Synchronization

Prof. Sirer and Van Renesse
CS 4410
Cornell University

Threads share global memory

- When a process contains multiple threads, they have
  - Private registers and stack memory (the *context switching* mechanism saves and restores registers when switching from thread to thread)
  - Shared access to the remainder of the process "state"

Two threads, one variable

- Two threads updating a single shared variable
  - One thread wants to decrement amount by $10K
  - The other thread wants to decrement amount by 50%

```
amount= 100000;

...
amount= amount - 10000;
...

...
amount = 0.50 * amount;
...
```

- What happens when two threads execute concurrently?

Two threads, one variable

amount= 100000;

```
...
r1 = load from amount
r1 = r1 - 10000;
store r1 to amount
...
```

```
...
r1 = load from amount
r1 = 0.5 * r1
store r1 to amount
...
```

amount= ?

Two threads, one variable

amount= 100000;

...
r1 = load from amount
r1 = 0.5 * r1
store r1 to amount
...

...
r1 = load from amount
r1 = r1 - 10000;
store r1 to amount
...

amount= ?

Two threads, one variable

amount= 100000;

```
...
r1 = load from amount
r1 = r1 - 10000;
store r1 to amount
...
```

```
...
r1 = load from amount
r1 = 0.5 * r1
store r1 to amount
...
```

amount= ?

Shared counters

- One possible result: everything works!
- Another possible result: lost update!
  - ⇒ Difficult to debug

- Called a “race condition”

Race conditions

- Def: *a timing dependent error involving shared state*
  - Whether it happens depends on how threads scheduled
  - In effect, once thread A starts doing something, it needs to "race" to finish it because if thread B looks at the shared memory region before A is done, A's change will be lost.
- Hard to detect:
  - All possible schedules have to be safe
    - Number of possible schedule permutations is huge
    - Some bad schedules? Some that will work sometimes?
  - they are intermittent
    - Timing dependent = small changes can hide bug

Scheduler assumptions

Process a:

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

Process b:

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

If i is shared, and initialized to 0

- Who wins?
- Is it guaranteed that someone wins?
- What if both threads run on identical speed CPU
  - executing in parallel

Critical Section Goals

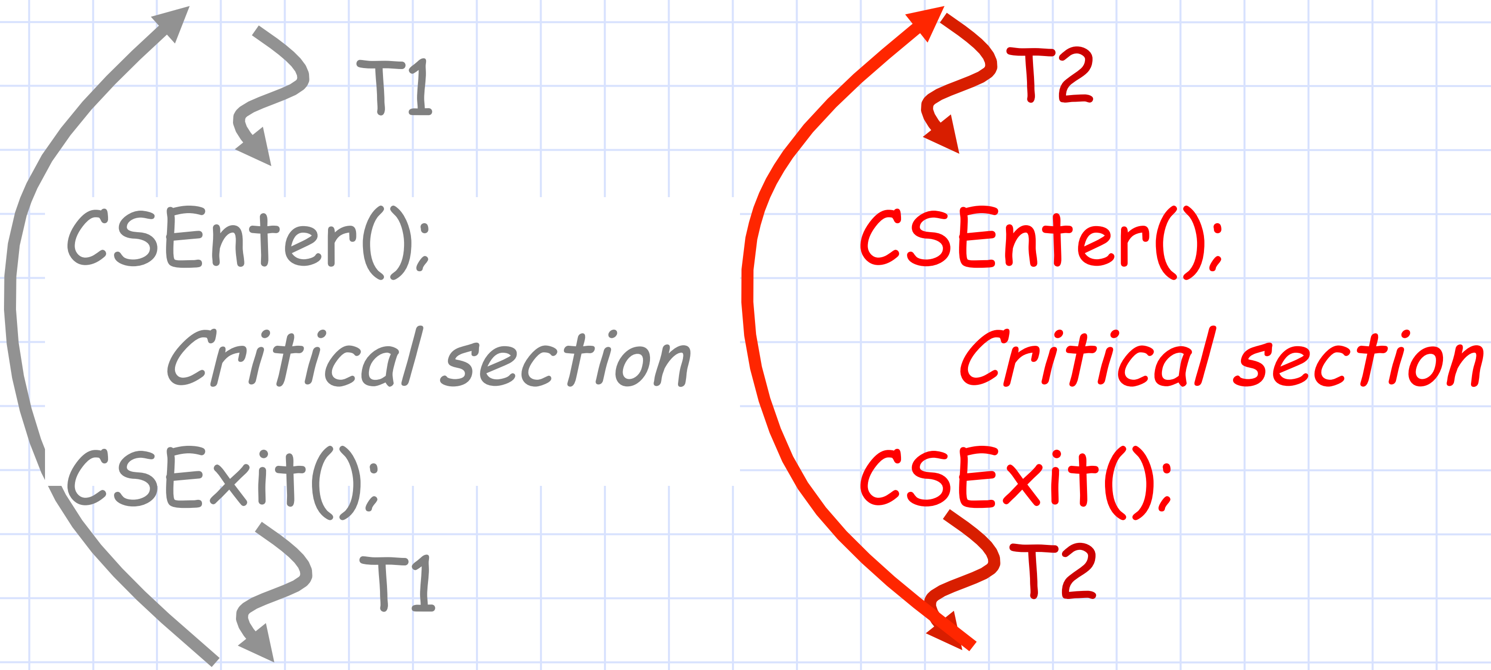
- Threads do some stuff but eventually *might* try to access shared data

time

	T1	T2
	CSEnter();	CSEnter();
	Critical section	*Critical section*
	CSExit();	CSExit();
	T1	T2

Critical Section Goals

- Perhaps they loop (perhaps not!)



