Synchronization

Chapter 6 and 7

Cooperating Threads

- We like our threads to cooperate
- Shared memory can be a nice way to do this
  - For example, shared data structures
  - For example, pool of threads to get work done
- Concurrent access to the same memory
  - Can leave contents in an unknown state
  - Examples, race1.c
  - Can be a problem on multiprocessor **and** uniprocessor systems
- We call these *race conditions*
  - Correctness of the program depends on the timing of execution

Race Conditions

- Consider, a single statement may expand to multiple machine instructions
- Execution of these instructions may be interleaved
- How about just val++ and val--

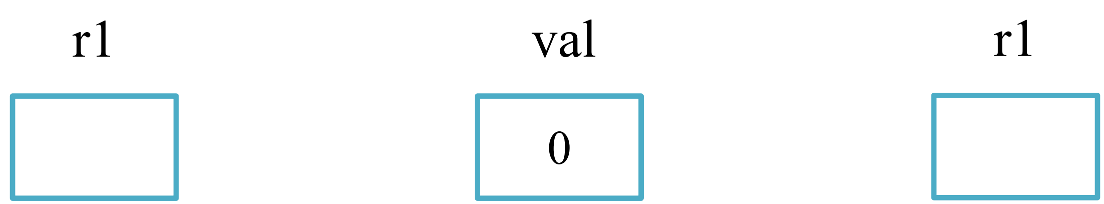
```
load r1, val          load r1, val
inc r1                dec r1
store val, r1         store val, r1
```

- Interleaving can leave 1 (or -1) in val
- This must be happening all the time in our example program!

What’s Going On?

```
load r1, val
inc r1
store val, r1
```

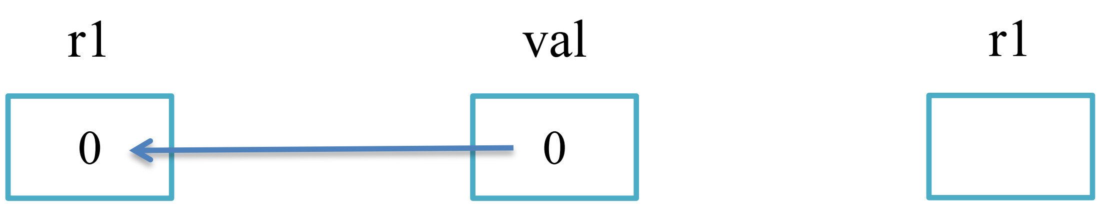
```
load r1, val
dec r1
store val, r1
```



What's Going On?

```
load r1, val
inc r1
store val, r1
```

```
load r1, val
dec r1
store val, r1
```



What's Going On?

```
load r1, val
inc r1
store val, r1
```

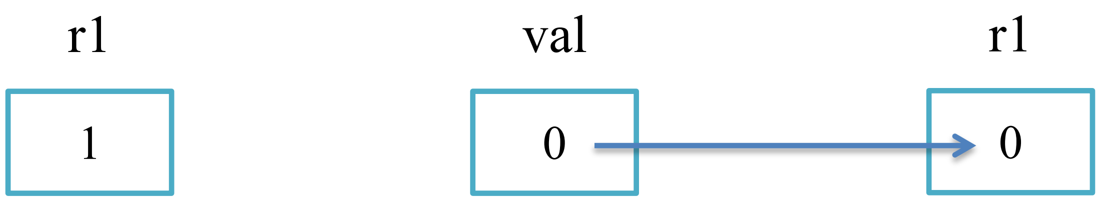
```
load r1, val
dec r1
store val, r1
```

r1	val	r1
1	0	

What's Going On?

```
load r1, val
inc r1
store val, r1
```

```
load r1, val
dec r1
store val, r1
```



What's Going On?

```
load r1, val
inc r1
store val, r1
```

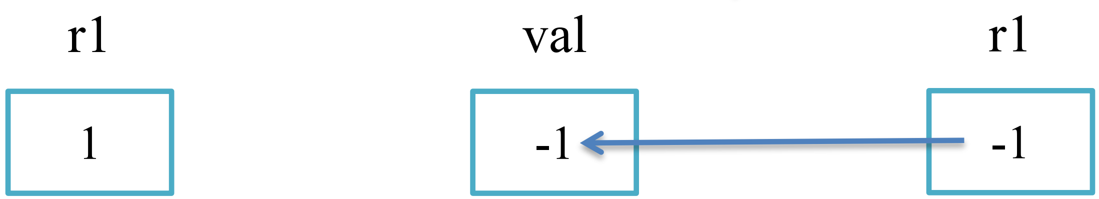
```
load r1, val
dec r1
store val, r1
```

r1	val	r1
1	0	-1

What's Going On?

```
load r1, val
inc r1
store val, r1
```

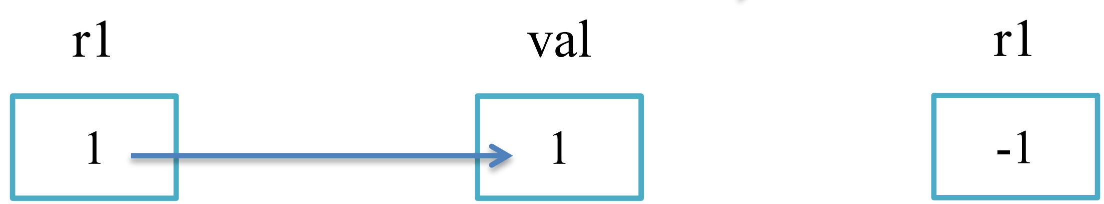
```
load r1, val
dec r1
store val, r1
```



What's Going On?

```
load r1, val
inc r1
store val, r1
```

```
load r1, val
dec r1
store val, r1
```



Why do we get race conditions?

- OS gets to make CPU scheduling decisions
  - May be difficult to predict what it will do
  - May vary from execution to execution
- Context switch can happen between any two instructions
- Any interleaving of thread execution is possible
- In general, a correct algorithm should:
  - Not depend on choices made by the CPU scheduler
  - Work under concurrent execution
  - i.e., it should be free of race conditions

The Critical Section Problem

- Identify *critical section*s within our code
  - Places where we access shared data
  - Only one thread permitted in any critical section at a time
  - OK, seems like this avoids potential problems
- How about concurrency?
  - Most of the time a thread won't be in a critical section, for typical applications.
  - It will spend most of its time in the *remainder section*

Critical Section Template

```
someThread() {
   while ( true ) {

      entry Section;

      critical Section;

      exit Section;

      remainder Section;
   }
}
```

We just need to figure out, once and for all, how to write these two sections so they are guaranteed to work.

