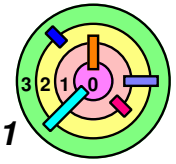
Ch 5: Synchronizing Access to Shared Objects

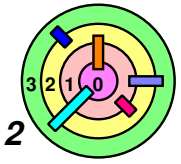
Bill Cheng

http://merlot.usc.edu/william/usc/



Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - behavior changes when re-run program
    - does it matter which thread runs first?
    - when would it be considered the behavior wrong/incorrect?
    - programs need to work for any possible interleaving
- Compiler/hardware instruction reordering
- Multi-word operations (such as `memcmp()`) are not atomic



Compiler/Hardware Can Reorder Instructions

- Modern compilers (and hardware) reorder instructions to improve performance

Thread 1

```
p = someComputation()
pInitialized = true;
```

Thread 2

```
while (!pInitialized)
    ;
q = anotherComputation(p)
```

  - can thread 2 use p before p is initialized?
    - doesn't look like it's possible, right?!

- If you have optimization turned on when you compile, the compiler may decide to do the following (since it doesn't understand that `p` and `pInitialized` are semantically related):

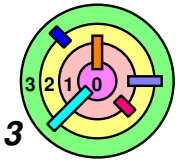
Thread 1

```
pInitialized = true;
p = someComputation()
```

Thread 2

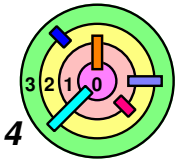
```
while (!pInitialized)
    ;
q = anotherComputation(p)
```

  - clearly, this is no good

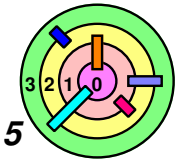


Why Reordering?

- Why do compilers reorder instructions?
  - efficient code generation requires analyzing control/data dependency
- Why do CPUs reorder instructions?
  - write buffering: allow next instruction to execute while write is being completed
- Fix: *memory barrier* (a.k.a. membar or memory fence)
  - instruction to compiler/CPU
  - all operations before barrier complete before barrier returns
  - no operation after barrier starts until barrier returns



(5.1) Challenges



Race Condition

- A *race condition* occurs when the behavior of a program depends on the interleaving of operations of different threads

Thread 1	Thread 2
`x = 1;`	`x = 2;`

  - possible final values of `x` are 1 or 2

- Ex: y is initialized to 12

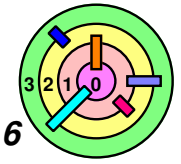
Thread 1	Thread 2
`x = y + 1;`	`y = y * 2;`

  - possible final values of `x` are 13 or 25

- Ex: x is initialized to 0

Thread 1	Thread 2
`x = x + 1;`	`x = x + 2;`

  - possible final values of `x` are 1, 2, and 3



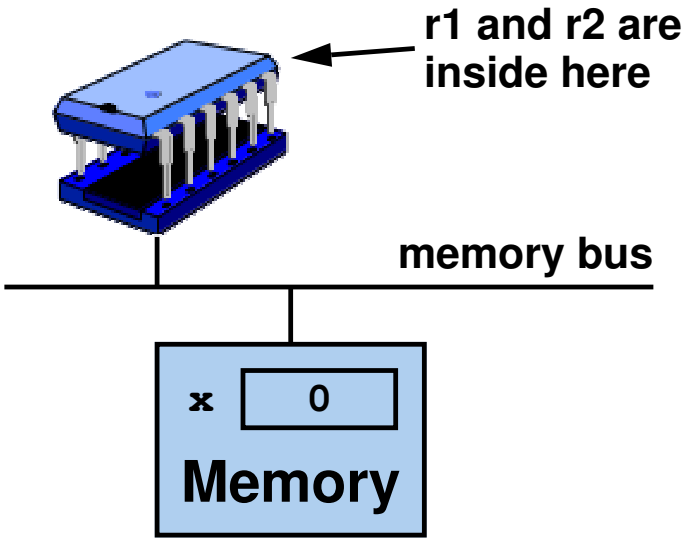
Race Condition

Thread 1:

```
x = x + 1;
 /*
  load  r1,x
  add   r2,r1,1
  store x,r2
  */
```

Thread 2:

```
x = x + 2;
 /*
  load  r1,x
  add   r2,r1,1
  store x,r2
  */
```

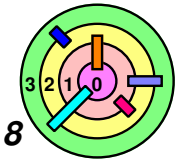


- Unfortunately, processors do not execute high-level language statements
  - they execute machine instructions
  - if thread 1 executes the first (or two) machine instructions
  - context switch can happen (to run a different thread)
    - this *can* happen if you have a *preemptive scheduler*
  - then thread 2 executes all 3 machine instructions
  - then later thread 1 executes the remaining machine instructions
  - `x` would end up to be 1

- Note: load and store are *atomic* (indivisible) operations

Too Much Milk Problem

- Two roommates want to make sure that the refrigerator is always well stocked with milk
  - what's the algorithm for each roommate?
- Correctness property
  - *liveness:* the program eventually enters a good state
    - if there is no milk, eventually someone would buy milk
  - *safety:* the program never enters a bad state
    - must not end up with more than one milk
- Unless otherwise specified, we will always assume that *neither the compiler nor the architecture reorders instructions*



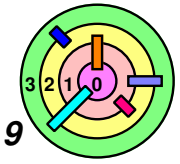
Too Much Milk Try #1: Leave A Note

- Algorithm:

```
// thread A or thread B
if(milk == 0){    // if no milk
  if(note == 0){  // if no noke
    note = 1;     // leave a note
    milk++;       // buy milk
    note = 0;     // remove note
  }
}
```

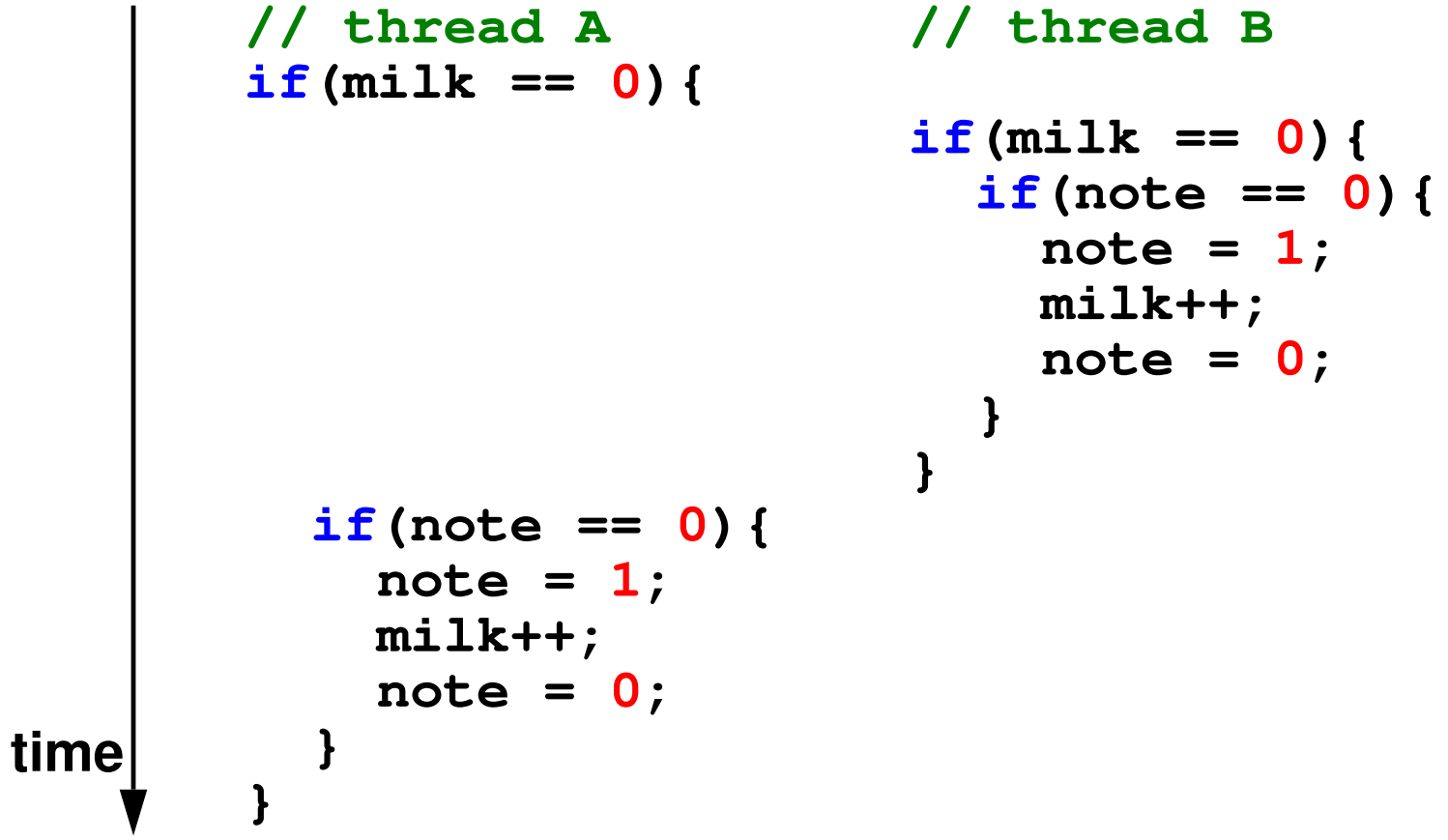
  - which statements are *atomic*?
    - the assumption here is that if a statement only access zero or one memory location, it's an atomic operation (because it cannot be preempted in the middle of that operation)

- Q: Does the above solution guarantees *safety* and *liveness*?



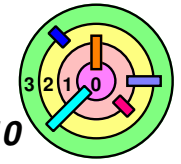
Too Much Milk Try #1: Leave A Note

- This solution satisfies liveness but violates safety



  - in this scenario, milk is 2 at the end

- Heisenbug!
  - occasionally fail in ways that may be difficult to reproduce

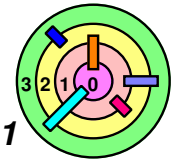


Too Much Milk Try #2: Two Notes

Algorithm:

```
// thread A
noteA = 1;          // leave note
if(noteB == 0){     // if no note
  if(milk == 0){    // if no milk
    milk++;         // buy milk
  }
}
noteA = 0;          // remove note

// thread B
noteB = 1;          // leave note
if(noteA == 0){     // if no note
  if(milk == 0){    // if no milk
    milk++;         // buy milk
  }
}
noteB = 0;          // remove note
```

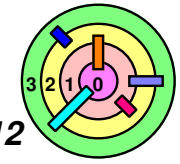
- does this solution guarantees ***safety*** and ***liveness***?



Too Much Milk Try #2: Two Notes

- To *prove safety*, need to look at all possible interleaving
  - *proof by contradiction:* assuming that the algorithm is *not safe*, i.e., both A and B will buy milk
- Consider the state of the two variables noteB and milk when thread A is at [A1]
  - given the assumption, thread A will be at [A3] and thread B will be at [B3]

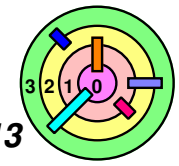
```
        // thread A
        noteA = 1;
[A1]    if(noteB == 0){
[A2]      if(milk == 0){
[A3]        milk++;
          }
        }
        noteA = 0;

        // thread B
        noteB = 1;
[B1]    if(noteA == 0){
[B2]      if(milk == 0){
[B3]        milk++;
[B4]      }
[B5]    }
        noteB = 0;
```



Too Much Milk Try #2: Two Notes

- To *prove safety*, need to look at all possible interleaving
  - *proof by contradiction:* assuming that the algorithm is *not safe*, i.e., both A and B will buy milk
- Consider the state of the two variables noteB and milk when thread A is at [A1]
  - given the assumption, thread A will be at [A3] and thread B will be at [B3]
- Case 1: noteB = 1, milk = don't care
  - contradiction, thread A will not reach [A3]
- Case 2: noteB = 0, milk > 0
  - contradiction, thread A will not reach [A3]
- Case 3: noteB = 0, milk = 0
  - contradiction, thread B will not reach [B3] ¤

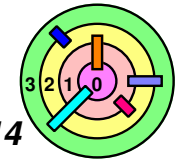
```
        // thread A
        noteA = 1;
[A1]    if(noteB == 0){
[A2]      if(milk == 0){
[A3]        milk++;
          }
        }
        noteA = 0;

        // thread B
        noteB = 1;
[B1]    if(noteA == 0){
[B2]      if(milk == 0){
[B3]        milk++;
[B4]      }
[B5]    }
        noteB = 0;
```



Too Much Milk Try #2: Two Notes

- Canno prove *liveness*
  - if thread A executes [A0] and switch to thread B to execute [B0], or vice versa, both will not buy milk

```
        // thread A
[A0]    noteA = 1;
[A1]    if(noteB == 0){
[A2]      if(milk == 0){
[A3]        milk++;
          }
        }
        noteA = 0;

        // thread B
[B0]    noteB = 1;
[B1]    if(noteA == 0){
[B2]      if(milk == 0){
[B3]        milk++;
[B4]      }
[B5]    }
        noteB = 0;
```

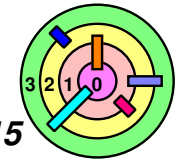


Too Much Milk Try #3: Waiting

Algorithm:

```
// thread A
noteA = 1;                  // leave noteA
while(noteB == 1){          // if no note from roommate
  ;                         // spin
}
if(milk == 0){              // if no milk
  milk++;                   // buy milk
}
noteA = 0;                  // remove noteA

// thread B
noteB = 1;                  // leave note
if(noteA == 0){             // if no note from roommate
  if(milk == 0){            // if no milk
    milk++;                 // buy milk
  }
}
noteB = 0;                  // remove note
```

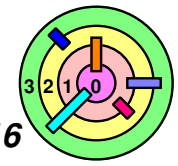
- does this solution guarantees *safety* and *liveness*?



