

# Operating Systems

## Lecture 10

### lock and conditional variable design

Prof. Mengwei Xu

Much of the contents are from Prof. Ion Stoica (cs162@Berkeley)

# Recap: Scheduling

---

- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- Shortest Job First (SJF) / Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair

# Recap: Scheduling

---

- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks  $\Rightarrow$  more tokens)
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Real-time scheduling
  - Need to meet a deadline, predictability essential
  - Earliest Deadline First (EDF) and Rate Monotonic (RM) scheduling



# Numbering from zero

Why numbering should start at zero

[EWD831.html](#)

To denote the subsequence of natural numbers  $2, 3, \dots, 12$  without the pernicious three dots, four conventions are open to us:

- a)  $2 \leq i < 13$
- b)  $1 < i \leq 12$
- c)  $2 \leq i \leq 12$
- d)  $1 \leq i < 13$

Are there reasons to prefer one convention to the other? Yes, there are. The observation that conventions a) and b) have the advantage that the difference between the bounds as mentioned equals the length of the subsequence is valid. So is the observation that, as a consequence, in either convention two subsequences are adjacent means that the upper bound of the one equals the lower bound of the other. Valid as these observations are, they don't enable us to choose between a) and b); so let us start afresh.

There is a smallest natural number. Exclusion of the lower bound -as in b) and d)- forces for a subsequence starting at the smallest natural number the lower bound as mentioned into the realm of the unnatural numbers. That is ugly, so for the lower bound we prefer  $\leq$  as in a) and c). Consider now the subsequences starting at the smallest natural number: inclusion of the upper bound would then force the latter to be unnatural by the time the sequence has shrunk to the empty one. That is ugly, so for the upper bound we prefer  $<$  as

in a) and d). We conclude that convention a) is to be preferred.

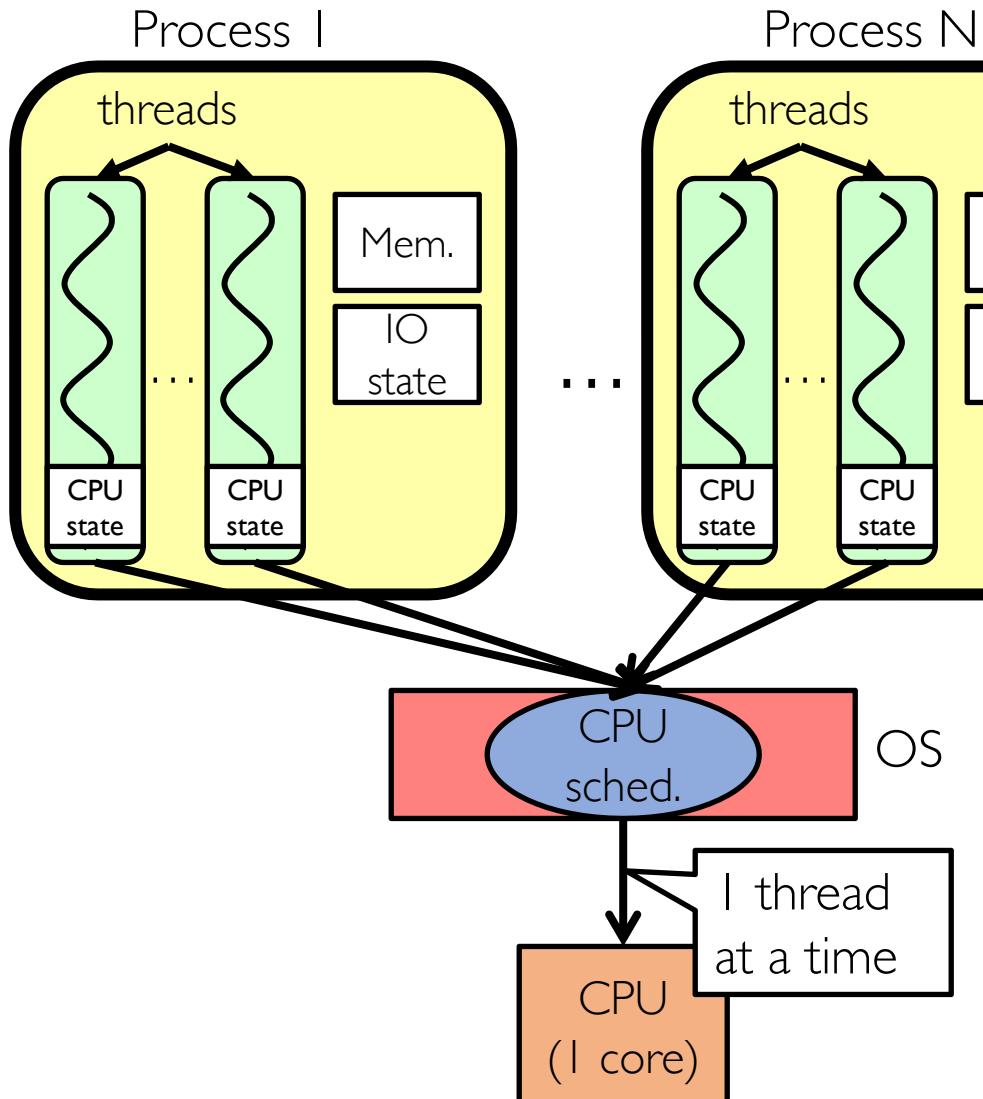
Remark. The programming language Mesa, developed at Xerox PARC, has special notations for intervals of integers in all four conventions. Extensive experience with Mesa has shown that the use of the other three conventions has been a constant source of clumsiness and mistakes, and on account of that experience Mesa programmers are now strongly advised not to use the latter three available features. I mention this experimental evidence -for what it is worth- because some people feel uncomfortable with conclusions that have not been confirmed in practice. (End of Remark.)