Critical Section Problem, let's solve it

```
bool inside[ 2 ] = { false, false };

someThread0() {
  while ( true ) {

    while ( inside[ 1 ] ) {
    }
    inside[ 0 ] = true;

    critical Section;

    inside[ 0 ] = false;

    remainder Section;
  }
}
```

```
someThread1() {
  while ( true ) {

    while ( inside[ 0 ] ) {
    }
    inside[ 1 ] = true;

    critical Section;

    inside[ 1 ] = false;

    remainder Section;
  }
}
```

Note, the body is empty.

- Does it work? Let's find out

Broken Solution 1

```
bool inside[ 2 ] = { false, false };

someThread0() {
  while ( true ) {

    while ( inside[ 1 ] ) {
    }
    inside[ 0 ] = true;

    critical Section;

    inside[ 0 ] = false;

    remainder Section;
  }
}
```

```
someThread1() {
  while ( true ) {

    while ( inside[ 0 ] ) {
    }
    inside[ 1 ] = true;

    critical Section;

    inside[ 1 ] = false;

    remainder Section;
  }
}
```

Another Candidate

```
bool wantIn[ 2 ] = { false, false };

someThread0() {
  while ( true ) {

    wantIn[ 0 ] = true;
    while ( wantIn[ 1 ] ) {
    }

    critical Section;

    wantIn[ 0 ] = false;

    remainder Section;
  }
}
```

```
someThread1() {
  while ( true ) {

    wantIn[ 1 ] = true;
    while ( wantIn[ 0 ] ) {
    }

    critical Section;

    wantIn[ 1 ] = false;

    remainder Section;
  }
}
```

- Does it work?

Another (Broken) Candidate

```
bool wantIn[ 2 ] = { false, false };

someThread0() {
  while ( true ) {

    wantIn[ 0 ] = true;
    while ( wantIn[ 1 ] ) {
    }

    critical Section;

    wantIn[ 0 ] = false;

    remainder Section;
  }
}
```

```
someThread1() {
  while ( true ) {

    wantIn[ 1 ] = true;
    while ( wantIn[ 0 ] ) {
    }

    critical Section;

    wantIn[ 1 ] = false;

    remainder Section;
  }
}
```

Requirements for a Critical-Section Solution

- Mutual Exclusion
  - Only one thread at a time in a critical section
- Progress
  - If two or more threads want to enter the critical section, eventually one will get it.
  - Without needing help from some other thread
  - i.e., threads can’t get stuck in the doorway
- Bounded Waiting
  - Can’t starve a thread in the entry section
  - A bound on the number of threads that can get ahead of some thread A

Another candidate, take turns

```
int turn = 0;

someThread0() {
  while ( true ) {

    while ( turn == 1 ) {
     }

    critical Section;

    turn = 1;

    remainder Section;
  }
}
```

```
someThread1() {
  while ( true ) {

    while ( turn == 0 ) {
    }

    critical Section;

    turn = 0;

    remainder Section;
  }
}
```

- Does it work?

Problems with Taking Turns

```
int turn = 0;

someThread0() {
  for ( int i = 0; i < 999;
        i++ ) {

    while ( turn == 1 ) {
    }

    critical Section;

    turn = 1;

    remainder Section;
  }
}
```

```
someThread1() {
  for ( int i = 0; i < 1000;
        i++ ) {

    while ( turn == 0 ) {
    }

    critical Section;

    turn = 0;

    remainder Section;
  }
}
```

Peterson's Solution

threads

- Two-~~process~~ solution
- Assume that Load and Store instructions are *atomic*
  - i.e., they can't be interrupted by another instruction
- Use flags, with a turn variable to break ties
  - bool wantIn[] = { false, false };
  - int turn = 0;
- Set wantIn[me] when you want to enter the critical section
- Then, set turn to let the other thread go first.
  - Think: "I want in, but you can go first if there's a tie."

Peterson’s Solution

```
bool wantIn[] = { false, false };
int turn = 0;
```

```
someThread0() {
  while ( true ) {
    wantIn[ 0 ] = true;
    turn = 1; // after you
    while ( wantIn[ 1 ] &&
            turn == 1 ) {
    }

    critical Section;

    wantIn[ 0 ] = false;

    remainder Section;
  }
}
```

```
someThread1() {
  while ( true ) {
    wantIn[ 1 ] = true;
    turn = 0; // no, after you
    while ( wantIn[ 0 ] &&
            turn == 0 ) {
    }

    critical Section;

    wantIn[ 1 ] = false;

    remainder Section;
  }
}
```

Beyond Peterson’s Solution

- So, we’re done, right?
  - Peterson’s algorithm solves the critical section problem.
- Not so fast
- What’s wrong with just using Peterson’s solution as the basis all our synchronization code?

What's Wrong with Peterson's Solution?

- Where should I start?
  - It depends on busy waiting
  - It just works for two threads
  - It's tedious to implement (and we're going to have critical sections all over the place)
  - It's tricky to read and understand (it's not obvious that this is even critical section code)
  - It may not actually work ☹

OS Synchronization Support

- I'd rather have a synchronization mechanism provided by the OS.
  - Then, I wouldn't have to write my own
  - And (more importantly) my process could block when it has to wait.
- We demand synchronization help from the Operating System!

The Semaphore

- Semaphore S, it's like an integer.
  - We'll declare a semaphore like:
    sem s = 1;
- Two standard operations on a semaphore s.
  - acquire(s) and release(s)
  - AKA wait(s) and release(s)
  - AKA wait(s) and post(s)
  - Originally, p(s) and v(s)
  - Lots of names for the same two operations . . . not my fault

This is the syntax we'll expect when we write pseudocode with semaphores.

