Synchronization

CS 4410, Operating Systems

Fall 2016
Cornell University

Rachit Agarwal
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.

Synchronization: Topic 1

Synchronization Motivation & Basics

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

Threads Share Memory

Threads have:

- Private registers

  Context switching saves and restores registers when switching from thread to thread

- Shared “global” memory

  Global means not stack memory

- Usually private stack

  Pointers into stacks across threads frowned upon

Two Theads, One Variable

2 threads updating a single 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 two threads execute concurrently?

Two Theads, One Variable

It won’t actually 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
. . .
```

Memory

amount 100,000

Two Theads, One Variable

It 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 100,000

Everything works:

although different, either order is correct

Two Theads, One Variable

Or it 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	100,000

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” (total order on machine instructions)

All possible interleavings should be safe!

Race Conditions are Hard to Debug

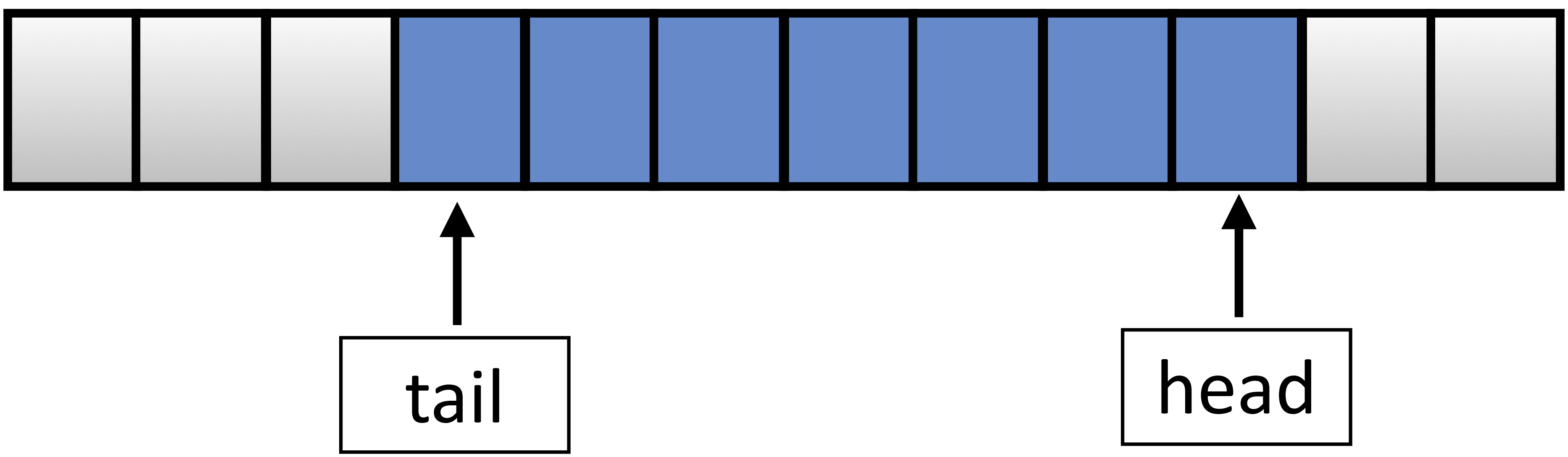
- Number of possible interleavings is huge
- Some interleavings are good
- Some interleavings are bad:
  - But bad interleavings may rarely happen!
  - ***Works 100x ≠ no race condition***
- Timing dependent = small changes can hide bug

Case Study: Therac-25

Example: Races with Queues

2 concurrent enqueue() operations?

2 concurrent dequeue() operations?



What could possibly go wrong?

Critical Section

Code that can be executed by only one thread at a time

T1

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

T2

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

Goals

Safety: 1 thread in a critical section at time

Liveness: all threads eventually make it into CS if desired

Fairness: all have equal chances of getting into CS

… in practice, fairness rarely guaranteed

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:

`outtobuymilk` | 0 |

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

Solution #1: No Protection

T1

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

T2

```
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)

`blueonit`, `greenonit`

T1

```
blueonit = 1
if not greenonit and
       fridge_empty():
    buy_milk()
blueonit = 0
```