Critical Section Goals

- We would like
  - Safety: No more than one thread can be in a critical section at any time.
  - Liveness: A thread that is seeking to enter the critical section will eventually succeed
  - Fairness: If two threads are both trying to enter a critical section, they have equal chances of success
- … in practice, fairness is rarely guaranteed

Too much milk problem

- Two roommates want to ensure that the fridge is always stocked with milk
  - If the fridge is empty, they need to restock it
  - But they don't want to buy too much milk
- Caveats
  - They can only communicate by reading and writing onto a notepad on the fridge
  - Notepad can have different cells, labeled by a string (just like variables)
- Write the pseudo-code to ensure that at most one roommate goes to buy milk

Solving the problem

- A first idea:
  - Have a boolean flag, *out-to-buy-milk*. Initially false.

```
while(outtobuymilk)
    continue;
 if fridge_empty():
   outtobuymilk =true
   buy_milk()
   outtobuymilk = false
```

```
while(outtobuymilk)
    continue;
 if fridge_empty():
   outtobuymilk =true
   buy_milk()
   outtobuymilk = false
```

– Is this Safe? Live? Fair?

Solving the problem

- A second idea:
  - Have a boolean flag, *out-to-buy-milk*. Initially false.

```
outtobuymilk = true
if fridge_empty():
    buy_milk()
outtobuymilk = false
```

```
outtobuymilk =true
if fridge_empty():
    buy_milk()
    outtobuymilk = false
```

– Is this Safe? Live? Fair?

Solving the problem

- A third idea:
  - Have two boolean flags, one for each roommate. Initially false.

```
greenbusy = true
if not redbusy and
    fridge_empty():
    buy_milk()
greenbusy = false
```

```
redbusy = true
if not greenbusy and
    fridge_empty():
    buy_milk()
redbusy = false
```

– Is this Safe? Live? Fair?

Solving the problem

- A final attempt:
  - Have two boolean flags, one for each roommate. Initially false.

```
greenbusy = true
while redbusy:
    do_nothing()
if fridge_empty():
    buy_milk()
greenbusy = false
```

```
redbusy = true
if not greenbusy and
    fridge_empty():
    buy_milk()
redbusy = false
```

– Is this Safe? Live? Fair?

Solving the problem

- A final attempt:
  - Have two boolean flags, one for each roommate. Initially false.

```
greenbusy = true
while redbusy:
    do_nothing()
if fridge_empty():
    buy_milk()
greenbusy = false
```

```
redbusy = true
if not greenbusy and
    fridge_empty():
    buy_milk()
redbusy = false
```

– Really complicated, even for a simple example, hard to ascertain that it is correct

– Asymmetric code, hard to generalize

Solving the problem, really

- The really final attempt:
  - Adding another binary variable: *turn*: { red, blue }

```
greenbusy = true
turn = red
while redbusy and
        turn == red:
    do_nothing()
if fridge_empty():
    buy_milk()
greenbusy = false
```

```
redbusy = true
turn = green
while greenbusy and
        turn == green:
    do_nothing()
if fridge_empty():
    buy_milk()
redbusy = false
```

– Really complicated, even for a simple example, hard to ascertain that it is correct

Solving the problem, really

```
greenbusy = true
turn = red
while redbusy and turn == red:
   do_nothing()
if fridge_empty():
      buy_milk()
greenbusy = false
```

```
redbusy = true
turn = green
while greenbusy and turn == green:
      do_nothing()
if fridge_empty():
      buy_milk()
redbusy = false
```

- Safe:
  - if both in critical section, greenbusy = redbusy = true
  - both found *turn* set favorable to self
  - but *turn* was set to an unfavorable value just before c.s.
- Live: thread never waits more than one turn
- Fair: symmetry

Spinlocks

- Use more powerful hardware primitives to provide a mutual exclusion primitive
- Typically relies on a multi-cycle bus operation that atomically reads and updates a memory location

```
acquire() {
   while(test_and_set(outtobuymilk) == 1)
      /* do nothing */;
}
release() {
   outtobuymilk = 0;
}
```

Spinlocks

0

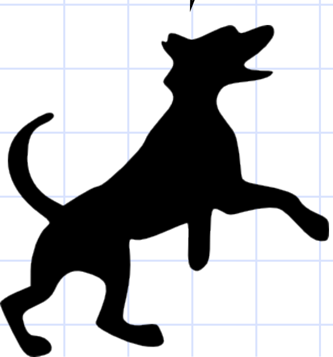
```
acquire(int *lock) {
   while(test_and_set(lock) == 1)
        /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```

Let me in!!!

No, Let *me* in!!!

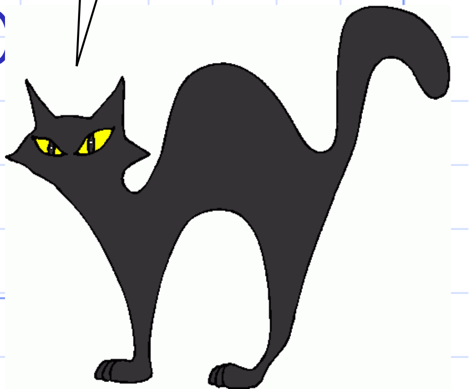
```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

```
acquire(houselock);
Nap_on_couch();
Release(houselock)
```