\* \* \*

When dealing with a sequence of length  $N$ , the elements of which we wish to distinguish by subscript, the next vexing question is what subscript value to assign to its starting element. Adhering to convention a) yields, when starting with subscript 1, the subscript range  $1 \leq i < N+1$ ; starting with 0, however, gives the nicer range  $0 \leq i < N$ . So let us let our ordinals start at zero: an element's ordinal (subscript) equals the number of elements preceding it in the sequence. And the moral of the story is that we had better regard -after all those centuries!- zero as a most natural number.

Remark. Many programming languages have been designed without due attention to this detail. In FORTRAN, subscripts always start at 1; in AL-

# OS Conceptual Framework



- Physical addresses shared
  - So: Processes and Address Translation
- CPU must be Shared
  - So: Threads
- Processes aren't trusted
  - So: Kernel/Userspace Split
- Threads might not cooperate
  - So: Use timer interrupts to context switch ("preemption")

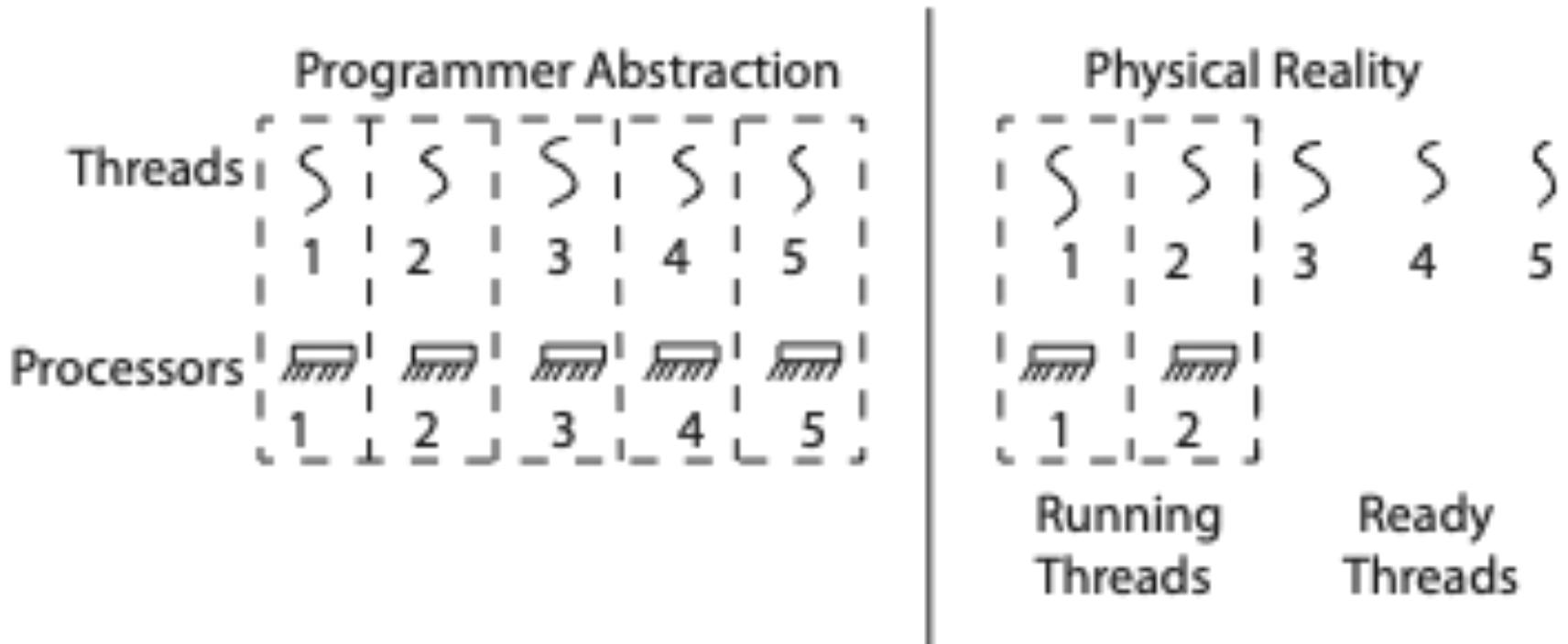


# Goals for Today

---

- Motivating synchronization: why it's difficult
- Locks
- Condition variables
- Semaphores