Semaphore Operations

- *Acquire* operation : wait until the semaphore has a positive value, then decrement it and proceed
- *Release* operation : increment the semaphore value
- You can think of them this way: (but they aren't really implemented like this)

```
def acquire( sem s ) {
 while (s <= 0 )
   ; // no-op
 s--;
}
```

```
def release( sem s ) {
 s++;
}
```

Mutual Exclusion with Semaphores

```
sem s = 1;

someThread() {
  while ( true ) {

    acquire( s );

    critical Section;

    release( s );

    remainder Section;
  }
}
```

```
someThread() {
  while ( true ) {

    acquire( s );

    critical Section;

    release( s );

    remainder Section;
  }
}
```

Critical Section with Semaphores

- It's a good thing
  - Efficient, we will figure out how to implement acquire() using blocking
  - Generalizes to any number of threads/processes
  - Generalizes to any number of critical sections
  - Portable (well, maybe)
  - Ease of use / readability (it looks like synchronization code)
- Plus, semaphores are good at solving lots of synchronization problems
- Let's try some

More than Just Critical Sections

Make sure B() happens after A()

```
someThread() {
  // Do some stuff here.



  A();


  // Do some more stuff.

}
```

```
otherThread() {
  // Do some stuff.

  // Watch TV.


  B();


  // Clean room.

}
```

More than Just Critical Sections

Make sure B() happens after A()

```
sem x = 0;

someThread() {
  // Do some stuff here.



  A();
  release( x );

  // Do some more stuff.

}
```

```
otherThread() {
  // Do some stuff.

  // Watch TV.

  acquire( x );
  B();


  // Clean room.

}
```

A Broken Solution

Make sure B() happens after A()

```
sem x = 1;

someThread() {
  // Do some stuff here.



  A();
  release( x );

  // Do some more stuff.

}
```

```
otherThread() {
  // Do some stuff.

  // Watch TV.

  acquire( x );
  B();


  // Clean room.

}
```

More than Just Critical Sections

Make sure B() happens after A(), and C() happens after B()

```
someThread() {

  A();

}
```

```
otherThread() {

  B();

}
```

```
otherOtherThread() {

  C();

}
```

More than Just Critical Sections

Make sure B() happens after A(), and C() happens after B()

```
sem x = 0;
sem y = 0;
someThread() {



  A();
  release(x);


}
```

```
otherThread() {


  acquire(x);
  B();
  release(y);


}
```

```
otherOtherThread() {


  acquire(y);
  C();



}
```

A Broken Solution

Make sure B() happens after A(), and C() happens after B()

```
sem x = 0;
```

```
someThread() {



  A();
  release(x);


}
```

```
otherThread() {


  acquire(x);
  B();
  release(x);


}
```

```
otherOtherThread() {


  acquire(x);
  C();



}
```

More than Just Critical Sections

Make sure B() and C() happen after A()

```
someThread() {

  A();

}
```

```
otherThread() {

  B();

}
```

```
otherOtherThread() {

  C();

}
```

Same Solution as Before

Make sure B() and C() happen after A()

```
sem x = 0;
sem y = 0;
someThread() {
    A();
    release(x);
}

otherThread() {
    acquire(x);
    B();
    release(y);
}

otherOtherThread() {
    acquire(y);
    C();
}
```

This doesn't permit as much concurrency.

More than Just Critical Sections

Make sure B() and C() happen after A()

```
sem x = 0;
sem y = 0;

someThread() {
  A();
  release(x);
  release(y);
}

otherThread() {
  acquire(x);
  B();
}

otherOtherThread() {
  acquire(y);
  C();
}
```

More than Just Critical Sections

Make sure A() happens after both B() and C()

```
someThread() {

  A();

}
```

```
otherThread() {

  B();

}
```

```
otherOtherThread() {

  C();

}
```

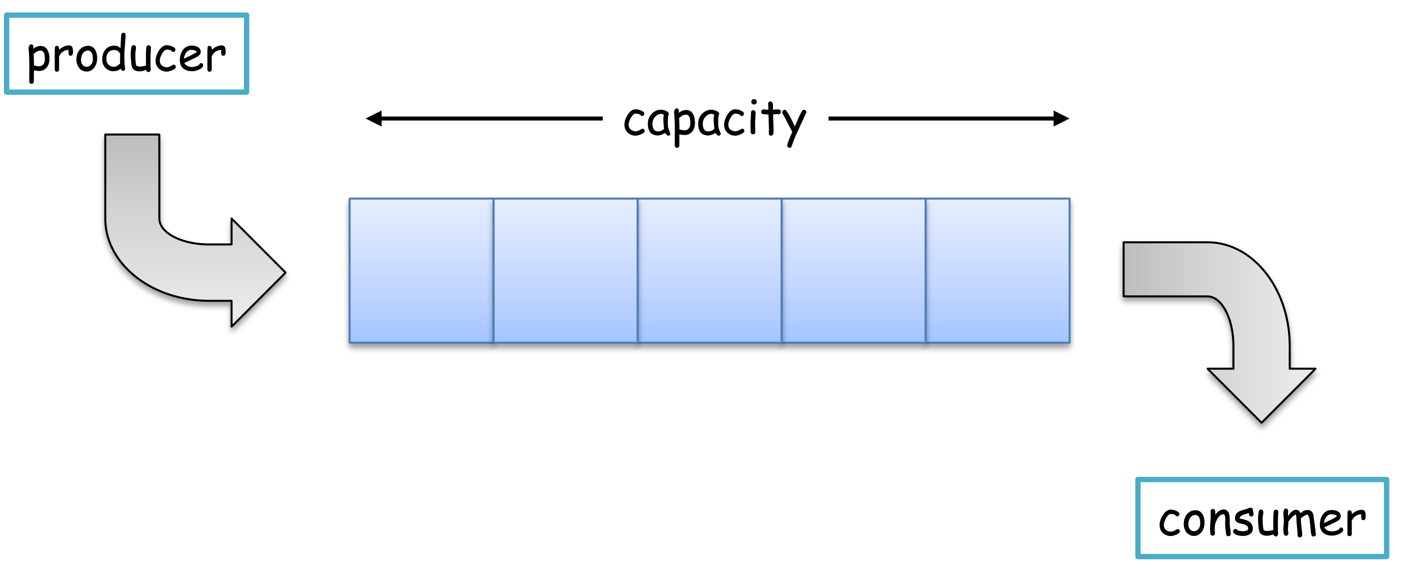
More than Just Critical Sections

Make sure A() happens after both B() and C()

```
sem x = 0;
sem y = 0;

someThread() {

  acquire(x);
  acquire(y);
  A();

}

otherThread() {

  B();
  release(x);

}

otherOtherThread() {

  C();
  release(y);

}
```