1

1



Spinlocks

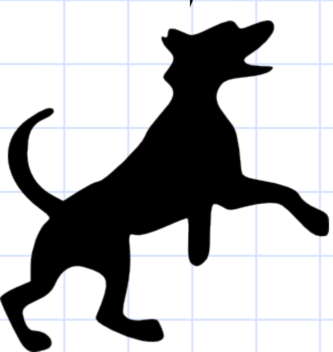
1

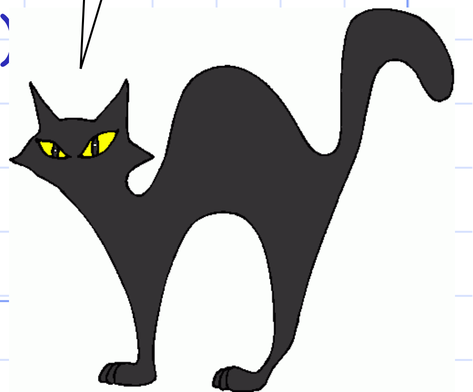
```
acquire(int *lock) {
   while(test_and_set(lock) == 1)
       /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```

```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

```
acquire(houselock);
Nap_on_couch();
Release(houselock)
```

1

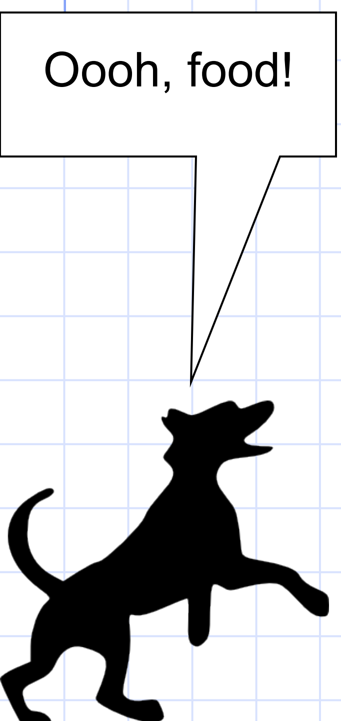




Spinlocks

1

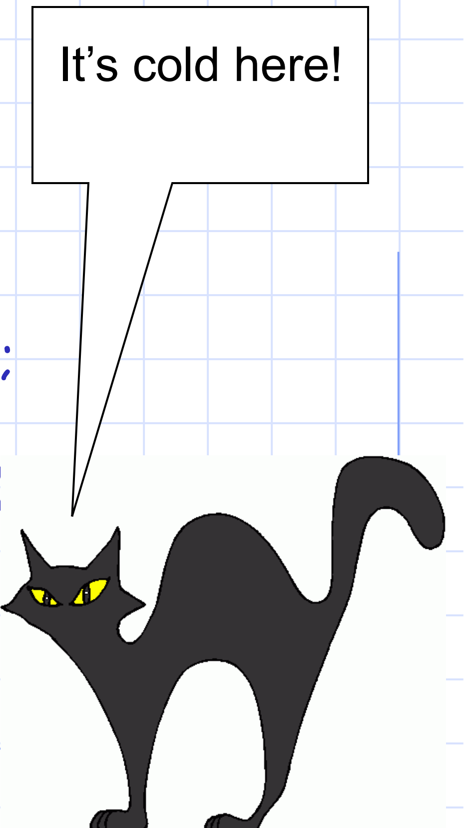
```
acquire(int *lock) {
   while(test_and_set(lock) == 1)
        /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```




```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

```
acquire(houselock);
Nap_on_couch();
Release(houselock);
```

1



Spinlock Issues

- Spinlocks require the participants that are not in the critical section to spin
  - We could replace the “do nothing” loop with a “yield()” call, but the processes would still be scheduled and descheduled
- We need a better primitive that will allow one process to pass through, and all others to go to sleep until they can be executed again

Semaphores

- Non-negative integer with atomic increment and decrement
- Integer 'S' that (besides init) can only be modified by:
  - P(S) or S.wait(): decrement or block if already 0
  - V(S) or S.signal(): increment and wake up process if any
- These operations are atomic, with the following rough semantics

```
P(S) {
    while(S ≤ 0)
        ;
    S--;
}
```

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

- But this implementation would also be terribly inefficient!

