# What is the problem?

# Bringing It All Together

- Processes
  - Abstraction for protection
  - Define address space
- Threads
  - Share (and communicate) through global and static data, share the heap, each has its own stack and full use of the registers
  - Race conditions may be a problem!
- CPU Schedulers
  - May pre-empt a process or thread at any time
- Ensuring correctness (OR eliminating race conditions and deadlock)
  - Safety and liveness
  - Atomic instructions
  - Synchronization: mutual exclusion, counted resources…
  - Locks, semaphores, monitors, transactions, conservative two-phase locking
  - The Six Commandments of multi-threaded programming
  - Common patterns: Bounded Buffer, Dining Philosophers, Readers/Writers

# So Far: One Big Lock

- Advantage: Simple
  - Relatively easy to get correct
  - This is a big advantage!
- Disadvantage: Performance
  - Eliminates advantages due to multi-threading for that part of your code
  - Eliminates advantages due to multicore in that part of your code

*There's another problem…*

# Multi-Object Synchronization

- Transfer $100 from account A to account B
  - A->subtract(100)
  - B->add(100)

Individual operations are atomic. Sequence is not.

- How should we ensure atomicity?

# Fine-Grained Locking

# Fine-Grained Locking

Wait!

What does it mean for a lock to be *fine-grained*?

What sort of locking have we been doing?


Picture from an [E-Bay listing by seller veles2365]

# Fine-Grained Locking

- Better for performance
  - This will matter more in the kernel than in an application (the kernel affects every application!)
- Complex
  - May need to acquire multiple locks to accomplish a task (a lock for each account?)
  - Incorrect code becomes more likely (Deadlock!)

# Conservative Two-Phase Locking

# Conservative Two-Phase Locking

- *A thread must*:
  - Acquire all locks it will need
    - If all locks cannot be acquired, release any already acquired and begin again
  - Make necessary changes, commit, and release locks
- Thus B cannot see any of A's changes until A commits and releases the lock
- Provides serializability
- Prevents deadlock

# Concurrency Quiz

If two threads execute this program concurrently, how many different final values of the global variable X are there?

Initially, X == 0.

Thread 1

```
void increment() {
    int tmp = X;
    tmp = tmp + 1;
    X = tmp;
}
```

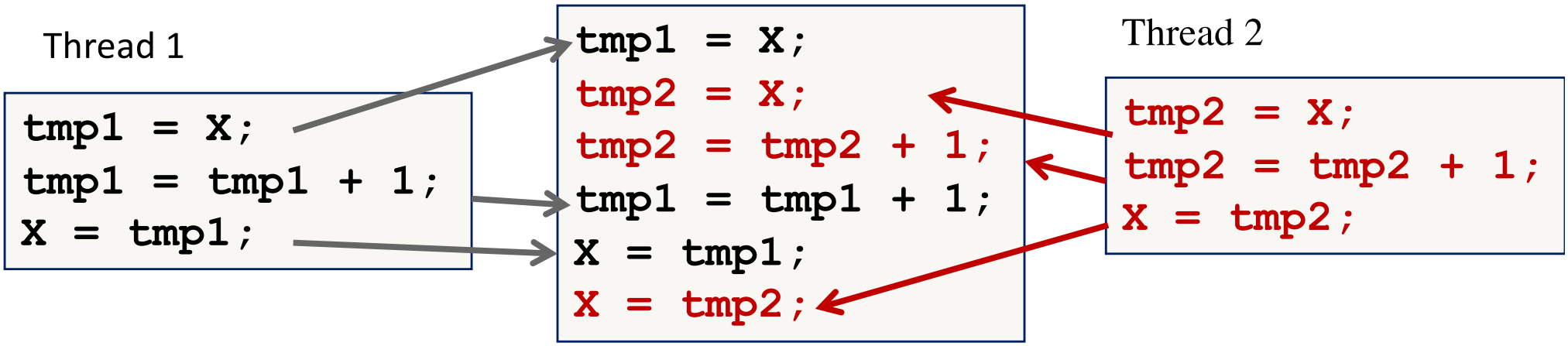
Thread 2

```
void increment() {
    int tmp = X;
    tmp = tmp + 1;
    X = tmp;
}
```

A. 0
B. 1
C. 2
D. More than 2

# Schedules/Interleavings

- Model of concurrent execution
- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, some synchronization is needed




The changes by Thread 1 and Thread 2 are not *serializable*.

