Synchronization 1: Monitors and Language Support for Concurrency

Lecture 7

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4103/

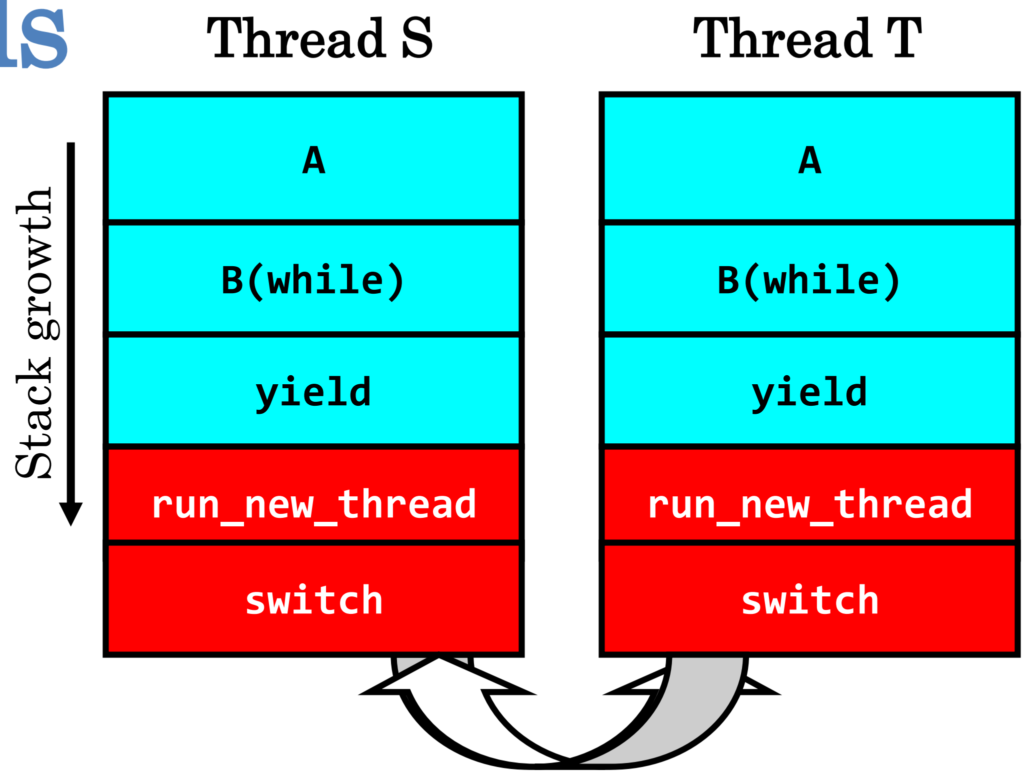
Context Switches

Switching Threads

- Consider the following code blocks:

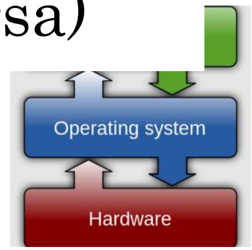
```
func A() {
  B();
}

func B() {
  while(TRUE) {
    yield();
  }
}
```

- Two threads, S and T, each run A

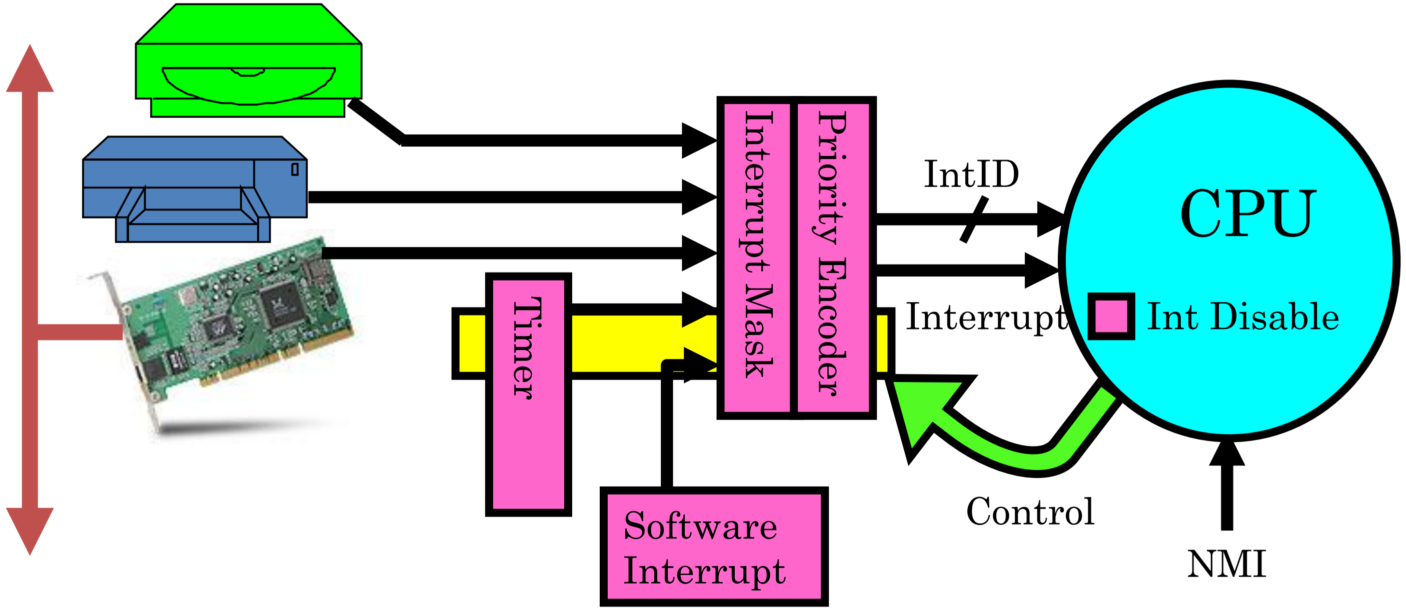


Thread S's `switch` returns to Thread T's (and vice versa)

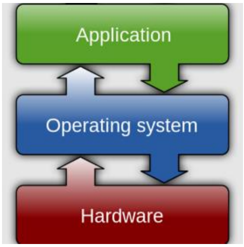
Pintos: switch.S



Interrupt Management



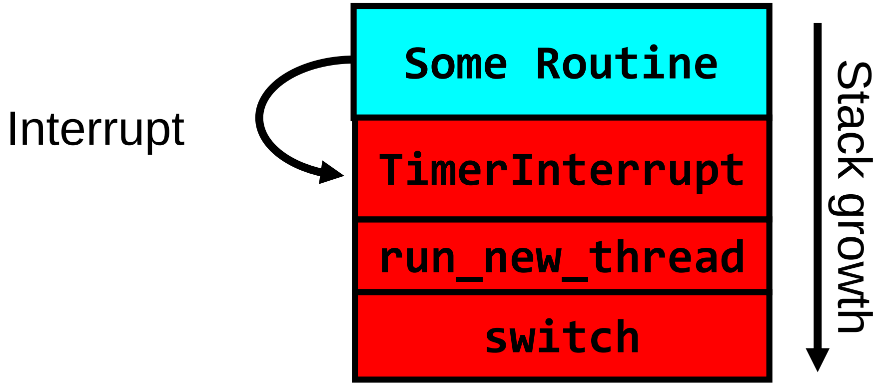
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can't be disabled