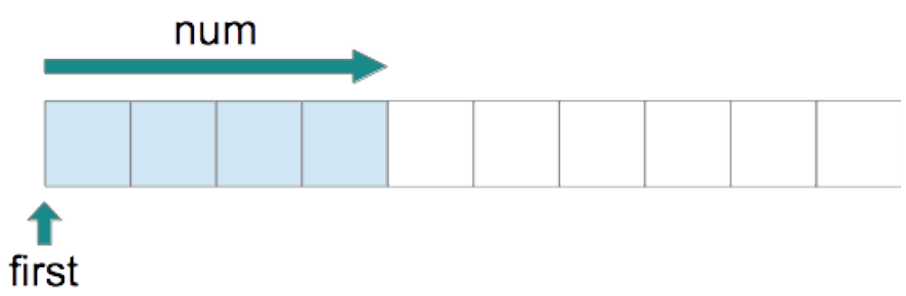
Bounded-Buffer Problem

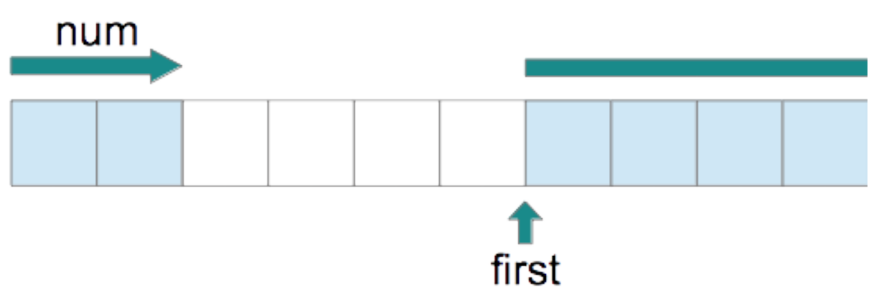
- Bounded buffer, a classic synchronization problem
  - Buffer with limited capacity (number of slots)
  - Producer waits if buffer is full
  - Consumer waits if buffer is empty
  - Queue order within the buffer



Implementing Bounded Buffer

```
#define BUFFER_SIZE 10

// A circular queue of items in the buffer.
Item buffer[ BUFFER_SIZE ];

int first = 0;                   // Position of first item in the buffer
int num = 0;                     // Number of items in the buffer.
```





A "Solution"
(If you like busy waiting and race conditions)

```
while ( true ) {
  Item it = makeSomething();
  while ( num >= BUFFER_SIZE )
     ; // just wait for a free spot
  buffer[ ( first + num ) %
           BUFFER_SIZE ] = it;
  num++;
}
```

Producer Thread

```
while ( true ) {
  while ( num == 0)
     ; // Wait for an item
  Item it = buffer[ first ];
  first = ( first + 1 ) % BUFFER_SIZE;
  num--;
  consumeItem( it );
}
```

Consumer Thread

Bounded-Buffer with Semaphores

```
sem emptyCount = BUFFER_SIZE;
sem fullCount = 0;
```

```
while ( true ) {
  Item it = makeSomething();
  acquire( emptyCount );
  buffer[ ( first + num ) %
           BUFFER_SIZE ] = it;
  num++;
  release( fullCount );
}
```

Producer Thread

```
while ( true ) {
  acquire( fullCount );
  Item it = buffer[ first ];
  first = ( first + 1 ) % BUFFER_SIZE;
  num--;
  release( emptyCount );
  consumeItem( it );
}
```

Consumer Thread

Fixed?

Bounded-Buffer Solution

```
sem emptyCount = BUFFER_SIZE;
sem fullCount = 0;
sem lock = 1;
```

```
while ( true ) {
  Item it = makeSomething();
  acquire( emptyCount );
  acquire( lock );
  buffer[ ( first + num ) %
           BUFFER_SIZE ] = it;
  num++;
  release( lock );
  release( fullCount );
}
```

Producer Thread

```
while ( true ) {
  acquire( fullCount );
  acquire( lock );
  Item it = buffer[ first ];
  first = ( first + 1 ) % BUFFER_SIZE;
  num--;
  release( lock );
  release( emptyCount );
  consumeItem( it );
}
```

Consumer Thread

Fixed!

Bounded-Buffer Solution Recap

- A Semaphore lock
  - For preventing concurrent access to the buffer
  - Initialized to 1
  - Just like semaphore-based critical section solution
- A Semaphore fullCount
  - Counts the number of filled buffer slots
  - Initialized to 0
- A Semaphore emptyCount
  - Counts the number of empty slots
  - Initialized to the buffer capacity
- Note, we can get a lot of value out semaphore counting behavior

Implementing Semaphores

- Inside the OS, we have to be able to:
  - efficiently implement semaphores
  - … and other synchronization mechanisms
- We'll need to:
  - Solve the critical section problem ourselves
  - Use this to block processes when they have to wait

DIY Synchronization Solutions

- Uniprocessor solutions
  - Could temporarily disable interrupts
  - If we let processes do this, say goodbye to CPU protection
    - Appropriate for *cooperative multitasking*
- Modern CPUs provide help
  - Instructions indented to simplify synchronization
  - Atomic instructions to
    - Test a memory word and set its value

      or
    - Compare and swap the contents of two memory words

Test-and-Set-Based Solution

- **Test-And-Set Instruction**

set condition codes based on memory location (is it zero)
set memory location to non-zero

Atomic

```
int lock = 0; // shared
while ( true ) {
    wait:   testAndSet lock
            branchIfNonZero wait

            // Critical section

            lock = 0;

            // Remainder
}
```