Too Much Milk Try #3: Waiting

- Can prove ***safety*** using a similar argument for solution 2
  - case 1: noteB = 1, milk = don't care
    - contradiction, B will not buy milk
  - case 2: noteB = 0, milk > 0
    - contradiction, A will not buy milk
  - case 3: noteB = 0, milk = 0
    - contradiction, B will not buy milk ¤
- ***Liveness:*** since thread B has no loop, noteB will eventually be 0 and thread A will get to decide to buy milk or not
- Solution 3 has both safety and liveness using only atomic load and store operations

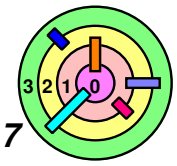
```
// thread A
noteA = 1;
while(noteB == 1) {
  ;
}
if(milk == 0){
  milk++;
}
noteA = 0;

// thread B
noteB = 1;
if(noteA == 0){
  if(milk == 0){
    milk++;
  }
}
noteB = 0;
```



Too Much Milk Try #3: Waiting

- Is solution 3 a "good" solution?
  - issues:
    - solution is complex (why the asymmetry?)
      - there is something called Peterson's algorithm that would work more generally
    - solution is inefficient: thread A is doing *busy-waiting* and consuming CPU resource
    - solution may fail if the compiler or hardware reorders instructions (although this limitation can be addressed by using memory barriers, which would increase the implementation complexity of the algorithm)

```
// thread A
noteA = 1;
while(noteB == 0)
  ;
}
if(milk == 0){
  milk++;
}
noteA = 0;

// thread B
noteB = 1;
if(noteA == 0){
  if(milk == 0){
    milk++;
  }
}
noteB = 0;
```



Too Much Milk: Use Synchronization Objects

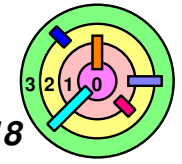
- ***Lock:*** a primitive that only one thread at a time can own

```
// thread A or thread B
Kitchen::buyIfNeeded() {
  lock.acquire();
  if (milk == 0) {
    milk++;
  }
  lock.release();
}
```

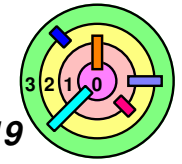
  - simple and symmetrical

- Unless otherwise specified, we use the term ***lock*** and ***mutex*** interchangeably (although in general, a lock may allow multiple threads to have concurrent access to a resource)

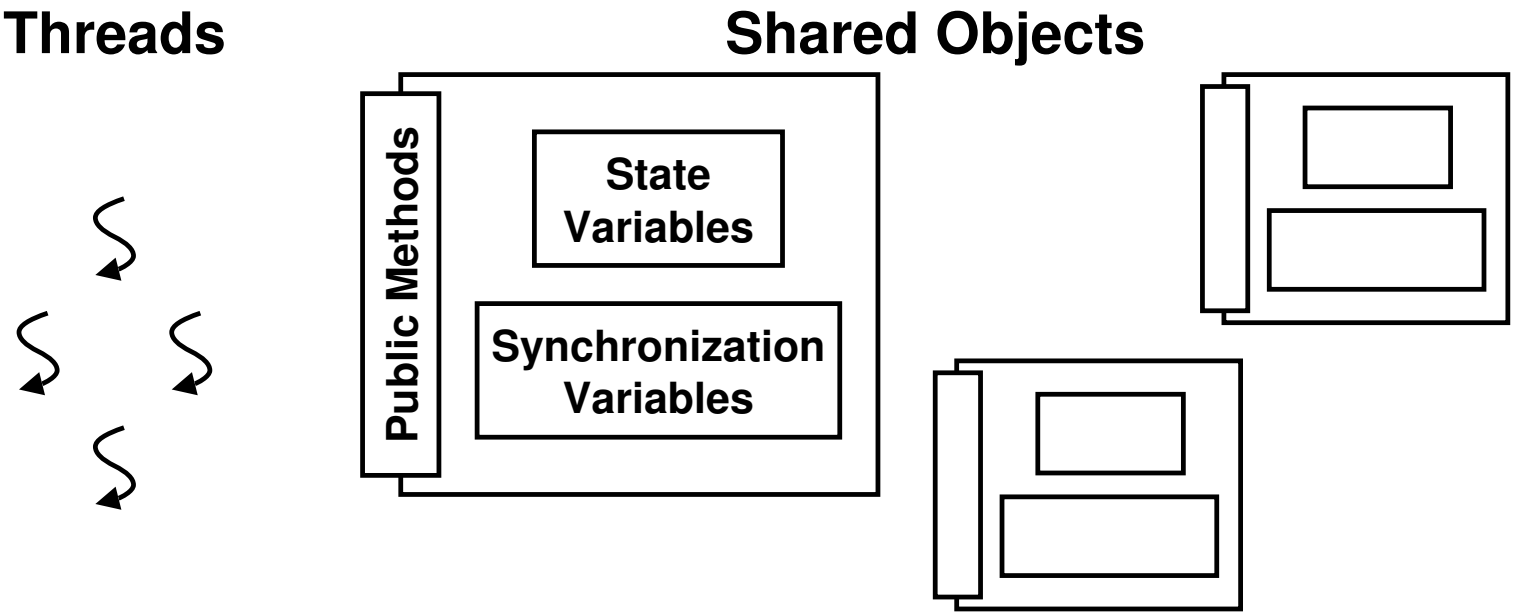
```
// thread A or thread B
Kitchen::buyIfNeeded() {
  mutex.lock();
  ...
  mutex.unlock();
}
```



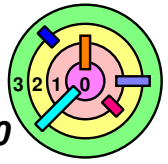
(5.2) Structuring Shared Objects



Threads And Shared Objects

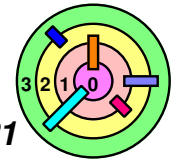


- In a multi-threaded program, threads are separate from shared objects and operate concurrently on shared objects
  - shared objects contain both shared state and *synchronization variables* (for controlling concurrent access to shared state)
- *Shared objects:* objects that can be accessed *safely* by *multiple threads*
  - all shared state in a program should be encapsulated in one or more shared objects



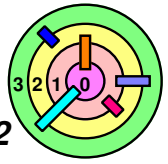
Monitors

- When a programming language includes support for shared objects, a shared object is often called a *monitor*
  - a monitor is a synchronization construct that allows executing entities to have both mutual exclusion and the ability to wait/block for a certain condition to become true
- Early programming languages with monitors include Birnch Hansen's Concurrent Pascal and Xerox PARC's Mesa
  - today, Java supports monitors via the `synchronized` keyword



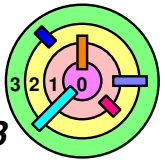
Shared Objects Are Implemented In Layers

Concurrent Applications:			
Shared Objects:	**Bounded Buffer**	**Readers/Writers**	**Barrier**
Synchronization Variables:	**Semaphores**	**Locks**	**Condition Variables**
Atomic Instructions:	**Interrupt Disable**	**Test-and-Set**	
Hardware:	**Multiple Processors**	**Hardware Interrupts**	



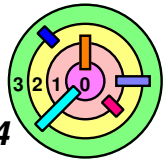
(5.3) Locks: Mutual Exclusion

Synchronization Variables:	Semaphores	*Locks*	Condition Variables



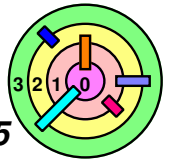
Locks

- A *lock* is a synchronization variable that provides mutual exclusion (when one thread holds a lock, no other thread can hold it, i.e., other threads are excluded)
  - while holding a lock, a thread can perform an arbitrary set of operations
    - those operations *appear to be atomic* to other threads
      - no other thread can observe an intermediate state
      - other threads can only observe the state after the lock is released
- A program associates each lock with some subset of shared state and requires a thread to hold the lock when accessing that state
  - as a result, only one thread can access the shared state at a time



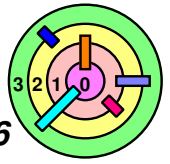
Locks: API and Properties

- A lock enables mutual exclusion by providing two methods: `Lock::acquite()` and `Lock::release()`
  - a lock can be in one of two states: BUSY or FREE
  - a lock is initially in the FREE state
  - `Lock::acquire()` waits until the lock is FREE and then *atomically* makes the lock BUSY
    - seeing the state is FREE *and* setting the state to BUSY are together an *atomic operation*
    - if multiple threads try to acquire the lock, at most one thread will succeed
      - one thread observes that the lock is FREE and sets it to BUSY while other threads just see that the lock is BUSY
  - `Lock::release()` makes the lock FREE
    - if there are pending `acquire()` operations, this state change causes *one* of them to proceed



Locks: API and Properties

- A lock implementation should ensure the following three properties
  - *mutual exclusion:* at most one thread holds the lock
    - this is a *safety* property - locks prevent more than one thread from accessing shared state
  - *progress:* if no thread holds the lock and any thread attempts to acquire the lock, then eveutually some thread succeeds in acquiring the lock
    - this is a *liveness* property - if a lock is FREE, some thread must be able to acquire it
  - *bounded waiting:* if a thread T attempts to acquire a lock, then there exists a bound on the number of times other threads can successfully acquire the lock before T does
    - this is a *liveness* property - any particular thread that wants to acquire the lock must eventually succeed in doing so
- Non-property: thread ordering
  - no promise that waiting threads acquire the lock in FIFO order



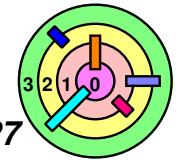
Case Study: Thread-Safe Bounded Queue

- Use a fixed size buffer to implement a FIFO queue

```
tryget() {
  item = NULL;
  lock.acquire();
  if (front < tail) {
    item = buf[front % MAX];
    front++;
  }
  lock.release();
  return item;
}
```

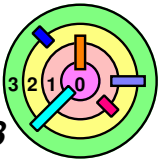
```
tryput(item) {
  lock.acquire();
  if ((tail - front) < size) {
    buf[tail % MAX] = item;
    tail++;
  }
  lock.release();
}
```

- initially, `front=tail=0`, `lock=FREE`, `buf[MAX]`
- for simplicity, assume no wraparound/overflow on array index
- `front` = total number of items removed
- `tail` = total number of items inserted/appended
- a thread cannot know the state of the bounded queue/buffer unless it's holding the lock
  - if `tryget()` returns NULL, we can only conclude that the buffer ***was*** empty



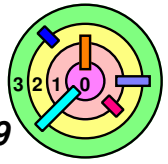
Critical Section

- A *critical section* is a sequence of code that *atomically* accesses shared state
  - a critical section with respect to lock L is code executed when holding lock L (code between `L.acquire()` and `L.release()`)



(5.4) Condition Variables: Waiting for a Change

Synchronization Variables:	Semaphores	Locks	*Condition Variables*

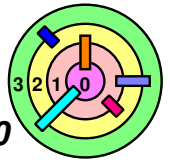


Condition Variables (CV)

- ***Busy waiting:***

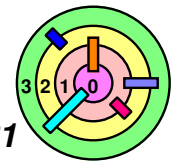
```
get()
{
  while ((data = tryget()) == NULL) ;
  return data;
}
```

- The right way to wait for a shared state variable to change value is to go *sleep on a queue* *(i.e., a condition variable queue)* and wait for a *wake up* call *(i.e., a notification)*
  - these threads are working together and helping each other
- Waiting inside a critical section
  - called only when holding a lock
- Wait: *atomically* release lock, placing the thread on the CV queue, and suspend the execution of the calling thread
  - *reacquire the lock* when wakened
- Signal: wake up a waiting thread, if any
- Broadcast: wake up all waiting threads, if any



Condition Variable Design Pattern

```
methodThatWaits() {
   lock.acquire();
   // read/write shared state
   while (!testSharedState()) {
      cv.wait(lock);
   }
   // read/write shared state
   lock.release();
}

methodThatSignals() {
   lock.acquire();
   // read/write shared state
   // if testSharedState() is true
   cv.signal(lock);
   // read/write shared state
   lock.release();
}
```

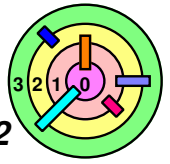


Example: Bounded Queue/Buffer

```
get() {
  lock.acquire();
  while (front==tail){
    empty.wait(lock);
  }
  item = buf[front%MAX];
  front++;
  full.signal(lock);
  lock.release();
  return item;
}
```

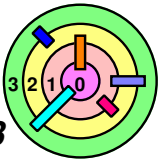
```
put(item) {
  lock.acquire();
  while ((tail-front)==MAX){
    fullf.wait(lock);
  }
  buf[tail%MAX] = item;
  tail++;
  empty.signal(lock);
  lock.release();
}
```

- Two CV queues
  - `empty`: threads sleep here because the buffer is empty (nothing to get, nothing to work on)
    - `empty.signal()` if the buffer is no longer empty
  - `full`: threads sleep here because the buffer is full (cannot add work, no space)
    - `full.signal()` if the buffer is no longer full



Pre/Post Conditions

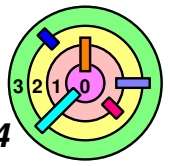
```
methodThatWaits() {
  lock.acquire();
  // pre-condition: State is consistent
  // read/write shared state
  while (!testSharedState()) {
    cv.wait(lock);
  }
  // WARNING: shared state may have changed,
  //      but testSharedState() is true and
  //      pre-condition is true
  // read/write shared state
  lock.release();
}

methodThatSignals() {
  lock.acquire();
  // pre-condition: State is consistent
  // read/write shared state
  // if testSharedState() is true
  cv.signal(lock);
  // NO WARNING: signal keeps lock
  // read/write shared state
  lock.release();
}
```



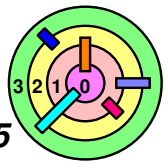
Condition Variables

- Always hold lock when calling `wait()`, `signal()`, `broadcast()`
  - always hold lock when accessing shared state
- Condition variable is memoryless
  - if signal when no one is waiting, it's as if nothing has happened
  - if wait before signal, waiting thread wakes up
- `wait()` atomically releases lock
- When a thread is woken up from `wait()`, it may not run immediately
  - `signal()/broadcast()` put the thread on the ready list
  - when lock is released, any waiting thread might acquire it
  - *lock is reacquired* before `wait()` returns
- `wait()` must be called in a loop since spurious wakeup can occur

```
while (needToWait()) {
  cv.wait(lock);
}
```



