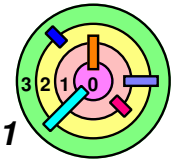
Housekeeping (Lecture 7 - 6/12/2025)

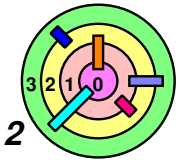
- **PA2 is due at 11:45pm on Tuesday, 7/1/2025**
  - **if you have code from current or a previous semester, *do not look at/copy/share* any code from it**
    - **it's best if you just get rid of it**
  - **if you include files that's not part of the original "`make pa2-submit`" command, the grader will delete them**

- ***Grading guidelines* is the *ONLY* way we will grade and we can only grade on a *standard 32-bit Ubuntu Linux 16.04 inside VirtualBox/UTM* or on AWS Free Tier**
  - **although not recommended, you can do your development on a different platform**
    - **you must test your code on the "standard" platform because those are the *only* platforms the grader is allowed to grade on**

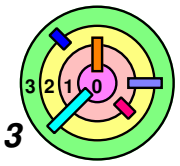


Housekeeping (Lecture 7 - 6/12/2025)

- PA2 and PA3 are sepeate assignments and they will be graded separately
  - if you only make a PA3 submission, you will get a score of 0 for your PA2 submission
  - I would suggest that when you are done with PA2, make a submission and keep the source in the cs350/pa2 directory and don't touch it
  - create a cs350/pa3 directory by copying everything from the cs350/pa2 directory and start working on PA3 in the cs350/pa3 directory
  - when you are working on PA3 and discovered bugs in your PA2 code, you need modify your PA2 code in both the pa2 directory and the pa3 directory and make another PA2 submission