This is called
a *spinlock*

Test-and-Set-Based Solution

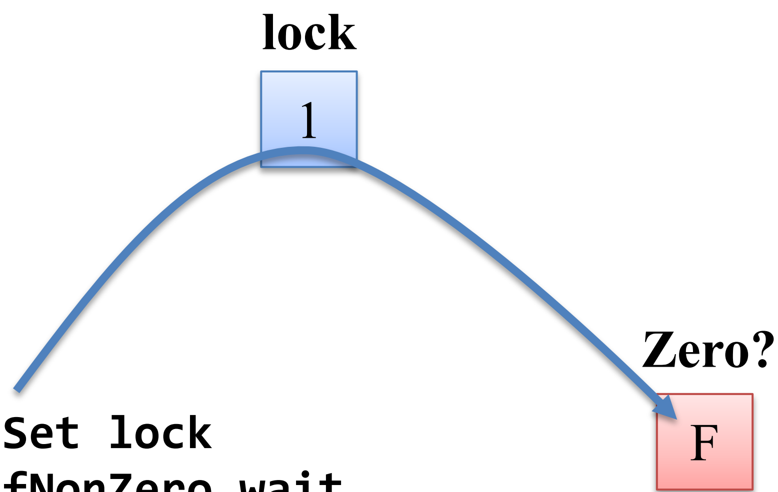
lock

1

Zero?

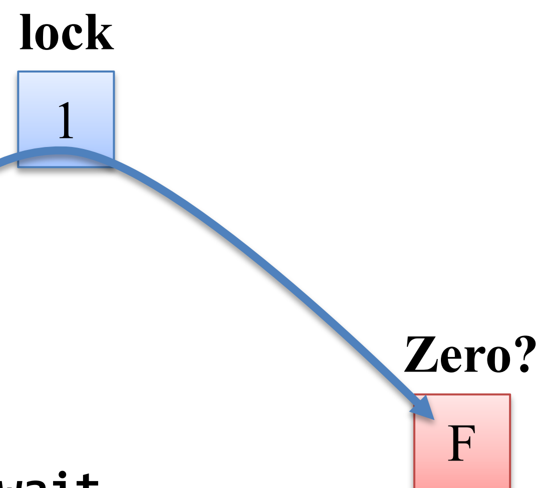
```
while( true ) {
    wait:  testAndSet lock
           branchIfNonZero wait

           Critical Section;

           lock = 0;

           Remainder Section;
}
```

Test-and-Set-Based Solution




```
while( true ) {
    wait:  testAndSet lock
           branchIfNonZero wait

           Critical Section;

           lock = 0;

           Remainder Section;
}
```

Test-and-Set-Based Solution




```
while( true ) {
    wait:  testAndSet lock
           branchIfNonZero wait

           Critical Section;

           lock = 0;

           Remainder Section;
}
```

Test-and-Set-Based Solution

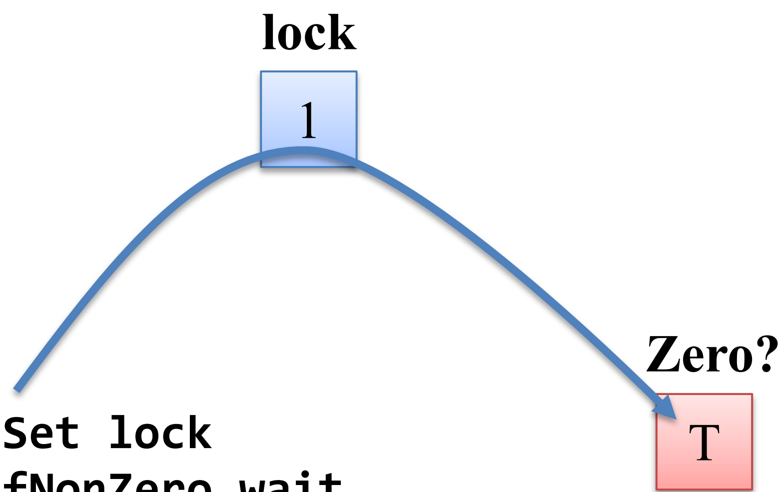
lock

0

Zero?

```
while( true ) {
    wait:   testAndSet lock
            branchIfNonZero wait

            Critical Section;

            lock = 0;

            Remainder Section;
}
```

Test-and-Set-Based Solution




```
while( true ) {
    wait:  testAndSet lock
           branchIfNonZero wait

           Critical Section;

           lock = 0;

           Remainder Section;
}
```

Test-and-Set-Based Solution

lock

0

Zero?

```
while( true ) {
    wait:  testAndSet lock
           branchIfNonZero wait

           Critical Section;

           lock = 0;

           Remainder Section;
}
```

Compare-and-Swap-Based Solution

- Compare-And-Swap Instruction

```
if ( target == expected ) {
   set condition codes to true
   target = newValue;
} else
    set condition codes to false
```

Atomic

```
int lock = 0;  // shared
while ( true ) {
    wait:  compareAndSwap lock, 0, 1
           branchIfFalse wait

           // Critical section

           lock = 0;

           // Remainder
}
```

Semaphore Implementation

- So, applications can use semaphores for synchronization
  - If we're the OS, we just need to figure out how to implement acquire() and release()
  - We could implement them like this, with a busy waiting *spinlock* in each.

```
def acquire( sem s ) {
   bool proceed = false;
   while ( !proceed ) {
      spinlock_entry_code;
      if ( s > 0 ) {
        s--;
        proceed = true;
      }
      spinlock_exit_code;
   }
}
```

```
def release( sem s ) {
   spinlock_entry_code;
   s++;
   spinlock_exit_code;
}
```

Semaphore Implementation

- Busy waiting spinlock in acquire() and release()
  - We know how to do this (and it will work)
  - Don’t need to duplicate the spinlock code everywhere
  - All code outside just uses acquire() / release()
- But, how long might a thread need to wait in acquire()?
  - Depends on the application.
- Busy waiting could be a problem

Blocking Semaphore Implementation

- We'd like acquire() to just block until the thread can make progress
- We're going to need help from the OS for this
  - The OS can block threads waiting to acquire
  - Keep a queue of PCBs (or threads) for each semaphore, like a device queue
  - So, we have some scheduling queues that aren't associated with any physical device

