

Recap: Synchronization

•What kind of knowledge and mechanisms do we need to get independent processes to communicate and get a consistent view of the world (computer state)?

Time	You	Your roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy milk
3:50		Arrive home, put up mlk
3:50		Oh no!

•Example: Too Much Milk

Recap: Synchronization Terminology

- **Synchronization:** use of atomic operations to ensure cooperation between threads
- **Mutual Exclusion:** ensure that only one thread does a particular activity at a time and *excludes* other threads from doing it at that time
- Critical Section: piece of code that only one thread can execute at a time
- Lock: mechanism to prevent another process from doing something
 - Lock before entering a critical section, or before accessing shared data.
 - Unlock when leaving a critical section or when access to shared data is complete
 - Wait if locked

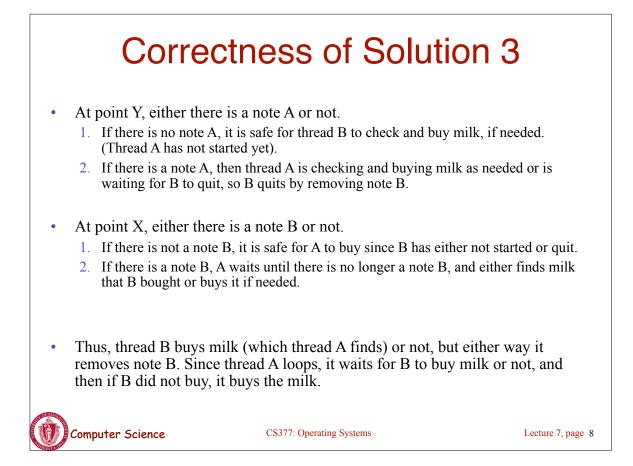
=> All synchronization involves waiting.



Тоо М	uch Milk: Soluti	on 1
Only one person buySomeone buys milk	if you need it. tomic loads and stores as building blo ion of lock) ion of unlock)	ocks.
Thread A	Thread B	
if (noMilk & NoNote) {	if (noMilk & NoNote) {	
leave Note;	leave Note;	
buy milk;	buy milk;	
remove note;	remove note;	
}	}	
Does this work?		
Computer Science	CS377: Operating Systems	Lecture 7, page 5

Too M	luch Milk: Solutior	า 2
How about using lab checking the the n	eled notes so we can leave a note be nilk?	efore
Thread A	Thread B	
leave note A	leave note B	
if (noNote B) {	if (noNote A) {	
if (noMilk){	if (noMilk){	
buy milk;	buy milk;	
}	}	
}	}	
remove note;	remove note;	
Does this work?		
Computer Science	CS377: Operating Systems	Lecture 7, page 6

Too Much Milk: Solution 3		
Thread A	Thread B	
<pre>leave note A X: while (Note B) { do nothing; } if (noMilk){ buy milk; } remove note A;</pre>	<pre>leave note B Y: if (noNote A) { if (noMilk){ buy milk; } } remove note B;</pre>	
Does this work?		
Computer Science	CS377: Operating Systems	Lecture 7, page 7



Is Solution 3 a good solution?

- It is too complicated it was hard to convince ourselves this solution works.
- It is asymmetrical thread A and B are different. Thus, adding more threads would require different code for each new thread and modifications to existing threads.
- A is *busy waiting* A is consuming CPU resources despite the fact that it is not doing any useful work.

CS377: Operating Systems

=> This solution relies on loads and stores being atomic.

Language Support for Synchronization

Have your programming language provide atomic routines for synchronization.

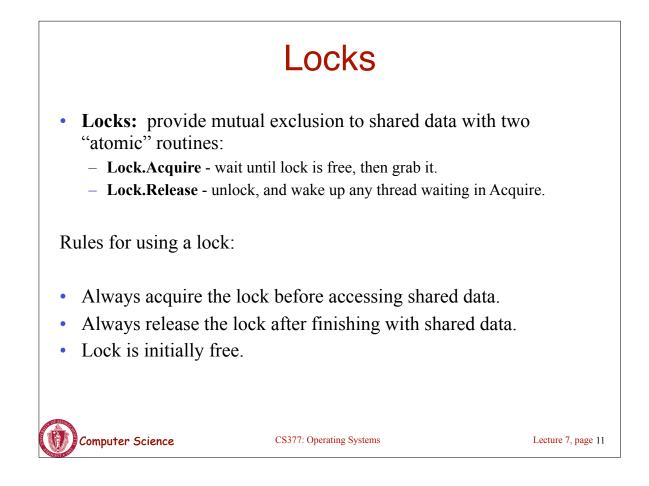
- Locks: one process holds a lock at a time, does its critical section releases lock.
- Semaphores: more general version of locks.
- Monitors: connects shared data to synchronization primitives.

=> All of these require some hardware support, and waiting.