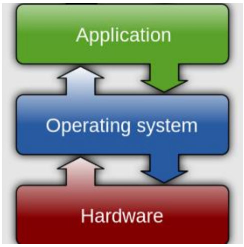


Preempting a Thread

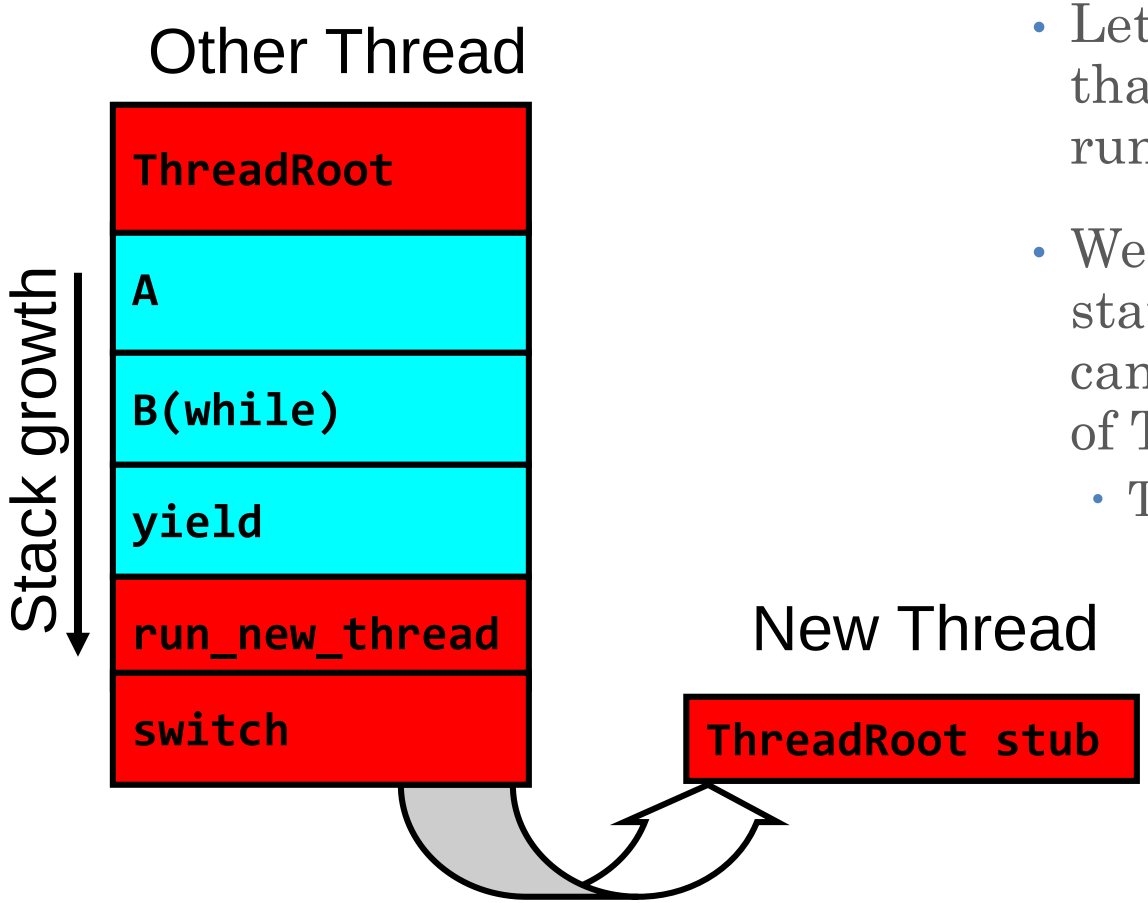
- Timer Interrupt routine:

```
TimerInterrupt() {
  DoPeriodicHouseKeeping();
  run_new_thread();
}
```

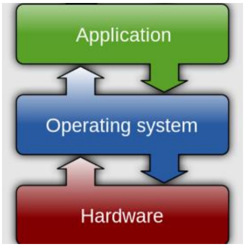




Creating a New Thread



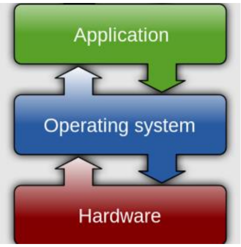
- Let ThreadRoot be the routine that the thread should start out running
- We need to set up the thread state so that, another thread can "return" into the beginning of ThreadRoot
  - This really starts the new thread



Bootstrapping Threads

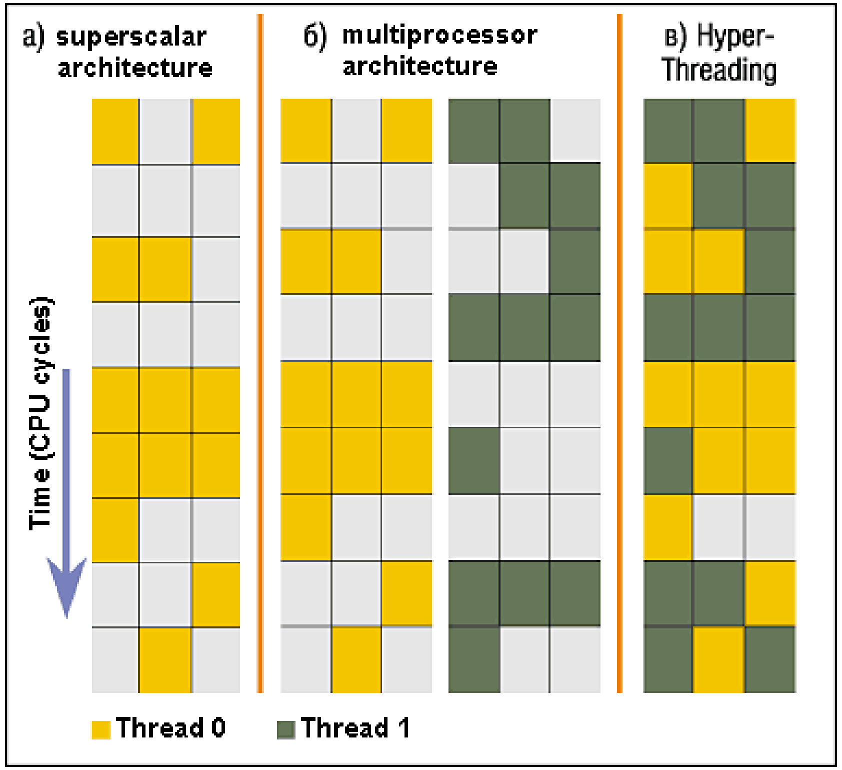
```
ThreadRoot() {
  DoStartupHousekeeping();
  UserModeSwitch();     /* enter user mode */
  call fcnPtr(fcnArgPtr);
  ThreadFinish();
}
```

- Stack will grow and shrink with execution of thread
- ThreadRoot() never returns
  - ThreadFinish() destroys thread, invokes scheduler

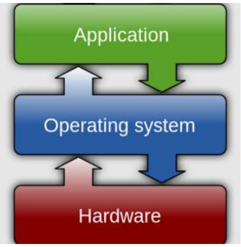


Aside: SMT/Hyperthreading

- Hardware technique
  - Superscalar processors try to execute multiple independent instructions in parallel
  - Hyperthreading allows a single core to process multiple instructions streams at once
  - But, sub-linear speedup
- Original called “Simultaneous Multithreading”
  - http://www.cs.washington.edu/research/smt/index.html
  - Intel, SPARC, Power (IBM)
- From the OS perspective, this just looks like multiple cores

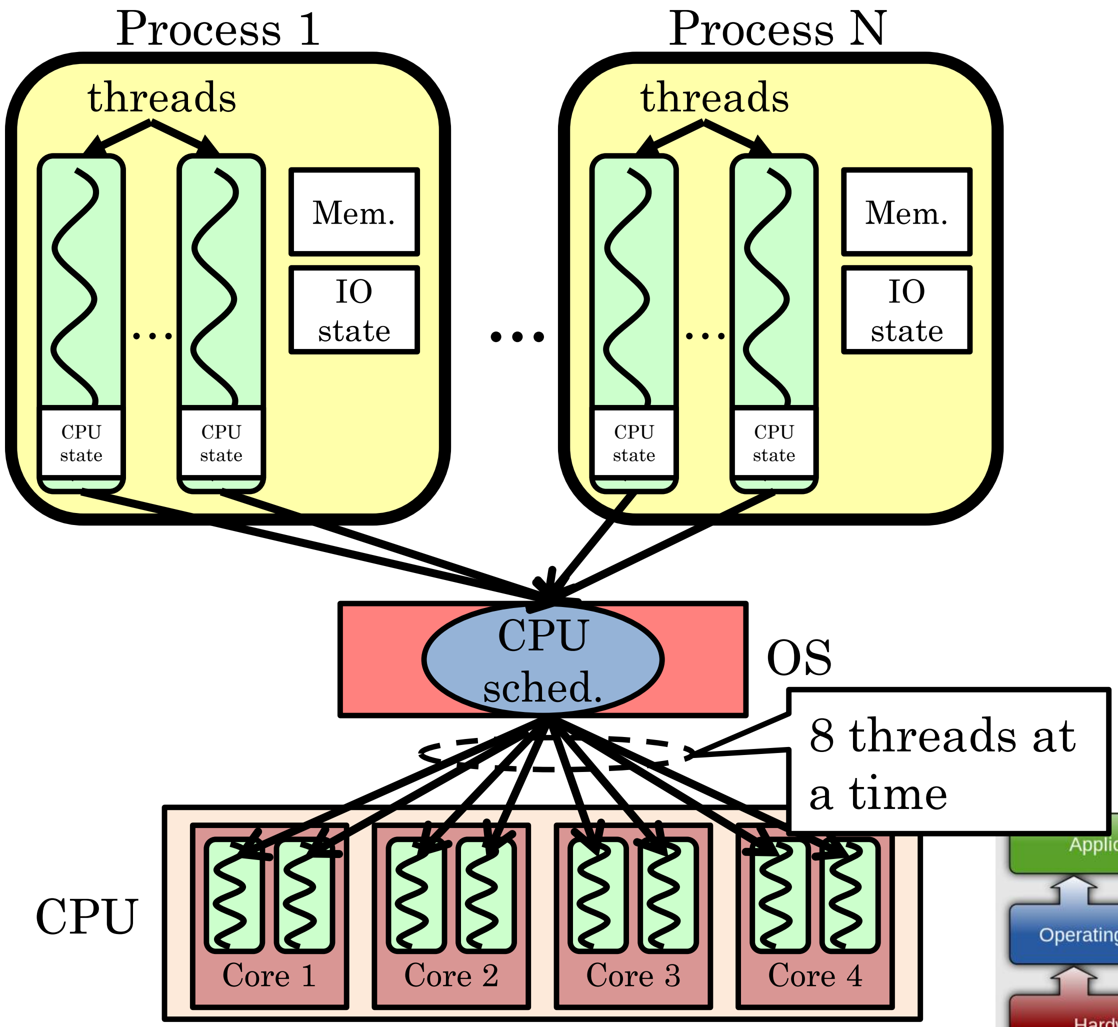


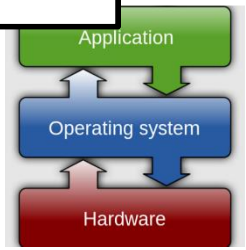
Colored blocks show instructions executed