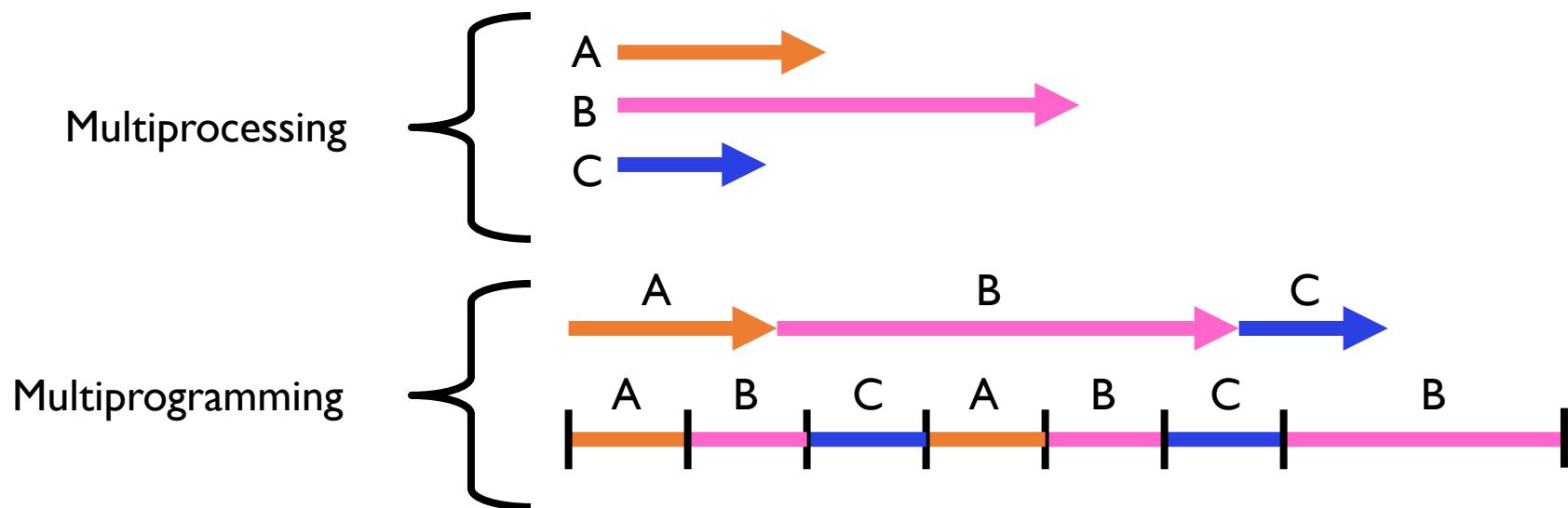
# Recall: Thread Abstraction



- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule

# Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs or cores or hyperthreads (HW per-instruction interleaving)
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads “concurrently”?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...





# Why Allow Cooperating Threads?

---

- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    - Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - To compile, for instance, **gcc** calls **cpp** | **cc1** | **cc2** | **as** | **ld**
    - Makes system easier to extend



# Correctness for Systems with Concurrency

---

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic  $\Rightarrow$  Input state determines results
  - Reproducible  $\Rightarrow$  Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called “Heisenbugs”

# Interactions Complicate Debugging

---

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc.
  - Extreme example: buggy device driver causes thread A to crash “independent thread” B
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    - Depends on scheduling, which depends on timer/other things
    - Original UNIX had a bunch of non-deterministic errors

# Problem is at the Lowest Level

---

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A

$x = 1;$

Thread B

$y = 2;$

- However, what about (Initially,  $x = 12$ ):

Thread A

$x = 1;$

Thread B

$x = 2;$

- $X$  could be 1 or 2 (non-deterministic!)
- Could even be 3 for serial processors:
  - Thread A writes 0001, B writes 0010 → scheduling order ABABABBA yields 3!

# Problem is at the Lowest Level

- A more complex case (Initially,  $x = 0$ ):

Thread A

$x = x + l;$

```
load r1, x  
add r2, r1, l  
store x, r2
```

Thread B

$x = x + 2;$

```
load r1, x  
add r2, r1, 2  
store x, r2
```

- What are the possible outputs?

# Problem is at the Lowest Level

- A more complex case (Initially,  $x = 0$ ):

Thread A

$x = x + 1;$

Thread B

$x = x + 2;$

```
load r1, x  
add r2, r1, 1  
store x, r2
```

```
load r1, x  
add r2, r1, 2  
store x, r2
```

```
load r1, x
```

```
add r2, r1, 1
```

```
store x, r2
```

```
load r1, x
```

```
add r2, r1, 2
```

```
store x, r2
```

```
load r1, x  
add r2, r1, 1
```

```
load r1, x
```

```
add r2, r1, 2
```

```
store x, r2
```

```
store x, r2
```

Final:  $x = 3$

Final:  $x = 2$

Final:  $x = 1$

# What's Worse: Reordered Instructions by Compiler

- Compilers could reorder the instructions to maximize the instruction level parallelism.
  - Yet, it only ensures the dependency correctness within a thread, not across threads.
  - `pInitialized` could be set to true before `funcA()`.

## Thread A

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

## Thread B

```
y = 2;  
while(!pInitialized); // wait  
q = funcB(p)
```

# Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- **Atomic Operation (原子操作)**: an operation that always runs to completion or not at all
  - It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
  - Consequently – weird example that produces “3” on previous slide can’t happen
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

# Motivation: “Too Much Milk”

- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:



Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

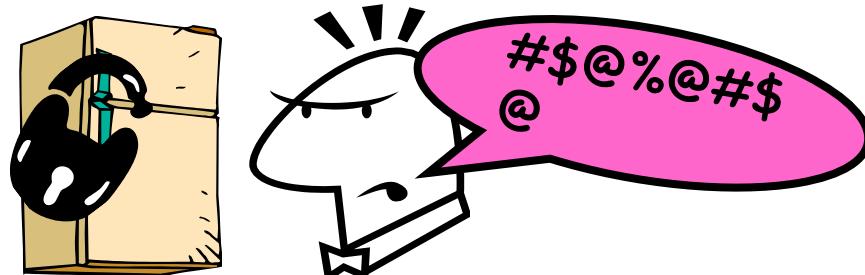
# Definitions

---

- **Synchronization (同步)**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that it's hard to build anything useful with only reads and writes
- **Mutual Exclusion (互斥)**: ensuring that only one thread does a particular thing at a time
  - One thread excludes the other while doing its task
- **Critical Section (临界区)**: piece of code that only one thread can execute at once.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing

# Definitions

- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants orange



- Of Course – We don't know how to make a lock yet



# Too Much Milk: Correctness Properties

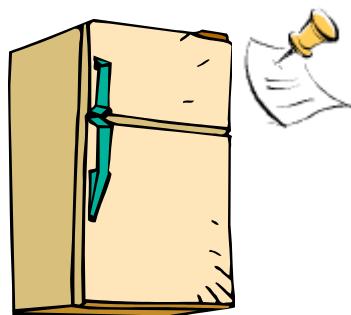
---

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
  - Always write down behavior first
- What are the correctness properties for the “Too much milk” problem??
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

# Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {  
    if (noNote) {  
        leave Note;  
        buy milk;  
        remove note;  
    }  
}
```



# Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

Thread A

```
if (noMilk) {  
    if (noNote) {  
        leave Note;  
        buy Milk;  
        remove Note;  
    }  
}
```

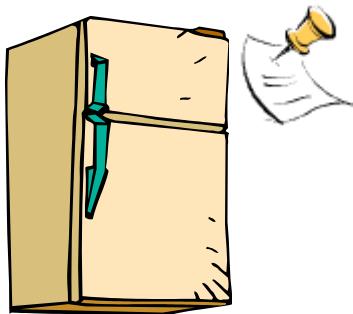
Thread B

```
if (noMilk) {  
    if (noNote) {  
        leave Note;  
        buy Milk;  
        remove Note;  
    }  
}
```

# Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {  
    if (noNote) {  
        leave Note;  
        buy milk;  
        remove note;  
    }  
}
```



- Result?
  - Still too much milk **but only occasionally!**
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails **intermittently**
  - Makes it really hard to debug...
  - Must work despite what the dispatcher does!

# Too Much Milk: Solution #1½

---

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;  
if (noMilk) {  
    if (noNote) {  
        buy milk;  
    }  
}  
remove Note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

# Too Much Milk Solution #2

---

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Thread A

```
leave note A;  
if (noNote B) {  
    if (noMilk) {  
        buy Milk;  
    }  
}  
remove note A;
```

Thread B

```
leave note B;  
if (noNoteA) {  
    if (noMilk) {  
        buy Milk;  
    }  
}  
remove note B;
```

- Does this work?

# Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Thread A

```
leave note A;  
if (noNote B) {  
    if (noMilk) {  
        buy Milk;  
    }  
}  
remove note A;
```

Thread B

```
leave note B;  
if (noNoteA) {  
    if (noMilk) {  
        buy Milk;  
    }  
}  
remove note B;
```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - *Extremely unlikely* this would happen, but will at worse possible time
  - Probably something like this in UNIX

# Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;  
while (note B) {\\"X  
    do nothing;  
}  
if (noMilk) {  
    buy milk;  
}  
remove note A;
```

Thread B

```
leave note B;  
if (noNote A) {\\"Y  
    if (noMilk) {  
        buy milk;  
    }  
    remove note B;
```

- Does this work?

# Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;  
while (note B) { \\X  
    do nothing;  
}  
if (noMilk) {  
    buy milk;  
}  
remove note A;
```

Thread B

```
leave note B;  
if (noNote A) { \\Y  
    if (noMilk) {  
        buy milk;  
    }  
    remove note B;
```

- Does this work? **Yes.** Both can guarantee that:

- It is safe to buy, or
- Other will buy, ok to quit

- At X:

- If no note B, safe for A to buy,
- Otherwise wait to find out what will happen

- At Y:

- If no note A, safe for B to buy
- Otherwise, A is either buying or waiting for B to quit

# Case I

- “leave note A” happens before “if (noNote A)”

```
leave note A;  
while (note B) {\\"X  
    do nothing;  
};  
  
leave note B;  
if (noNote A) {\\"Y  
    if (noMilk) {  
        buy milk;  
    }  
    remove note B;
```

A blue arrow points from the "leave note A" block to the "if (noNote A)" block, labeled "happened before".

```
if (noMilk) {  
    buy milk; }  
}  
remove note A;
```

# Case I

- “leave note A” happens before “if (noNote A)”