Computer Science

Computer Science

Lecture 7, page 9



Implementing	g Too Much Milk v	with Locks
Too Much Milk		
Thread A	Thread B	
Lock.Acquire();	Lock.Acquire();	
if (noMilk){	if (noMilk){	
buy milk;	buy milk;	
}	}	
Lock.Release();	Lock.Release();	
• This solution is cle	an and symmetric.	
• How do we make I	Lock.Acquire and Lock.Release	e atomic?
Computer Science	CS377: Operating Systems	Lecture 7, page 12

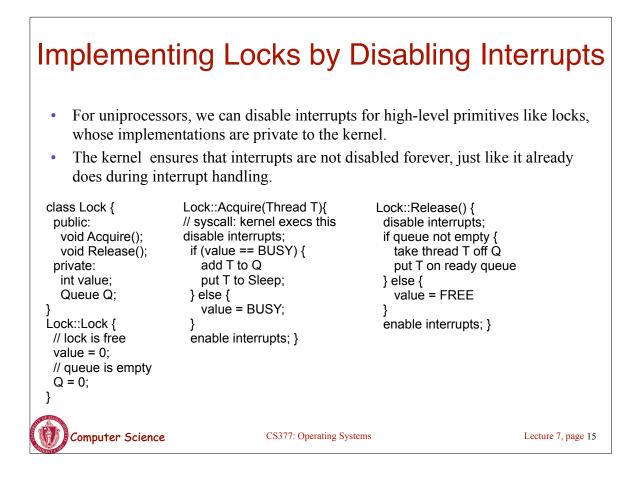
Hardware Support for Synchronization

Implementing high level primitives requires low-level hardware supportWhat we have and what we want

	Concurrent programs		
Low-level atomic operations (hardware)	load/store	interrupt disable	test&set
High-level atomic operations (software)	lock monitors	semaphore send & receive	
Computer Science	CS377: Operati	ng Systems	Lecture 7, page 1

Implementing Locks By Disabling Interrupts

- There are two ways the CPU scheduler gets control:
 - Internal Events: the thread does something to relinquish control (e.g., I/O).
 - **External Events:** interrupts (e.g., time slice) cause the scheduler to take control away from the running thread.
- On uniprocessors, we can prevent the scheduler from getting control as follows:
 - **Internal Events:** prevent these by not requesting any I/O operations during a critical section.
 - **External Events:** prevent these by disabling interrupts (i.e., tell the hardware to delay handling any external events until after the thread is finished with the critical section)



Atomic read-modify-write Instructions

- Atomic read-modify-write instructions *atomically* read a value from memory into a register and write a new value.
 - Straightforward to implement simply by adding a new instruction on a uniprocessor.
 - On a multiprocessor, the processor issuing the instruction must also be able to *invalidate* any copies of the value the other processes may have in their cache, i.e., the multiprocessor must support some type of *cache coherence*.

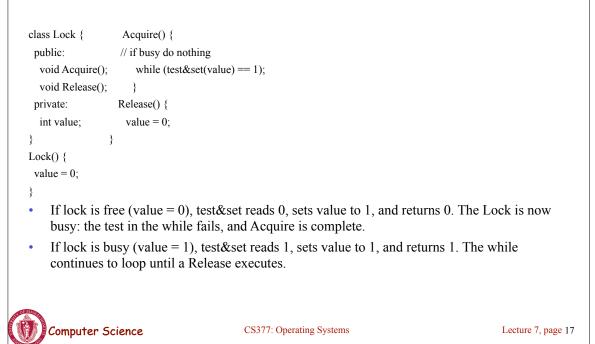
• Examples:

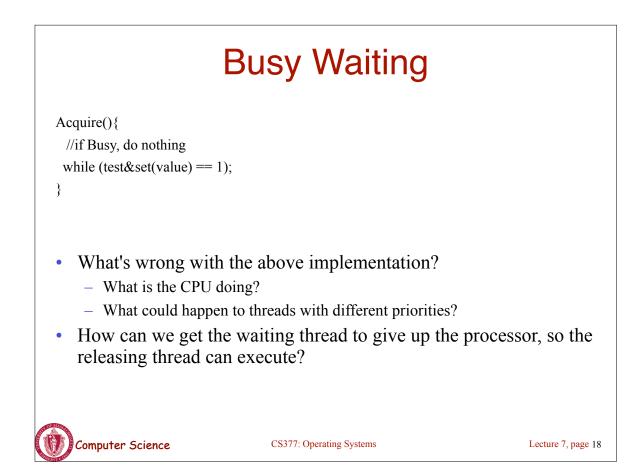
- Test&Set: (most architectures) read a value, write '1' back to memory.
- **Exchange:** (x86) swaps value between register and memory.
- **Compare&Swap:** (68000) read value, if value matches register value r1, exchange register r2 and value.

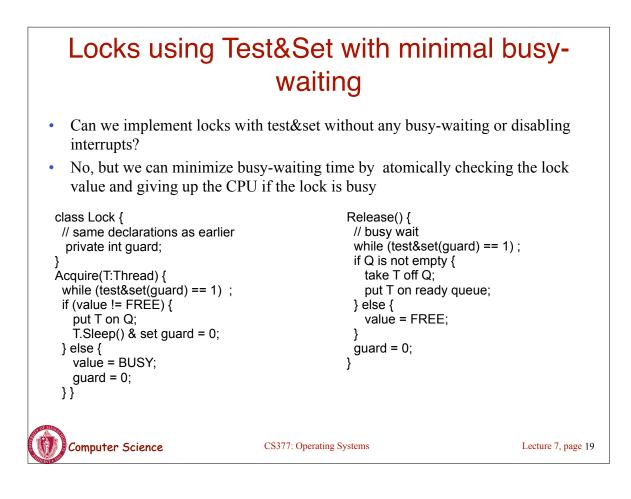
Computer Science

Implementing Locks with Test&Set

• Test&Set: reads a value, writes '1' to memory, and returns the old value.









- Communication among threads is typically done through shared variables.
- Critical sections identify pieces of code that cannot be executed in parallel by multiple threads, typically code that accesses and/or modifies the values of shared variables.
- Synchronization primitives are required to ensure that only one thread executes in a critical section at a time.
 - Achieving synchronization directly with loads and stores is tricky and errorprone
 - Solution: use high-level primitives such as locks, semaphores, monitors

Computer Science