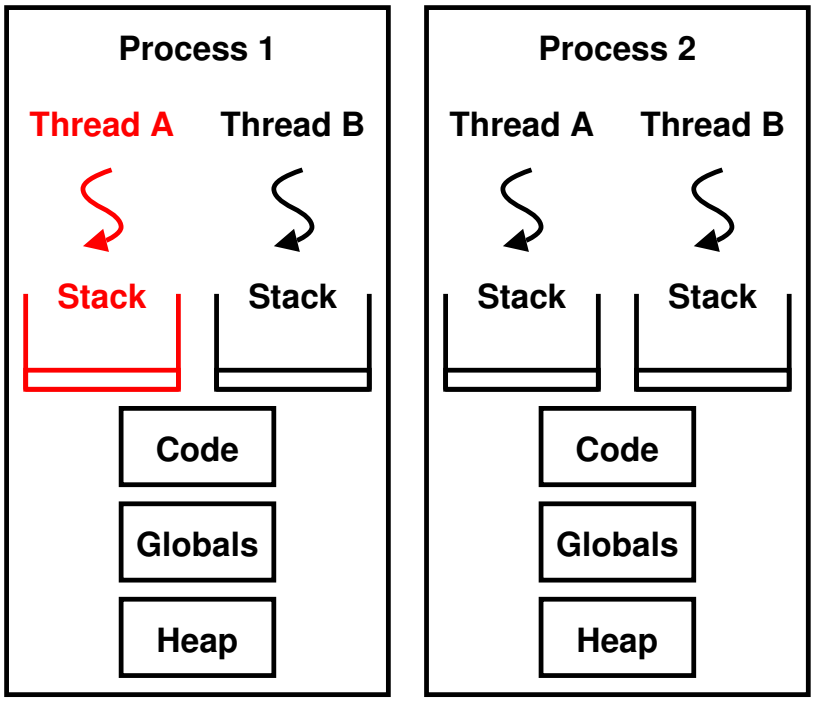


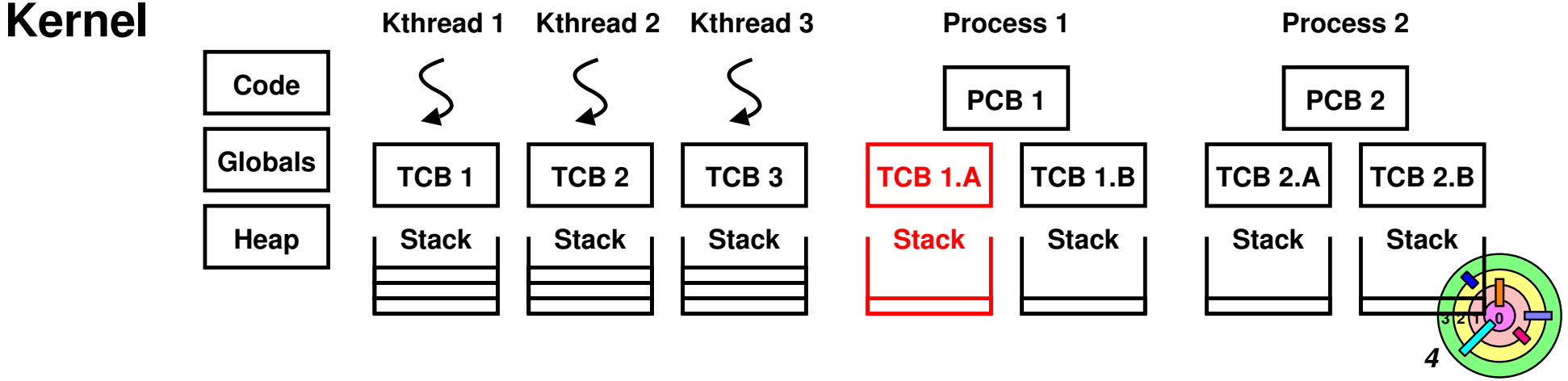
(4.8) Implementing Multi-Threaded Processes



Implementing Multi-threaded User Processes Using Kernel Threads

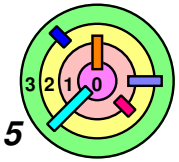
- User thread = kernel thread (Linux, MacOS)
  - all thread-related function calls are system calls
    - kernel does context switch
  - simple, but a lot of transitions between user and kernel mode





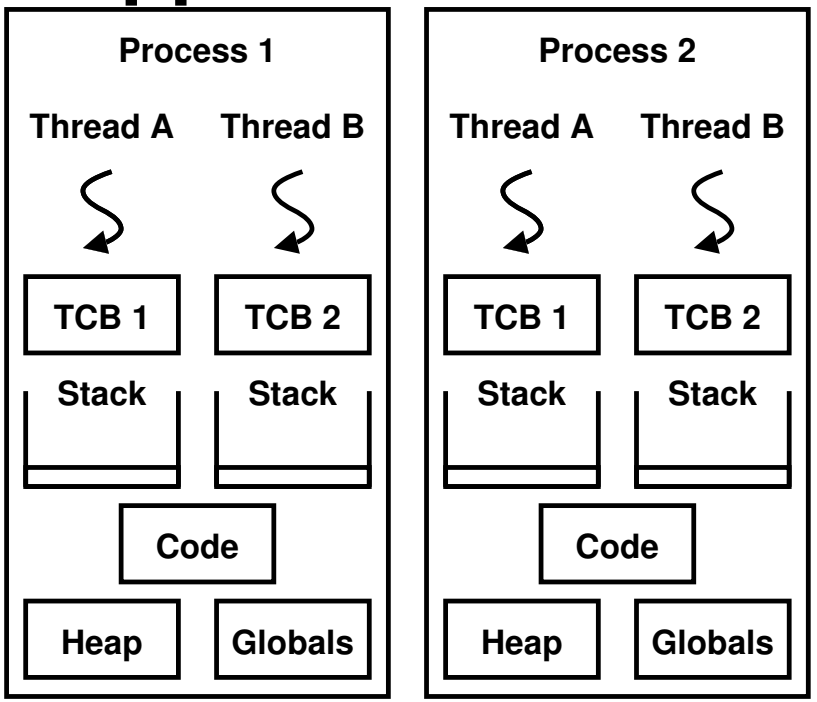
Implementing User-Level Threads Without Kernel Support