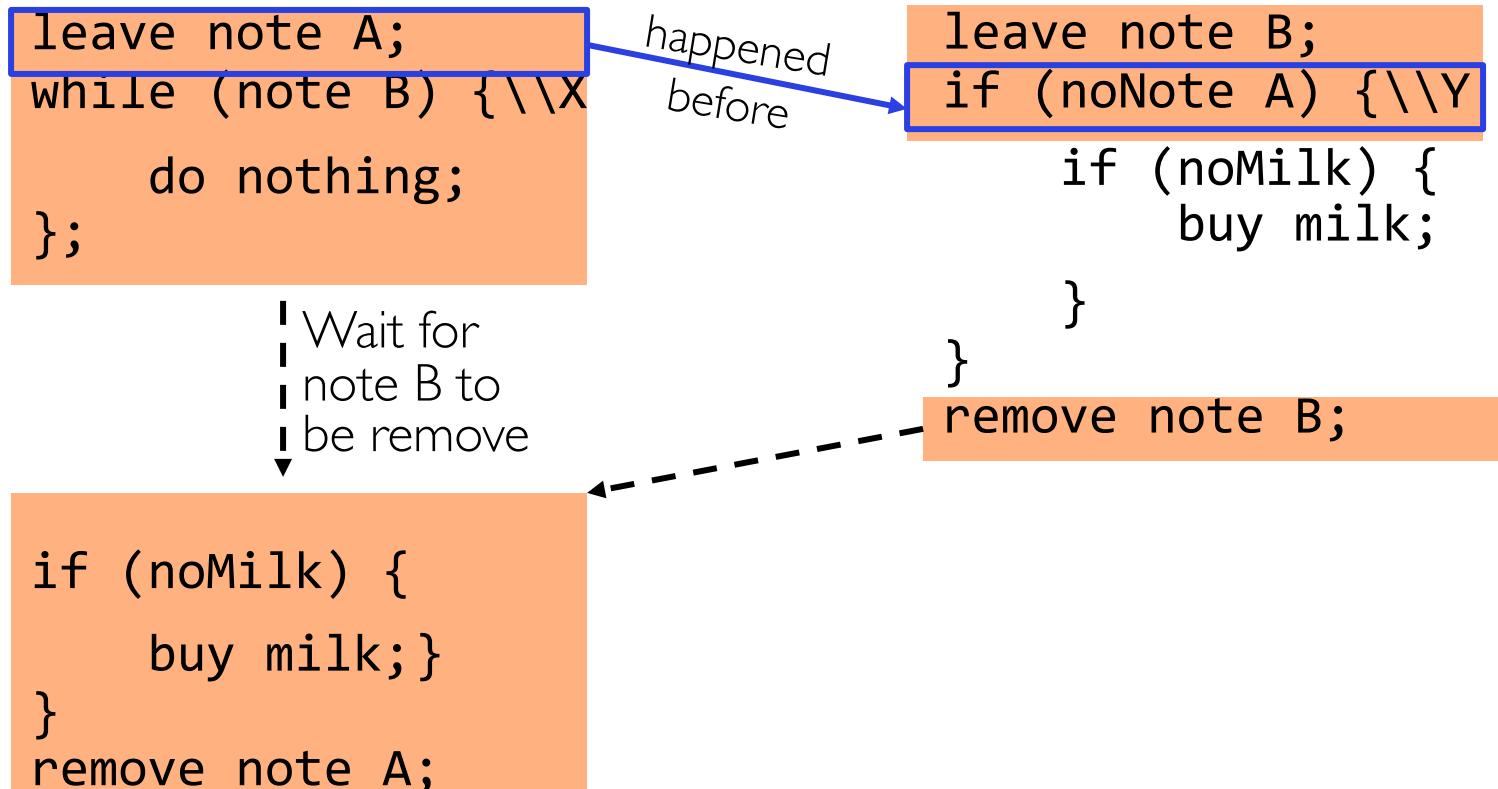
```
leave note A;  
while (note B) {\\"X  
    do nothing;  
};  
  
leave note B;  
if (noNote A) {\\"Y  
    if (noMilk) {  
        buy milk;  
    }  
    remove note B;
```

A blue arrow points from the "leave note A" line in the first code block to the "if (noNote A)" line in the second code block, with the label "happened before" written above it.

```
if (noMilk) {  
    buy milk; }  
}  
remove note A;
```

# Case I

- “leave note A” happens before “if (noNote A)”



## Case 2

- “if (noNote A)” happens before “leave note A”

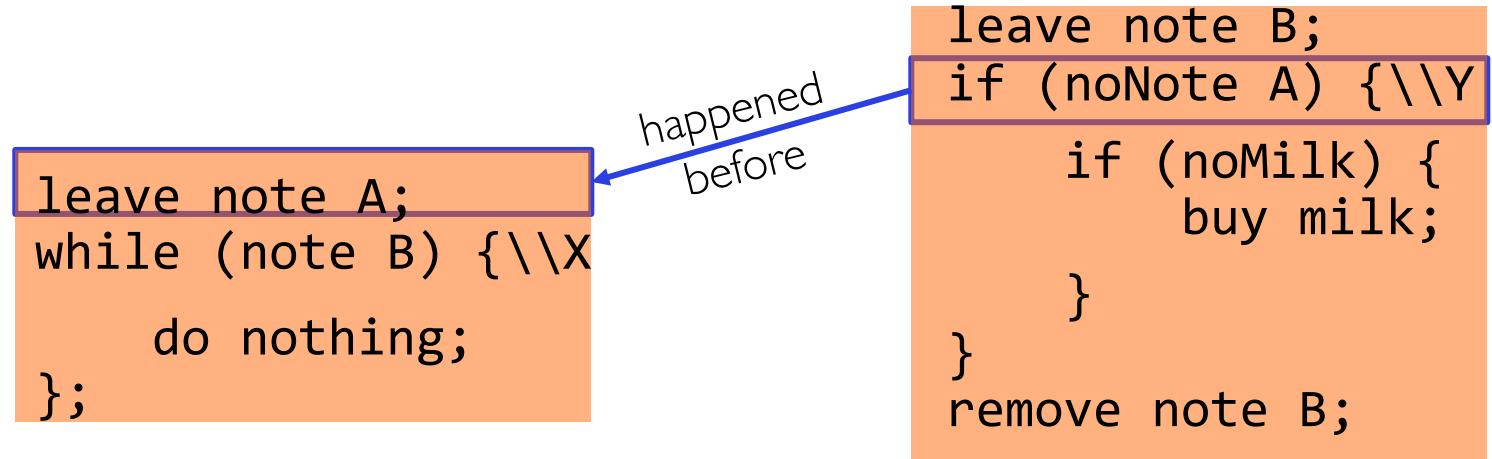
```
leave note A;           leave note B;  
while (note B) {\\"X      if (noNote A) {\\"Y  
    do nothing;          if (noMilk) {  
};                      buy milk;  
                           }  
                           remove note B;
```

A blue arrow points from the "leave note A;" line to the "if (noNote A) {\\"Y" line, with the text "happened before" written above it.

```
if (noMilk) {  
    buy milk; }  
}  
remove note A;
```