T2

```
greenonit = 1
if not blueonit and
       fridge_empty():
    buy_milk()
greenonit = 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)

blueonit, greenonit

T1

```
blueonit = 1
while greenonit:
    do_nothing()
if fridge_empty():
  buy_milk()
blueonit = 0
```

T2

```
greenonit = 1
if not blueonit and
    fridge_empty():
  buy_milk()
greenonit = 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, green}`

T1

```
blueonit = 1
turn = green
while (greenonit and
   turn==green):
     do_nothing()
if fridge_empty():
   buy_milk()
blueonit = 0
```

T2

```
greenonit = 1
turn = blue
while (blueonit and
   turn==blue):
     do_nothing()
if fridge_empty():
   buy_milk()
greenonit = 0
```

Safe? Live? Fair?

– complicated (and this is a simple example!)
– hard to ascertain that it is correct
– hard to generalize, inefficient

Hardware Solution

- Hardware primitives to provide mutual exclusion
- Typically relies on a multi-cycle bus operation that atomically reads and updates a memory location
- Example Spec of Test-And-Set:

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

sets the value to 1, returns former value

Spinlocks

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

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

Buying Milk with Spinlock

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

T1

```
SL_acquire(&buyingmilk)
if fridge_empty():
  buy_milk()
SL_release(&buyingmilk)
```

T2

```
SL_acquire(&buyingmilk)
if fridge_empty():
  buy_milk()
SL_release(&buyingmilk)
```

SpinLock Issues

Participants not in critical section must **spin**

→ **wasting CPU cycles**

- Replace the “do nothing” loop with a “yield()”?
  Processes would still be scheduled and descheduled

Need better primitive:

- allows one process to pass through
- all others to sleep until they can be executed again

Synchronization: Topic 2

Semaphores

- Definition
- Binary Semaphores
- Counting Semaphores
- Implementing Semaphores
- Classic Synchronization Problems (w/Semaphores)
    - Producer-Consumer (w/ a bounded buffer)
    - Readers/Writers Problems
- Classic Semaphore Mistakes
- Semaphores Considered Harmful

What is a Semaphore?

[Dijkstra 1962]

Non-negative integer w/atomic increment, decrement

```
S = new Semaphore(init)   // must initialize!
```

Can only be modified by:

- P(S): decrement or block if already 0
- V(S): increment & wake up any waiting threads
- No interface to read the value

Operations have the following semantics:

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

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

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

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

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

T2

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

Counting Semaphores

Sema count can be any integer

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

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

T1

```
PacketProcessor():
x = get_packet_from_card()
enqueue(packetq, x);
V(packetarrived);
```

T2

```
NetworkingThread():
P(packetarrived);
x = dequeue(packetq);
print_contents(x);
```

Starter Semaphore implementation

Assume a thread "stuck" in P() will be eventually interrupted

```
P(Sema *s) {
    while(TRUE) {
        if s.count big enough
            break;
    }
   decrement s.count
}

V(Sema *s) {
 increment s.count
}
```

```
struct Semaphore
{
  int count;
}
```

Do we like this?

1. Shared state is not protected

   Threads can be interrupted at any point? → need a lock

2. **Busy-waiting is bad!**

   Count not big enough? → make note of thread's interest and stop it. (have V start it again when count big enough)

Producer-Consumer Problem

2+ threads communicate:

some threads **produce** data that others **consume**

Bounded buffer: size **N**

Producer process writes data to buffer

- Writes to **`in`** and moves rightwards
- Don't write more than **N**!

Consumer process reads data from buffer

- Reads from **`out`** and moves rightwards
- Don't consume if there is no data!

Example: "pipe" (|) in Unix `> cat file | sort | uniq | more`

Solution #1: 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!

Solution #2: Add Mutex Semaphores

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

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

}
```

now atomic

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

Solution #3: Add Communication Semaphores

```
Shared:
int buf[N];
int in, out;
Semaphore mutex_prod(1), mutex_cons(1);
Semaphore nRoom(N), NData(0);
```

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

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

Readers-Writers Problem

[Courtois+ 1971]

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

Want to allow:

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

Example: making an airline reservation

- When you browse to look at flight schedules the web site acts as a **reader** on your behalf
- When you reserve a seat, web site has to write into database to make the reservation

Readers-Writers Constraints

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

Questions:

1. Writer active & combo of readers/writers arrive.
   Who should get in next?
2. Writer 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() {

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

}
```

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

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 @ P(rw_lock)

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…)

Classic Semaphore Mistakes

I

```
P(S)
CS
P(S)  ⬅typo
```

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

J

```
V(S)  ⬅typo
CS
V(S)
```

Undermines mutex:

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

K

```
P(S)
CS  ⬅omission
```

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!