- Implement user-level threads completely at user level, without any OS support
  - e.g., *green threads* in the earliest implementation of Sun's Java Virtual Machine (JVM)
    - to the kernel, a multi-threaded application using green threads appears to be a normal single-threaded process
    - if a user thread makes a system call and get blocked waiting for I/O, the kernel cannot run a different user thread
      - to get true parallelism, you have to run multiple processes



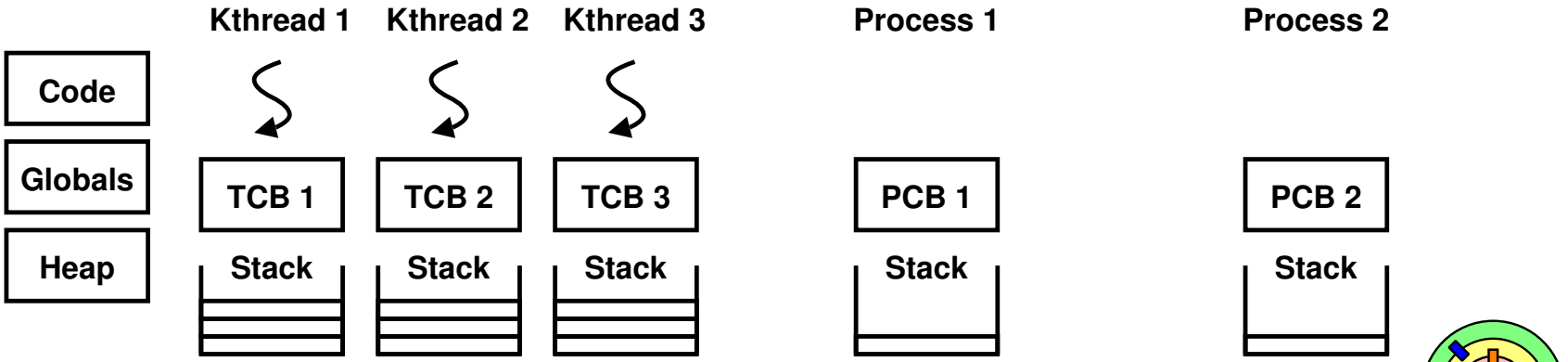
Implementing User-Level Threads Without Kernel Support

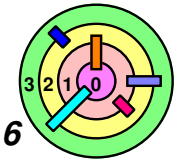
- The kernel doesn't know about user-level threads



User

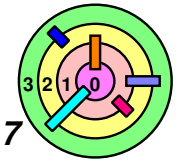
Kernel





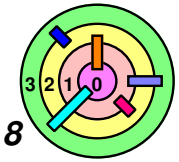
Implementing User-Level Threads Without Kernel Support

- Preemptive user-level threads: implementation for process P
  - user-level thread library makes a system call to register a timer signal handler and signal stack with the kernel
  - when a hardware timer interrupt occurs, the hardware saves P's register state and runs the kernel's handler
  - instead of restoring P's register state and resuming P where it was interrupted, the kernel's handler copies P's saved registers onto P's signal stack
  - the kernel resumes execution in P at the registered signal handler on the signal stack
  - the signal handler copies the processor state of the preempted user-level thread from the signal stack to that thread's TCB
  - the signal handler chooses the next thread to run, re-enables the signal handler (similar to re-enabling interrupts), and restores the new thread's state from its TCB into the processor; execution with the state (newly) stored on the signal stack



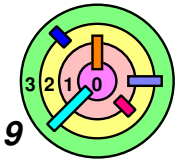
Implementing User-Level Threads With Kernel Support

- Today, most programs use kernel-supported threads rather than pure user-level threads
  - major operating systems support threads using standard abstractions, so the issue of portability is less of an issue than it once was



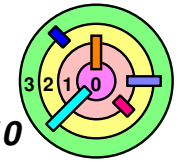
Implementing User-Level Threads With Kernel Support