## Case 2

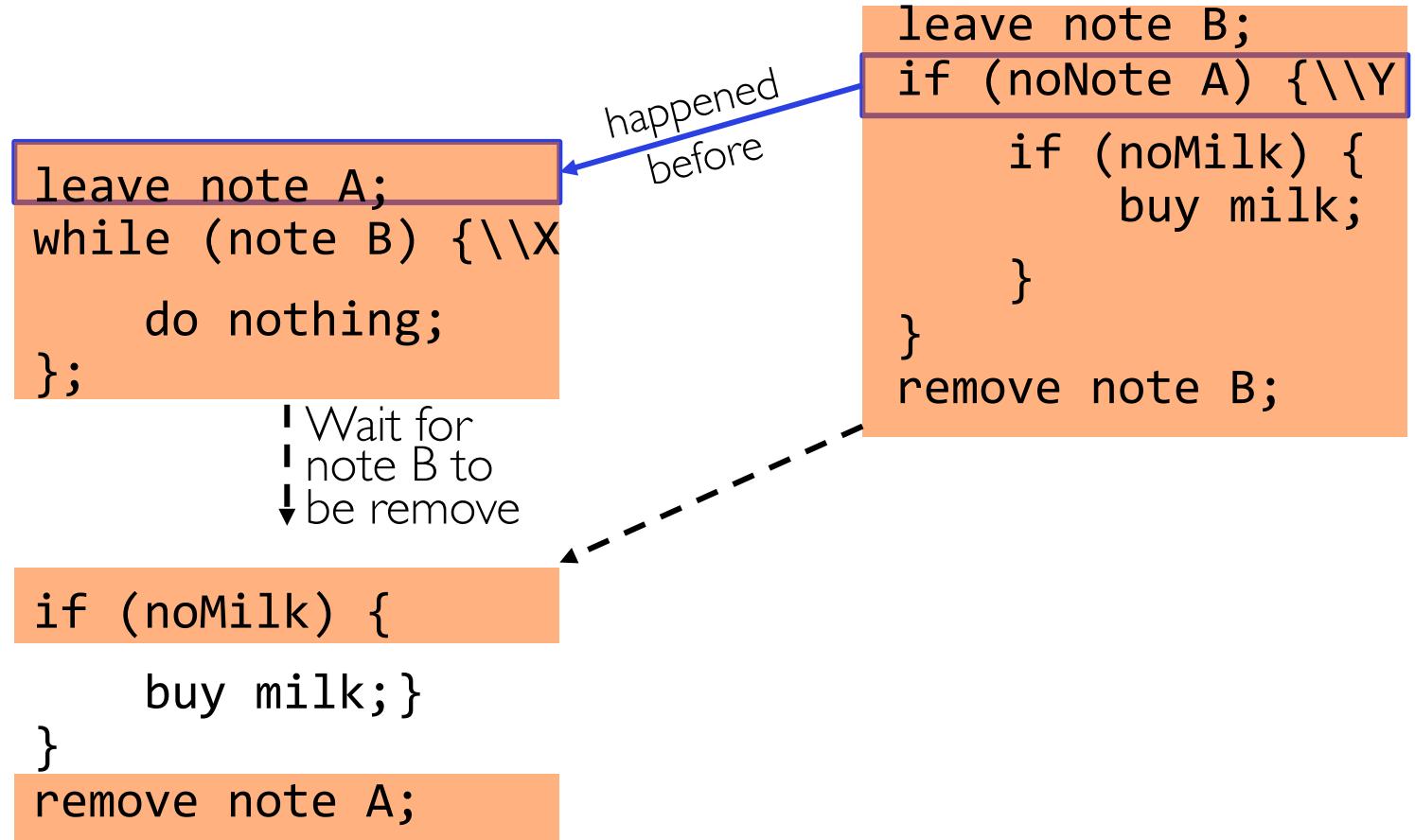
- “if (noNote A)” happens before “leave note A”



```
if (noMilk) {
    buy milk;
}
remove note A;
```

## Case 2

- “if (noNote A)” happens before “leave note A”





# Solution #3 Discussion

---

- Our solution protects a single “Critical-Section” piece of code for each thread:

```
if (noMilk) {  
    buy milk;  
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple example
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called “busy-waiting”
- There's a better way
  - Have hardware provide higher-level primitives than atomic load & store
  - Build even higher-level programming abstractions on this hardware support



# Too Much Milk: Solution #4

---

- Suppose we have some sort of implementation of a lock
  - `lock.Acquire()` – wait until lock is free, then grab
  - `lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:

```
milklock.Acquire();  
if (nomilk)  
    buy milk;  
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a “`Critical Section`”

# Where are we going with synchronization?

Programs	Shared Programs			
Higher-level API	Locks Semaphores Monitors Send/Receive			
Hardware	Load/Store Disable Ints Test&Set Compare&Swap			

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

# Lock

---

- Suppose we have some sort of implementation of a lock
  - `lock.Acquire()` – wait until lock is free, then grab
  - `lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- 3 formal properties
  - Mutual exclusion: at most one thread holds the lock
  - Progress: if no thread holds the lock and any thread attempts to acquire the lock, then eventually some thread succeeds in acquiring the lock
  - Bounded waiting: if 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
    - Yet, it does not promise that waiting threads acquire the lock in FIFO order.