(5.5) Designing and Implementing Shared Objects

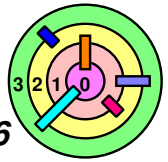


Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
- Add locks to object/module
  - grab lock on start to every method/procedure
  - release lock on finish
- If need to wait:

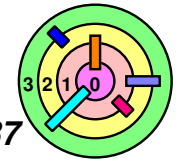
```
while (needToWait()) {
  cv.wait(lock);
}
```

  - do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - `signal()` or `broadcast()`
- Always leave shared state variables in a consistent state when lock is released, or when waiting



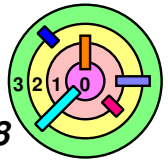
Remember The Rules

- Use consistent structure
- Always 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()`



Mesa vs. Hoare Semantics

- **Mesa**
  - `signal()` puts waiting thread on the ready list
  - signaller keeps lock and processor
- **Hoare**
  - `signal()` gives processor and lock to waiting thread
  - when waiting thread finishes, processor/lock given back to signaller
  - nested signals possible
- In general, Mesa semantics makes it easier to write application code, and therefore, more widely used
  - with Hoare semantics, liveness properties is easier to prove
- We generally use the term *monitor* to mean that we are using *locks* and *CVs* for synchronization
  - *message-passing* is another way for synchronization

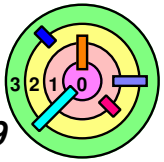


Bounded Buffer (Mesa Semantics)

```
get() {
  lock.acquire();
  while (front==tail){
    empty.wait(lock);
  }
  item = buf[front%MAX];
  front++;
  full.signal(lock);
  lock.release();
  return item;
}
```

```
put(item) {
  lock.acquire();
  while ((tail-front)==MAX){
    full.wait(lock);
  }
  buf[tail%MAX] = item;
  tail++;
  empty.signal(lock);
  lock.release();
}
```

This is what we had before

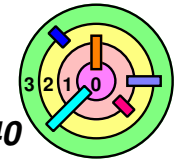


Bounded Buffer (Hoare Semantics)

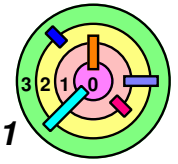
```
get() {
  lock.acquire();
  if (front==tail){
    empty.wait(lock);
  }
  item = buf[front%MAX];
  front++;
  full.signal(lock);
  lock.release();
  return item;
}
```

```
put(item) {
  lock.acquire();
  if ((tail-front)==MAX){
    full.wait(lock);
  }
  buf[tail%MAX] = item;
  tail++;
  empty.signal(lock);
  lock.release();
}
```

- No need to loop since the lock is ***transferred*** from the thread calling `empty.signal()` to the thread that was sleeping in `empty.wait()`
  - but we said before that `wait()` must be called in a loop since spurious wakeup is permitted to occur
    - under Hoare semantics, the implementation must make sure that spurious wakeup cannot occur (and this may be difficult to implement on some systems)

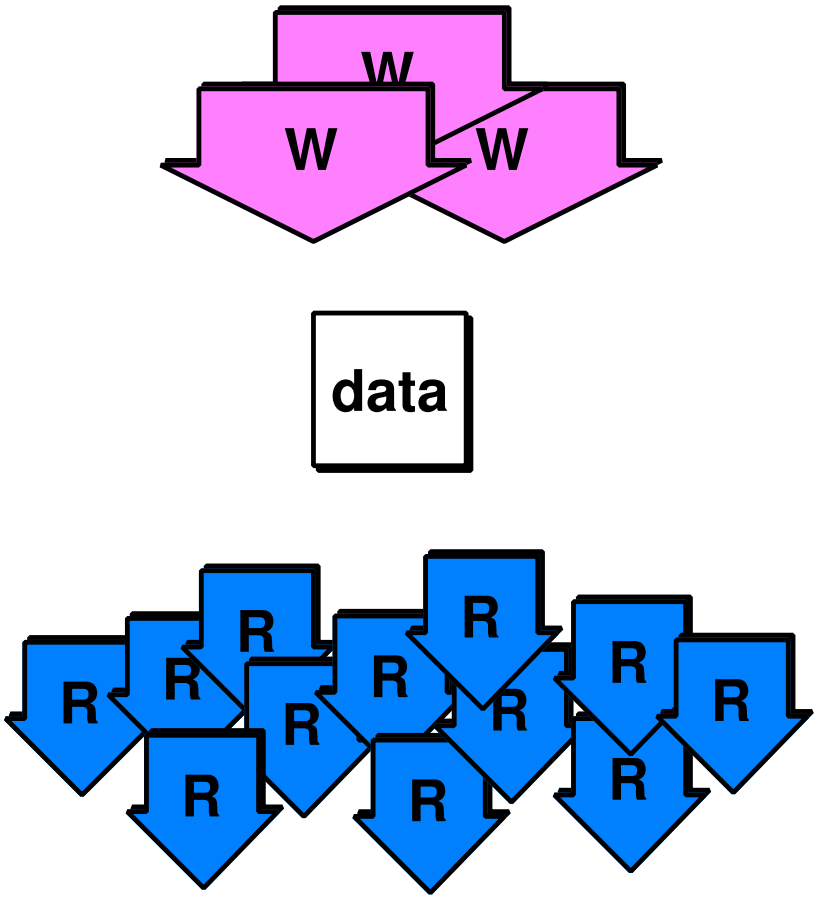


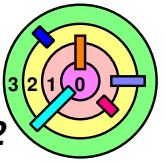
(5.6) Three Case Studies



Readers/Writers Lock

Readers/writers lock (RWLock): **a lock which allows multiple *reader* threads to access shared data concurrently, but still provides mutual exclusion whenever a *writer* thread is reading or modifying the shared data**



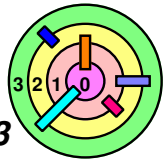


Readers/Writers Lock

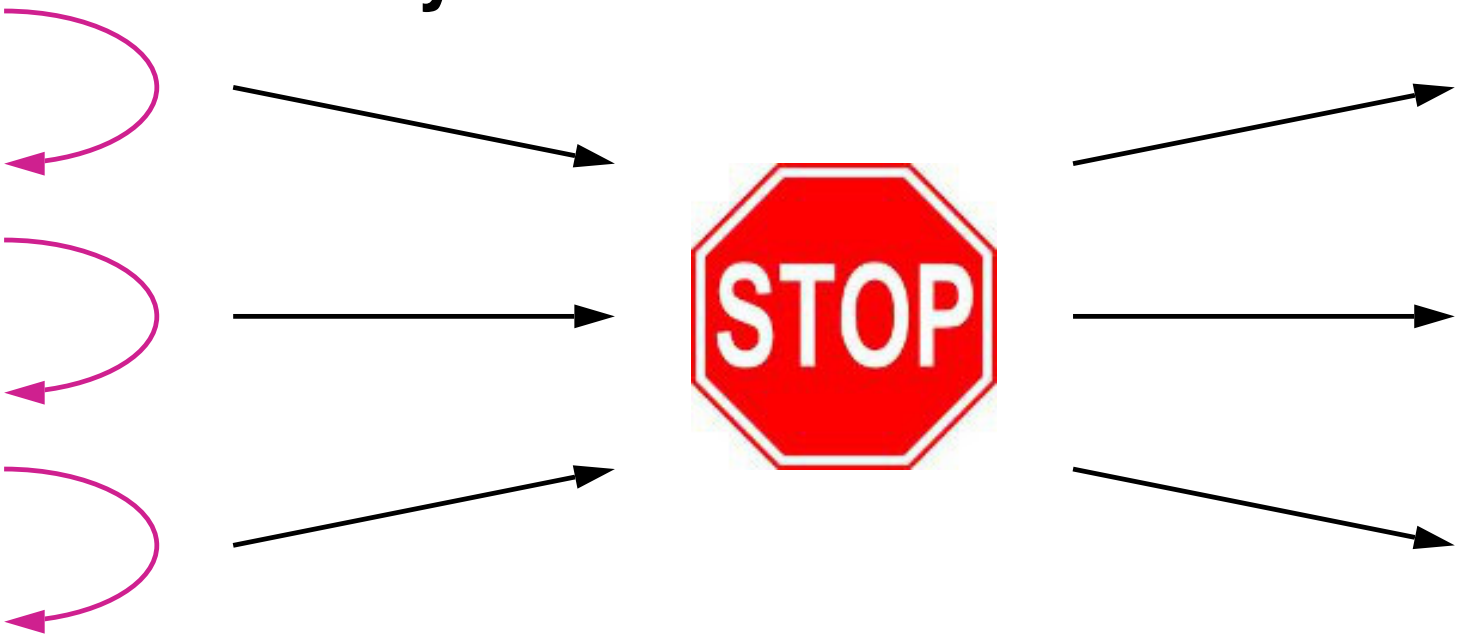
```
reader() {
  lock.acquire();
  waitingReaders++;
  while (!(writers==0))
    readersCV.wait(lock);
  waitingReaders--;
  activeReaders++;
  lock.release();
  /* read data */
  lock.acquire();
  if (--activeReaders==0)
    writersCV.signal(lock);
  lock.release();
}
```

```
writer() {
  lock.acquire();
  waitingWriters++;
  while(!((readers==0)
       && (writers==0)))
    writersCV.wait(lock);
  waitingWriters--;
  activeWriters++;
  lock.release();
  /* write data */
  lock.acquire();
  activeWriters--;
  assert(activeWriters==0);
  writersCV.signal(lock);
  readersCV.broadcast(lock);
  lock.release();
}
```

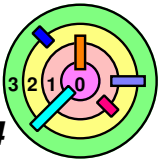
- readers read *concurrently*
- a writer writes *exclusively* (no concurrent reading or writing by other threads)



Synchronization Barriers

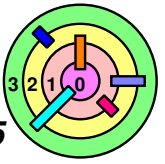


- When a thread is done with its work, it must call `checkin()`
  - `checkin()` doesn't return until all threads have checked in
- Ex: MapReduce, signal processing
- Synchronization barrier is differrent from a memory barrier
  - memory barrier is to synchronize memory operations for ***one thread***



MapReduce

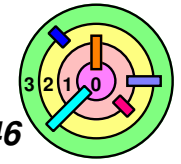
- MapReduce pseudo-code:
  - create N threads
  - create barrier to synchronize N threads
  - each thread executes map operation in parallel
  - `barrier.checkin()`
  - each thread sends data in parallel to reducers
  - `barrier.checkin()`
  - each thread executes reduce operation in parallel
  - `barrier.checkin()`



Synchronization Barriers

```
int numEntered = 0;
void checkin() {
  lock.acquire();
  if (++numEntered < barrierN) {
    while(numEntered < barrierN)
      barrierCV.wait(lock);
  } else {
    barrierCV.broadcast(lock);
  }
  lock.release();
}
```

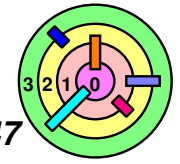
- The above implementation of `checkin()` results in a barrier that can only be used once
  - where can you reset `numEntered` to 0 so you can reuse this barrier?



Synchronization Barriers

- One solution is to use two barriers, one for checking in and one for checking out

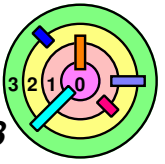
```
int numEntered = 0, numLeft = 0;
void checkin() {
  lock.acquire();
  if (++numEntered < barrierN) {
    while(numEntered < barrierN)
      checkinCV.wait(lock);
  } else {
    numLeft = 0;
    checkinCV.broadcast(lock);
  }
  if (++numLeft < barrierN) {
    while(numLeft < barrierN)
      checkoutCV.wait(lock);
  } else {
    numEntered = 0;
    checkoutCV.broadcast(lock);
  }
  lock.release();
}
```



Synchronization Barriers

A more efficient solution

```
int numEntered = 0, generation = 0;
void checkin() {
  lock.acquire();
  if (++numEntered < barrierN) {
    int my_generation = generation;
    while(my_generation == generation)
      barrierCV.wait(lock);
  } else {
    numEntered = 0;
    generation++;
    barrierCV.broadcast(lock);
  }
  lock.release();
}
```



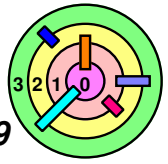
Starvation

- ***Starvation:*** the possibility that a thread doesn't get to run
  - the ***bounded buffer solution*** is not starvation-free

```
get() {
  lock.acquire();
  while (front==tail){
    empty.wait(lock);
  }
  item = buf[front%MAX];
  front++;
  full.signal(lock);
  lock.release();
  return item;
}
```

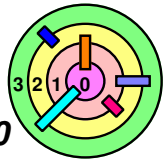
```
put(item) {
  lock.acquire();
  while ((tail-front)==MAX){
    fullf.wait(lock);
  }
  buf[tail%MAX] = item;
  tail++;
  empty.signal(lock);
  lock.release();
}
```

- Let's say that thread X calls `get()` and goes to sleep because the buffer is empty (and some other threads are doing the same thing)
  - starvation can happen if every time another thread calls `put()`, a 3rd thread that has called `get()` got the item and thread X always see an empty buffer



Starvation

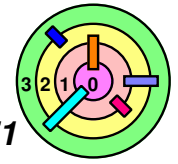
- The bounded buffer solution needs liveness constraints:
  - *starvation-freedom:* if a thread waits in `get()`, it is guaranteed to proceed after a bounded number of `put()` calls
  - *first-in-first-out (FIFO):* the k^{th} thread to acquire the lock in `get()` retrieves the item inserted by the k^{th} to acquire the lock in `put()`
    - this is a much stronger constraint than starvation-freedom
    - if the FIFO constraint is satisfied, starvation-freedom is guaranteed
    - you can build your own queue for the threads to sleep on (with a queue of CVs)