Aside: SMT/Hyperthreading

- Switch overhead:
  - Same process: low
  - Different proc.: high
- Protection
  - Same proc: low
  - Different proc: high
- Sharing overhead
  - Same proc: low
  - Different proc: high





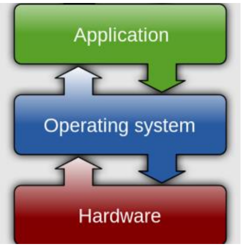
Synchronization: Monitors

Recall: Race Conditions

- What are the possible values of `x` below?
- Initially `x == 0` and `y == 0`

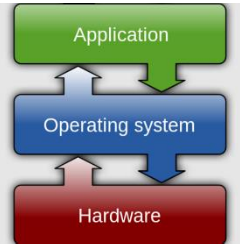
```
Thread A        Thread B
x = y + 1;      y = 2;
                y = y * 2;
```

- `1` or `3` or `5` (non-deterministic)
- Race Condition: Thread A races against Thread B



Recall: Locks

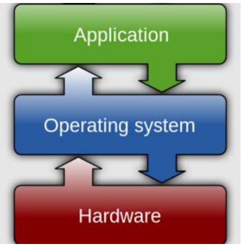
- Locks provide two atomic operations:
  - `Lock.acquire()` – wait until lock is free; then mark it as busy
    - After this returns, we say the calling thread holds the lock
  - `Lock.release()` – mark lock as free
    - Should only be called by a thread that currently holds the lock
    - After this returns, the calling thread no longer holds the lock
- For now, don't worry about how to implement locks!
  - We'll cover that in substantial depth later on in the class



Mutual Exclusion between Thread and Interrupt Handler

- Interrupt handler must run to completion without interruptions
  - Can't acquire a lock in an interrupt handler (why?)
- Solution: Disable interrupts and restore them afterwards

```
int state = intr_disable();
<code manipulating shared data>
intr_restore(state);
```

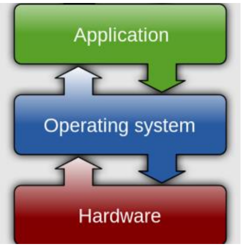


Is Mutual Exclusion Enough?

No…

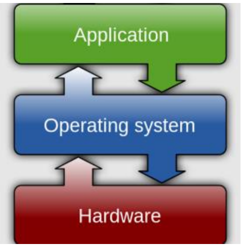
Recall: Relevant Definitions

- Synchronization: Coordination among threads, usually regarding shared data
- Mutual Exclusion: Ensuring only one thread does a particular thing at a time (one thread excludes the others)
  - Type of synchronization
- Critical Section: Code exactly one thread can execute at once
  - Result of mutual exclusion
- Lock: An object only one thread can hold at a time
  - Provides mutual exclusion

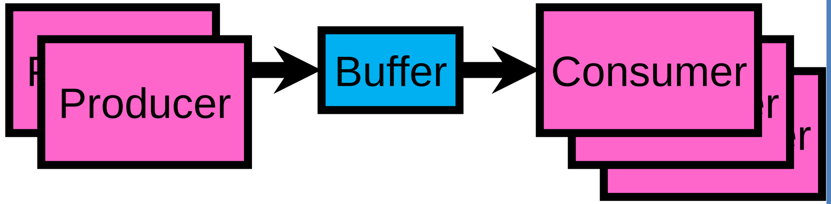


The Producer-Consumer Problem

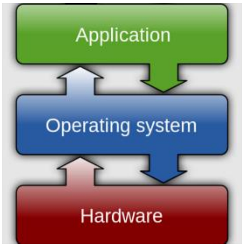
- Some processes/threads produce output that is consumed as input by other processes/threads
- Where have we seen this?
  - Pipes
  - Sockets
- GCC compiler – simple 1-1
  - `cpp | cc1 | cc2 | as | ld`



Producer-Consumer with a Bounded Buffer

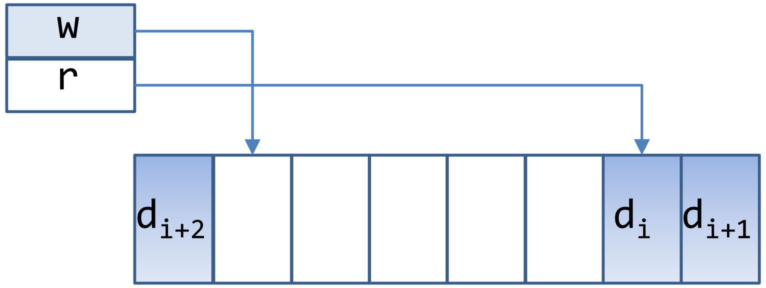


- Problem Definition
  - Producers puts things into a shared buffer
  - Consumers takes them out
- Don't want producers and consumers to have to work in lockstep, so put a buffer (bounded) between them
  - Need synchronization to maintain integrity of the data structure and coordinate producers/consumers
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

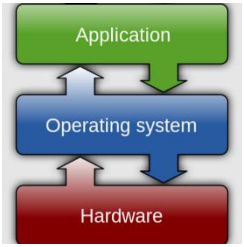


Circular Buffer Data Structure (Sequential Case)

```c
typedef struct buf {
  int write_index;   /* = 0 */
  int read_index;    /* = 0 */
  <type> entries[BUFSIZE + 1];
} buf_t;
```



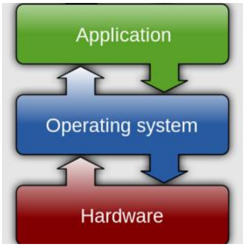
- Insert: if not full:
  - write & bump (wrap around) write ptr (enqueue)
- Remove: if not empty:
  - read & bump (wrap around) read ptr (dequeue)
- How to tell if empty (on dequeue)?
- How to tell if full (on enqueue)?
- And what do you do if it is?
- What needs to be atomic?



Circular Buffer: Attempt #1

```
mutex buf_lock = <initially unlocked>
Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {} // Wait for a free slot
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {} // Wait for a used slot
  item = dequeue();
  release(&buf_lock);
  return item;
}
```

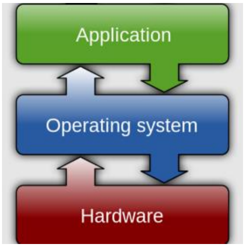
Will we ever come out of the wait loop?



Circular Buffer: Attempt #2

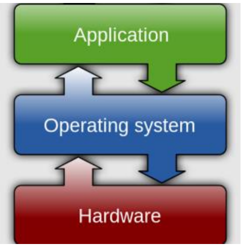
```
mutex buf_lock = <initially unlocked>
Producer(item) {
  acquire(&buf_lock);
  while (buffer full) { release(&buf_lock); acquire(&buf_lock); }
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) { release(&buf_lock); acquire(&buf_lock); }
  item = dequeue();
  release(&buf_lock);
  return item;
}
```

What happens when one is waiting for the other?