- Various systems take more of a hybrid model (best of both worlds)
  - *hybrid thread join*
  - *per-processor kernel threads*
  - *scheduler activations* (in Windows): user-level thread scheduler is notified/activated for every kernel event that might affect the user-level thread system

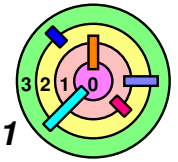


(4.9) Alternative Abstractions

- Asynchronous I/O and event-driven programming
  - allows a single-threaded program to cope with high-latency I/O devices by overlapping I/O with processing and other I/O
- Data parallel programming
  - all processors perform the same instruction in parallel on different parts of a data set



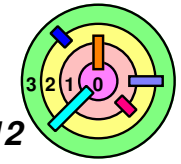
Extra Slides



Ch 5: Synchronizing Access to Shared Objects

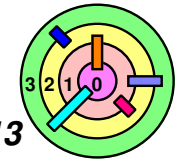
Bill Cheng

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):

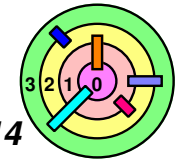
Thread 1

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

Thread 2

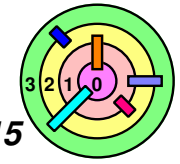
```
while (!pInitialized)
    ;
q = anotherComputation(p)
```

  - clearly, this is no good

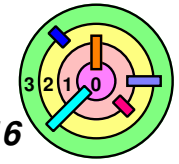


Why Reordering?

- 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

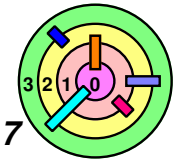
Thread 1	Thread 2
`x = y + 1;`	`y = y * 2;`

  - 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



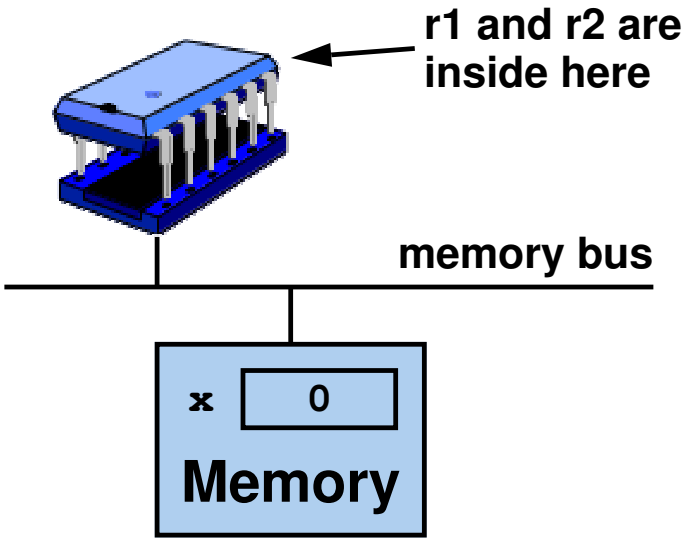
Race Condition

Thread 1:

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

Thread 2:

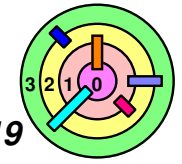
```
x = x + 2;
 /*
  load  r1,x
  add   r2,r1,1
  store x,r2
  */
```