Semaphores

- Atomicity of semaphore operations is achieved by including a spinlock in the semaphore

```
Struct Sema {
    int lock;
    int count;
    Queue waitq;
};
P(Sema *s) {
    while(test_and_set(&s->lock) == 1) /* do nothing or yield */;
    if (--s->count < 0) { enqueue on wait list, s->lock = 0; run something else; }
    else { s->lock = 0; }
}
V(Sema *s) {
    while(test_and_set(&s->lock) == 1) /* do nothing or yield */;
    if (++s->count <= 0) { dequeue from wait list, make runnable; }
    s->lock = 0;
}
```

Binary Semaphores

- Semaphore value is limited to 1 or less
  - Used for mutual exclusion (sema as a more efficient mutex)
  - Same thread performs both the P() and the V() on the same semaphore

```
semaphore S
S.init(1);

Process1():                Process2():
    P(S);                      P(S);
    Modifytree();              Modifytree();
    V(S);                      V(S);
```

Semaphores

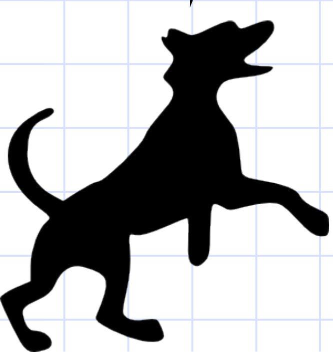
0 1 Queue: empty

```
P(Sema *s) {
  while(test_and_set(&s->lock) == 1)
        /* do nothing */;
  if (--s->count < 0) { enqueue on wait list,
                        s->lock = 0; run something else; }
  else s->lock = 0;
}
```

Let me in!!!

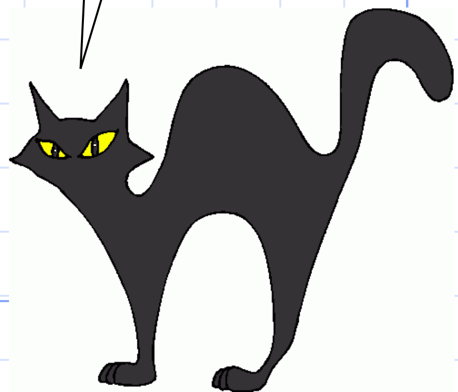
No, Let *me* in!!!

```
P(house);                   P(house);
Jump_on_the_couch();        Nap_on_couch();
V(house);                   V(house);
```



0 0

1



Semaphores

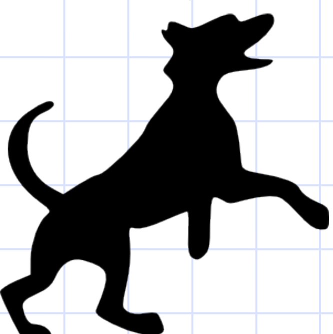
0 0 Queue:

```
P(Sema *s) {
  while(test_and_set(&s->lock) == 1)
        /* do nothing */;
  if (--s->count < 0) { enqueue on wait list,
                        s->lock = 0; run something else; }
  else s->lock = 0;
}
```

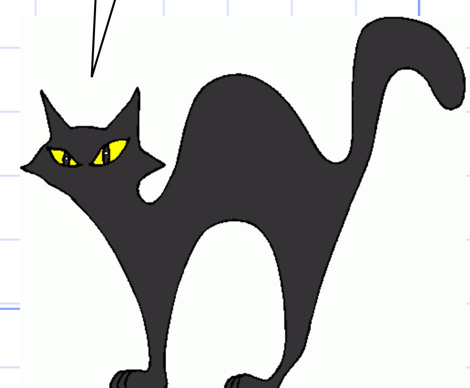
Yay, couch!!!

No, Let *me* in!!!

```
P(house);                      P(house);
Jump_on_the_couch();           Nap_on_couch();
V(house);                      V(house);
```



-1 1



Counting Semaphores

- Sema count can be any integer
  - Used for signaling, or counting resources
  - Typically, one thread performs a P() to wait for an event, another thread performs a V() to alert the waiting thread that an event occurred

```
semaphore packetarrived
packetarrived.init(0);
```

```
PacketProcessor():

  p = retrieve_packet_from_card();
  enqueue(packetq, p);
  V(packetarrived);
```

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

Semaphores

- Semaphore count keeps state and reflects the sequence of past operations
  - A negative count reflects the number of processes on the sema wait queue
  - A positive count reflects number of future P operations that will succeed
- No way to read the count! No way to grab multiple semaphores at the same time! No way to decrement/increment by more than 1!
- All semaphores must be initialized!

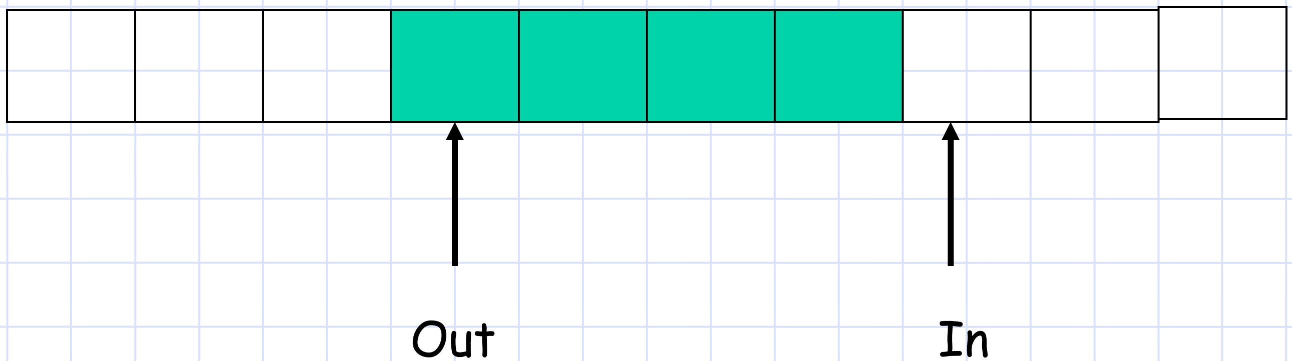
Classical Synchronization Problems

Bounded Buffer

- Bounded buffer:
  - Arises when two or more threads communicate with some threads “producing” data that others “consume”.
  - Example: preprocessor for a compiler “produces” a preprocessed source file that the parser of the compiler “consumes”