# What does a Lock Guarantee?

- A simple case of lock.
  - Assuming  $x$  is shared among threads
  - Other threads only access  $x$  with lock

```
int x = 0;  
// T1: can we ensure x = 0 here?  
lock.acquire();  
// T2: can we ensure x = 0 here?  
x = 1;  
// T3: can we ensure x = 1 here?  
lock.release();  
// T4: can we ensure x = 1 here?  
x = 2;  
// T5: can we ensure x = 2 here?
```

# What does a Lock Guarantee?

- A simple case of lock.
  - Assuming x is shared among threads
  - Other threads only access x with lock

**If a lock is not held, nothing can be guaranteed!**

```
int x = 0;  
// T1: can we ensure x = 0 here?  
  
lock.acquire();  
// T2: can we ensure x = 0 here?  
  
x = 1;  
// T3: can we ensure x = 1 here?  
  
lock.release();  
// T4: can we ensure x = 1 here?  
  
x = 2;  
// T5: can we ensure x = 2 here?
```



# Condition Variable

---

- Condition Variable (条件变量): a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
- Operations:
  - **Wait(&lock)**: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - **Signal()**: Wake up one waiter, if any
  - **Broadcast()**: Wake up all waiters
  - Differentiate them from UNIX `wait` and `signal`

# Condition Variable Example

- Condition Variable (条件变量): a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
- A common pattern:

```
FuncA_wait() {  
    lock.acquire();  
    // read/write shared state here  
    while (!testOnSharedState())  
        cv.wait(&lock);  
    assert(testOnSharedState());  
    lock.release();  
}
```

```
FuncB_signal() {  
    lock.acquire();  
    // read/write shared state here  
    // If state has changed that allows  
    // another thread to make progress, call  
    // signal or broadcast  
    cv.signal();  
    lock.release();  
}
```



# Condition Variable Example

- A concrete example of bounded queue implementation (or producer-consumer, 生产者消费者)

```
class bounded_queue {  
    Lock lock;  
    CV itemAdded;  
    CV itemRemoved;  
    void insert(int item);  
    int remove();  
}
```

```
void bounded_queue::insert(int item) {  
    lock.acquire();  
    while (queue.full()) {  
        itemRemoved.wait(&lock);  
    }  
    add_item(item);  
    itemAdded.signal();  
    lock.release();  
}
```

How to implement remove()?



# Condition Variable Example

- A concrete example of bounded queue implementation (or producer-consumer, 生产者消费者)
- Two key principles
  - CV is always used with lock acquired
  - CV is put in a while loop. Why?

```
void bounded_queue::insert(int item) {  
    lock.acquire();  
    while (queue.full()) {  
        itemRemoved.wait(&lock);  
    }  
    add_item(item);  
    itemAdded.signal();  
    lock.release();  
}
```

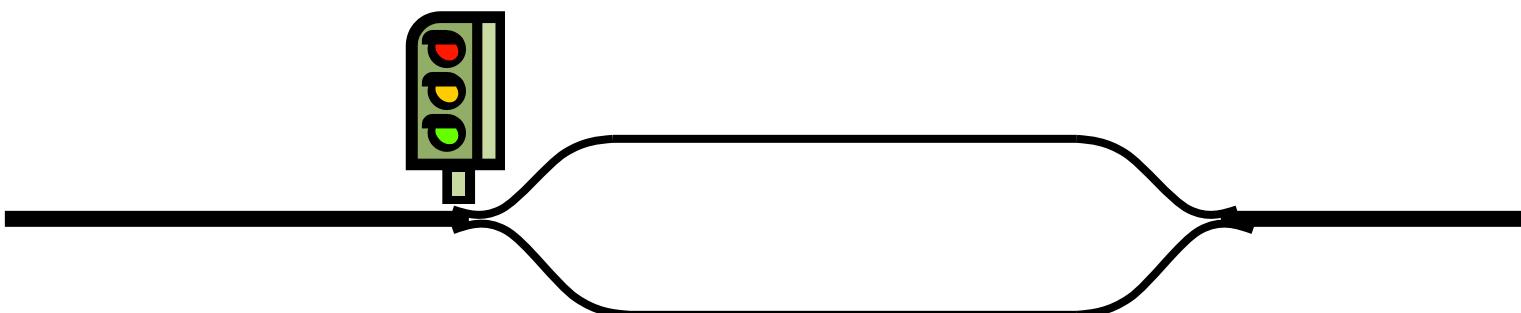
# Semaphores

---

- Semaphores (信号量) are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - $P()$ : an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
  - $V()$ : an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - Think of this as the signal() operation
  - Note that  $P()$  stands for “proberen” (to test) and  $V()$  stands for “verhogen” (to increment) in Dutch

# Semaphores vs. Integers

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - Two P's together can't decrement value below zero
    - Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



# Two Uses of Semaphores

## I. Mutual Exclusion (initial value = 1)

- Also called “Binary Semaphore”.
- Can be used for mutual exclusion:

```
semaphore.P();  
// Critical section goes here  
semaphore.V();
```

## 2. Scheduling Constraints (initial value = 0)

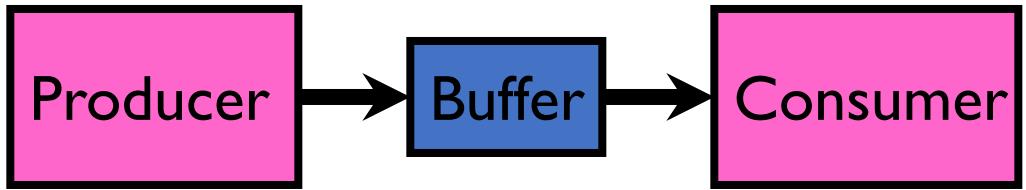
- Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 **schedules** thread 1 when a given **event** occurs
- Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0  
ThreadJoin {  
    semaphore.P();  
}  
ThreadFinish {  
    semaphore.V();  
}
```



# Producer-Consumer with a Bounded Buffer

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- Example 1: GCC compiler
  - `cpp | cc1 | cc2 | as | ld`
- Example 2: Coke machine
  - Producer can put limited number of Cokes in machine
  - Consumer can't take Cokes out if machine is empty



# Correctness constraints for solution

---

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb:  
**Use a separate semaphore for each constraint**
  - Semaphore fullSlots; // consumer's constraint
  - Semaphore emptySlots; // producer's constraint
  - Semaphore mutex; // mutual exclusion



# Full Solution to Bounded Buffer