- 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*



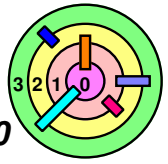
Too Much Milk Try #1: Leave A Note

- 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
  }
}
```

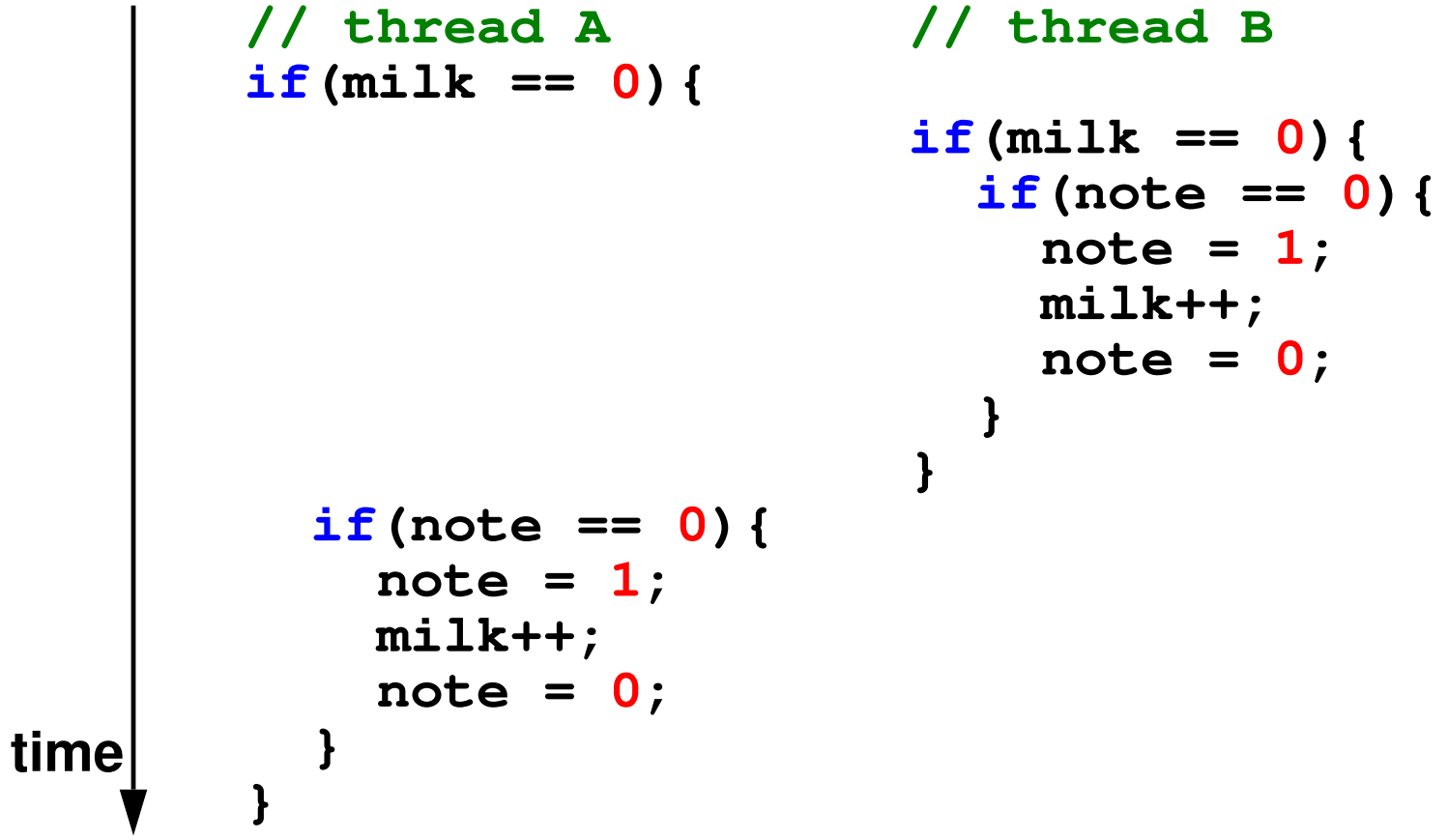
  - which statements are *atomic*?
    - the assumption here is that if a statement only access zero or one memory location, it's an atomic operation (because it cannot be preempted in the middle of that operation)

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



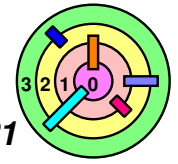
Too Much Milk Try #1: Leave A Note

- 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

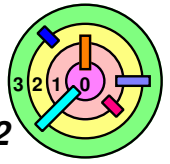


Too Much Milk Try #2: Two Notes

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
```

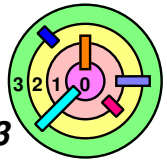
- does this solution guarantees ***safety*** and ***liveness***?