Producer-Consumer Problem

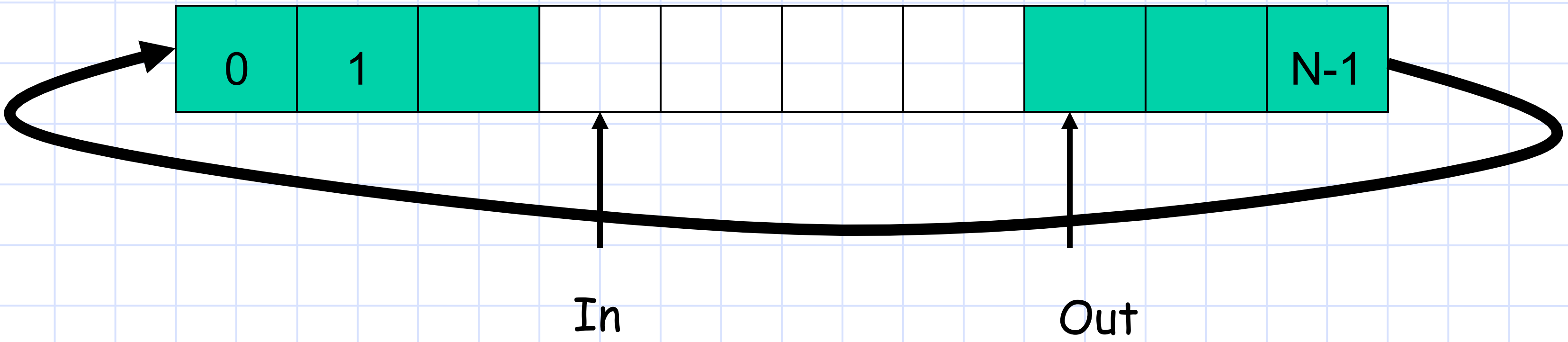
- Start by imagining an unbounded (infinite) buffer
- Producer process writes data to buffer
  - Writes to *In* and moves rightwards
- Consumer process reads data from buffer
  - Reads from *Out* and moves rightwards
  - Should not try to consume if there is no data



Need an infinite buffer

Producer-Consumer Problem

- Bounded buffer: size 'N'
  - Access entry 0… N-1, then "wrap around" to 0 again
- Producer process writes data to buffer
  - Must not write more than 'N' items more than consumer "ate"
- Consumer process reads data from buffer
  - Should not try to consume if there is no data



Producer-Consumer Problem

- A number of applications:
  - Data from bar-code reader consumed by device driver
  - Data in a file you want to print consumed by printer spooler, which produces data consumed by line printer device driver
  - Web server produces data consumed by client's web browser
- Example: so-called "pipe" (|) in Unix

  > cat file | sort | uniq | more

  > prog | sort
- Thought questions: where's the bounded buffer?
- How "big" should the buffer be, in an ideal world?

Producer-Consumer Problem

- Solving with semaphores
  - We’ll use two kinds of semaphores
  - We’ll use *counters* to track how much data is in the buffer
    - One counter counts as we add data and stops the producer if there are N objects in the buffer
    - A second counter counts as we remove data and stops a consumer if there are 0 in the buffer
  - Idea: since general semaphores can count for us, we don’t need a separate counter variable
- Why do we need a second kind of semaphore?
  - We’ll also need a mutex semaphore

Producer-Consumer Problem

Shared: Semaphores mutex, empty, full;

Init: mutex = 1; /* for mutual exclusion*/

empty = N; /* number empty buf entries */

full = 0; /* number full buf entries */

Producer

```
do {
    . . .
    // produce an item in nextp
    . . .
    P(empty);
    P(mutex);
    . . .
    // add nextp to buffer
    . . .
    V(mutex);
    V(full);
} while (true);
```

Consumer

```
do {
    P(full);
    P(mutex);
    . . .
    // remove item to nextc
    . . .
    V(mutex);
    V(empty);
    . . .
    // consume item in nextc
    . . .
} while (true);
```

Readers and Writers

- In this problem, threads share data that some threads "read" and other threads "write".
- Goal: allow multiple concurrent readers but only a single writer at a time, and if a writer is active, readers wait for it to finish

Readers-Writers Problem

- Courtois et al 1971
- Models access to a database
  - A reader is a thread that needs to look at the database but won't change it.
  - A writer is a thread that modifies the database
- Example: making an airline reservation
  - When you browse to look at flight schedules the web site is acting as a reader on your behalf
  - When you reserve a seat, the web site has to write into the database to make the reservation

Readers-Writers Problem

- Many threads share an object in memory
  - Some write to it, some only read it
  - Only one writer can be active at a time
  - Any number of readers can be active simultaneously
- Key insight: generalizes the critical section concept

- One issue we need to settle, to clarify problem statement.
  - Suppose that a writer is active and a mixture of readers and writers now shows up. Who should get in next?
  - Or suppose that a writer is waiting and an endless of stream of readers keeps showing up. Is it fair for them to become active?