Blocking Semaphore Implementation

- Pretend the OS has two operations
  - block(), mark the current process as blocked
  - wakeup(P), mark process P as ready and place its PCB back on the ready queue

```
acquire(S){
  spinlock_entry_code;
  if (s.value <= 0) {
    enqueue this process on s.list;
    block();
    spinlock_exit_code;
    CPU_Scheduler();
  } else {
    s.value--;
    spinlock_exit_code;
  }
}
```

```
release(S){
  spinlock_entry_code;
  if (s.value >= 0 && s.list.length > 0) {
    dequeue a process P from s.list;
     wakeup(P);
  } else
    s.value++;
  spinlock_exit_code;
}
```

OS Semaphore Implementation

```
acquire(S){
  spinlock_entry_code;
  if (s.value <= 0) {
    enqueue this process on s.list;
    block();
    spinlock_exit_code;
    CPU_Scheduler();
  } else {
    s.value--;
    spinlock_exit_code;
  }
}
```

```
release(S){
  spinlock_entry_code;
  if (s.value >= 0 && s.list.length > 0) {
    dequeue a process P from s.list;
    wakeup(P);
  } else
    s.value++;
  spinlock_exit_code;
}
```

- Busy waiting, but just for a little while
  - Just long enough to inspect the semaphore state …
  - … and make a scheduling decision
  - **Doesn’t** depend on how long the application needs to wait in acquire().

Flavors of Semaphores

- *Counting Semaphore*
  - Integer value is unrestricted
- *Binary Semaphore*
  - Integer value can only be 0 or 1
  - Good enough, and may be easier to implement
  - Sometimes called a *mutex*
  - Still works fine for critical section solution

```
sem S; // Implicitly set to 1.

acquire( S );
criticalSection;
release( S );
```

Do Semaphores Really Work?

- Yes. They aren’t just an idea, they’re a readily available synchronization mechanism
- POSIX semaphores
  - Header file: semaphore.h
  - Type: sem_t
  - int sem_init(sem_t *sem, int pshared, unsigned int value);
  - int sem_destroy(sem_t * sem);
  - int sem_wait(sem_t * sem);
  - int sem_post(sem_t * sem);
  - sem_t * sem_open(char *name, flags, permissions, int value);
  - int sem_close(sem_t *sem);

Do Semaphores Really Work?

- Yes. They aren't just an idea, they're a readily available synchronization mechanism
- POSIX semaphores
  - Header file: `semaphore.h`
  - Type: `sem_t`
  - `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - `int sem_destroy(sem_t * sem);`
  - `int sem_wait(sem_t * sem);`
  - `int sem_post(sem_t * sem);`
  - `sem_t * sem_open(char *name, flags, permissions, int value);`
  - `int sem_close(sem_t *sem);`
- Working examples

Within the same process.

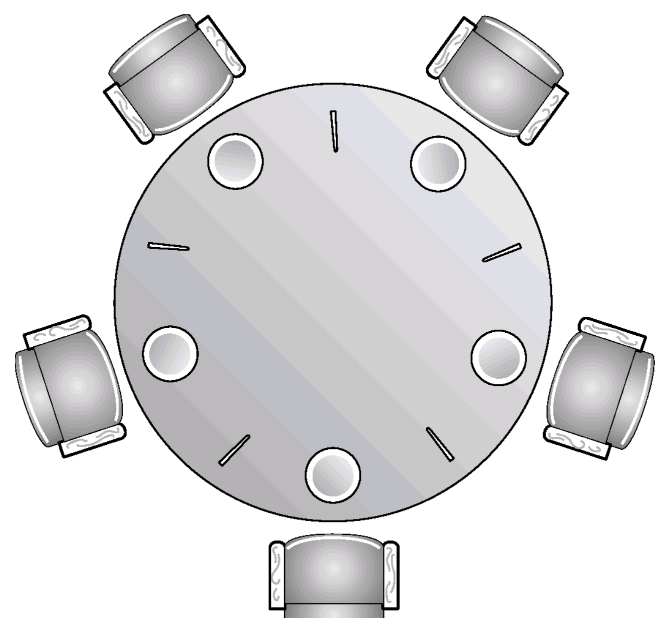
Between processes.

Classic Synchronization Problems

- Some more difficult problems
  - To help us think about synchronization
  - To get some practice using semaphores
  - To see standard types of solutions
  - To improve our computer literacy
- Bounded buffer problem (already seen that)
- Dining-philosophers problem
- Readers-writers problem (later)

Dining-Philosophers Problem

- 5 Philosophers: think or eat
  - Think when you want, eat when you want
  - Each philosopher needs two chopsticks to eat
  - Shared resources: the chopsticks
  - Models cooperating threads that need more than one resource at a time



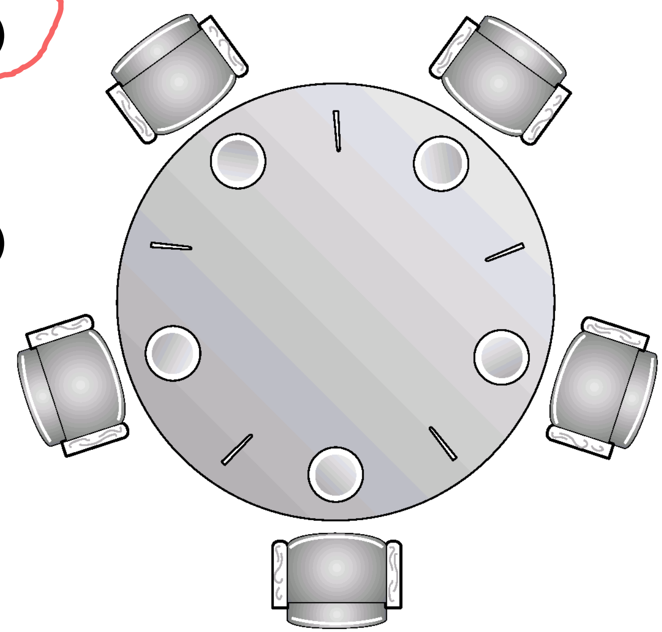
Semaphore Solution

```
sem chopstick[5] = {1,1,1,1,1}

philospher(int i) {
  while ( true ) {
    think();

    acquire( chopstick[ i ] );
    acquire( chopstick[(i+1)%5] )
    eat();
    release( chopstick[i] );
    release( chopstick[(i+1)%5] )
  }
}
```

repeat

grap筷子

2根



Semaphore Solution

- How’s my solution
  - Well, it may deadlock
  - How?
- Solutions?
  - Keep an extra chopstick around?
  - Only one at a time can eat? low efficiency
  - Only pick up a chopstick if you can get both?
  - Break the symmetry, apply a global ordering to chopstick allocation (this is a reusable trick)