```
Semaphore fullSlots = ?;  
Semaphore emptySlots = ?;  
  
Semaphore mutex = 1;           // No one using machine  
  
Producer(item) {  
    emptySlots.P();           // Wait until space  
    mutex.P();                // Wait until machine free  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();            // Tell consumers there is  
                            // more coke  
}  
  
Consumer() {  
    fullSlots.P();            // Check if there's a coke  
    mutex.P();                // Wait until machine free  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();           // tell producer need more  
    return item;  
}
```



# Full Solution to Bounded Buffer

```
Semaphore fullSlots = 0;      // Initially, no coke
Semaphore emptySlots = bufSize;           // Initially, num empty slots
Semaphore mutex = 1;                // No one using machine

Producer(item) {
    emptySlots.P();                  // Wait until space
    mutex.P();                      // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V();                 // Tell consumers there is
                                    // more coke
}

Consumer() {
    fullSlots.P();                  // Check if there's a coke
    mutex.P();                      // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V();                // tell producer need more
    return item;
}
```

# Discussion about Solution

Why asymmetry?

- Producer does: `emptySlots.P()`  
`fullSlots.V()`
- Consumer does: `fullSlots.P()`,  
`emptySlots.V()`

Decrease # of  
empty slots

Increase # of  
occupied slots

Decrease # of  
occupied slots

Increase # of  
empty slots



# Discussion about Solution

---

Is order of P's important?

Is order of V's important?

What if we have 2 producers or 2 consumers?

```
Producer(item) {  
    mutex.P();  
    emptySlots.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}  
  
Consumer() {  
    fullSlots.P();  
    mutex.P();  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();  
    return item;  
}
```



# Discussion about Solution

Is order of P's important?

- Yes! Can cause deadlock

Is order of V's important?

- No, except that it might affect scheduling efficiency

What if we have 2 producers or 2 consumers?

- Do we need to change anything?

```
Producer(item) {  
    mutex.P();  
    emptySlots.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}  
  
Consumer() {  
    fullSlots.P();  
    mutex.P();  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();  
    return item;  
}
```



# Some Advices

---

- Always acquire the lock at the beginning of a method and release it right before the return
  - Consistent behavior makes it easier to program
  - Also makes it easier to read and debug



# Some Advices

- Always acquire the lock at the beginning of a method and release it right before the return
  - Consistent behavior makes it easier to program
  - Also makes it easier to read and debug
- A case: double-checked locking

```
Singleton* Singleton::instance() {  
    if (pInstance == NULL) {  
        pInstance = new Instance();  
    }  
    return pInstance;  
}
```

An unsafe solution

```
Singleton* Singleton::instance() {  
    lock.acquire();  
    if (pInstance == NULL) {  
        pInstance = new Instance();  
    }  
    lock.release();  
    return pInstance;  
}
```

A safe solution

```
Singleton* Singleton::instance() {  
    if (pInstance == NULL) {  
        lock.acquire();  
        if (pInstance == NULL) {  
            pInstance = new Instance();  
        }  
        lock.release();  
    }  
    return pInstance;  
}
```

An ``optimized'' solution.  
**Is it safe?**

# Common Concurrency Problems

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: Bugs In Modern Applications

[1] “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics”, Shan Lu, et al. ASPLOS’08



# Common Concurrency Problems

- Atomicity-Violation Bugs

```
1 Thread 1::  
2 if (thd->proc_info) {  
3     fputs(thd->proc_info, ...);  
4 }  
5  
6 Thread 2::  
7 thd->proc_info = NULL;
```



# Common Concurrency Problems

- Atomicity-Violation Bugs

```
1 Thread 1::  
2 if (thd->proc_info) {  
3     fputs(thd->proc_info, ...);  
4 }  
5  
6 Thread 2::  
7 thd->proc_info = NULL;
```

```
1 pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;  
2  
3 Thread 1::  
4 pthread_mutex_lock(&proc_info_lock);  
5 if (thd->proc_info) {  
6     fputs(thd->proc_info, ...);  
7 }  
8 pthread_mutex_unlock(&proc_info_lock);  
9  
10 Thread 2::  
11 pthread_mutex_lock(&proc_info_lock);  
12 thd->proc_info = NULL;  
13 pthread_mutex_unlock(&proc_info_lock);
```



# Common Concurrency Problems

---

- Order-Violation Bugs

```
1 Thread 1::  
2 void init() {  
3     mThread = PR_CreateThread(mMain, ...);  
4 }  
5  
6 Thread 2::  
7 void mMain(...) {  
8     mState = mThread->State;  
9 }
```



# Common Concurrency Problems

- Order-Violation Bugs

```
1 Thread 1::  
2 void init() {  
3     mThread = PR_CreateThread(mMain, ...);  
4 }  
5  
6 Thread 2::  
7 void mMain(...) {  
8     mState = mThread->State;  
9 }
```

```
1 pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;  
2 pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;  
3 int mtInit = 0;  
4  
5 Thread 1::  
6 void init() {  
7     ...  
8     mThread = PR_CreateThread(mMain, ...);  
9  
10    // signal that the thread has been created...  
11    pthread_mutex_lock(&mtLock);  
12    mtInit = 1;  
13    pthread_cond_signal(&mtCond);  
14    pthread_mutex_unlock(&mtLock);  
15    ...  
16 }  
17  
18 Thread 2::  
19 void mMain(...) {  
20     ...  
21     // wait for the thread to be initialized...  
22     pthread_mutex_lock(&mtLock);  
23     while (mtInit == 0)  
24         pthread_cond_wait(&mtCond, &mtLock);  
25     pthread_mutex_unlock(&mtLock);  
26  
27     mState = mThread->State;  
28     ...  
29 }
```