- We'll favor a kind of back-and-forth form of fairness:
  - Once a reader is waiting, readers will get in next.
  - If a writer is waiting, one writer will get in next.

Readers-Writers

```
mutex = Semaphore(1)
wrl = Semaphore(1)
rcount = 0;
```

Writer

```
while  True:
    wrl.P();
    . . .
    /*writing is performed*/
    . . .
    wrl.V();
```

Reader

```
while True:
    mutex.P();
    rcount++;
    if (rcount == 1)
        wrl.P();
    mutex.V();
   . . .
   /*reading is performed*/
   . . .
    mutex.P();
    rcount--;
    if (rcount == 0)
        wrl.V();
    mutex.V();
```

Readers-Writers Notes

- If there is a writer
  - First reader blocks on **wrl**
  - 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 **wrl**, and writer exits
  - Who gets to go in first?
- Why doesn't a writer need to use **mutex**?

Does this work as we hoped?

- If readers are active, no writer can enter
  - The writers wait doing a P(wrl)
- While writer is active, nobody can enter
  - Any other reader or writer will wait
- But back-and-forth switching is buggy:
  - Any number of readers can enter in a row
  - Readers can "starve" writers
- With semaphores, building a solution that has the desired back-and-forth behavior is really, really tricky!
  - We recommend that you try, but not too hard…

Common programming errors

Whoever next calls P() will freeze up. The bug might be confusing because that other process could be perfectly correct code, yet that's the one you'll see hung when you use the debugger to look at its state!

A typo. … second ti… every *othe*… to enter the critical sectio…

… won't respect mutual … the other processes … orrectly. Worse still, … e've done two "extra" V() ope… this way, other processes might g… o the CS inappropriately!

Proces…

P(S)
CS
P(S)

V(S)
CS
V(S)

P(S)
CS

More common mistakes

- Conditional code that can break the normal top-to-bottom flow of code in the critical section
- Often a result of someone trying to maintain a program, e.g. to fix a bug or add functionality in code written by someone else

```
P(S)
if(something or other)
   return;
CS
V(S)
```

Language Support for Concurrency

Revisiting semaphores!

- Semaphores are very “low-level” primitives
  - Users could easily make small errors
  - Similar to programming in assembly language
    - Small error brings system to grinding halt
  - Very difficult to debug
- Also, we seem to be using them in two ways
  - For mutual exclusion, the “real” abstraction is a critical section
  - But the bounded buffer example illustrates something different, where threads “communicate” using semaphores
- Simplification: Provide concurrency support in compiler
  - Monitors

Monitors

- Hoare 1974
- Abstract Data Type for handling/defining shared resources
- Comprises:
  - Shared Private Data
    - The resource
    - Cannot be accessed from outside
  - Procedures that operate on the data
    - Gateway to the resource
    - Can only act on data local to the monitor
  - Synchronization primitives
    - Among threads that access the procedures

Monitor Semantics

- Monitors guarantee mutual exclusion
  - Only one thread can execute monitor procedure at any time
    - "in the monitor"
  - If second thread invokes monitor procedure at that time
    - It will block and wait for entry to the monitor
      - $\Rightarrow$ Need for a wait queue
  - If thread within a monitor blocks, another can enter

Structure of a Monitor

```
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 instance of stack can be modified at a time

Synchronization Using Monitors

- Defines Condition Variables:
  - condition x;
  - Provides a mechanism to wait for events
    - Resources available, any writers
- 3 atomic operations on *Condition Variables*
  - x.wait(): release monitor lock, sleep until woken up

    $\Rightarrow$ condition variables have a waiting queue
  - x.notify(): wake one process waiting on condition (if there is one)
    - No history associated with signal
  - x.notifyAll(): wake all processes waiting on condition
    - Useful for resource manager

Producer Consumer using Monitors

```
Monitor Producer_Consumer {
    any_t buf[N];
    int n = 0, tail = 0, head = 0;
    condition not_empty, not_full;
    void put(char ch) {
        if(n == N)
            wait(not_full);
        buf[head%N] = ch;
        head++;
        n++;
            signal(not_empty);
    }
    char get()  {
        if(n == 0)
            wait(not_empty);
        ch = buf[tail%N];
        tail++;
        n--;
        signal(not_full);
        return ch;
    }
}
```

What if no thread is waiting when signal is called?

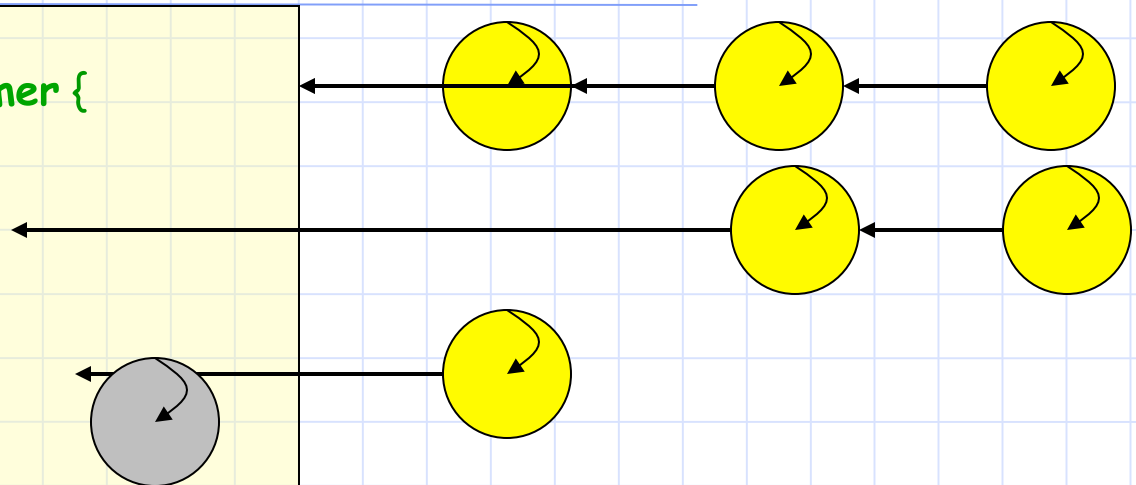
Signal is a “no-op” if nobody is waiting. This is very different from what happens when you call V() on a semaphore – semaphores have a “memory” of how many times V() was called!

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 so they can enter
  - Condition variables: each condition variable has a queue of threads waiting on the associated condition

Producer Consumer using Monitors