Another POSIX API: Mutex/Lock

- Initialize a mutex lock:
  - `pthread_mutex_t lock;`
  - **`pthread_mutex_init( &lock, NULL );`**
- Or, we can do it at declaration time:
  - `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`
- Acquire the lock:
  - **`pthread_mutex_lock( &lock );`**
- Release the lock:
  - **`pthread_mutex_unlock( &lock );`**

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

Condition Variables (CVs)

- To wait for a condition to become true, a thread can make use of a CV.
- A CV is an explicit queue that threads can put themselves on when some condition is not as desired by **waiting** on the condition.
- Some other thread, when it changes condition, can then wake one (or more) of those waiting threads and thus allow them to continue by **signaling** on the condition.

wait() and signal()

- wait(cond_t *cv, mutex_t *lock)
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
- signal(cond_t *cv)
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return w/o doing anything

POSIX Condition Variables

- Declare then initialize a condition variable:
  - `pthread_cond_t cond;`
  - **`pthread_cond_init( &cond, NULL);`**
- Or, do it all at once.
  - **`pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`**
- Wait/signal on a condition.
  - **`pthread_cond_wait( &cond, &lock );`**
  - **`pthread_cond_signal( &cond );`**
- A handy function for waking everyone.
  - **`pthread_cond_broadcast( &cond );`**

Main thread wants to wait for its spawned thread

```
void *child(void *arg) {
  printf("child\n");
  // XXX how to indicate we are done?
  return NULL;
}

int main(int argc, char *argv[]) {
  printf("parent: begin\n");
  pthread_t c;
  pthread_create(&c, NULL, child, NULL);
  // XXX how to wait for child?
  printf("parent: end\n");
  return 0;
}
```

Main thread wants to wait for its spawned thread

```
volatile int done = 0;

void *child(void *arg) {
  printf("child\n");
  done = 1;
  return NULL;
}

int main(int argc, char *argv[]) {
  printf("parent: begin\n");
  pthread_t c;
  pthread_create(&c, NULL, child, NULL);
  while (done == 0)
    ; // spin
  printf("parent: end\n");
  return 0;
}
```

thread_exit();

thread_join();

Implementing Join with CVs (Correct)

```
void thread_exit() {
  Mutex_lock(&m);
  done = 1;             // a
  Cond_signal(&c);      // b
  Mutex_unlock(&m);
}

void thread_join() {
  Mutex_lock(&m);       // w
  while (done == 0)     // x
    Cond_wait(&c, &m); // y
  Mutex_unlock(&m);     // z
}
```

Implementing Join with CVs (Wrong 1)

```
void thread_exit() {
  Mutex_lock(&m);    // a
  Cond_signal(&c);   // b
  Mutex_unlock(&m);  // c
}

void thread_join() {
  Mutex_lock(&m);    // x
  Cond_wait(&c, &m); // y
  Mutex_unlock(&m);  // z
}
```

Implementing Join with CVs (Wrong 2)

```
void thread_exit() {
  done = 1;              // a
  Cond_signal(&c);       // b
}

void thread_join() {
  if (done == 0)         // x
    Cond_wait(&c);       // y
}
```

Implementing Join with CVs (Wrong 3)

```
void thread_exit() {
  done = 1;             // a
  Cond_signal(&c);      // b
}

void thread_join() {
  Mutex_lock(&m);       // w
  if (done == 0)        // x
    Cond_wait(&c, &m);  // y
  Mutex_unlock(&m);     // z
}
```

Implementing Join with CVs (Correct?)

```
void thread_exit() {
  Mutex_lock(&m);
  done = 1;              // a
  Cond_signal(&c);       // b
  Mutex_unlock(&m);
}

void thread_join() {
  Mutex_lock(&m);        // w
  if (done == 0)         // x
    Cond_wait(&c, &m);   // y
  Mutex_unlock(&m);      // z
}
```

Good Rule of Thumb

- Keep state in addition to CVs!
  - CVs are used to nudge threads when state changes.
  - If state is already as needed, don’t wait for a nudge!
- Always do wait and signal while holding the lock!

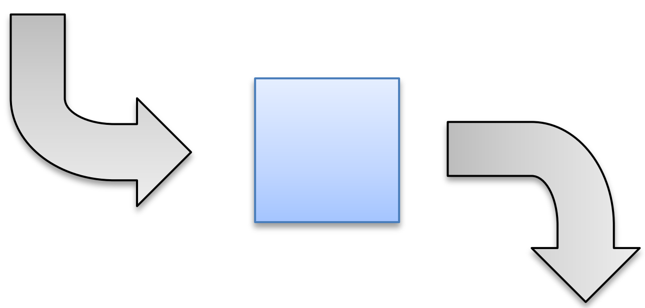
Alternative Solution for Producer/Consumer

```
int buffer;
int count = 0;
```

```
void put(int value) {
  assert(count == 0);
  count = 1;
  buffer = value;
}
```

```
int get() {
  assert(count == 1);
  count = 0;
  return buffer;
}
```

producer: put()



consumer: get()

Solution V1 (Broken)

```
void *producer(int loops) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);            //p1
    if (count == 1)            //p2
      Cond_wait(&C, &m);       //p3
    put(i);                    //p4
    Cond_signal(&C);           //p5
    Mutex_unlock(&m);          //p6
  }
}
```

```
void *consumer(int loops) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);          //c1
    if (count == 0)          //c2
      Cond_wait(&C, &m);     //c3
    int tmp = get();         //c4
    Cond_signal(&C);         //c5
    Mutex_unlock(&m);        //c6
    printf("%d\n", tmp);
  }
}
```

Thread Trace: Solution V1 (Broken)