# Schedules/Interleavings

Now let's add a (global) lock…

```
void increment() {
    lock.acquire();
    int tmp = X;
    tmp = tmp + 1;
    X = tmp;
    lock.release();
}
```

# Schedules/Interleavings

Thread 1

```
lock.acquire();
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;
lock.release();
```

```
lock.acquire()
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;
lock.release()
lock.acquire()
tmp2 = X;
tmp2 = tmp2 + 1;
X = tmp2;
lock.release()
```

Thread 2

```
lock.acquire();
tmp2 = X;
tmp2 = tmp2 + 1;
X = tmp2;
lock.release();
```

The changes by Thread 1 and Thread 2 are *serializable*.

# Conservative Two-Phase Locking

- *A thread must*:
  - Acquire all locks it will need
    - If all locks cannot be acquired, release any already acquired and begin again
  - Make necessary changes, commit, and release locks
- Thus B cannot see any of A's changes until A commits and releases the lock
- Provides serializability
- Prevents deadlock

# Transactions

# Transactions

- *Transactions* group actions together so that they are:
  - *atomic*: they all happen or they all don't
  - *serializable*: transactions appear to happen one after the other
  - *durable*: once it happens, it sticks
- Critical sections give us atomicity and serializability, but not durability

# Achieving Durability

Key idea: Ensure multiple updates are *all* written to disk. (Or none are.)

To get durability, we need to be able to:

- Write the changes to disk
- Certify that all the changes are successfully written
  - If they are not *all* written, then reverse the partial changes

# Achieving Durability, Formally

To get durability, we need to be able to:

- *Commit*: indicate when a transaction is finished (or written entirely to disk)
- *Rollback*: recover from an *aborted* transaction
  - If we have a failure in the middle of a transaction, we need to be able to undo what we have done so far
- In other words, we do a set of operations tentatively.
  - If we get to the commit stage, we are okay.
  - If not, roll back operations as if the transaction never happened.

# Achieving Durability, In Reality

Reversing changes on disk (*rolling back*) is hard

- How do we know what was there before?

So instead…

- Keep *write-ahead* log on disk of all changes in the transaction
- Once all changes are written to the log, the transaction is committed
  - Which means the word “Commit” is written at the end
  - If a crash happens before the commit, then the log can just be ignored
- W*rite-behind* changes to the disk
  - As in, later on a thread will actually put those changes in their proper spot in the filesystem
  - If a crash occurs during write-behind, the log will still contain the transaction, and the write-behind step can be restarted on restart

# The Write-Ahead Log

- All changes are written to the log (a special part of disk) prior to being written to their true location
- Changes must be *idempotent*

```
begin transaction
x = 300
y = 512
Commit
```

# iClicker Question

Imagine two threads executing this code, where x and y are global and begin as 0:

```
write begin transaction
lock
    x = x + 3
    y = y + 5
    write x, y to log
unlock
write Commit
```

Does this code work?

A. Yes
B. No

# In the transaction log…

Assuming all goes well and initial values of x and y are 0:

```
begin transaction
x=3
y=5
Commit
begin transaction
x=6
y=10
Commit
```

# In the transaction log…

But what if thread 1 crashes before the commit?

```
begin transaction
x=3
y=5
begin transaction
x=6
y=10
Commit
```

# How does the problem occur?

```
write begin transaction
lock
    x = x + 3
    y = y + 5
    write x,y to log
unlock
write Commit
```

Given two threads A & B that execute that code in the following sequence:

1. A gets the lock, reads an modifies x and y, writes to the log, and unlocks
2. The B grabs the lock before A commits
3. B reads A's modifications, then modifies x and y, writes to the log, unlocks, and commits
4. Then the system crashes before A commits