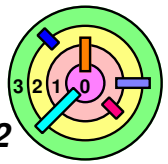
get() for FIFO Bounded Buffer

```
queue<CV> getQueue, putQueue;
int numGetCalled=0, numPutCalled=0;
get() {
  lock.acquire();
  myGetPos = numGetCalled++;
  myGetCV = new CV;
  getQueue.push_back(myGetCV);
  while (front<myGetPos || front==tail){
    myGetCV.wait(lock);
  }
  getQueue.delete_front();
  item = buf[front%MAX];
  front++;
  // wake up the next thread waiting in put(), if any
  nextPutThreadCV = putQueue.front();
  if (nextPutThreadCV != NULL) {
    nextPutThreadCV.signal(lock);
  }
  lock.release();
  return item;
}
```

- put() is similar

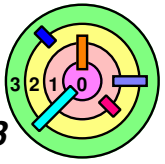


(5.7) Implementing Synchronization Objects



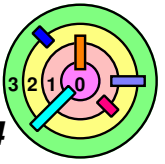
Shared Objects Are Implemented In Layers

Concurrent Applications:			
Shared Objects:	Bounded Buffer	Readers/Writers	Barrier
Synchronization Variables:	**Semaphores**	**Locks**	**Condition Variables**
Atomic Instructions:	**Interrupt Disable**	**Test-and-Set**	
Hardware:	**Multiple Processors**	**Hardware Interrupts**	



Implementing Synchronization

- We will talk about kernel thread synchronization first (and will briefly talk about user thread synchronization)
- Use memory load/store
  - see too much milk solution - too complex
- Modern systems have hardware support for:
  - *disabling interrupts:* on a uniprocessor, we can make any sequence of instructions atomic by disabling interrupts
  - *atomic read-modify-write instructions:* globally atomic instructions on a multiprocessor system
  - note that each of these primitives are also *memory barriers*, i.e., all prior instructions must complete before one of these instructions is executed

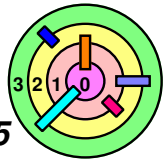


Implementing Locks By Disabling Interrupts

```
Lock::acquire() {
  disableInterrupts();
}
```

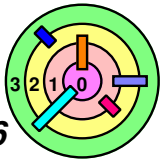
```
Lock::release() {
  enableInterrupts();
}
```

- This implementation does not provide mutual inclusion even on a uniprocessor (if the thread calls `yield()`)
- What else is wrong with the above implementation?
  - what if the thread calling `acquite()` doesn't yield the processor after the lock is acquired?
    - it's not a good idea to run the processor with interrupt disable for a long period of time
      - some interrupt needs attention of the processor right away
    - other threads do not get to run and make the system unresponsive to handling user inputs or other real-time tasks
    - malicious or buggy program can monopolize the processor



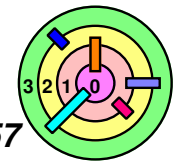
Implementing Uniprocessor Queueing Locks

- If the lock is BUSY, no point in running the acquiring thread until the lock is free
  - yield the processor (by calling `thread_switch()`)
- Still need to disable the interrupt briefly to protect the lock's data structure
  - need to re-enable the interrupt before we put the thread to sleep



Implementing Uniprocessor Queueing Locks

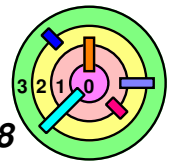
```
class Lock {
  private:
    int value = FREE;
    queue waiting; // hence the name "queueing lock"
  public:
    void acquire();
    void release();
};

Lock::acquire()
{
  disableInterrupts();
  if (value == BUSY) {
    waiting.add(runningThread);
    runningThread->state = WAITING;
    TCB *nextThread=readyList.remove();
    thread_switch(runningThread, nextThread);
    runningThread->state = RUNNING;
  } else {
    value = BUSY;
  }
  enable_interrupt();
}
```



Implementing Uniprocessor Queueing Locks

```
class Lock {
  private:
    int value = FREE;
    queue waiting; // hence the nar
  public:
    void acquire();
    void release();
};

Lock::acquire()
{
  disableInterrupts();
  if (value == BUSY) {
    waiting.add(runningThread);
    runningThread->state = WAITING;
    TCB *nextThread=readyList.remove();
    thread_switch(runningThread, nextThread);
    runningThread->state = RUNNING;
  } else {
    value = BUSY;
  }
  enable_interrupt();
}
```

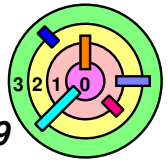
- our convention is that `thread_switch()` must only be called when interrupt is disabled
- if some code calls `thread_switch()`, it must disable interrupt first and enable interrupt when `thread_switch()` returns



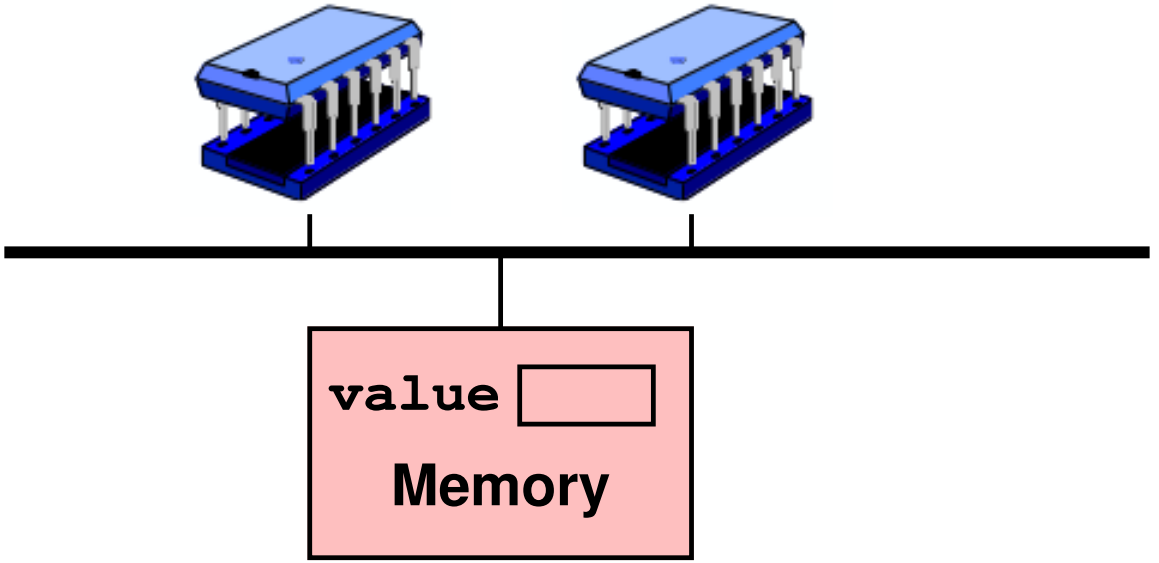
Implementing Uniprocessor Queueing Locks

```
Lock::release()
{
  disableInterrupts();
  if (!waiting.empty()) {
    TCB *nextThread=waiting.remove();
    nextThread->state = READY;
    readyList.add(nextThread);
  } else {
    value = FREE;
  }
  enable_interrupt();
}
```

⇨ Is it a bug that we don't change `value` to FREE when lock is released and the waiting queue is not empty?



Implementing Multiprocessor Spinlocks

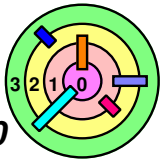


- When there are multiple processors, the difficulty lies in locking

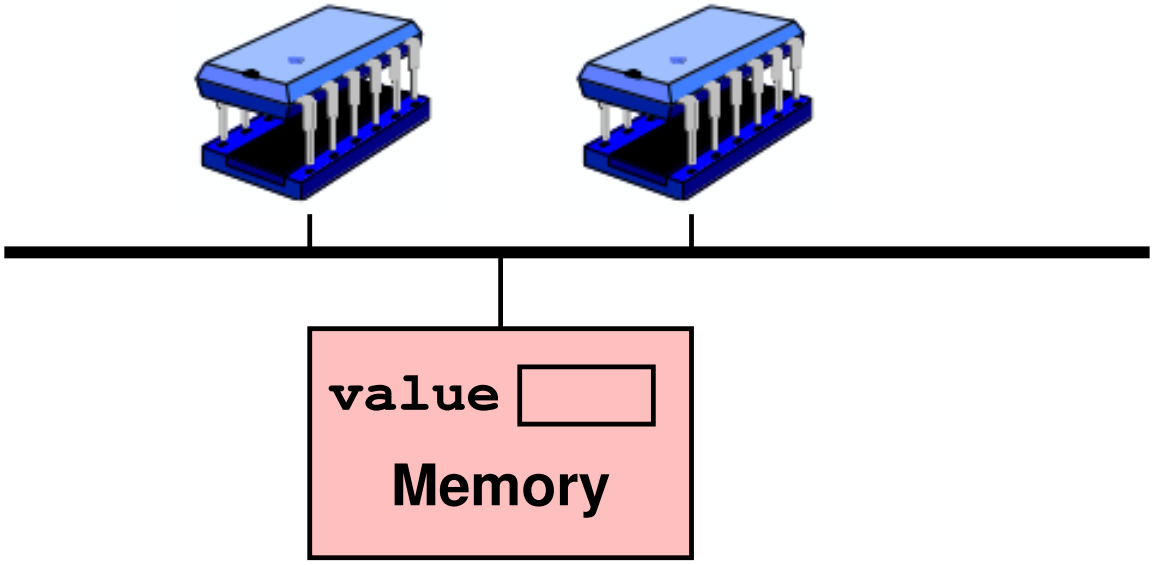
```
if (value == FREE) {
  value = BUSY;
}
```

  - if both threads execute the above code concurrently in different processors, both threads think they got the lock

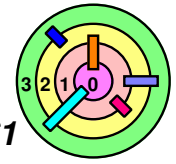
- No way to implement this with only software



Atomic Hardware Instructions



- Most processor architectures provide *atomic read-modify-write* instructions to support synchronization
  - *compare-and-swap* and *test-and-set* are other names for such an instruction (two operations locked together as an atomic instruction over the bus)



Atomic Test-And-Set Machine Instruction

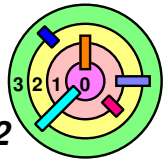
- Lock is represented as a bit, 0 if FREE, 1 if BUSY

```
bool test_and_set(int *lock)
{
  bool previous_value=*lock;
  *lock = BUSY;
  return previous_value;
}
```

  - if `test_and_set()` returns BUSY (i.e., 1), it means that the calling thread does *not* own the lock
  - if `test_and_set()` returns FREE (i.e., 0), it means that the calling thread is now the exclusive owner of the lock

```
Spinlock::acquire()
{
  // while BUSY
  while (test_and_set(&value))
    ; // spin
}
```

```
Spinlock::release()
{
  value = FREE;
  memory_barrier();
}
```

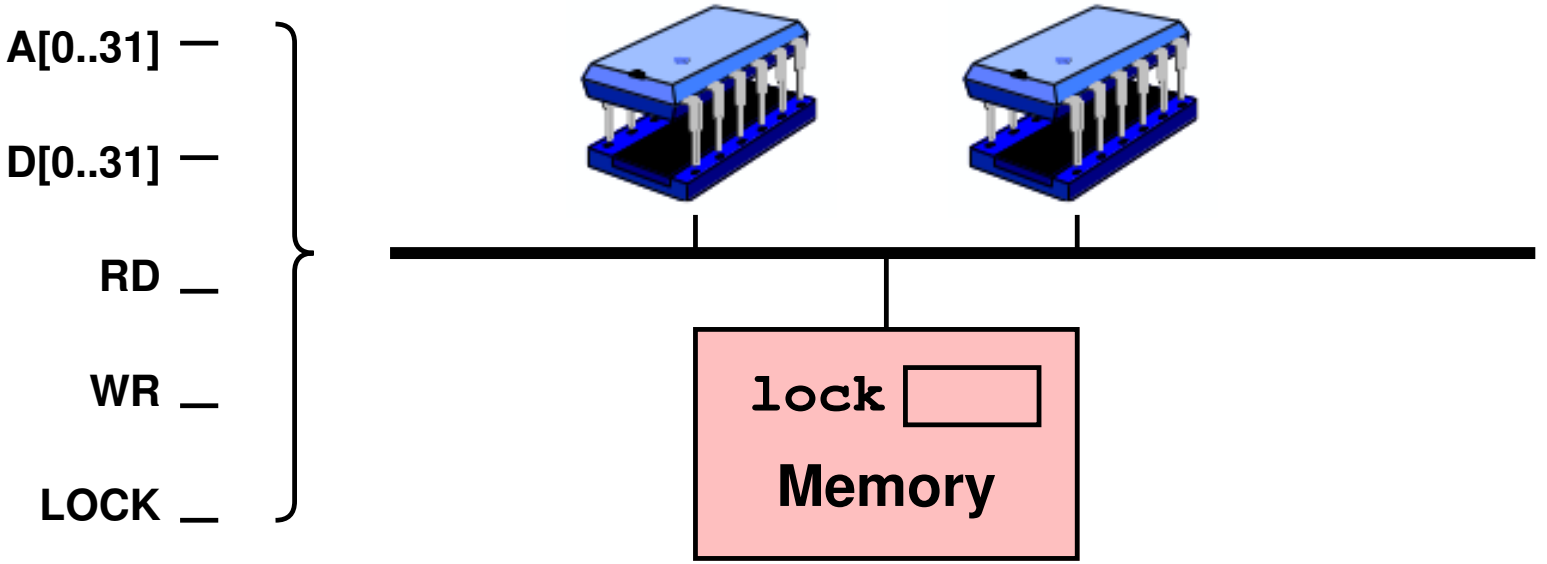


Atomic Test-And-Set Machine Instruction

- Lock is represented as a bit, 0 if FREE, 1 if BUSY

```
bool test_and_set(int *lock)
{
  bool previous_value=*lock;
  *lock = BUSY;
  return previous_value;
}
```