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

Solution V2 (Better, But Still Broken)

```
void *producer(int loops) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);               //p1
    while (count == 1)            //p2
      Cond_wait(&C, &m);          //p3
    put(i);                       //p4
    Cond_signal(&C);              //p5
    Mutex_unlock(&m);             //p6
  }
}
```

```
void *consumer(int loops) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);          //c1
    while (count == 0)       //c2
      Cond_wait(&C, &m);     //c3
    int tmp = get();         //c4
    Cond_signal(&C);         //c5
    Mutex_unlock(&m);        //c6
    printf("%d\n", tmp);
  }
}
```

Thread Trace: Solution V2 (Broken)

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Better Solution (usually): use two CVs
Solution V3

```
void *producer(int loops) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);            //p1
    while (count == 1)         //p2
      Cond_wait(&E, &m);       //p3
    put(i);                    //p4
    Cond_signal(&F);           //p5
    Mutex_unlock(&m);          //p6
  }
}
```

```
void *consumer(int loops) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);          //c1
    while (count == 0)       //c2
      Cond_wait(&F, &m);     //c3
    int tmp = get();         //c4
    Cond_signal(&E);         //c5
    Mutex_unlock(&m);        //c6
    printf("%d\n", tmp);
  }
}
```

Summary: rules of thumb

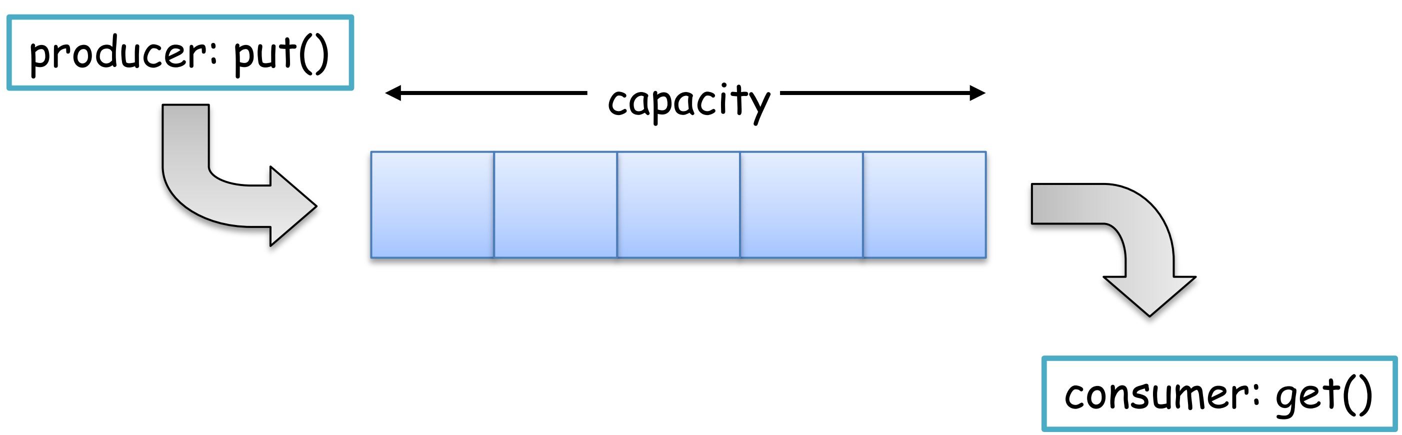
- Keep state in addition to CVs
- Always do wait/signal with lock held
- Whenever you acquire a lock, recheck state

Final Correct Solution for Producer/Consumer

```
int buffer[MAX];
int fill= 0;
int use = 0;
int count = 0
```

```
void put(int value) {
  buffer[fill] = value;
  fill = (fill + 1) % max;
  count ++;
}
```

```
int get() {
  int tmp = buffer[use];
  use = (use + 1) % max;
  count--;
  return tmp;
}
```



Solution V4 (Final)

```
void *producer(void *arg) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);             //p1
    while (count == max)        //p2
      Cond_wait(&E, &m);        //p3
    put(i);                     //p4
    Cond_signal(&F);            //p5
    Mutex_unlock(&m);           //p6
  }
}
```

```
void *consumer(void *arg) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);             //c1
    while (count == 0)          //c2
      Cond_wait(&F, &m);        //c3
    int tmp = get();            //c4
    Cond_signal(&E);            //c5
    Mutex_unlock(&m);           //c6
    printf("%d\n", tmp);
  }
}
```

How to wake the right thread?

- wait(cond_t *cv, mutex_t *lock)
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
- signal(cond_t *cv)
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing
- broadcast(cond_t *cv)
  - wake all waiting threads (if >= 1 thread is waiting)
  - if there are no waiting thread, just return, doing nothing

Another Correct Solution V5, But Not As Good As V4

```
void *producer(int loops) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);          //p1
    while (count == max)     //p2
      Cond_wait(&C,&m);      //p3
    put(i);                  //p4
    Cond_ broadcast(&C);     //p5
    Mutex_unlock(&m);        //p6
  }
}
```

```
void *consumer(int loops) {
  for (int i=0; i < loops; i++){
    Mutex_lock(&m);            //c1
    while (count == 0)         //c2
      Cond_wait(&C,&m);        //p3
    int tmp = get();           //c4
    Cond_ broadcast(&C);       //c5
    Mutex_unlock(&m);          //c6
    printf("%d\n", tmp);
  }
}
```

Readers-Writers Problem

- Make a distinction for type of concurrent access
  - Two or more threads read concurrently ☺
  - Two or more threads write concurrently ☹
  - A thread writes and some read concurrently ☹
  - Good for controlling access to database fields
  - If we can implement this, we can get more concurrency
- Formally:
  - nw = number of active writers
  - nr = number of active readers
  - Safety condition: ($nr == 0$ and $nw \leq 1$) or $nw == 0$

Reader-Writer Locks

1

```
typedef struct _rwlock_t {
  sem_t lock;
  sem_t writelock;
  int readers;
} rwlock_t;
void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}
```

2

```
acquire_r (rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}
```

3

```
release_r (rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}
```

release lock

4

```
acquire_w (rwlock_t *rw) {
  sem_wait (&rw->writelock);
}
release_w (rwlock_t *rw) {
  sem_post (&rw->writelock);
}
```

What Have We Learned?

- Potential for race conditions
- Critical section problem and DIY solutions
- Synchronization mechanisms
  - Semaphores
  - Mutex + CVs
- Classic problems
  - Bounded-Buffer
  - Dining-Philosophers
  - Readers-Writers