‘em all, let them sort it out

T

Get from list

```
lock(mx);
while (empty(list) ||
       list.head.type != what_I_want)
       wait(mx, cv);
element = list.pop();
unlock(mx);
```

T2

Put to list

```
element.type = not_what_T_wants;
lock(mx);
list.push(element);
broadcast(cv);
unlock(mx);
```

- If we use **broadcast** (notifyAll or pulseAll), then both T and T1 wake.
- T goes back to wait, and T1 gets its element.
- If we loop before leaping, broadcast is **always** safe!
- In this example, it is signal that is unsafe! Broadcast works.
- But still spurious wakeups, which have a cost: **thundering herd**.
- “Signal is just a performance hint.” [Hauser et. al.]

Thundering herd

Thundering herd: an event occurs (e.g., a CV broadcast) and many threads wake up as a result, where only one of them can actually consume/handle the event.

The others go back to sleep, and their work is wasted.

Thus the system is briefly under heavy load while a "herd of threads thunders through", like a herd of buffalo stampeding.

Image: http://www.artbarbarians.com/gallery2/images/13/thunder-and-dust-american-bison-buffalo-fine-art-print-daniel-smith-large11318193.jpg

Producer/consumer bounded buffer

Let's apply what we know to an example problem.

- Manage flow of items from producers to consumers.
- Use a "buffer" to store the items in transit.
- The buffer decouples producer and consumer so they can be fully asynchronous: avoids forcing a **rendezvous**.
- But if buffer is empty, consumer must wait for item(s).
- And any practical buffer is **bounded**: it can max out.
- If the buffer is full, producer must drop item(s) or wait.
- Producer/consumer bounded buffers occur **everywhere** in systems: I/O, pipes, networking, services.
- And in a **soda machine.** Slides from Landon Cox.

Example: the soda/HFCS machine

Delivery person
(producer)

Vending machine
(buffer)

Soda drinker
(consumer)

Producer-consumer code

```
consumer () {

   take a soda from machine

}
```

```
producer () {

   add one soda to machine

}
```

Solving producer-consumer

1. **What are the variables/shared state?**
   - Soda machine buffer
   - Number of sodas in machine (≤ MaxSodas)
2. **Locks?**
   - One, to protect all shared state (sodaLock)
3. **Mutual exclusion?**
   - Only one thread can manipulate machine at a time
4. **Ordering constraints?**
   - Consumer must wait if machine is empty (CV hasSoda)
   - Producer must wait if machine is full (CV hasRoom)

Producer-consumer code

```
consumer () {
  lock


  take a soda from machine


  unlock
}
```

```
producer () {
  lock


  add one soda to machine


  unlock
}
```

Producer-consumer code

```
consumer () {
   lock
   wait if empty

   take a soda from machine

   notify (not full)

   unlock
}
```

```
producer () {
   lock
   wait if full

   add one soda to machine

   notify (not empty)

   unlock
}
```

Producer-consumer code

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasSoda)
  }          Mx       CV1

  take a soda from machine

  signal (hasRoom)
             CV2
  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock, hasRoom)
  }             Mx        CV2

  add one soda to machine

  signal (hasSoda)
             CV1
  unlock (sodaLock)
}
```

Producer-consumer code

```
consumer () {
   lock (sodaLock)

   while (numSodas == 0) {
     wait (sodaLock,hasSoda)
   }

   take a soda from machine

   signal(hasRoom)

   unlock (sodaLock)
}
```

```
producer () {
   lock (sodaLock)

   while(numSodas==MaxSodas){
     wait (sodaLock, hasRoom)
   }

   fill machine with soda

   broadcast(hasSoda)

   unlock (sodaLock)
}
```

The signal should be a broadcast if the producer can produce more than one resource, and there are multiple consumers.

Variations: one CV?

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasRorS)
  }          Mx       CV

  take a soda from machine

  signal (hasRorS)
            CV

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock,hasRorS)
  }            Mx       CV

  add one soda to machine

  signal(hasRorS)
            CV

  unlock (sodaLock)
}
```

Two producers, two consumers: who consumes a signal?

ProducerA and ConsumerB wait while ConsumerC signals?

Variations: one CV?

```
consumer () {
   lock (sodaLock)

   while (numSodas == 0) {
     wait (sodaLock,hasRorS)
   }

   take a soda from machine

   signal (hasRorS)

   unlock (sodaLock)
}
```

```
producer () {
   lock (sodaLock)

   while(numSodas==MaxSodas){
     wait (sodaLock,hasRorS)
   }

   add one soda to machine

   signal (hasRorS)

   unlock (sodaLock)
}
```

Is it possible to have a producer and consumer both waiting? Suppose MaxSodas==1
cA, cB wait, pC adds/signals/leaves, pD waits, cA wakes/signals/leaves, cB wakes (!), waits
Now: pC, cA gone, cB, pD both waiting!

Variations: one CV?

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasRorS)
  }

  take a soda from machine

  broadcast (hasRorS)

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock,hasRorS)
  }

  add one soda to machine

  broadcast (hasRorS)

  unlock (sodaLock)
}
```

Use broadcast instead of signal: safe but slow.

Notes from soda machine

- Producer/consumer illustrates using multiple CVs to reduce **spurious wakeups**: can be more selective/targeted with wait/notify.
- If we use one CV for multiple conditions, a signal/notify might wake up the “wrong” thread (“spurious”), leaving a thread blocked forever.
- Broadcast gets it done: “wake ‘em all and let them sort it out”.
- “**Loop before you leap!**” makes broadcast safe to “sort it out”.
- But broadcast may be slow: **thundering herd**.
- “Signal is just a performance hint.” Tells system waking just one is good enough: program functions the same but performs better.

Thinking about threads

- Threads don't "know" about one another: they coordinate through shared locks and CVs.
- Threads don't have agency or intention; they don't "decide" when to lock, signal, or wait; they just execute the program.
- Thread calls on locks and CVs are conventions in the program to protect (synchronize) their accesses to the shared data.
- Concurrency control is about using synchronization in the right ways and in all the right places in the program's code.
- A program is **correctly synchronized** or **data-race-free** iff it excludes all dangerous schedules that cause unintended behavior.
- Its threads "do the right thing" in any remaining **legal** schedule, on any system, on any machine.