- e.g., `value` is *FREE*, call `test_and_set(&value)`

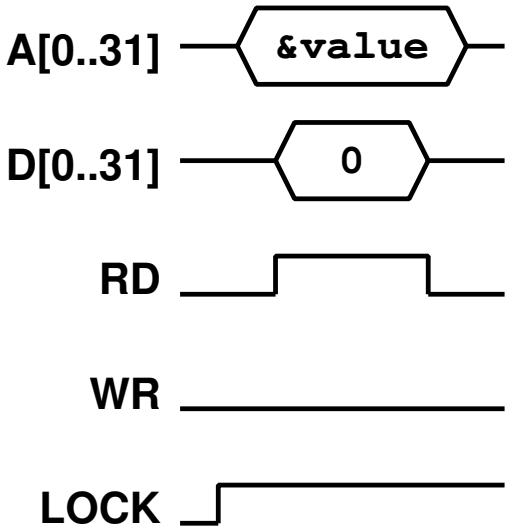


Atomic Test-And-Set Machine Instruction

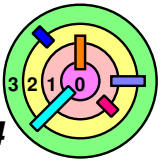
- Lock is represented as a bit, 0 if FREE, 1 if BUSY

```
bool test_and_set(int *lock)
{
  bool previous_value=*lock;
  *lock = BUSY;
  return previous_value;
}
```

  - e.g., `value` is *FREE*, call `test_and_set(&value)`



    - if LOCK signal is asserted on the bus, no otherprocessor can perform any operation over the bus

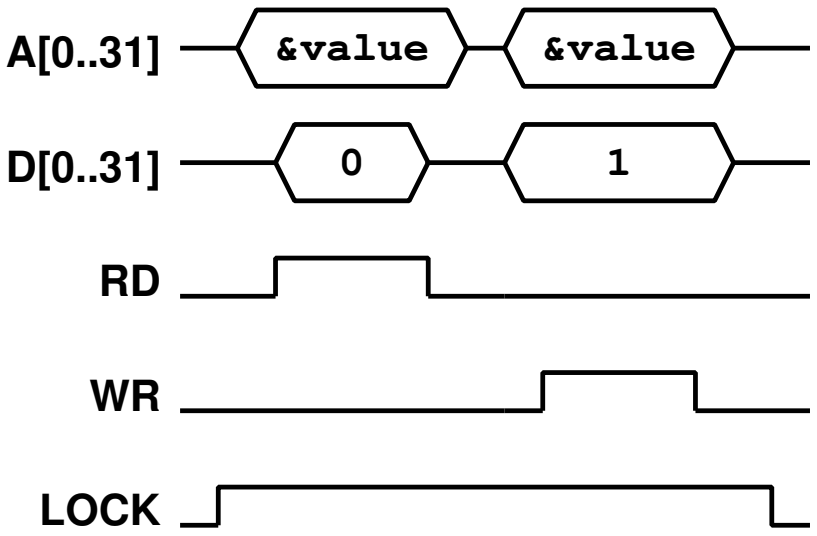


Atomic Test-And-Set Machine Instruction

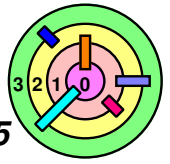
- Lock is represented as a bit, 0 if FREE, 1 if BUSY

```
bool test_and_set(int *lock)
{
  bool previous_value=*lock;
  *lock = BUSY;
  return previous_value;
}
```

- e.g., `value` is *FREE*, call `test_and_set(&value)`



- if LOCK signal is asserted on the bus, no otherprocessor can perform any operation over the bus
  - read from the bus and write to the bus, together, is an *atomic* operation

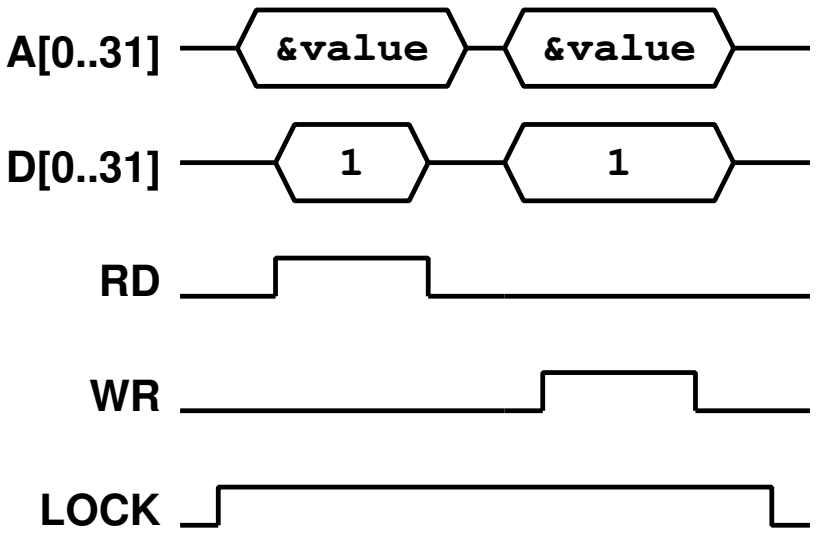


Atomic Test-And-Set Machine Instruction

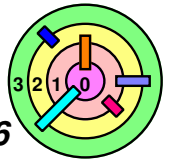
- Lock is represented as a bit, 0 if FREE, 1 if BUSY

```
bool test_and_set(int *lock)
{
  bool previous_value=*lock;
  *lock = BUSY;
  return previous_value;
}
```

- e.g., `value` is ***BUSY***, call `test_and_set(&value)`



- if LOCK signal is asserted on the bus, no otherprocessor can perform any operation over the bus
  - read from the bus and write to the bus, together, is an ***atomic*** operation



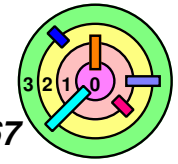
Better Spinlock

- Naive spinlock (lock out the bus way too much)

```
Spinlock::acquire()
{
  while (test_and_set(&value))
    ; // spin
}
```

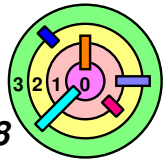
- Better spinlock (try to minimize locking out the bus)

```
Spinlock::acquire()
{
  while (1) {
    if (value == FREE) {
      // lock was at least momentarily unlocked
      if (test_and_set(&value) == FREE) {
        // we have locked the spinlock
        break;
      }
      // some other thread beat us, try again
    }
  }
}
```



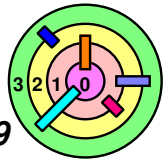
Spinlocks

- Spinlocks are *wasteful*
  - processor time wasted waiting for the lock to be released
  - barely acceptable if locks are held only briefly
- *Interrupt service routine* must run to completion without blocking
  - must use a spinlock to access shared data (since interrupt handlers are not threads)
  - in addition, before using a spinlock, must disable interrupts first
- Most OSes keep interrupt handlers extremely simple
  - minimal work that must be done inside an interrupt handler are
    - wake up a thread waiting for the I/O completion interrupt
      - this would require accessing the ready list (access protected by first disabling interrupts then lock a spinlock)
    - start the next I/O operation (to keep the I/O device as busy as possible)



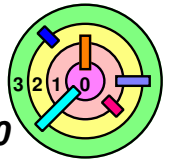
How Many Spinlocks

- Various data structures need synchronized access
  - need a queue of waiting threads on lock X
  - need a queue of waiting threads on lock Y
  - list of threads ready to run
- What's wrong with one spinlock for the entire kernel?
  - bottleneck
- Instead:
  - one spinlock per lock
  - one spinlock for the ready list
    - per-processor ready list: one spinlock per processor/core



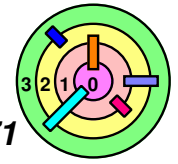
Which Thread Is Currently Running?

- Thread scheduler needs to find the TCB of the currently running thread
  - to suspend and switch to a new thread
  - to check if the current thread holds a lock before acquiring or releasing it
- On a uniprocessor, easy: use a global variable (currentThread)
- On a multiprocessor system, varioius methods:
  - compiler can dedicate a register: e.g., use r31 to point to the TCB of the running thread and every processor has its own r31
  - if stack sizes are fixed: can put a pointer to the TCB of the running thread at the bottom of its stack (assuming the base address of the stack is page-aligned)
    - find the address of this pointer by masking the current stack pointer (i.e., zeroing out the least-significant bits)



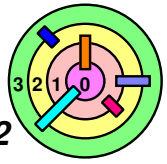
Implementing Multiprocessor Queueing Locks

```
class Lock {
  private:
    int value = FREE;
    SpinLock spinLock; // need this for multiprocessor
    queue waiting; // queueing lock
  public:
    void acquire();
    void release();
};
```



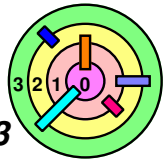
Implementing Multiprocessor Queueing Locks

```
Lock::acquire()
{
  spinLock.acquire();
  if (value == BUSY) {
    waiting.add(runningThread);
    // spinLock released inside suspend()
    scheduler.suspend(&spinLock);
  } else {
    value = BUSY;
    spinLock.release();
  }
}

Lock::release()
{
  spinLock.acquire();
  if (!waiting.empty()) {
    TCB *nextThread=waiting.remove();
    scheduler.makeReady(nextThread);
  } else {
    value = FREE;
  }
  spinLock.release();
}
```



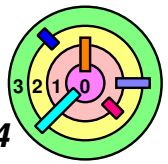
Scheduler Implementing

```
class Scheduler {
  private:
    // access only when owning schedulerSpinLock
    queue readyList;
    // access only when interrupt is disabled
    SpinLock schedulerSpinLock;
  public:
    void suspend(SpinLock *lock);
    void makeReady(TCB *thread);
};
```



Scheduler Implementing

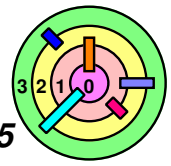
```
Scheduler::suspend(SpinLock *lock)
{
  disableInterrupts();
  schedulerSpinLock.acquire();
  lock->release();
  runningThread->state = WAITING;
  TCB *nextThread=readyList.remove();
  thread_switch(runningThread, nextThread);
  runningThread->state = RUNNING;
  schedulerSpinLock.release();
  enableInterrupts();
}

Scheduler::makeReady(TCB *thread)
{
  disableInterrupts();
  schedulerSpinLock.acquire();
  nextThread->state = READY;
  readyList.add(thread);
  schedulerSpinLock.release();
  enable_interrupt();
}
```



Scheduler Implementing

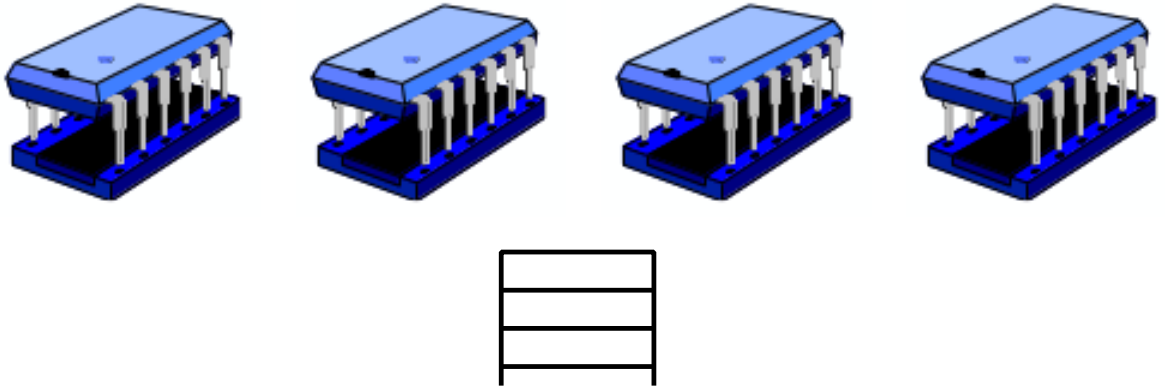
```
Scheduler::suspend(SpinLock *lock)
{
  disableInterrupts();
  schedulerSpinLock.acquire();
  lock->release();
  runningThread->state = WAITING;
  TCB *nextThread=readyList.remove();
1 thread_switch(runningThread, nextThread);
  runningThread->state = RUNNING;
  schedulerSpinLock.release();
  enableInterrupts();
}

Scheduler::makeReady(TCB *thread)
{
  disableInterrupts();
  schedulerSpinLock.acquire();
  nextThread->state = READY;
2 readyList.add(thread);
  schedulerSpinLock.release();
  enable_interrupt();
}
```

- without using `schedulerSpinLock`, thread 1 can be running in two CPUs simultaneously

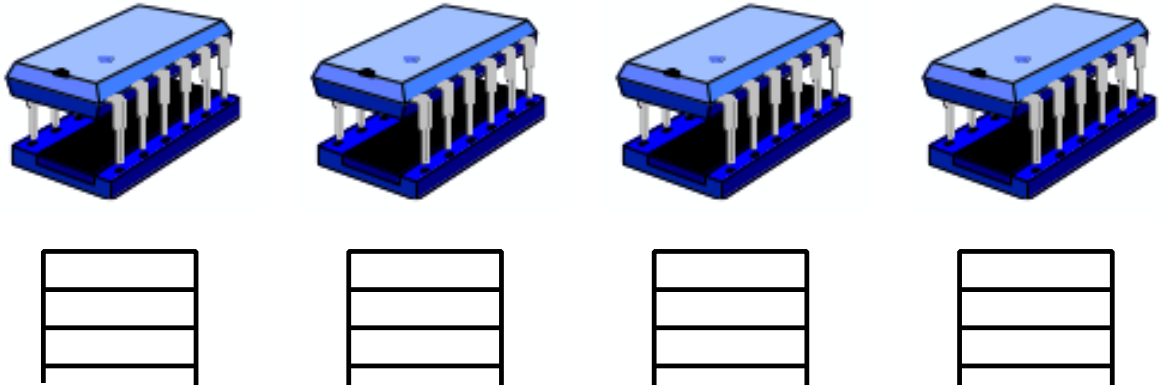


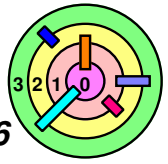
Cache Affinity

- After a thread has run on a particular processor, next time it runs, it would be cheaper to run it on the same processor
  - *cache affinity*