L

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

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 Monitors

Synchronization: Topic 3

Monitors & Condition Variables

- Definition
- Semantics
- Simple Monitor Example
- vs. Semaphores
- Classic Synchronization Problems (w/Monitors)
  - Bounded Buffer Producer-Consumer
  - Readers/Writers Problems
- Classic Synchronization Mistakes (w/Monitors)

What is a Monitor?

[Hoare 1974]

Abstract Data Type for handling shared resources, comprising:

1. Shared Private Data
    - the resource
    - cannot be accessed from outside
2. Procedures that operate on the data
    - gateway to the resource
    - can only act on data local to the monitor
3. Synchronization primitives
    - among threads that access the procedures

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 stack
{
    int top;

    void push(any_t *) {
    }

    any_t *pop() {
    }

    initialization_code() {
    }
}
```

only one operation can execute at a time

Monitors can define ***Condition Variables***

A mechanism to wait for events

3 operations on Condition Variable `Condition x;`

- `x.wait()`: atomically release monitor lock and relinquish processor, sleep until woken up (or you 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)

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

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

Using Condition Variables

To wait for some condition:

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

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

When the condition becomes satisfied:

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

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

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

Kid and Cook Threads

not always this explicit

```
Monitor BurgerKing {
  Lock mlock;

  int numburgers = 0
  condition hungrykid;


  void kid_eat() {
     mlock.acquire()
     while (numburgers==0)
        hungrykid.wait()
     numburgers -= 1
     mlock.release()
  }

  void makeburger() {
     mlock.acquire()
     ++numburger;
     hungrykid.signal();
     mlock.release()
  }
}
```

```
kid_main() {

   dig_in_mud();
   BK.kid_eat();
   bathe();
   draw_on_walls();
   BK.kid_eat();
   facetime_Karthik();
   facetime_oma();
   BK.kid_eat();

}
```

```
cook_main() {

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

}
```

Ready

Running

Synchronization: The Big Picture

Concurrent Applications

Shared Objects

Bounded Buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt Disable Test-and-Set

Hardware

Multiple Processors Hardware Interrupts

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

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;
  void 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;
  }
}
```

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

Readers and Writers

[slide 31]

```
Monitor ReadersNWriters

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


 void BeginWrite()
  assert(nReaders==0 or nWriters==0)
  ++waitingWriters
  while (nWriters >0 or nReaders >0)
     canWrite.wait();
  --waitingWriters
  nWriters = 1;

 void BeginRead()
    assert(nReaders==0 or nWriters==0)
    ++waitingReaders
    while (nWriters>0 or waitingWriters>0)
       canRead.wait();
    --waitingReaders
    ++nReaders


 void EndWrite()
  assert(nWriters==1 and nReaders==0)
  nWriters = 0
  if WaitingWriters > 0
     canWrite.signal();
  else if waitingReaders > 0
    canRead.broadcast();

 void EndRead()
    assert(nReaders>0 and nWriters==0)
    --nReaders;
    if (nReaders==0 and waitingWriters>0)
       canWrite.signal();
```

Understanding the Solution

A writer can enter if:

- no other active writer
  and
- no waiting readers

A reader can enter if:

- no active writer
  and
- no waiting writers

When a writer finishes:

checks to see if readers waiting:

Y ➙ lets all enter

N ➙ if writer waiting, lets 1 enter

When last reader finishes:

- it lets 1 writer in (if any)

Fair?