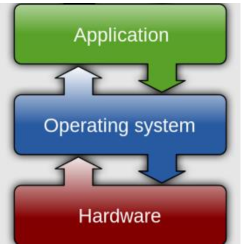
Producer-Consumer: Correctness

- Mutual exclusion:
  - Only one thread manipulates the buffer data structure at a time
- Synchronization requirements other than mutual exclusion:
  - If buffer is empty, consumer waits for the producer
  - If buffer is full, producer waits for consumer



Recall: Semaphore

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX (& Pintos)
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - `P()` or `down()`: atomic operation that waits for semaphore to become positive, then decrements it by 1
  - `V()` or `up()`: an atomic operation that increments the semaphore by 1, waking up a waiting `P`, if any
- `P()` stands for "proberen" (to test) and `V()` stands for "verhogen" (to increment) in Dutch



Recall: Two Important Semaphore Patterns

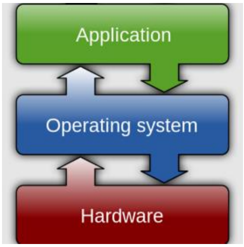
- Mutual Exclusion: (Like lock)
  - Called a "binary semaphore“

```
initial value of semaphore = 1;

semaphore.down();
   // Critical section goes here
semaphore.up();
```

- Signaling other threads, e.g. ThreadJoin

```
Initial value of semaphore = 0

ThreadJoin {
    semaphore.down();
}
```

```
ThreadFinish {
    semaphore.up();
}
```



Producer-Consumer Synchronization

- Mutual exclusion:
  - Only one thread manipulates the buffer data structure at a time

```
Lock mutex;
```

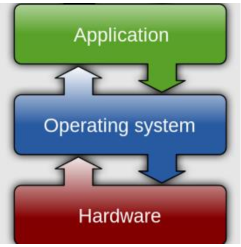
- Synchronization requirements other than mutual exclusion:
  - If buffer is empty, consumer waits for the producer

```
Semaphore usedSlots;
```

  - If buffer is full, producer waits for consumer

```
Semaphore freeSlots;
```

- Rule of thumb: use a separate semaphore for each constraint

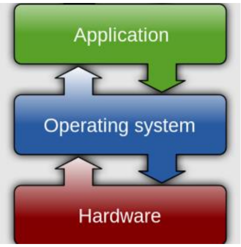


Producer-Consumer Code

```
Semaphore usedSlots = 0;           // No slots used
Semaphore freeSlots = bufSize;     // All slots free
Lock mutex = <initially unlocked>; // Nobody in critical sec.
```

```
Producer(item) {
  freeSlots.down();
  mutex.acquire();
  Enqueue(item);
  mutex.release();
  usedSlots.up();
}
```

```
Consumer() {
  usedSlots.down();
  mutex.acquire();
  item = Dequeue();
  mutex.release();
  freeSlots.up();
  return item;
}
```



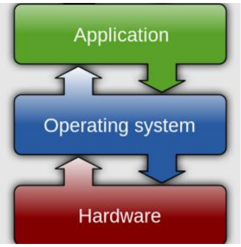
Discussion

- What if we wrote the following?

```
Producer(item) {
  mutex.acquire();
  freeSlots.down();
  Enqueue(item);
  mutex.release();
  usedSlots.up();
}
```

```
Consumer() {
  usedSlots.down();
  mutex.acquire();
  item = Dequeue();
  mutex.release();
  freeSlots.up();
  return item;
}
```

- Deadlock… more on this later



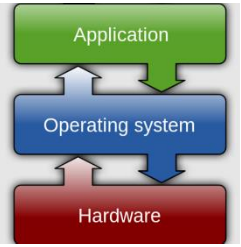
Discussion

- What if we wrote the following?

```
Producer(item) {
  freeSlots.down();
  mutex.acquire();
  Enqueue(item);
  usedSlots.up();
  mutex.release();
}
```

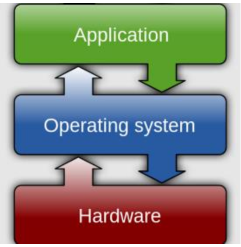
```
Consumer() {
  usedSlots.down();
  mutex.acquire();
  item = Dequeue();
  mutex.release();
  freeSlots.up();
  return item;
}
```

- Still correct!



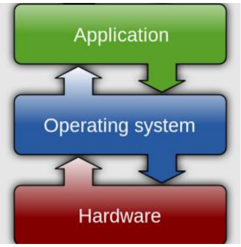
Problems with Semaphores

- More powerful (and primitive) than locks
- Argument: Clearer to have separate constructs for
  - Mutual Exclusion: One thread can do something at a time
  - And waiting for a condition to become true
- But: need to make sure a thread calls `down()` for every `up()`
  - Other tools are more flexible than this



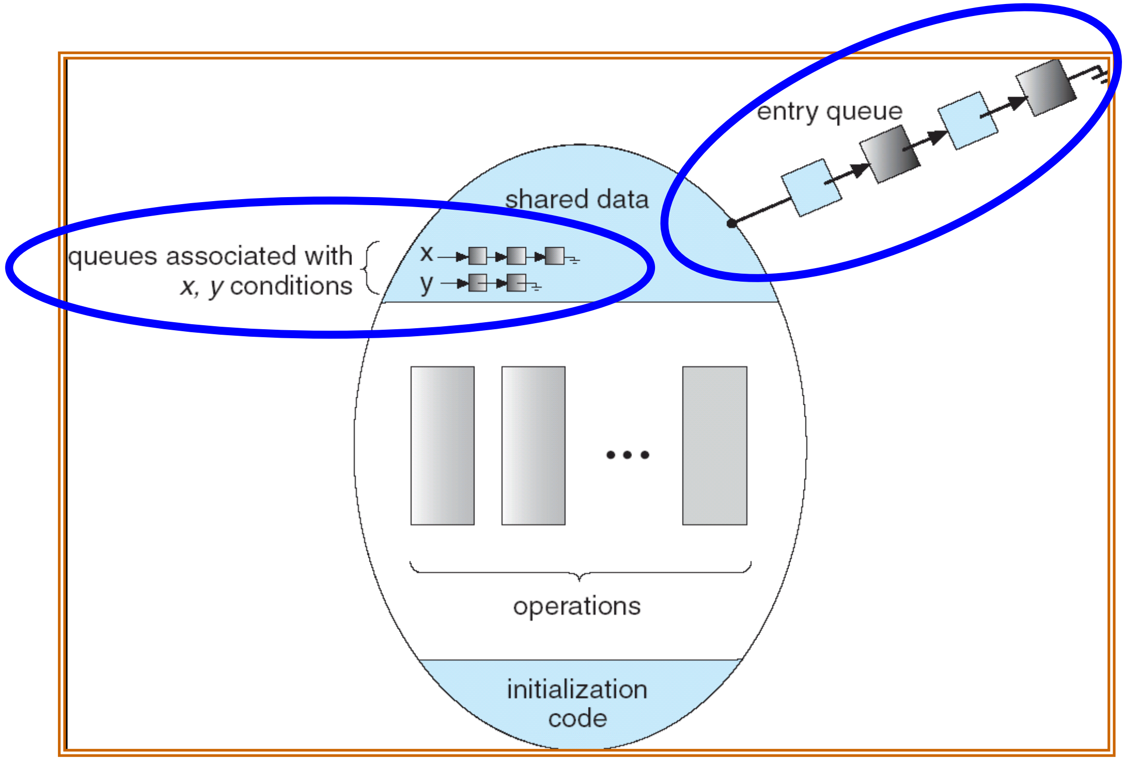
Condition Variables

- Queue of threads waiting inside a critical section
  - Typically, waiting until a condition on some variables becomes true
  - Variables typically are protected by a `mutex`
- Operations:
  - `wait(&lock)`: Atomically release lock and go to sleep until condition variable is signaled. Re-acquire the lock before returning.
  - `signal()`: Wake up one waiting thread (if there is one)
  - `broadcast()`: Wake up all waiting threads
- Rule: Hold lock when using a condition variable

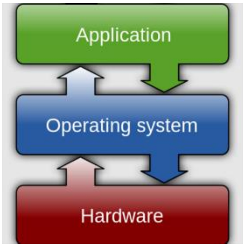


Monitors

- A monitor consists of a lock and zero or more condition variables used for managing concurrent access to shared data

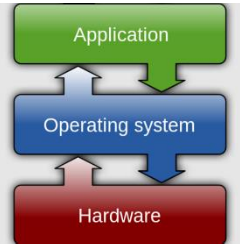


- Lock: the lock provides mutual exclusion to shared data
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at the time we go to sleep



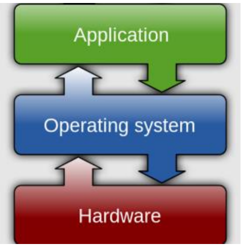
Producer-Consumer with Condition Variables

```
mutex buf_lock = <initially unlocked>
condvar no_longer_empty = <initially empty>
condvar no_longer_full = <initially empty>
Producer(item) {
  acquire(&buf_lock);
  while (buffer full) { cond_wait(&no_longer_full, &buf_lock); }
  enqueue(item);
  cond_signal(&no_longer_empty);
  release(&buf_lock);
}
Consumer() {
  acquire(&buf_lock);
  while (buffer empty) { cond_wait(&no_longer_empty, &buf_lock); }
  item = dequeue();
  cond_signal(&no_longer_full);
  release(&buf_lock);
  return item;
}
```



Why the while Loop?