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]

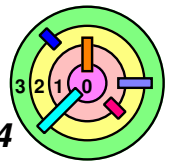
```
        // thread A
        noteA = 1;
[A1]    if(noteB == 0){
[A2]      if(milk == 0){
[A3]        milk++;
          }
        }
        noteA = 0;

        // 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] ¤

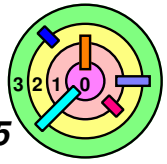
```
        // thread A
        noteA = 1;
[A1]    if(noteB == 0){
[A2]      if(milk == 0){
[A3]        milk++;
          }
        }
        noteA = 0;

        // 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;
```

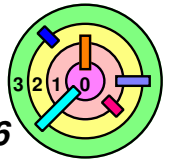


Too Much Milk Try #3: Waiting

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
```

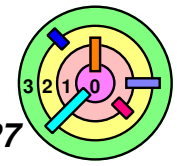
- does this solution guarantees *safety* and *liveness*?



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

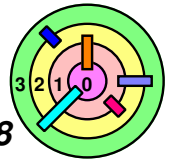
```
// thread A
noteA = 1;
while(noteB == 1) {
  ;
}
if(milk == 0){
  milk++;
}
noteA = 0;

// 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

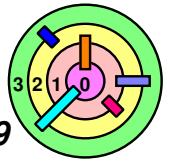
- ***Lock:*** a primitive that only one thread at a time can own

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

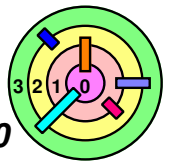
  - simple and symmetrical

- 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)

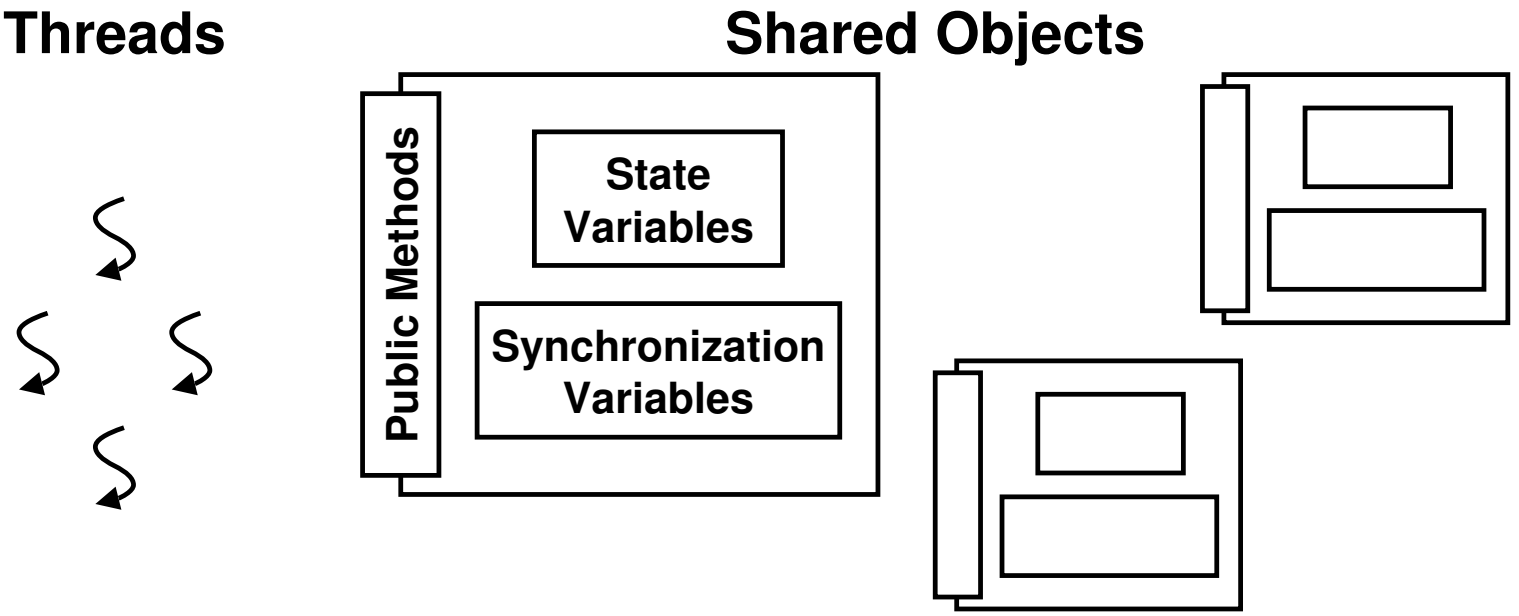
```
// thread A or thread B
Kitchen::buyIfNeeded() {
  mutex.lock();
  ...
  mutex.unlock();
}
```



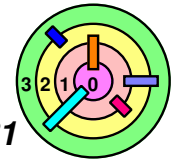
(5.2) Structuring Shared Objects



Threads And Shared Objects

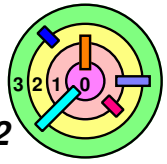


- In a multi-threaded program, threads are separate from shared objects and operate concurrently on shared objects
  - shared objects contain both shared state and *synchronization variables* (for controlling concurrent access to shared state)
- *Shared objects:* objects that can be accessed *safely*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



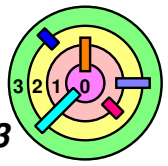
Monitors

- 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



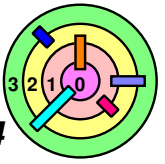
Shared Objects Are Implemented In Layers

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**	



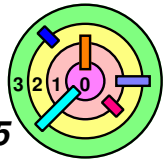
(5.3) Locks: Mutual Exclusion

Synchronization Variables:	Semaphores	*Locks*	Condition Variables



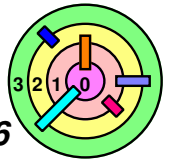
Locks

- 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



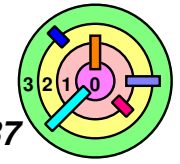
Locks: API and Properties

- 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



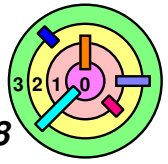
Case Study: Thread-Safe Bounded Queue

- 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;
}
```