Wants 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

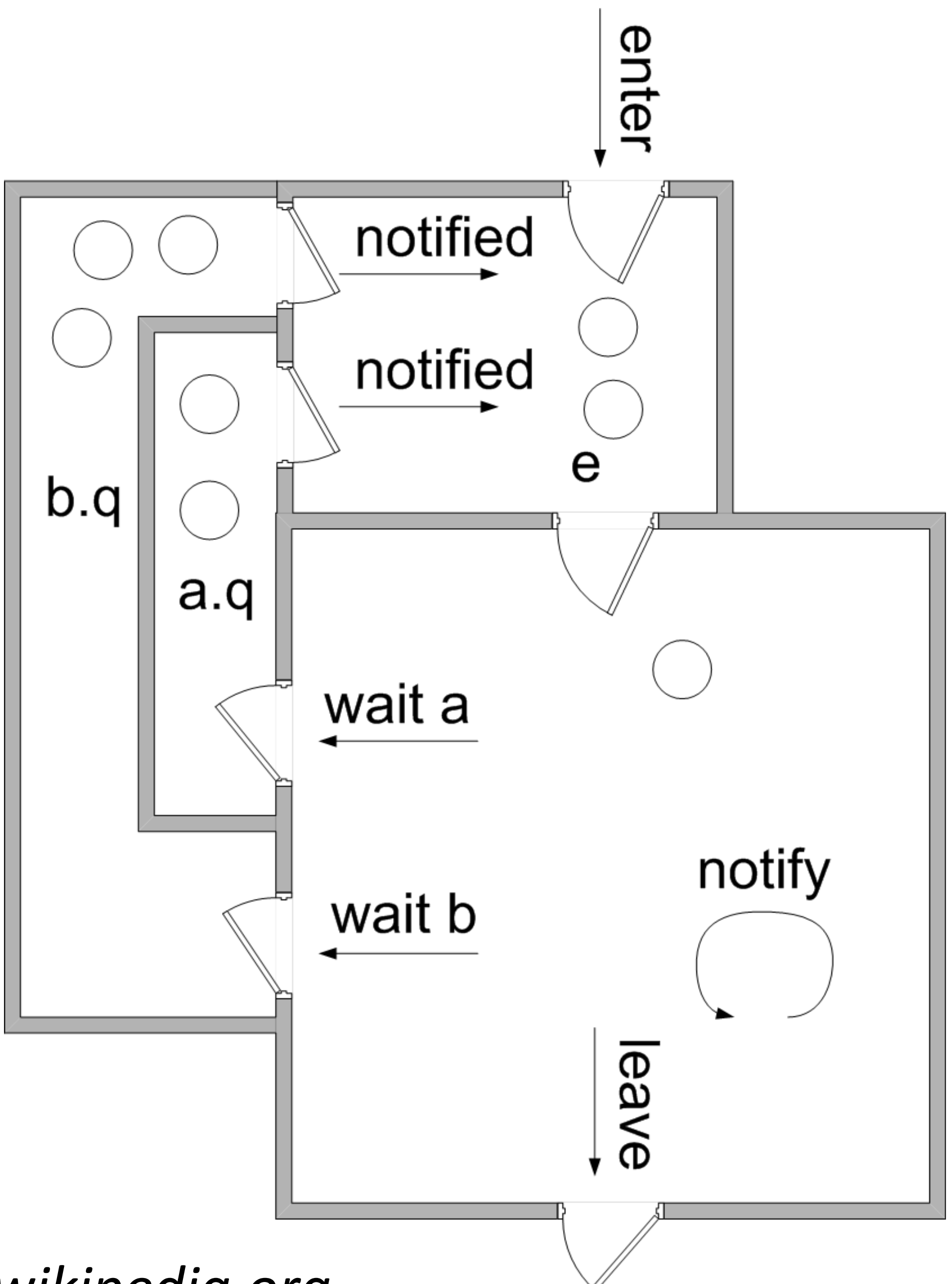
Hoare vs. Mesa Semantics for Monitors

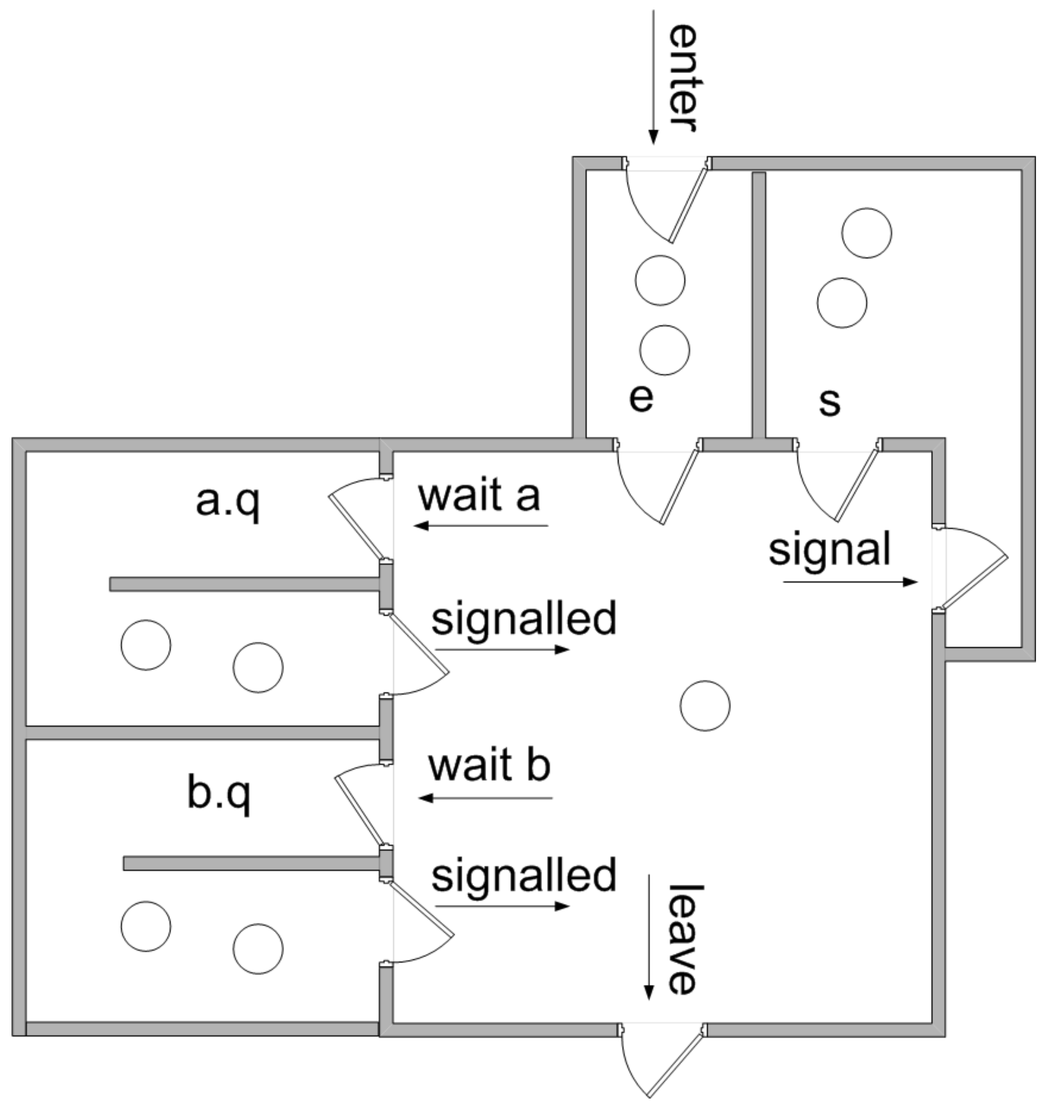
Hoare Semantics: monitor lock is transferred directly from the signaling thread to the newly woken up thread

- not usually desirable to force signaling thread to relinquish the monitor lock immediately to a woken up thread
- confounds scheduling with synchronization, penalizes threads

Mesa 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? Which is Hoare?





wikipedia.org

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()
```

Synchronization: Final Topics

Mapping to Real Languages

Barrier Synchronization

Another Classic: Barbershop Problem

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 10-P/P2/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”

Barbershop Problem

One possible version:

- A barbershop holds up to k clients
- N barbers work on clients
- M clients total want their hair cut
- Each client will have their hair cut by the first barber available

Implementing the Barbershop

(1) Identify the waits

Customers?

Barbers?

(2) Create condition variables for each

(3) Create counters to trigger the waiting

(4) Create signals for the waits

Rules to Live By

- Use consistent structure
- When possible, use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()

Go read the 12 Commandments of Synchronization

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”