- When a thread is woken up by `cond_signal()`, it is simply marked as eligible to run
- It may or may not reacquire the lock immediately!
  - Another thread could be scheduled and "sneak in" make the condition it's waiting for no longer true
  - Need a loop to re-check condition on wakeup
- This is called Mesa Scheduling (Mesa-style Monitors)
- Most operating systems use Mesa-style Monitors!



Why the while Loop? (Example)

Thread A (Consumer)

```
acquire(&buf_lock);
while (buffer empty) {
  cond_wait(&not_empty, &buf_lock);



  // while loop checks condition
  // again, goes back to sleep
}
```

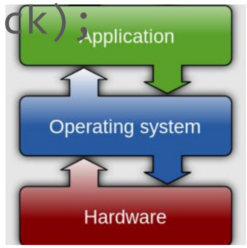
Thread B (Producer)

```
acquire(&buf_lock)
enqueue(item)
cond_signal(&not_empty);
release(&buf_lock);
```

Thread C (Consumer)

```
acquire(&buf_lock);
while (buffer empty)
    cond_wait(...);
dequeue();
release(&buf_lock);
```

This is why the while loop is necessary!



Mesa Monitors

- Signaler keeps lock and CPU
- Waiter placed on ready queue with no special priority

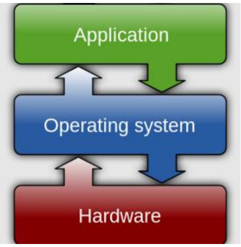
```
…
acquire(&buf_lock)
…
cond_signal(&not_empty);
…
release(&buf_lock);
```

Put waiting thread on ready queue

schedule thread (sometime later!)

```
acquire(&buf_lock);
…
while (is_empty(&queue)) {
  cond_wait(&not_empty, &buf_lock);
}
…
release(&buf_lock);
```

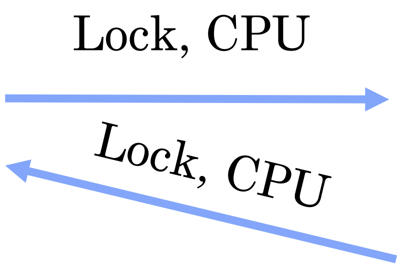
- Practically, need to check condition again after wait
  - By the time the waiter gets scheduled, condition may be false again – so, just check again with the "while" loop
- Most real operating systems do this!
  - Efficient, easy to implement



Alternative: Hoare Monitors

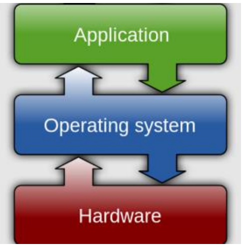
- Named after British logician Tony Hoare
- When a thread calls `signal()`:
  - It releases the lock and the OS context-switches to the waiter, which acquires the lock immediately
  - When waiter releases lock, the OS switches back to signaler

```
…
acquire(&buf_lock);
…
cond_signal(&buf_CV);
…
release(&buf_lock);
```




```
acquire(&buf_lock);
…
if (is_empty(&queue)) {
  cond_wait(&buf_CV, &buf_lock);
}
…
release(&buf_lock);
```

- Academically interesting, but not necessary!
  - Introduces complexity into the scheduler
  - Adds additional context switches



Mesa Monitors vs. Hoare Monitors

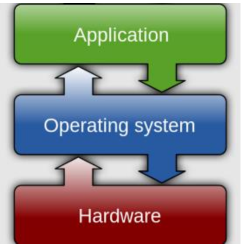
Mesa Monitor

```
while (buffer empty) {
  cond_wait(&not_empty, &buf_lock);
}
```

Hoare Monitor

```
if (buffer empty) {
  cond_wait(&not_empty, &buf_lock);
}
```

- In practice, almost all OSes implement Mesa monitors



Summary: Monitors

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed

- Basic structure of monitor-based program:

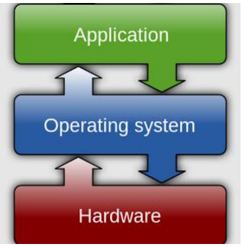
```
lock
while (need to wait) {
    condvar.wait();
}
unlock
```

Check and/or update state variables Wait if necessary

- Trigger doing something so no need to wait

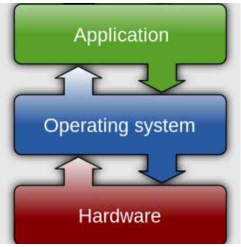
```
lock
condvar.signal();
unlock
```

Check and/or update state variables



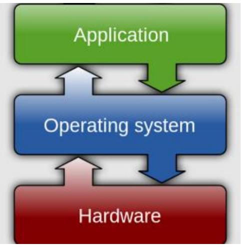
Conclusion

- We studied synchronization primitives to wait until an event
  - Mutual exclusion isn't enough!
- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
    - Some languages support monitors directly
- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed



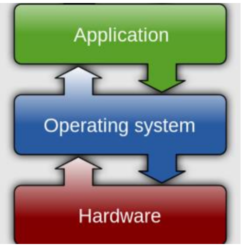
Announcements

- Project 1 was posted, due March 24 (design document due March 10)
  - Walkthrough for project 1 will be March 17
  - Don’t postpone work for this
- Assignment 2 will be available later this week
- Mardi-Gras break: no lecture on March 3
- Midterm review: March 10, midterm exam: March 12



Programming Language Support for Concurrency and Synchronization

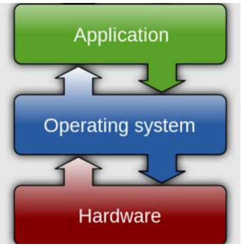
- Synchronization operations
- Exceptional conditions



Concurrency and Synchronization in C

- Standard approach: use pthreads, protect access to shared data structures
- One pitfall: consistently unlocking a mutex

```
int Rtn() {
  lock.acquire();
  …
  if (error) {
    lock.release();
    return errCode;
  }
  …
  lock.release();
  return OK;
}
```



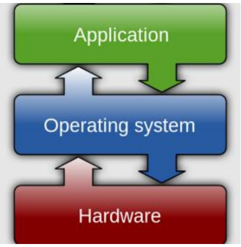
Concurrency and Synchronization in C

- Harder with more locks

```
void Rtn() {
  lock1.acquire();
  …
  if (error) {
    lock1.release();
    return;
  }
  …
  lock2.acquire();
  …
  if (error) {
    lock2.release()
    lock1.release();
    return;
  }
  …
  lock2.release();
  lock1.release();
}
```

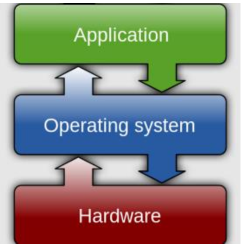
- Is goto a solution???

```
void Rtn() {
  lock1.acquire();
  …
  if (error) {
    goto release_lock1_and_return;
  }
  …
  lock2.acquire();
  …
  if (error) {
    goto release_both_and_return;
  }
  …
release_both_and_return:
  lock2.release();
release_lock1_and_return:
  lock1.release();
}
```



C++ Lock Guards

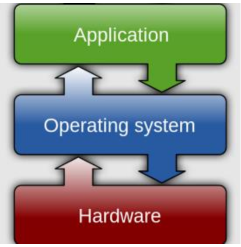
```
#include <mutex>

int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
  std::lock_guard<std::mutex> lock(global_mutex);
  …
  ++global_i;
  // Mutex released when ‘lock’ goes out of scope
}
```



Python Keyword `with`

- More versatile than we'll show here (can be used to close files, database connections, etc.)