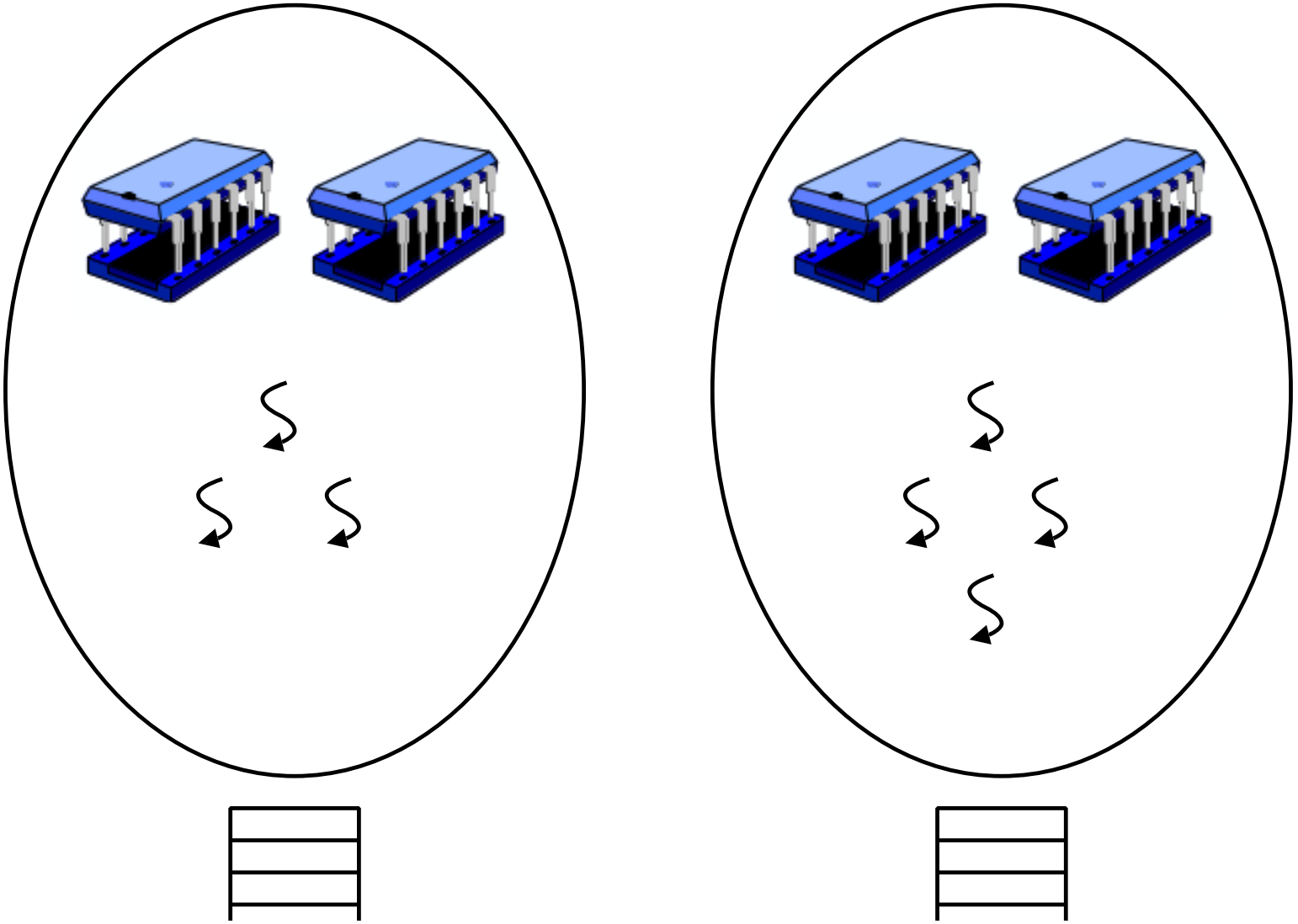


- This means that if you use a shared ready list for multiple processors, you will not be able to take advantage of cache affinity
  - therefore, you should use one ready list per processor
  - scheduler may do *load balancing* occasionally

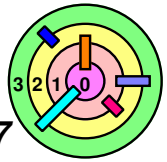




Solaris: Processor Sets

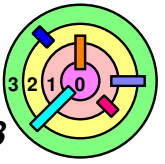


- Somewhere between the two extremes
  - reducing the frequency of requiring load balancing



Case Study: Linux 2.6 Kernel Mutex Lock

- Linux kernel implementation is optimized for the common case
  - *common case (fast path):* locks are in the FREE state
    - use read-modify-write instruction to acquire the lock and be optimistic that most of the time, it would be successful and be done with acquiring the lock
      - x86 processor has atomic decrment/increment (return previous value), atomic exchange (swap content of memory location and CPU register), atomic test-and-set
  - *rare case (slow path):* locks are in the BUSY state and lots of threads are waiting in a long queue to acquire it
    - if the read-modify-write instruction failed to acquire the lock, then use the previous approach for multiprocessor queueing locks

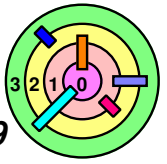


Fast Path Acquire

```
struct mutex {
  atomic_t count;
    // 1: unlocked
    // 0: locked with no waiting thread
    // <0: locked, with possible waiting threads
  spinlock_t wait_lock;
  struct list_head wait_list;
};
```

⇨ **Use a macro for the fast path to save procedure call overhead:**

```
    lock decl (%eax)
            // %eax contains the address of lock->count
    jns 1f // jump if result is >= 0, i.e., not signed
    call slowpath_acquire
1:
```

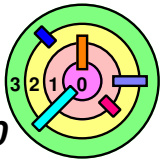


Slow Path Acquire

- The slow path uses the previous approach for multiprocessor queueing locks (start by disabling preemption, acquiring spinlock, adding thread to wait list)
  - main difference is that count is no longer 1 or 0 and can go negative
  - also, when thread returns from `scheduler.suspend()`, it cannot not assume that it owns the lock and must try to acquire the lock again

```
for (;;) {
  if (atomic_xchg(&lock->count, -1) == 1)
    break;
  // go to sleep
}
// count is now -1
if (list_empty(&lock->wait_list)) {
  atomic_set(&lock->count, 0);
}
```

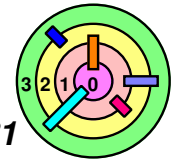
  - release spinlock and enable preemption



Fast Path Release

- Use a macro for the fast path to save procedure call overhead:

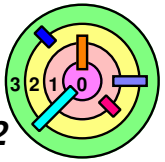
```
        lock incl (%eax)
                // %eax contains the address of lock->count
        jg 1f // jump if result is = 0, i.e., FREE
        call slowpath_release
1:
```



Slow Path Release

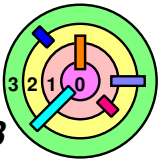
- The slow path code is similar to the previous approach for multiprocessor queueing locks

```
lock->wait_lock.acquire();
// now unlock the lock
atomic_set(&lock->count, 1);
if (!list_empty(&lock->wait_list)) {
  // wake up one thread on lock->wait_list
}
lock->wait_lock.release();
```



Linux 2.6 Kernel Mutex Lock

- Acquiring and releasing a lock can be inexpensive
- Programmers sometimes go to great lengths to avoid acquiring a lock
  - when there is little contention, avoiding locks is unlikely to significantly improve performance
  - it's often better to just keep things simple and use locks to ensure mutual exclusion when accesing shared state

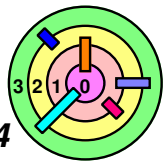


Implementing Condition Variables (Mesa Semantics)

```
void CV::wait(Lock *lock)
{
  assert(lock.isHeld());
  waiting.add(currentThread);
  scheduler.suspend(lock);
  lock->acquire();
}

void CV::signal()
{
  if (waiting.notEmpty()) {
    TCB *thread = waiting.remove();
    scheduler.makeReady(thread);
  }
}

void CV::broadcast()
{
  while (waiting.notEmpty()) {
    TCB *thread = waiting.remove();
    scheduler.makeReady(thread);
  }
}
```

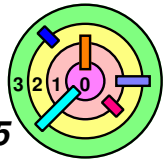
```
class CV {
  private:
    queue waiting;
  public:
    void wait(Lock *lock);
    void signal();
    void broadcast();
};
```

- since lock is already acquired, the code is simple

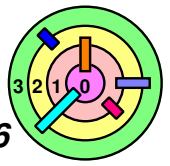


Implementing Application-Level Synchronization

- Recall from Ch 4 that there are two ways to support application-level concurrency:
  - via system calls
  - via user-level thread scheduler
- Kernel-managed threads
  - can split the lock data structure:
    - `count` is kept in the address space of the user process
    - kernel holds the `spinlock` and `waiting` queue
- User-managed threads
  - lock data structure is kept completely in the address space of the user process
  - what about disabling interrupts?
    - only need to disable upcalls from the OS kernel (upcalls in user space is analygous to interrupts in kernel)

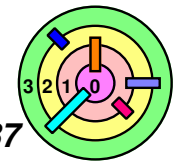


(5.8) Semaphores Considered Harmful



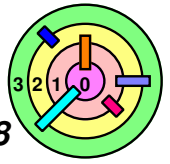
Synchronization Primitives

- Many different synchronization primitives have been proposed
  - semaphores
  - communication sequential processes
  - event delivery
  - message passing
  - etc.
- None of these are more powerful than using locks and condition variables
  - a program using other paradigms can be mapped to *threads and monitors* using straight-forward transformations



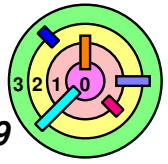
Semaphores

- Semaphores are defined as follows:
  - a semaphore is a non-negative integer value
  - when a semaphore is created, its value can be initialized to any non-negative integer
  - a semaphore has only two operations and no other operations are allowed to access the semaphore value
    - *atomic decrement:* `Semaphore::P()` waits until the semaphore value is positive, then atomically decrements the semaphore value by 1 and returns
    - *atomic increment:* `Semaphore::V()` atomically increments the semaphore value by 1
      - if some threads are waiting in `P()`, one of them is enabled so that its call to `P()` succeeds at decrementing the semaphore value and returns



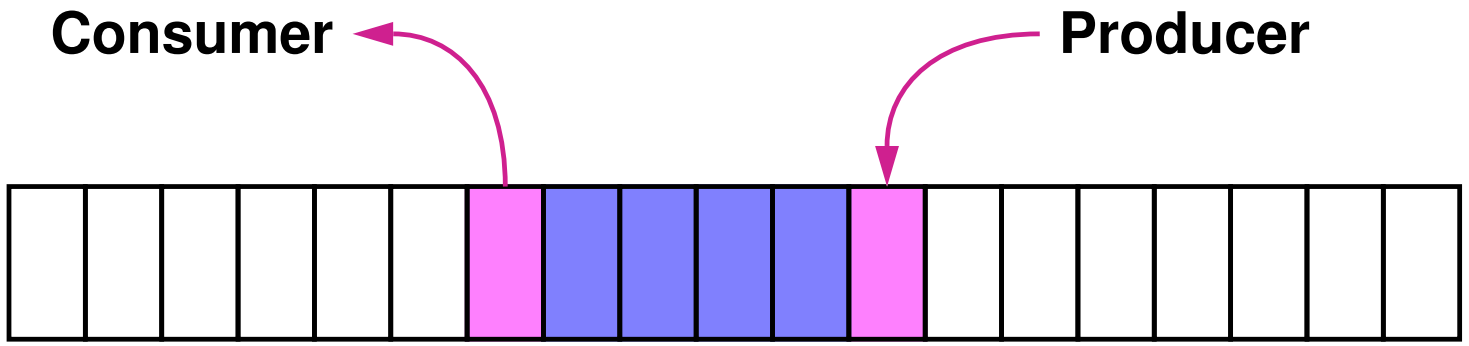
Semaphores Considered Harmful

- Why are semaphores considered harmful?
  - because it's used in two different ways
- If semaphore value is initialized to be > 0, it can be used for mutual exclusion
  - e.g., can be used to solve the FIFO bounded buffer problem
- If semaphore value is initialized to be 0, it can be used for general waiting
  - e.g., thread A calls `P()` to wait for thread B to call `V()`
    - thread B can call `V()` first, then when thread A calls `P()`, it would return right away
      - `P()` is like `CV::wait()` and `V()` is like `CV::signal()`, but not exactly the same because if `CV::signal()` is called first, `CV::wait()` can get stuck forever
  - for general waiting, it's more clean to use CV+lock/mutex and not a semaphore

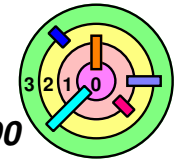


FIFO Bounded Buffer with Semaphores

- Semaphores are often used to synchronize communication between an I/ O device and threads waiting for I/ O completion
  - one producer (I/O device) and one consumer (thread waiting for I/O completion)

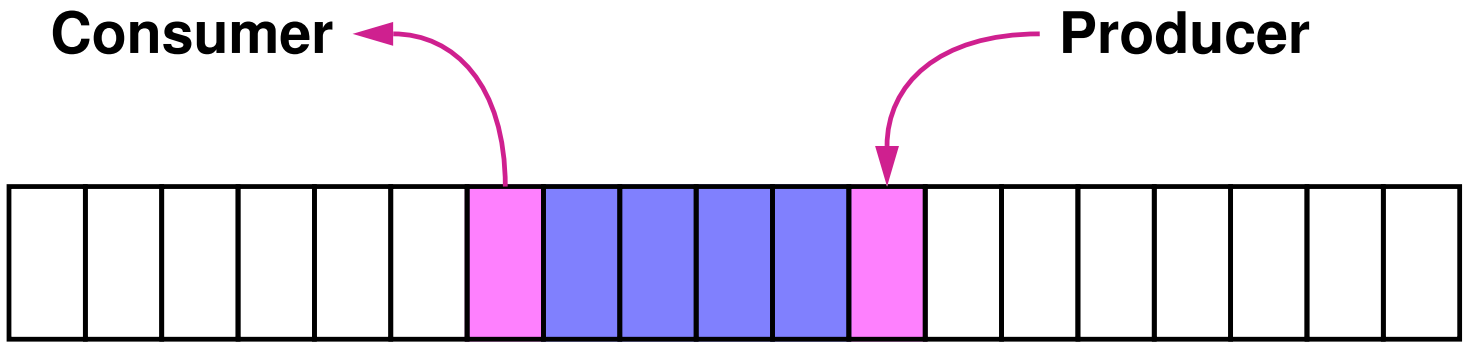


- Conveyor belt (in hardware)
  - perfect parallelism between producer and consumer
- A circular buffer is used in software implementation
- Most of the time, no interference
  - if you use a ***single mutex*** to lock the entire array of buffers, it's an overkill *(i.e., too **inefficient**)*