```
Monitor Producer_Consumer {
    condition not_full;
    /* other vars */
    condition not_empty;
    void put(char ch) {
        wait(not_full);
        . . .
        signal(not_empty);
    }
    char get() {
        . . .
    }
}
```



Condition Variables & Semaphores

- Condition Variables != semaphores
- Access to monitor is controlled by a lock
  - Wait: blocks thread and gives up the monitor lock
    - To call wait, thread has to be in monitor, hence the lock
    - Semaphore P() blocks thread only if value less than 0
  - Signal: causes waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - V() increments value, so future threads need not wait on P()
    - Condition variables have no history!
- However they can be used to implement each other

Language Support

- Can be embedded in programming language:
  - Synchronization code added by compiler, 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 and safer than semaphores
  - Compiler can check
  - Lock acquire and release are implicit and cannot be forgotten

Monitor Solutions to Classical Problems

A Simple Monitor

```
Monitor EventTracker {
   int numburgers = 0;
   condition hungrycustomer;

   void customerenter() {
      if (numburgers == 0)
            hungrycustomer.wait()
      numburgers -= 1
   }
   void produceburger() {
      ++numburger;
      hungrycustomer.signal();
   }
}
```

- Because condition variables lack state, all state must be kept in the monitor
- The condition for which the threads are waiting is necessarily made explicit in the code
  - Numburgers > 0
- Hoare vs. Mesa semantics
  - What happens if there are lots of customers?

A Simple Monitor

```
Monitor EventTracker {
    int numburgers = 0;
    condition hungrycustomer;

    void customerenter() {
        while(numburgers == 0)
            hungrycustomer.wait()
        numburgers -= 1
    }
    void produceburger() {
        ++numburger;
        hungrycustomer.signal();
    }
}
```

- Because condition variables lack state, all state must be kept in the monitor
- The condition for which the threads are waiting is necessarily made explicit in the code
  - Numburgers > 0
- Hoare vs. Mesa semantics
  - What happens if there are lots of customers?

Hoare vs. Mesa Semantics

- Hoare envisioned that the monitor lock would be transferred directly from the signalling thread to the newly woken up thread (Hoare semantics)
  - Yields clean semantics, easy to reason about
- But it is typically not desirable to force the signal'ing thread to relinquish the monitor lock immediately to a woken up thread
  - Confounds scheduling with synchronization, penalizes threads
- Every real system simply puts a woken up thread to be put on the run queue, but does not immediately run that thread, or transfer the monitor lock (known as Mesa semantics)
  - So, the thread is forced to re-check the condition upon wake up!

Producer Consumer using Monitors

```
Monitor Producer_Consumer {
    any_t buf[N];
    int n = 0, tail = 0, head = 0;
    condition not_empty, not_full;

    void put(char ch) {
        if(n == N)
            wait(not_full);
        buf[head%N] = ch;
        head++;
        n++;
            signal(not_empty);
    }
}
```

```
char get() {
        if(n == 0)
            wait(not_empty);
        ch = buf[tail%N];
        tail++;
        n--;
        signal(not_full);
        return ch;
    }
```

Readers and Writers

```
Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()
 {

 }
 Void EndWrite()
 {

 }

 Void BeginRead()
 {

 }

 Void EndRead()
 {

 }
```

Readers and Writers

```
Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()
 {

       NWriters = 1;
 }
 Void EndWrite()
 {
       NWriters = 0;

 }

Void BeginRead()
 {

      ++NReaders;

 }

 Void EndRead()
 {
       --NReaders

 }
```

Readers and Writers

```
Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()
 {

     NWriters = 1;
 }
 Void EndWrite()
 {
      NWriters = 0;
      if(WaitingReaders)
          Signal(CanRead);
      else
          Signal(CanWrite);
 }

 Void BeginRead()
 {

     ++NReaders;

 }

 Void EndRead()
 {
      if(--NReaders == 0)
              Signal(CanWrite);