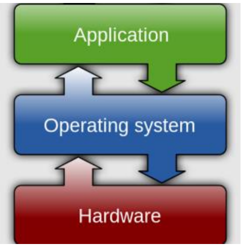
```
lock = threading.Lock()
…
with lock: # Automatically calls acquire()
  some_var += 1
  …
# release() called however we leave block
```



Java synchronized Keyword

- Every Java object has an associated lock:
  - Lock is acquired on entry and released on exit from a synchronized method
  - Lock is properly released if exception occurs inside a synchronized method
  - Mutex execution of synchronized methods (beware deadlock)

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```



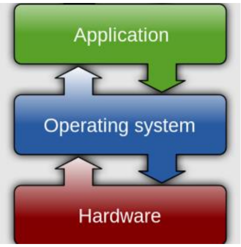
Java Support for Monitors

- Along with a lock, every object has a single condition variable associated with it
- To wait inside a synchronized method:

```
void wait();
void wait(long timeout);
```

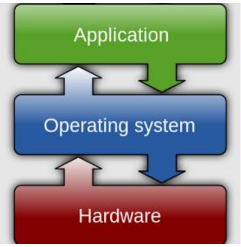
- To signal while in a synchronized method:

```
void notify();
void notifyAll();
```



Go Language Support for Concurrency

- Go was designed with concurrent applications in mind
- Some language aspects we'll talk about today:
  - defer keyword
  - Channels
- Some language aspects we won't talk about today (but may revisit):
  - Goroutines
  - select keyword
  - Contexts



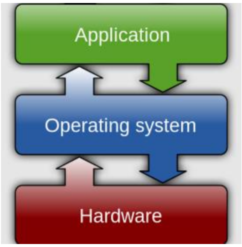
Go defer Keyword

- func Rtn() {

```
func Rtn() {
  lock.Lock()
  …
  if error {
    lock.Unlock()
    return
  }
  …
  lock.Unlock()
  return
}
```

- Solution: use defer
- func Rtn() {
- defer lock.Unlock()

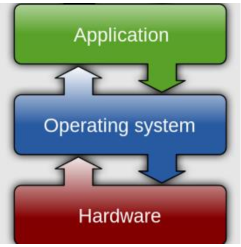
```
func Rtn() {
  lock.Lock()
  defer lock.Unlock()
  …
  if error {
    return
  }
  …
  return
}
```



Go defer Keyword

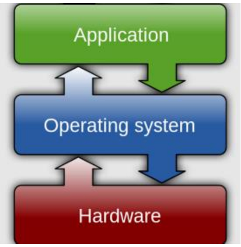
- The queue of “deferred” calls is maintained dynamically
- func Rtn() {
  lock1.Lock()
- defer lock1.Lock()
  …
  if condition {
  lock2.Lock()
- defer lock2.Unlock()
  }
  …
  return
  }

- lock1 is always unlocked here
- lock2 is unlocked here only if the condition was true earlier



Go Channels

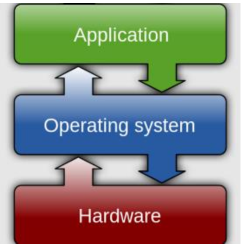
- A channel is a bounded buffer in userspace
  - Writes block if buffer is full
  - Reads block if buffer is empty
- "Do not communicate by sharing memory; instead, share memory by communicating."
  - From Effective Go
- Channels are the preferred mechanism for synchronization
  - Mutexes and condition variables are still supported, as in pthreads



Go Channels

- Semantics similar to pipes, with the following differences:
  - Used within a single process (not across processes)
  - Carries language objects/structs, not bytes (no marshalling/unmarshalling)

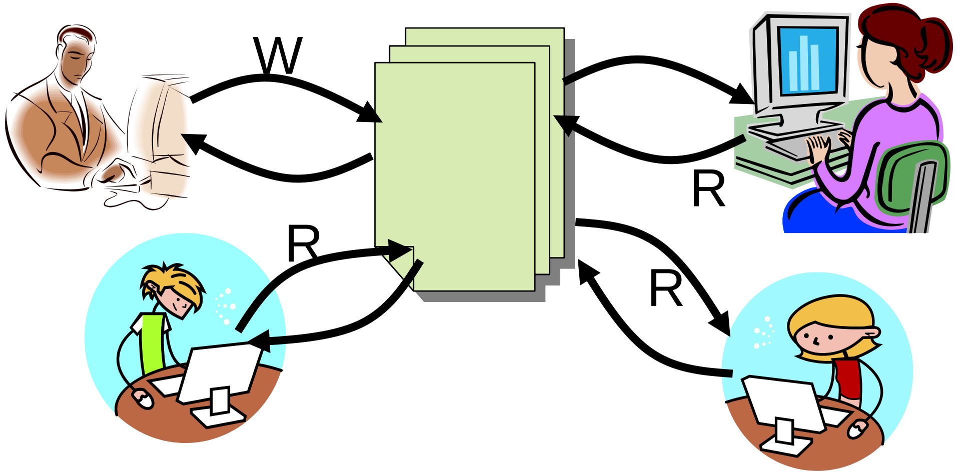
```
var x chan int = make(chan int, 5)
x <- 162
y := <- x
fmt.Println(y) // Prints 162
```

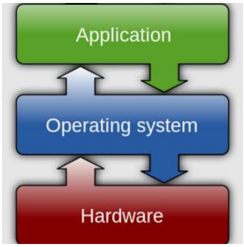


Readers-Writers Lock

Readers-Writers Problem

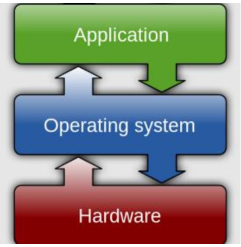
- Consider a shared database
- Two classes of users:
  - Readers – never modify DB
  - Writers – read and modify DB
- Is using a single lock on the whole DB sufficient?
  - Yes, but not ideal
  - Want to allow multiple concurrent readers





Reader-Writer Correctness

- Readers can access when no writers
- Writers can access when no readers and no other writers
- A lock will satisfy these requirements
  - But we want to allow multiple readers
  - Better efficiency

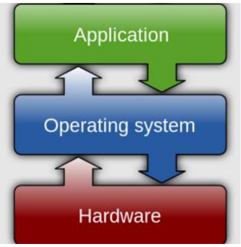


Reader-Writer with Monitors

```
Reader() {
  Wait until no active writers
  Access database
  Maybe wake up a writer
}

Writer() {
  Wait until no active readers or writers
  Access database
  Maybe wakeup reader or writer
}
```

- What synchronization elements do we need?

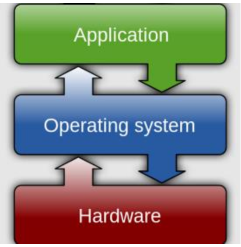
```
Lock (for mutual exclusion)

int activeReaders
condvar okToRead

int activeWriters
condvar okToWrite
```



Reader Version 1

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0) {          // Is it safe to read?
    okToRead.wait(&lock);   // Sleep on cond var
  }
  ++AR;                     // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0)              // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

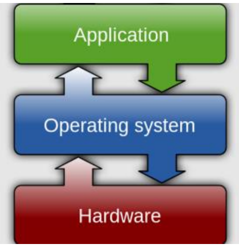
- AR = “Active Readers”
- AW = “Active Writers”



Writer Version 1

```
Writer() {
  // First check self into system
  lock.Acquire();
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    okToWrite.wait(&lock);     // Sleep on cond var
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  okToWrite.signal();          // Wake up one writer
  okToRead.broadcast();        // Wake up all readers
  lock.Release();
}
```

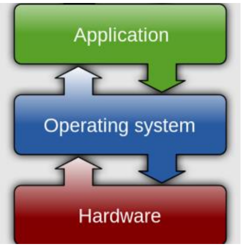
- AR = "Active Readers"
- AW = "Active Writers"



Writer Version 1: Starvation

```
Writer() {
  // First check self into system
  lock.Acquire();
  while (AR > 0 || AW > 0) {    // Is it safe to write?
    okToWrite.wait(&lock);      // Sleep on cond var
  }
  ++AW;                         // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                         // No longer active
  okToWrite.signal();           // Wake up one writer
  okToRead.broadcast();         // Wake up all readers
  lock.Release();
}
```

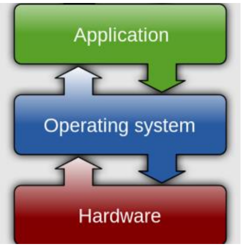
If there are always readers, this is always locked! Writer starves



Writer Version 1: Conflict

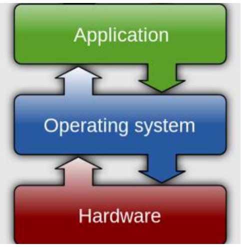
```
Writer() {
  // First check self into system
  lock.Acquire();
  while (AR > 0 || AW > 0) {    // Is it safe to write?
    okToWrite.wait(&lock);      // Sleep on cond var
  }
  ++AW;                         // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                         // No longer active
  okToWrite.signal();           // Wake up one writer
  okToRead.broadcast();         // Wake up all readers
  lock.Release();
}
```

If a writer gets the lock, all the readers wake up anyway, re-check the condition, and go to sleep



Reader-Writer with Monitors, Version 2

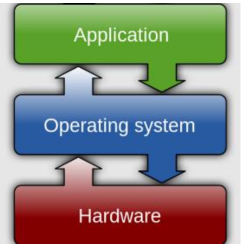
```
Reader() {
  Wait until no active or waiting writers
  Access database
  Maybe wake up a writer
}

Writer() {
  Wait until no active readers or writers
  Access database
  If waiting writer, wake it up
  Otherwise, wakeup readers
}
```



Reader Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

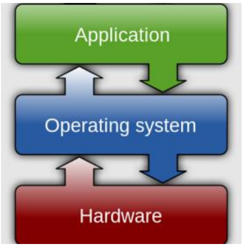
- AR = "Active Readers"
- AW = "Active Writers"
- WR = "Waiting Readers"
- WW = "Waiting Writers"



Writer Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

- AR = "Active Readers"
- AW = "Active Writers"
- WR = "Waiting Readers"
- WW = "Waiting Writers"

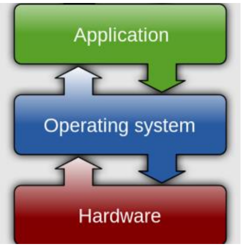


Reader-Writer Design Choices

- Reader starvation:

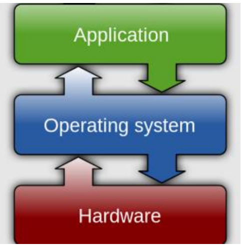
```
while (AW > 0 || WW > 0) {    // Safe to read?
  okToRead.wait(&lock);       // Sleep on cond var
}
```

- “Writer-biased” Lock
  - Can favor readers by changing conditions on wait loops
  - Other possibilities, e.g. track readers waiting since before current writer started



Fair Solution to the Reader-Writer Problem?