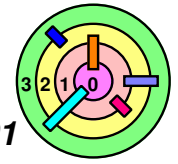


FIFO Bounded Buffer with Semaphores

- Semaphores are often used to synchronize communication between an I/ O device and threads waiting for I/ O completion
  - one producer (I/O device) and one consumer (thread waiting for I/O completion)



- When does it require synchronization?
  - producer needs to be blocked when all slots are full
  - consumer needs to be blocked when all slots are empty
- We will look at the solution for multiple producer and multiple consumer
  - will use a semaphore as a mutex to atomically accessing some shared variables



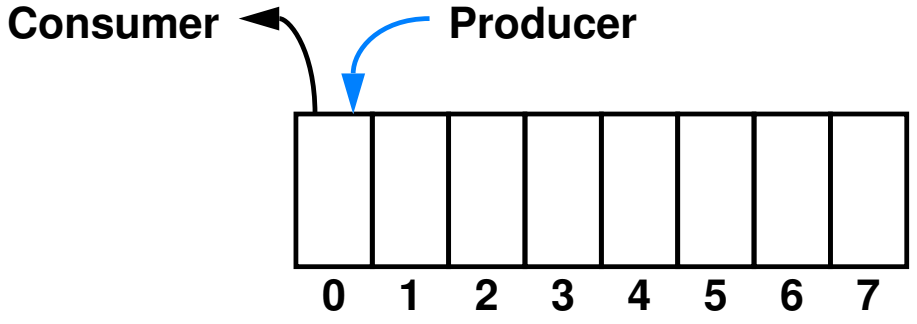
FIFO Bounded Buffer with Semaphores

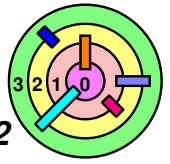
```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	8
occupied	0
mutex	1
tail	0
front	0





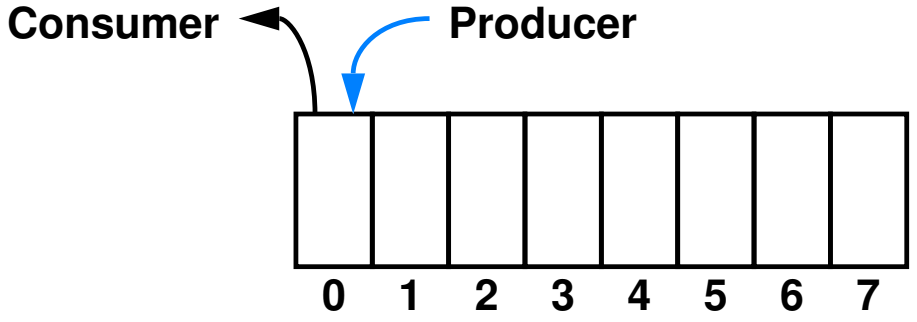
FIFO Bounded Buffer with Semaphores

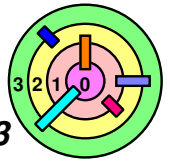
```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	8
occupied	0
mutex	1
tail	0
front	0





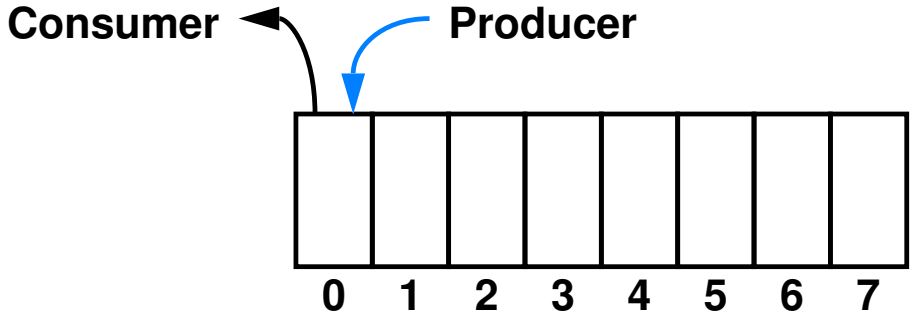
FIFO Bounded Buffer with Semaphores

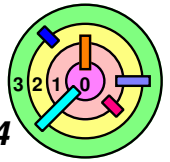
```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
   empty.P();
   mutex.P();
   buf[tail%MAX] = item;
   tail++;
   mutex.V();
   occupied.V();
}
```

```
char get( ) {
   occupied.P();
   mutex.P();
   item = buf[front%MAX];
   front++
   mutex.V();
   empty.V();
   return item;
}
```

empty	7
occupied	0
mutex	1
tail	0
front	0



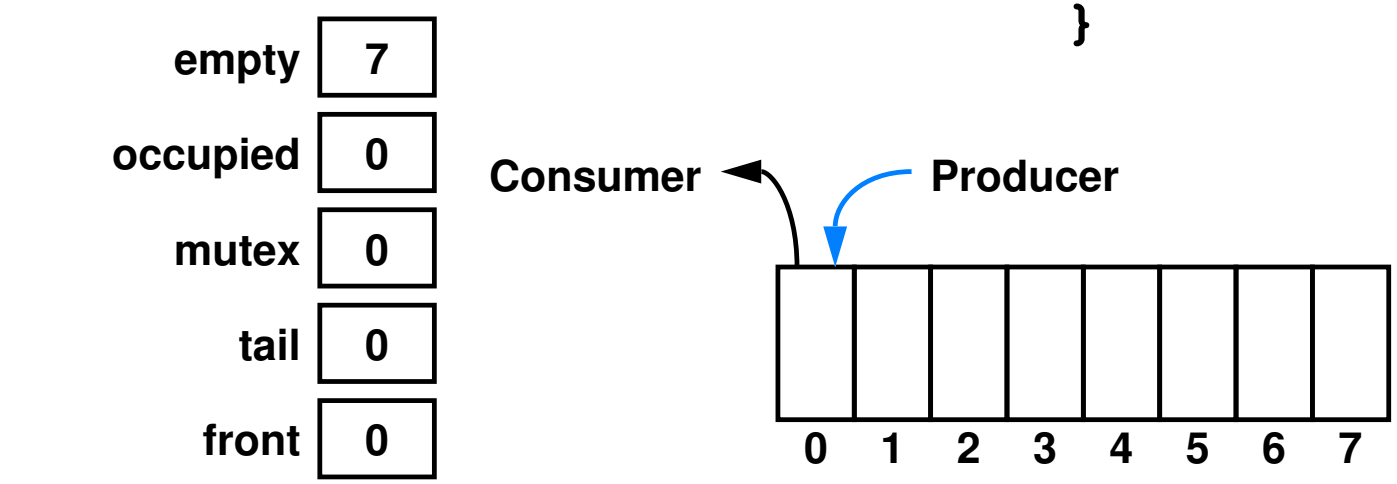


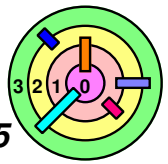
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
    mutex.P();
    item = buf[front%MAX];
    front++
    mutex.V();
    empty.V();
    return item;
}
```



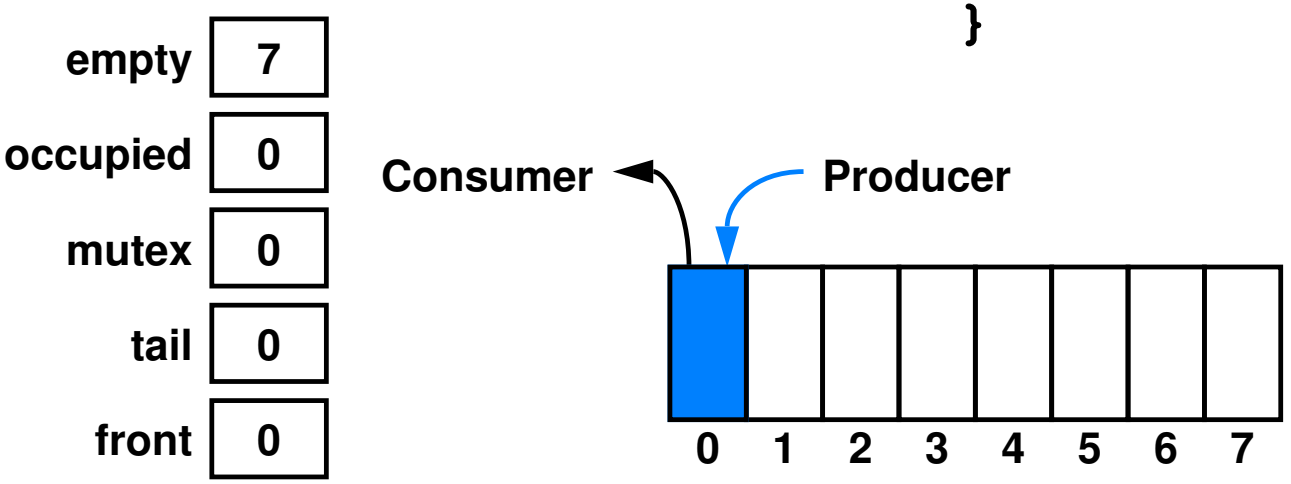


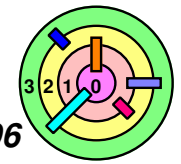
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
    mutex.P();
    item = buf[front%MAX];
    front++
    mutex.V();
    empty.V();
    return item;
}
```



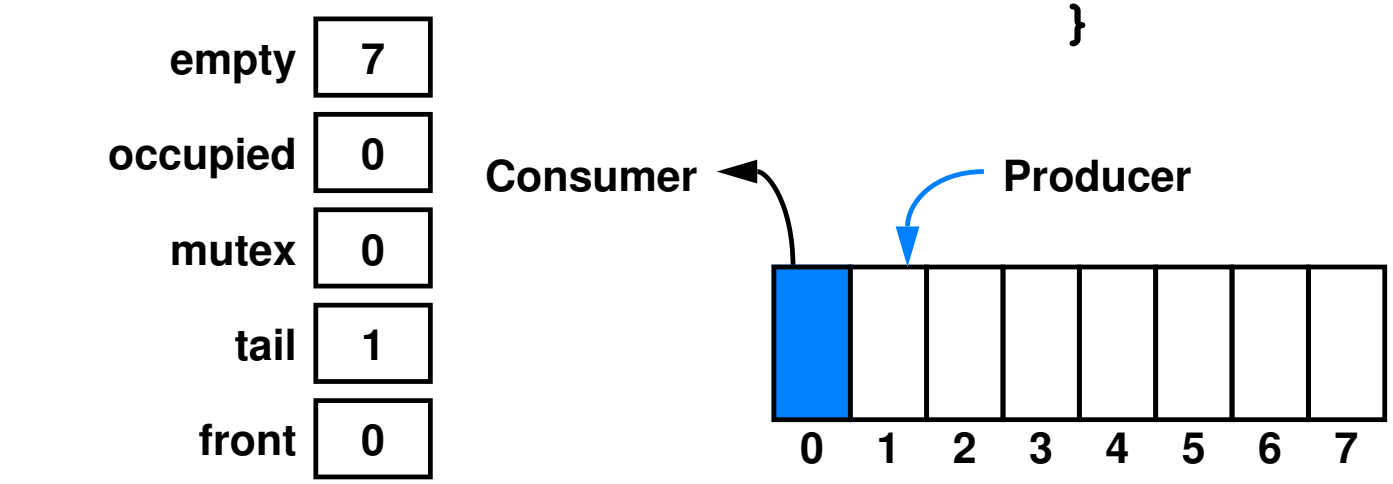


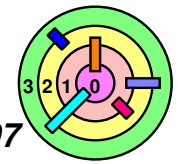
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
    mutex.P();
    item = buf[front%MAX];
    front++
    mutex.V();
    empty.V();
    return item;
}
```





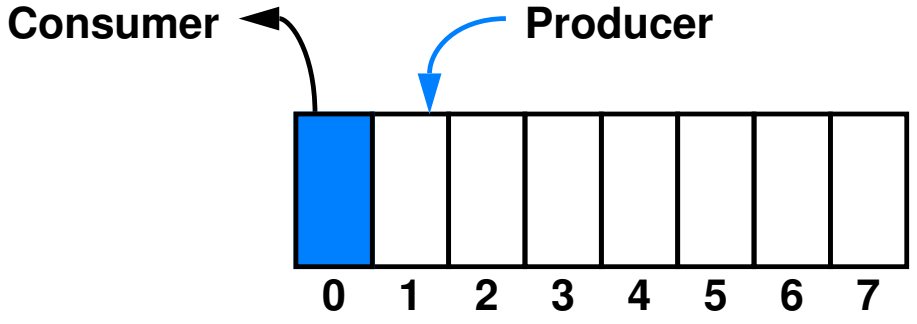
FIFO Bounded Buffer with Semaphores

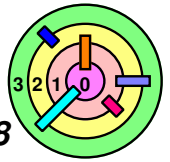
```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	7
occupied	0
mutex	1
tail	1
front	0





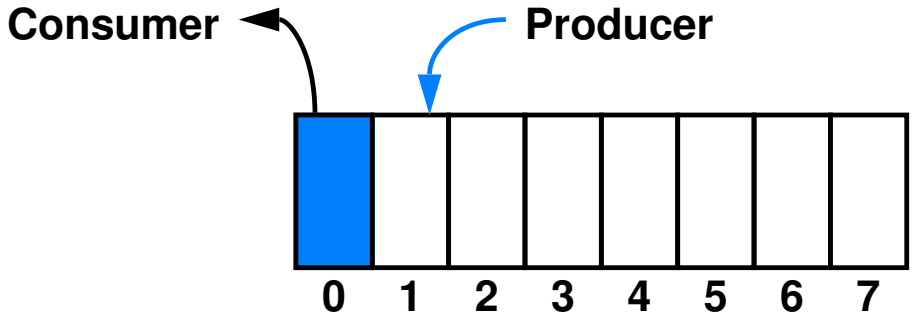
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