 }
```

Readers and Writers

```
Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()
 {
       if(NWriters == 1 || NReaders > 0)
       {
             ++WaitingWriters;
            wait(CanWrite);
            --WaitingWriters;
       }
       NWriters = 1;
 }
 Void EndWrite()
 {
        NWriters = 0;
        if(WaitingReaders)
             Signal(CanRead);
        else
             Signal(CanWrite);
 }

Void BeginRead()
 {
      if(NWriters == 1 || WaitingWriters > 0)
      {
            ++WaitingReaders;
            Wait(CanRead);
          --WaitingReaders;
      }
      ++NReaders;
      Signal(CanRead);
 }

 Void EndRead()
 {
       if(--NReaders == 0)
               Signal(CanWrite);

 }
```

Understanding the Solution

- A writer can enter if there are no other active writers and no readers are waiting

Readers and Writers

```
Monitor ReadersNWriters {
 int WaitingWriters, WaitingReaders,NReaders, NWriters;
 Condition CanRead, CanWrite;

 Void BeginWrite()
 {
     if(NWriters == 1 || NReaders > 0)
     {
         ++WaitingWriters;
        wait(CanWrite);
        --WaitingWriters;
     }
     NWriters = 1;
 }
 Void EndWrite()
 {
      NWriters = 0;
      if(WaitingReaders)
          Signal(CanRead);
      else
          Signal(CanWrite);
 }

Void BeginRead()
 {
     if(NWriters == 1 || WaitingWriters > 0)
     {
          ++WaitingReaders;
          Wait(CanRead);
        --WaitingReaders;
     }
     ++NReaders;
     Signal(CanRead);
 }

 Void EndRead()
 {
      if(--NReaders == 0)
            Signal(CanWrite);

 }
```

Understanding the Solution

- A reader can enter if
  - There are no writers active or waiting
- So we can have many readers active all at once
- Otherwise, a reader waits (maybe many do)

Readers and Writers

```
Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()
 {
      if(NWriters == 1 || NReaders > 0)
      {
           ++WaitingWriters;
          wait(CanWrite);
          --WaitingWriters;
      }
      NWriters = 1;
 }
 Void EndWrite()
 {
       NWriters = 0;
       if(WaitingReaders)
           Signal(CanRead);
       else
           Signal(CanWrite);
 }

Void BeginRead()
 {
     if(NWriters == 1 || WaitingWriters > 0)
     {
          ++WaitingReaders;
          Wait(CanRead);
        --WaitingReaders;
     }
     ++NReaders;
     Signal(CanRead);
 }

 Void EndRead()
 {
      if(--NReaders == 0)
            Signal(CanWrite);

 }
```

Understanding the Solution

- When a writer finishes, it checks to see if any readers are waiting
  - If so, it lets one of them enter
  - That one will let the next one enter, etc…
- Similarly, when a reader finishes, if it was the last reader, it lets a writer in (if any is there)

Readers and Writers

```
Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()
 {
       if(NWriters == 1 || NReaders > 0)
       {
             ++WaitingWriters;
            wait(CanWrite);
            --WaitingWriters;
       }
       NWriters = 1;
 }
 Void EndWrite()
 {
        NWriters = 0;
        if(WaitingReaders)
             Signal(CanRead);
        else
             Signal(CanWrite);
 }
```

```
Void BeginRead()
 {
      if(NWriters == 1 || WaitingWriters > 0)
      {
            ++WaitingReaders;
            Wait(CanRead);
          --WaitingReaders;
      }
      ++NReaders;
      Signal(CanRead);
 }

 Void EndRead()
 {
       if(--NReaders == 0)
              Signal(CanWrite);

 }
```

Understanding the Solution

- It 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
  - … this is 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

Subtle aspects?

- Condition variables force the actual conditions that a thread is waiting for to be made explicit in the code
  - The comparison preceding the "wait()" call concisely specifies what the thread is waiting for
- The fact that condition variables themselves have no state forces the monitor to explicitly keep the state that is important for synchronization
  - This is a good thing

Mapping to Real Languages

```
Monitor ReadersNWriters {
 int x;

 Void func()
 {
     if(x == 0)
     {
         …
     }
     x = 1
 }
```

```
Class ReadersNWriters:
    def __init__(self):
        self.lock = Lock()

    def  func():
        with self.lock:
            if x == 0:
                ….
            x = 1
```

- Python monitors are simulated by explicitly allocating a lock and acquiring and releasing it (with the "with" statement) when necessary
  - More flexible than Hoare's approach

Mapping to Real Languages

```
Monitor ReadersNWriters {
  int x;
  Condition foo

 Void func()
 {
     if(x == 0)
     {
         foo.wait()
     }
     x = 1
 }
```

```
Class ReadersNWriters:
    def __init__(self):
        self.lock = Lock()
        self.foo = Condition(self.lock)

    def  func():
        with self.lock:
            if x == 0:
                self.foo.wait()
            x = 1
```

- Python condition variables retain a pointer to the monitor lock so they can release it when the thread goes to wait
- signal() -> notify(); broadcast() -> notifyAll()

To conclude

- Race conditions are a pain!
- We studied several ways to handle them
  - Each has its own pros and cons
- Support in Python, Java, C# has simplified writing multithreaded applications
  - Java and C# support at most one condition variable per object, so are slightly more limited
- Some new program analysis tools automate checking to make sure your code is using synchronization correctly
  - The hard part for these is to figure out what "correct" means!