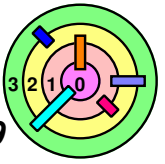
```
tryput(item) {
  lock.acquire();
  if ((tail - front) < size) {
    buf[tail % MAX] = item;
    tail++;
  }
  lock.release();
}
```

- 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



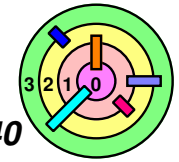
Critical Section

- 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*

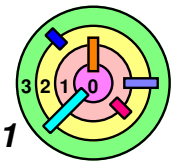


Condition Variables (CV)

- ***Busy waiting:***

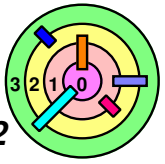
```
get()
{
  while ((data = tryget()) == NULL) ;
  return data;
}
```

- 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();
}
```

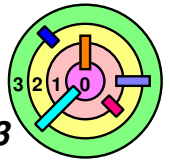


Example: Bounded Queue/Buffer

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

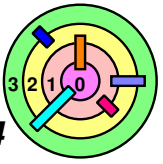
```
put(item) {
  lock.acquire();
  while ((tail-front)==MAX){
    fullf.wait(lock);
  }
  buf[tail%MAX] = item;
  tail++;
  empty.signal(lock);
  lock.release();
}
```

- 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

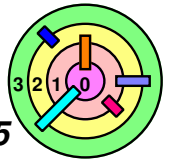
```
methodThatWaits() {
  lock.acquire();
  // pre-condition: State is consistent
  // read/write shared state
  while (!testSharedState()) {
    cv.wait(lock);
  }
  // WARNING: shared state may have changed,
  //      but testSharedState() is true and
  //      pre-condition is true
  // read/write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



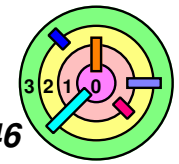
Condition Variables

- 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

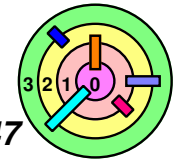


Structured Synchronization

- 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()`