Oh no!
Arrive home, put milk in fridge
Buy milk
Leave for grocery
Look in fridge, no milk
Arrive home

Your Roommate

Example: Too Much Milk
communicate and get a consistent view of the world (computer state)?

What kind of knowledge and mechanisms do we need to get independent processes to

Today: Synchonization

Review questions:
- Advantages? Disadvantages?
- How does each work?

- Lottery Scheduling
- Multilevel Feedback Queues
- SJF
- Round Robin
- FCFS

Scheduling Algorithms

Last Class: CPU Scheduling
Thread A

- Do not buy any milk if there is note (wait)
- Remove note (a version of unlock)
- Leave a note (a version of lock)

Restrict ourselves to atomic reads and stores as building blocks.

- Someone buys milk if you need it.
- Only one person buys milk at a time.

What are the correctness properties for this problem?

---

Too Much Milk: Solution 1

---

All synchronization involves waiting.

\[ \text{Lock} \]

- Wait if locked
- Unlock when leaving a critical section or when access to shared data is complete
- Lock before entering a critical section, or before accessing shared data

\[ \text{Critical Section} \]

- Piece of code that only one thread can execute at a time
- Excludes other threads from doing it at that time

\[ \text{Mutual Exclusion} \]

- Ensure that only one thread does a particular activity
- Between threads

\[ \text{Synchronization} \]

- Use of atomic operations to ensure cooperation

---

Synchronization Terminology
Does this work?

```
remove note A;
{
    {buy_with lk;}
    if (not lk)
        {do nothing;}
    x: if (not lk (Note B))
        leave note A
    Thread B

    Thread A
```

Too Much Milk: Solution 3

----

Does this work?

```
remove note A:
{
    {buy_with lk;}
    if (not lk)
        {do nothing;}
    x: if (notlk (Note B))
        leave note A
    Thread B

    Thread A
```

The milk?

How about using labeled notes so we can leave a note before checking the

Too Much Milk: Solution 2
This solution relies on loads and stores being atomic.

is not doing any useful work.

3. A is busy waiting - A is consuming CPU resources despite the fact that it

modifications to existing threads.

threads would require different code for each new thread and

2. It is symmetrical - thread A and B are different. Thus, adding more

works.

1. It is too complicated - it was hard to convince ourselves this solution

---

Is Solution 3 a good solution?

and then if B did not buy, it buys the milk.

removes note B. Since thread A loops, it waits for B to buy milk or not.

Thus, thread B buys milk (which thread A finds) or not, but either way it

that B bought or buys if it needed.

2. If there is a note B, A waits until there is no longer a note B, and either finds milk

1. If there is not a note B, it is safe for A to buy since B has either not started or quit.

At point X, either there is a note B or not.

for B to quit, so B quits by removing note B.

2. If there is a note A, then thread A is checking and buying milk as needed or is waiting

1. If there is no note A, it is safe for thread B to check and buy milk if needed. (Thread

At point Y, either there is a note A or not.

Correctness of Solution 3
• Lock is initially free.
• Always release the lock after finishing with shared data.
• Always acquire the lock before accessing shared data.

Rules for using a lock:

- lock::release: unlock and wake up any thread waiting in acquire.
- lock::acquire: wait until lock is free, then grab it.

Locks: provide mutual exclusion to shared data with two "atomic"

 Algorithms: connected shared data to synchronization primitives.

• Monitors: more general version of locks.

• Semaphores: one process holds a lock at a time, does its critical section

Have your programming language provide atomic routines for

Language Support for Synchronization
Concurrent Programs

What we have and what we want

Implementing high-level primitives requires low-level hardware support

Hardware Support for Synchronization

How do we make Lock::Acquire and Lock::Release atomic?

This solution is clean and symmetric:

```
lock->Acquire();
{
    // Busy wait
    if (monit()
        lock->Acquire();
}
```

Too Much Milk

Implementing Too Much Milk with locks
while not have the OS support Lock::Acquire() and Lock::Release as

section

delay handling any external events until after the thread is immersed with the critical

- External Events: Prevent these by disabling interrupts (i.e., tell the hardware to

- Internal Events: Prevent these by not requesting any I/O operations during a critical

- On multi-processor, we can prevent the scheduler from getting control as

away from the running thread.

- External Events: Interrupts (e.g., time slice) cause the scheduler to take control

- Internal Events: The thread does something to relinquish control (e.g., 0).

There are two ways the CPU scheduler gets control:

Implementing Locks by Disabling Interrupts
We have a problem with multithreading.

- Enable interrupts every time it selects a new process to run.
- If the next thread that executes to enable interrupts, thread dispatcher can switch thread to another thread, but it should not wake up the thread from sleep.
- When the thread wakes up, does it return to sleep?
- When should acquire re-enable interrupts when going to sleep?
- After putting the thread on the ready queue, but before going to sleep?
- Check the queue, and not wake up the thread.
- When acquiring re-enable interrupts when going to sleep?
- Lock and turn on interrupts.

Interrupts are still disabled. So it is ok to check the lock value, and if it is free, grab the lock.

When the sleeping thread wakes up, it returns from sleep back to acquire.
Summary

- Solution: use high-level primitives such as locks, semaphores, monitors
- Achieving synchronization directly with reads and stores is tricky and error-prone

- Execute in a critical section at a time.
- Synchronization primitives are required to ensure that only one thread
- Values of shared variables.
- By multiple threads, by explicitly code that accesses and/or modifies the
- Critical sections identify pieces of code that cannot be executed in parallel
- Communication among threads is typically done through shared variables.