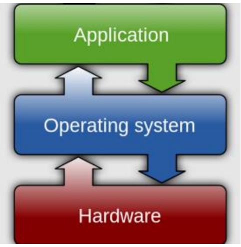
- Ideas?



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);     // Sleep on cond var
    --WR;
  }
  ++AR;                       // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

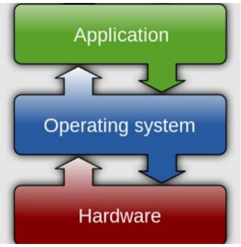
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 0, WR = 0, AW = 0, WW = 0
- R1 comes along (nobody waiting)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);     // Sleep on cond var
    --WR;
  }
  ++AR;                       // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

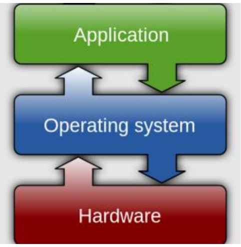
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 0, WR = 0, AW = 0, WW = 0
- R1 comes along (nobody waiting)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

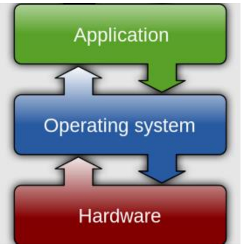
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R1 comes along (nobody waiting)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

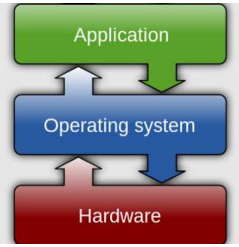
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R1 comes along (nobody waiting)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

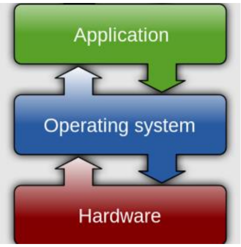
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R1 accesses DB



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

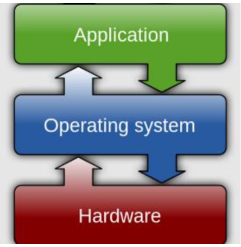
- Sequence of arrivals: R1, **R2**, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R2 comes along (R1 accessing DB)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);     // Sleep on cond var
    --WR;
  }
  ++AR;                       // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

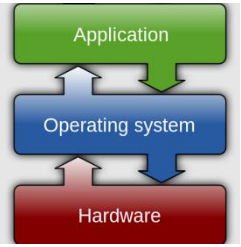
- Sequence of arrivals: R1, R2, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R2 comes along (R1 accessing DB)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

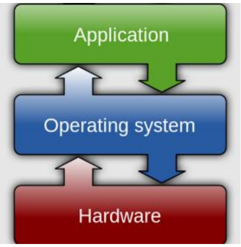
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 0, AW = 0, WW = 0
- R2 comes along (R1 accessing DB)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

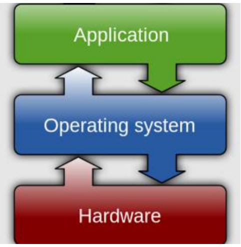
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 0, AW = 0, WW = 0
- R2 comes along (R1 accessing DB)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

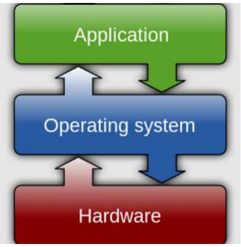
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 0, AW = 0, WW = 0
- R2 & R1 accessing DB



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();       // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

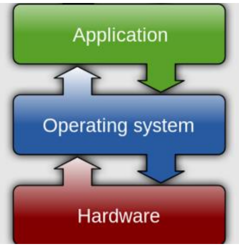
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 0, AW = 0, WW = 0
- W1 comes along (R1 & R2 accessing DB)



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

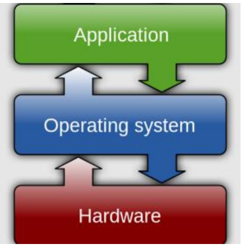
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 0, AW = 0, WW = 0
- W1 comes along (R1 & R2 accessing DB)



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

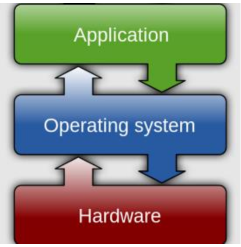
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 0, AW = 0, WW = 1
- W1 comes along (R1 & R2 accessing DB)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

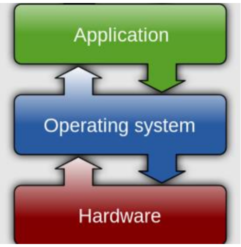
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 0, AW = 0, WW = 1
- R3 comes along (R1 & R2 accessing DB, W1 waiting)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

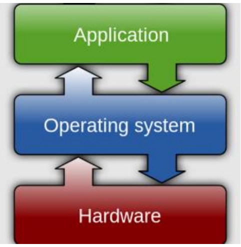
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 0, AW = 0, WW = 1
- R3 comes along (R1 & R2 accessing DB, W1 waiting)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

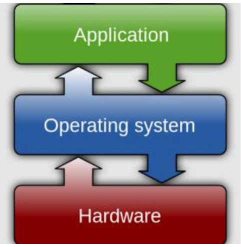
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 1, AW = 0, WW = 1
- R3 comes along (R1 & R2 accessing DB, W1 & R3 waiting)



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

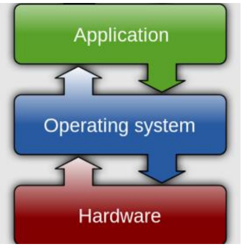
- Sequence of arrivals: R1, R2, W1, R3
- AR = 2, WR = 1, AW = 0, WW = 1
- R2 finishes reading DB
- R1 accessing DB, W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

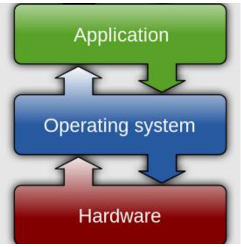
- Sequence of arrivals: R1, R2, W1, R3
- AR = 1, WR = 1, AW = 0, WW = 1
- R2 finishes reading DB
- R1 accessing DB, W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

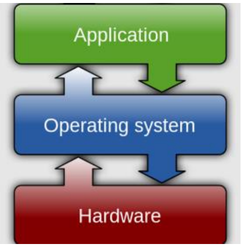
- Sequence of arrivals: R1, R2, W1, R3
- AR = 1, WR = 1, AW = 0, WW = 1
- R2 finishes reading DB
- R1 accessing DB, W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

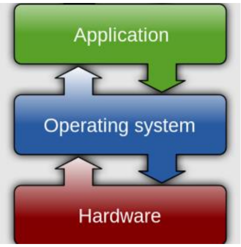
- Sequence of arrivals: R1, R2, W1, R3
- AR = 1, WR = 1, AW = 0, WW = 1
- R2 finishes reading DB
- R1 accessing DB, W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