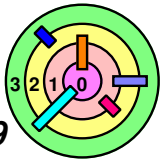
```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	7
occupied	1
mutex	1
tail	1
front	0



- note: producer continue to produce



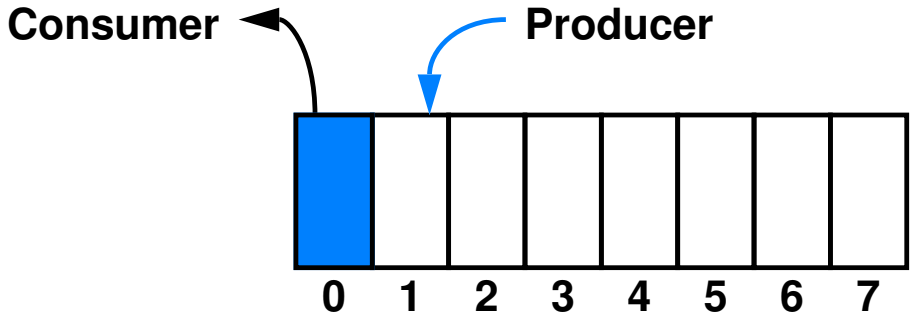
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

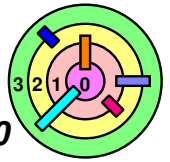
```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	7
occupied	0
mutex	1
tail	1
front	0



- note: producer continue to produce



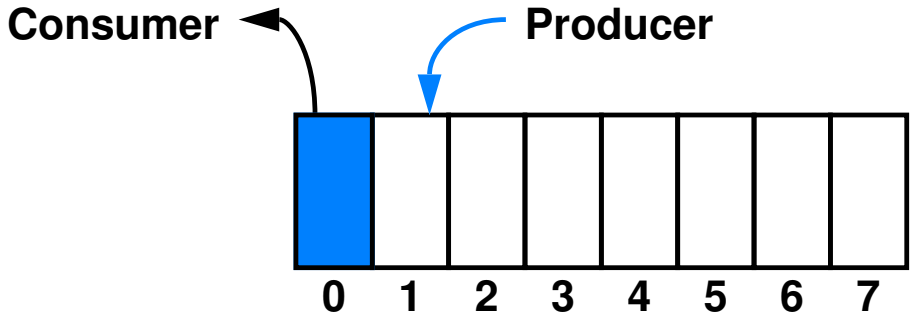
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

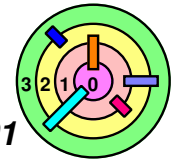
```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	7
occupied	0
mutex	0
tail	1
front	0



- note: producer continue to produce



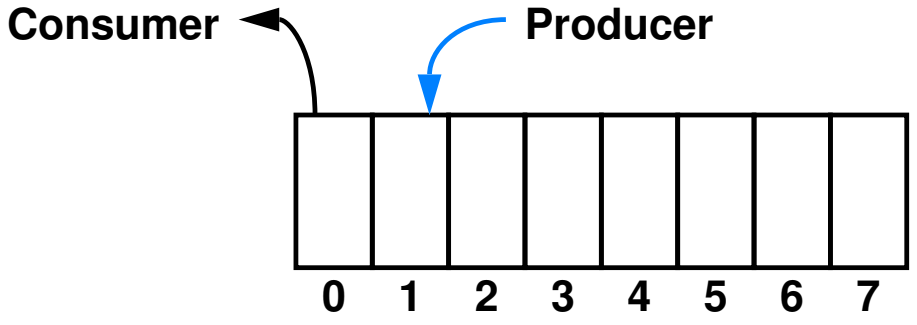
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

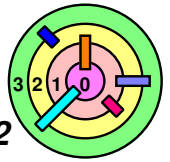
```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	7
occupied	0
mutex	0
tail	1
front	0



- note: producer continue to produce



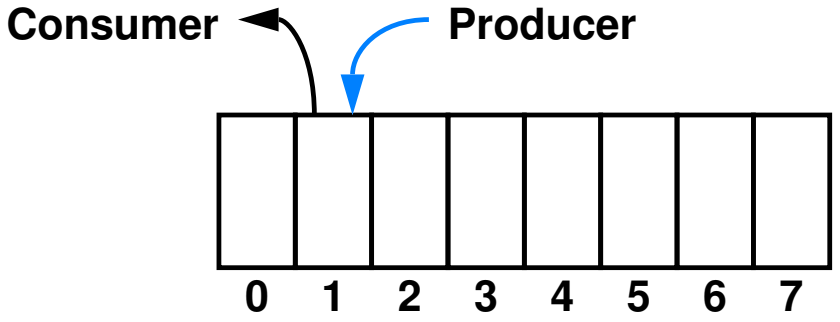
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

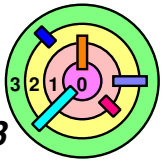
```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	7
occupied	0
mutex	0
tail	1
front	1



- note: producer continue to produce



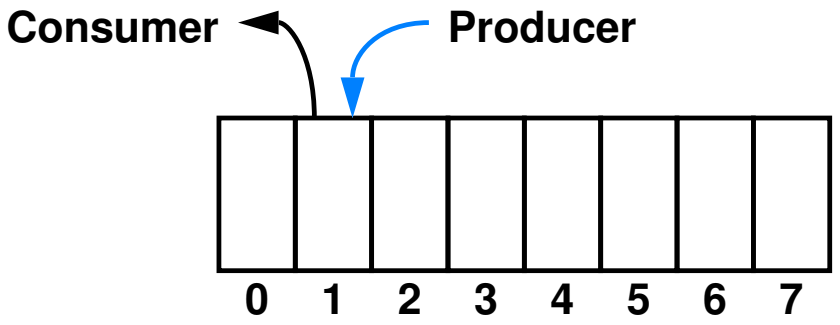
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

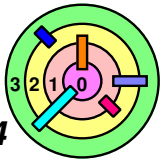
```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	7
occupied	0
mutex	1
tail	1
front	1



- note: producer continue to produce



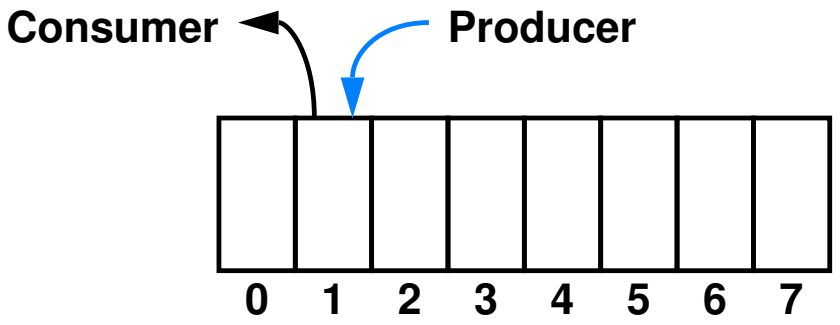
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

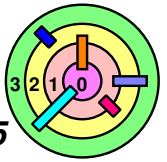
```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	8
occupied	0
mutex	1
tail	1
front	1



- note: producer continue to produce



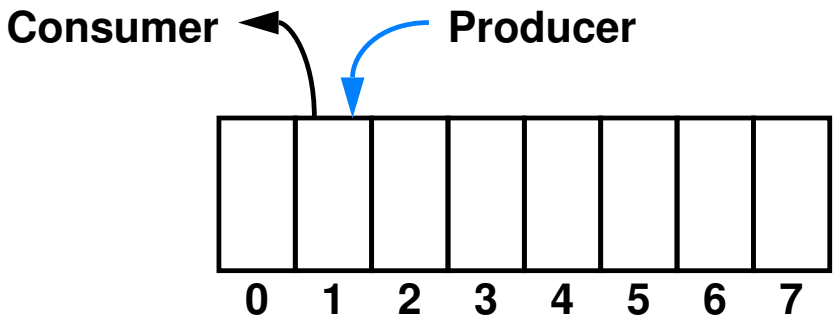
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

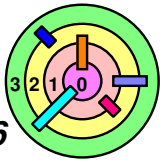
```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

empty	8
occupied	0
mutex	1
tail	1
front	1



- note: producer continue to produce



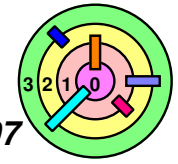
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

- if produce and consume run at same rate and work at different spots, no producer may ever wait for a consumer and vice versa
  - although threads of same type must be synchronized
- if producer is fast and consumer slow, producer may wait
- if consumer is fast and producer slow, consumer may wait



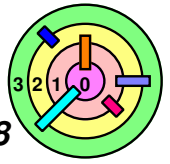
FIFO Bounded Buffer with Semaphores

```
Semaphore empty=MAX;
Semaphore occupied=0;
Semaphore mutex=1;// synchronize threads
int front=0, tail=0;
```

```
void put(item) {
  empty.P();
  mutex.P();
  buf[tail%MAX] = item;
  tail++;
  mutex.V();
  occupied.V();
}
```

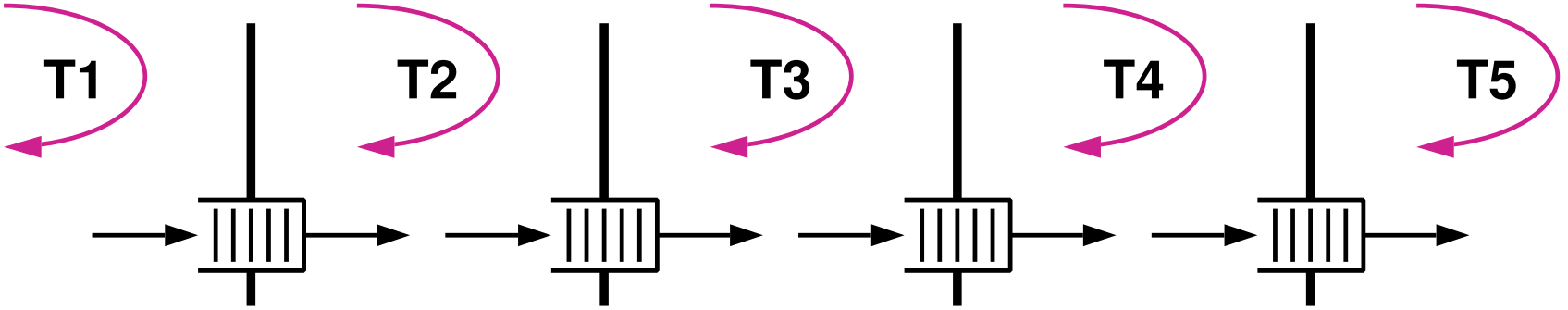
```
char get( ) {
  occupied.P();
  mutex.P();
  item = buf[front%MAX];
  front++
  mutex.V();
  empty.V();
  return item;
}
```

- ***Mutex*** **by itself is more** ***"coarse grain"***
  - **you may use one mutex to control access to the number of empty and occupied cells,** `front`**, and** `tail`
- ***Semaphore*** **gives more** ***"fine grain parallelism"***

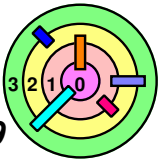


Semaphores Considered Harmful

- ***Semaphore*** **has limited use**
  - **difficult to use perfectly due to semaphore's memory property**
  - **pretty much the only place it's really good for is producer-consumer**
    - **although it's a very important application of semaphores**

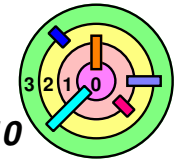


    - **the queues above are *queues with bounded buffer***
  - ***"pipelined parallelism"***



Communicating Sequential Processes (CSP / Google Go)

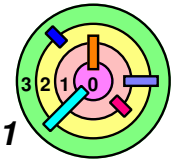
- One thread per shared object
  - only that thread is allowed to touch object's data
  - to call a method on the object, ***send thread a message*** with method name and arguments
  - thread waits in a loop, get message and perform operation
- No memory race conditions!
  - since data access is sequential/serialized



CSP Example: FIFO Bounded Buffer

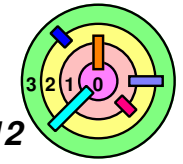
```
CV empty, full;
int front=0, tail=0;

get() {
  lock.acquire();
  while(front==tail){
    empty.wait(&lock);
  }
  item = buf[front%MAX];
  front++
  full.signal(lock);
  lock.release();
  return item;
}

void put(item) {
  lock.acquire();
  while((tail-front)==MAX){
    full.wait(&lock);
  }
  buf[tail%MAX] = item;
  tail++;
  empty.signal(&lock);
  lock.release();
}
```

- producer thread sleeps in the `full` CV queue to wait for an empty spot to appear in the buffer
- consumer thread sleeps in the `empty` CV queue to wait for a data item to appear in the buffer



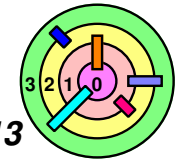
CSP Example: FIFO Bounded Buffer

```
while (cmd = getNext()){
  if (cmd == GET) {
    if (front < tail) {
      // do get()
      // send reply
      // if pending put, do it and send reply
    } else {
      // queue get operation
    }
  } else { // cmd == PUT
    if ((tail-front) < MAX) {
      // do put()
      // send reply
      // if pending get, do it and send reply
    } else {
      // queue put operation
    }
  }
}
```



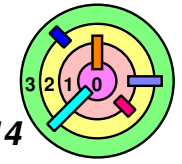
Locks/CVs vs. CSP

- Create a lock on shared data = create a single thread to operate on data
- Create a method on a shared object = send a message and wait for reply
- Wait on a CV = queue an operation that cannot be completed just yet
- Signal a condition = perform a queued operation that has been enabled to proceed



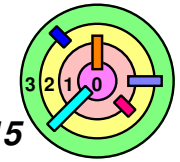
Message-Passing vs. Shared Memory

- Which approach is better for programming concurrency, message-passing or shared memory?
  - as it turns out, any program using monitors can be recast into CSP using a simple transmation, and vice versa
    - executing a procedure while holding a monitor lock is equivalent to processing a message in CSP
    - a monitor is single-threaded while it's holding the lock
- It's just a matter of style!



Review: Remember The Rules

- Use consistent structure
- Always 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()`



Extra Slides