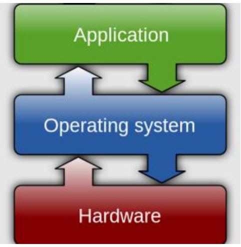
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 1, WR = 1, AW = 0, WW = 1
- R1 finishes reading DB
- W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);     // Sleep on cond var
    --WR;
  }
  ++AR;                       // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                       // No longer active
  if (AR == 0 && WW > 0)      // No other active readers
    okToWrite.signal();       // Wake up one writer
  lock.Release();
}
```

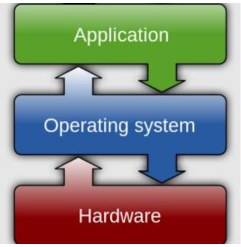
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 1
- R1 finishes reading DB
- W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

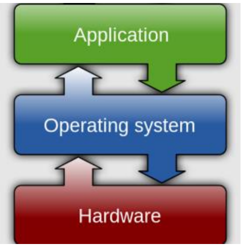
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 1
- R1 finishes reading DB
- W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

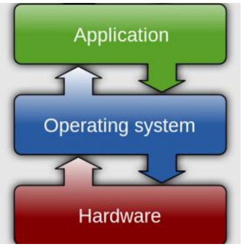
- Sequence of arrivals: **R1**, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 1
- R1 finishes reading DB
- W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

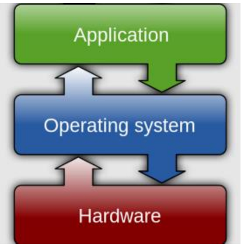
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 1
- R1 finishes reading DB
- W1 & R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

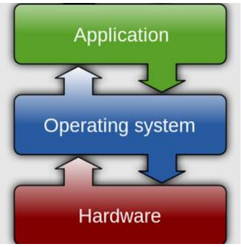
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 0
- W1 continues
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

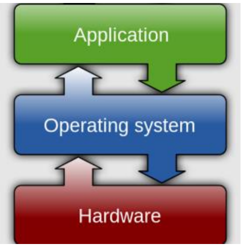
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 1, WW = 0
- W1 continues
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

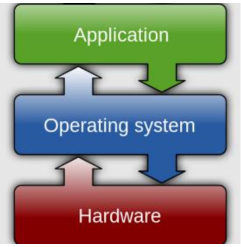
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 1, WW = 0
- W1 continues
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

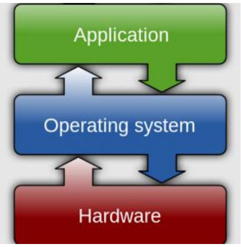
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 1, WW = 0
- W1 accessing DB
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

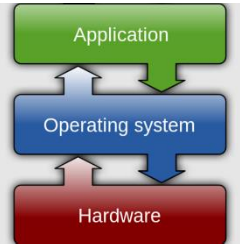
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 1, WW = 0
- W1 finishes
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

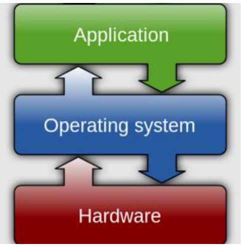
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 0
- W1 finishes
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

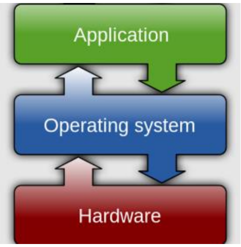
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 0
- W1 finishes
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

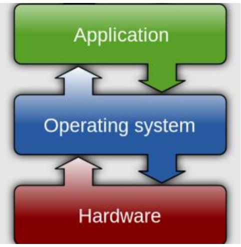
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 0
- W1 finishes
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Writer() {
  lock.Acquire();     // First check self into system
  while (AR > 0 || AW > 0) {   // Is it safe to write?
    ++WW;
    okToWrite.wait(&lock);     // Sleep on cond var
    --WW;
  }
  ++AW;                        // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  --AW;                        // No longer active
  if (WW > 0)
    okToWrite.signal();        // Wake up one writer
  else
    okToRead.broadcast();      // Wake up all readers
  lock.Release();
}
```

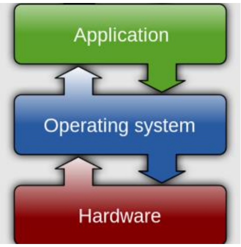
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 1, AW = 0, WW = 0
- W1 finishes
- R3 waiting



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

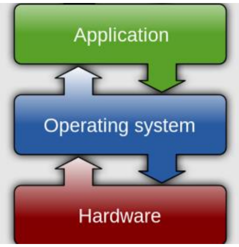
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 0, AW = 0, WW = 0
- R3 continues



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

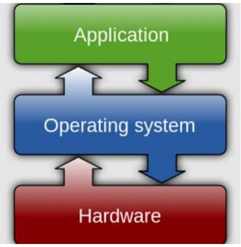
- Sequence of arrivals: R1, R2, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R3 continues



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                      // No longer active
  if (AR == 0 && WW > 0)     // No other active readers
    okToWrite.signal();      // Wake up one writer
  lock.Release();
}
```

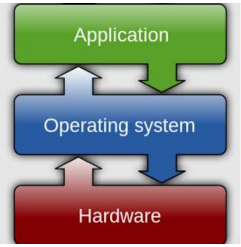
- Sequence of arrivals: R1, R2, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R3 continues



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

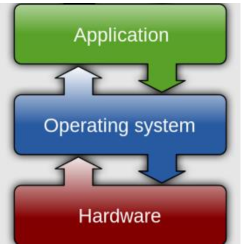
- Sequence of arrivals: R1, R2, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R3 accessing the DB



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

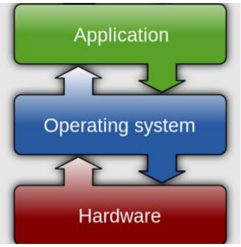
- Sequence of arrivals: R1, R2, W1, R3
- AR = 1, WR = 0, AW = 0, WW = 0
- R3 finishes



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

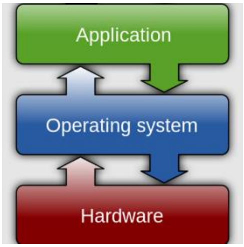
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 0, AW = 0, WW = 0
- R3 finishes



Simulation of Reader-Writer, Version 2

```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

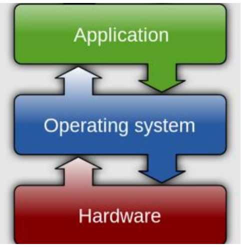
- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 0, AW = 0, WW = 0
- R3 finishes

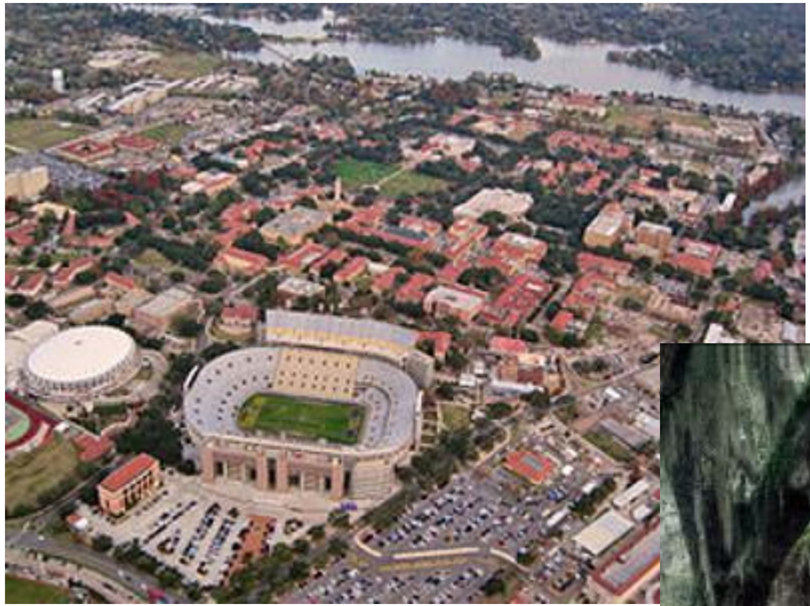


Simulation of Reader-Writer, Version 2

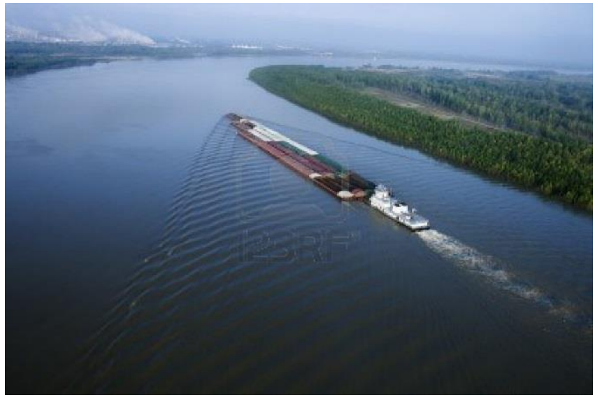
```
Reader() {
  // First check self into system
  lock.Acquire();
  while (AW > 0 || WW > 0) { // Is it safe to read?
    ++WR;
    okToRead.wait(&lock);    // Sleep on cond var
    --WR;
  }
  ++AR;                      // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  --AR;                     // No longer active
  if (AR == 0 && WW > 0)    // No other active readers
    okToWrite.signal();     // Wake up one writer
  lock.Release();
}
```

- Sequence of arrivals: R1, R2, W1, R3
- AR = 0, WR = 0, AW = 0, WW = 0
- R3 finishes





LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP