# Concurrency Review

15-440/640 Spring 2015

February 5

# What is Concurrency?

- Execution of more than one sequence of instructions at the same time
- If two instruction flows overlap in time, then they are concurrent

# Why do we want concurrency?

- Example server:

```
while(1) {
        sessfd = accept(sockfd,...);
        handle_client(sessfd);
}
```

- What happens if first client runs for hours?
- What happens to other clients?
- This is a sequential / iterative server

# Why do we want concurrency?

- Concurrent server:

```
while(1) {
        sessfd = accept(sockfd,...);
        if (fork()==0) {
                close(sockfd);
                handle_client(sessfd);
                exit();
        }
        close (sessfd);
}
```

Clone our process

New process (child) executes this

Original process (parent) continues here

- Now, each client handled by a separate process
- Each handler can block / wait without affecting other clients

# Working with processes

- `fork()` – create a new, identical copy of process
  - Except for process id, return value of fork (child gets 0, parent get's child's id)
  - All open files, sockets remain open in both!
- `exec()` – replace running program with a different one in this same process
  - Lose memory state from original program
  - Open files, sockets, and environment inherited
- `waitpid()` – wait for child process to exit and get status

# Communicating between processes

- `waitpid()`
- Signals:
  - E.g., `kill -SIGUSR1 pid`
- Pipes:
  - Using stdin/out, e.g., `cat foo | hexdump`
  - Named pipes – special "files" opened before fork, shared by child and parent
- Write to files
- Messaging, IPC, sockets, RPCs
- Shared memory

# Pros & Cons of Process-based Designs

+ Handle concurrent activities
+ Clean sharing model
    - Global variables: no
    - File tables: yes
    - Descriptors: no
+ Simple, straightforward
- Additional overhead of process control
- Not easy to share data

# Working with threads

- Pthreads interface standard for C
  - `pthread_create()` – start new thread
  - `pthread_join()` – wait for thread to end
  - `pthread_self()` – get thread id
  - `pthead_exit()`, `return` – exit thread
  - `exit()` – terminate all threads

- Many, many more functions defined!

# Threaded Server Example

- Concurrent server using pthreads:

```
while(1) {
        int *sessfd = (int*)malloc(sizeof(int));
        *sessfd = accept(sockfd,...);
        pthread_create(&tid, NULL,
              handle_client, sessfd);
}
```

Call this function in new thread

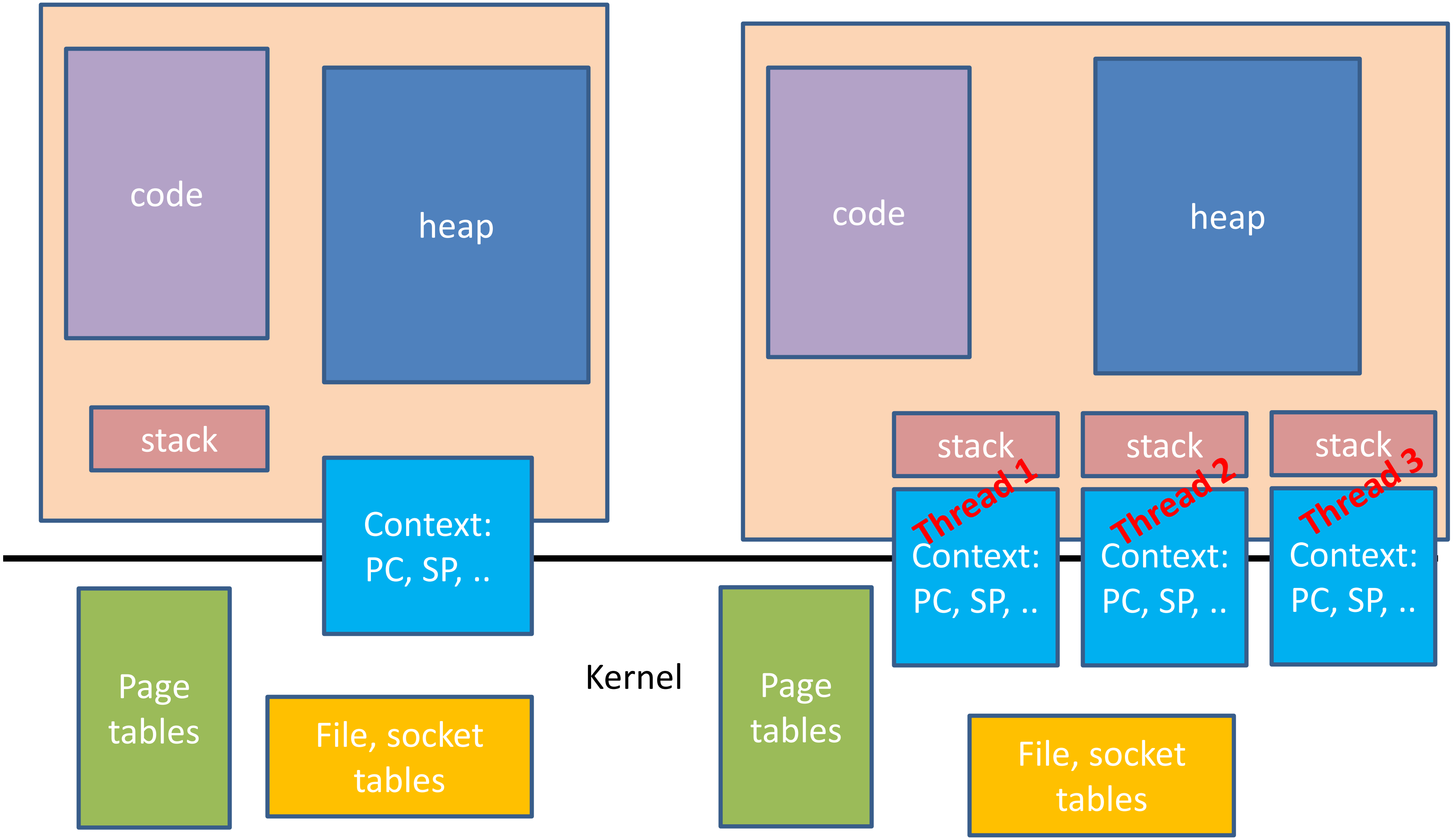
Parameter to function

- Very similar to processes, but a few key differences:
  - `handle_client` must return `void*`, take 1 `void*` parameter
  - Thread exits if `handle_client` returns; don't call `exit()`!
  - Don't close `sessfd` from main thread or `sockfd` from child

# Processes vs. threads

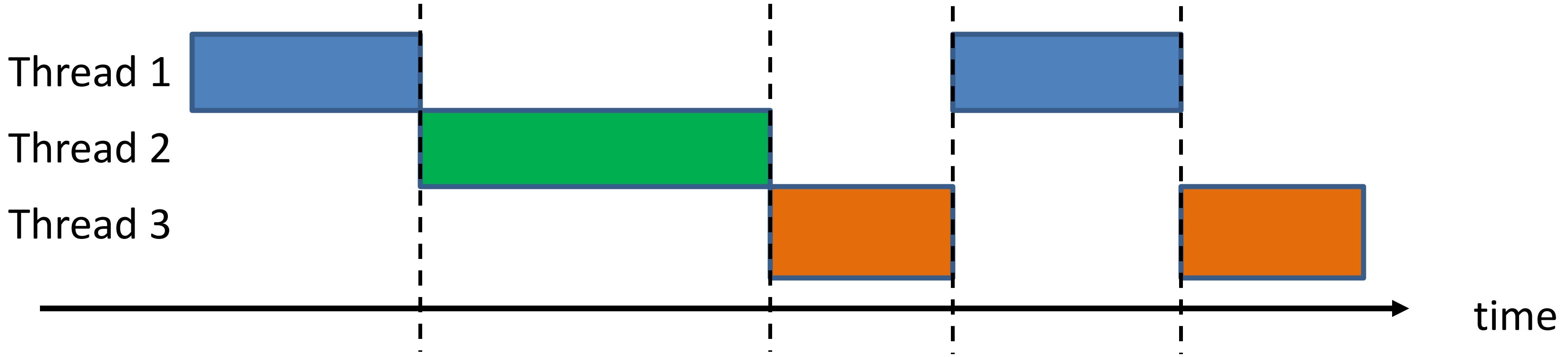
- Similarity:
  - Each has its own logical flow and context
  - Each can run concurrently on its own core
- Difference:
  - Threads share code and some data
    - Processes typically don't – more independent
  - Different set of control operations
    - Thread operations tend to be lower overhead

# Processes vs. Threads

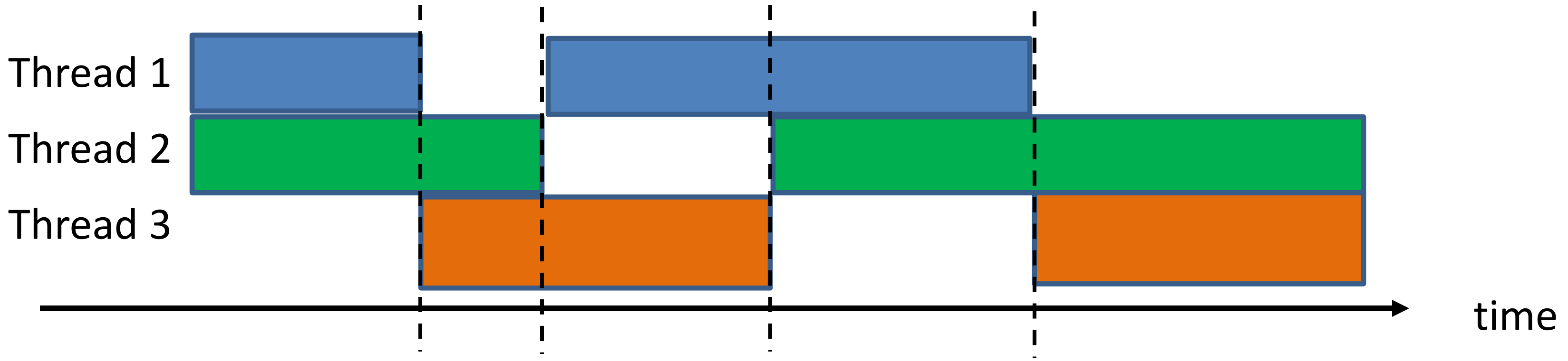


# Thread Execution

- Single core:  time slice to emulate parallel operation




- Multi-core: true parallel execution



# Pros & Cons of Thread-based Designs

\+ Easy to share data between thread

- Can pass by reference, access memory directly

\+ Threads are more efficient than processes

- Don’t need full context switch between threads
- Don’t need OS syscall for everything

\- Unintentional sharing can cause bugs

- Too easy to share – both the greatest strength and weakness of threads
- Hard to detect – nondeterministic bugs

\- Shared fate

- No isolation – one bad thread takes all of them down

# Alternative: I/O Multiplexing

- Single process handles all I/O
- Use select() call to wait for events on a set of sockets/fds
- Typically, an event handling loop:

```
while(1) {
    select(max_fd, &fd_set,…);
    for (i=0; i<max_fd; i++) {
        if (!FD_ISSET(i, &fd_set)) continue;
        if (i==sockfd) {
            sessfd=accept(sockfd,...);
            // add sessfd to fd_set, fix max_fd
        } else {
            // read more bytes from fd i
            // keep track of partial message state
            // track “state machine” for each client
            // watch for close events, etc.
            // generate replies
        }
    }
}
```

- Really hard to write!!!
- Very low overheads, total control

# Approaches to Concurrency

- Processes
  - Hard to share / communicate – avoids accidental sharing
  - Higher overheads
- Threads
  - (Too?) Easy to share / communicate
  - Medium overheads
  - Difficult to debug
  - Shared fate (exit() kills everything)
- Multiplexed I/O
  - Tedious, difficult to implement
  - Usually limited to I/O dominated tasks
  - Complete control over scheduling
  - Low overheads
  - Doesn’t use multiple cores

# Concurrency is great

BUT …

Doing it right is HARD!!!

# Managing Concurrency

- Coordinate sharing
- Coordinate execution timings

# Data sharing between threads

- Suppose you have lots of threads
- They need access to shared data

    Global shared data:

        pi = 3.14159

        e = 2.71828

        TheKing = “Elvis”

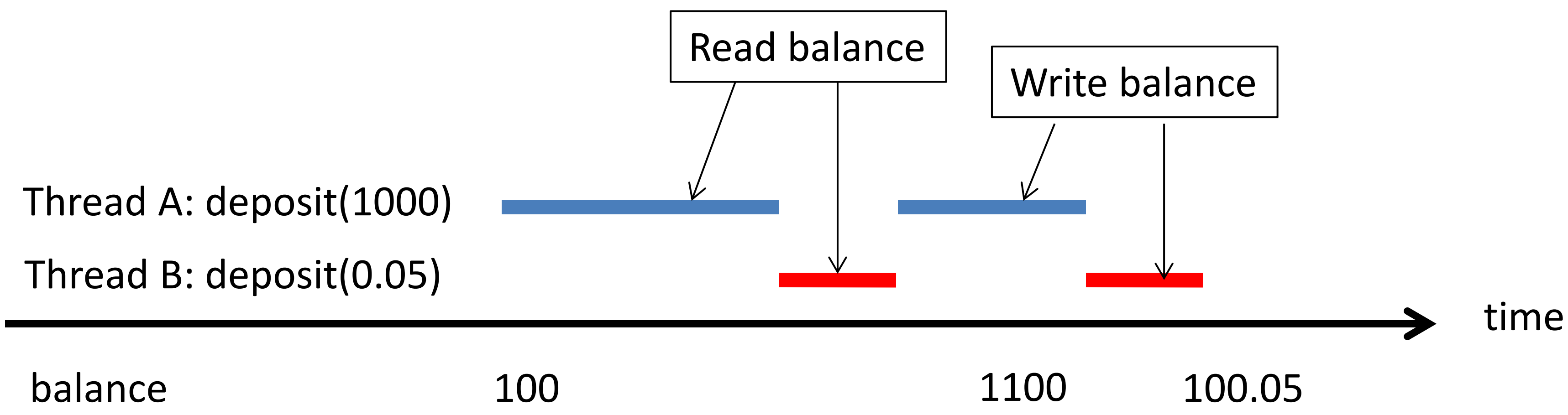
- If they only read data, is there a problem?
- Immutable objects can be shared freely without a problem

# What if data is modified?

- Example

```
float balance;
void deposit( float amount ) {
      balance = balance + amount;
}
```

- Bad things can happen



# Root causes of the problem

```
balance = balance + amount;
```

- This is actually multiple operations
    - Read – modify – write
- Even things that appear to be a single operation may not be: e.g., `i++`

- Murphy controls scheduling of threads!

# Sharing bugs are hard to diagnose

- Particularly nasty bugs
- Nondeterministic – may or may not happen for the same inputs

- “Heisenbugs” – disappear when you look for them using logging, printf, or gdb

- Proactively fix sharing issues in your designs to have a happier, healthier life!

# Need to control concurrency

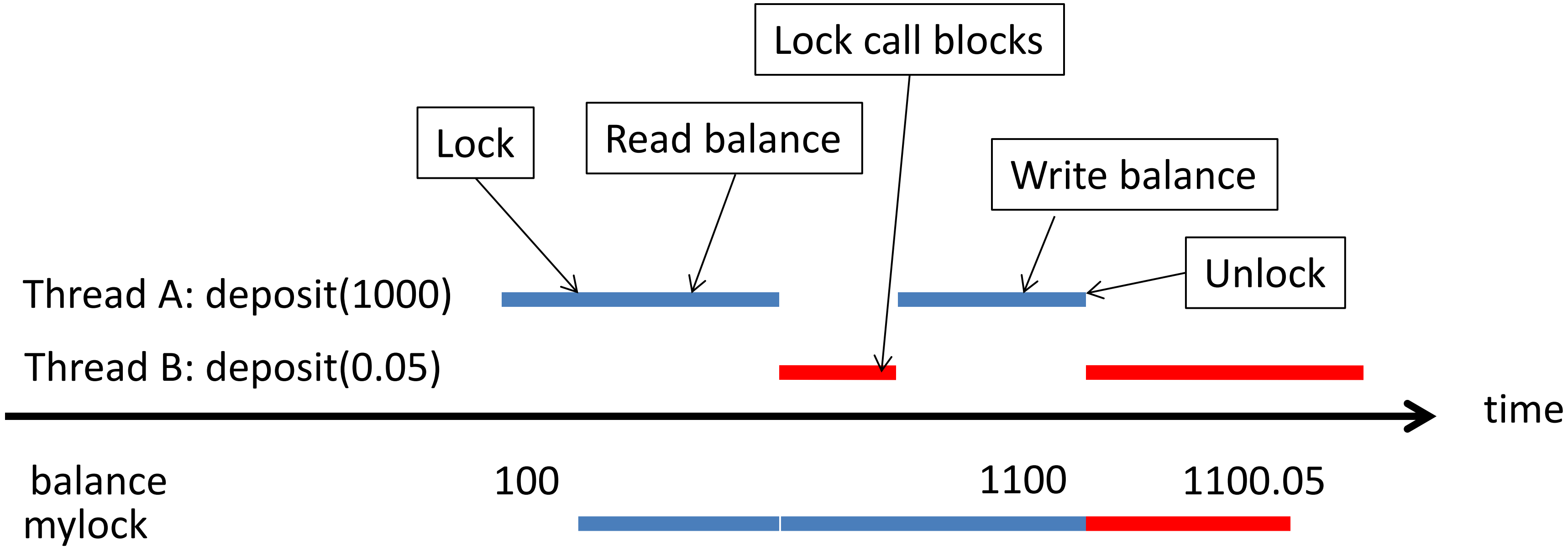
- Ensure Mutual Exclusion using a mutex
- Example:

```
pthread_mutex_t mylock;
float balance;
void deposit( float amount ) {
      pthread_mutex_lock(&mylock);
      balance = balance + amount;
      pthread_mutex_unlock(&mylock);
}
```

- lock operation blocks if mutex already held

# Now what happens

```
pthread_mutex_t mylock;
float balance;
void deposit( float amount ) {
        pthread_mutex_lock(&mylock);
        balance = balance + amount;
        pthread_mutex_unlock(&mylock);
}
```



# What locks do

- Locks help solve concurrent writes
- Locks don't actually "protect" shared data
    - Only work when everyone follows the protocol

- Locks actually prevent more than one thread from executing in a "critical section"

# Single Writer

- Locks help coordinate multiple writers
- What if there is just one writer?

```
float balanceA;
float balanceB;
void transferAtoB( float amount ) {
      balanceA = balanceA - amount;
      balanceB = balanceB + amount;
}
```

What if reader tries to sum balances here?

- Still a problem!

# Locks can fix this

- Fixed code:

```
float balanceA;
float balanceB;
pthread_mutex_t mylock;
void transferAtoB( float amount ) {
        pthread_mutex_lock(&mylock);
        balanceA = balanceA - amount;
        balanceB = balanceB + amount;
        pthread_mutex_unlock(&mylock);
}
```

- Readers need to lock as well!!
- Lock makes critical sections appear “atomic” relative to other critical sections
- Critical sections are serialized, not concurrent

# Lock granularity

- Suppose we use the previous code with 100s of threads updating thousands of accounts?

- Most threads may be blocked waiting for lock

- Lock granularity is too big

# One Big Global Lock

+ Easy to get correctness
- Can severely hurt performance
- May not be able to use multiple cores well

- Real-world example systems:
  - Linux kernel in the old days had big kernel lock
  - Python has Global Interpreter Lock
    - Can use many threads, but only one can actually be executing Python code at a time!!
    - Forced to use processes for concurrency

# Fine-grained locks

- Use multiple locks, tied to different data
- E.g., one lock per account
    - Threads need to hold lock A to read or write account A
    - CS touching A will be serialized only with other CS touching A
    - Access to B can happen concurrently
- Now, multiple threads less likely to block

# Transfer example revisited

- Code with fine grained locks:

```
struct account {
        float balance;
        pthread_mutex_t lock;
} acc[1000];

void transfer( int src, int dest, float amount ) {
        pthread_mutex_lock(&acc[src].lock);
        pthread_mutex_lock(&acc[dest].lock);
        acc[src].balance = acc[src].balance - amount;
        acc[dest].balance = acc[dest].balance + amount;
        pthread_mutex_unlock(&acc[dest].lock);
        pthread_mutex_unlock(&acc[src].lock);
}
```

- We need to hold multiple locks if the CS touches more than one account

# Still not quite right…

- What can happen if transfer from X to Y runs concurrently with transfer from Y to X?
- Murphy’s schedule of events:
    - Thread A grabs lock X
    - Thread B grabs lock Y
    - Thread A tries to get lock Y, blocks
    - Thread B tries to get lock X, blocks
- No further progress possible
- This is a “deadlock”

# Deadlocks

- Happens when there is a cyclic dependency between threads blocking on locks
- Can be due to arbitrarily long chains:
    - A waits for lock held by B, B waits for lock held by C, …. waits for lock held by A
- Can completely halt program, or parts of it
    - In the account example, thread A and B deadlock; any thread trying to touch X or Y will block indefinitely

# Avoiding deadlock

- Key goal: avoid forming a dependency cycle
- Option 1 – never hold more than one lock at a time
    - Can’t use this with fine grained account lock example
    - Big global lock doesn’t sound so bad now, does it?

# Avoiding deadlock

- Option 2 – order the locks in some way; acquire locks in order, release in reverse order
    - E.g., if a thread needs lock 7,3,and 6, first lock 3, then 6, then 7; when finished, unlock 7, then 6, then 3.
- Will this work? Why?
    - E.g., A holds lock 5 and blocks on lock 7
      B holds lock 7; it won't block on 5
      (if it wanted 5, it would have got it before it got 7)
    - Any dependency chain will have monotonically increasing lock ids
    - Cycles are not possible!!

# Avoiding Deadlock example

- Improved code with fine grain locks:

```
struct account {
        float balance;
        pthread_mutex_t lock;
} acc[1000];

void transfer( int src, int dest, float amount ) {
        if (src>dest) {
                int tmp=dest; dest=src; src=tmp; amount=-amount;
        }
        pthread_mutex_lock(&acc[src].lock);
        pthread_mutex_lock(&acc[dest].lock);
        acc[src].balance = acc[src].balance - amount;
        acc[dest].balance = acc[dest].balance + amount;
        pthread_mutex_unlock(&acc[dest].lock);
        pthread_mutex_unlock(&acc[src].lock);
}
```

- Are we done yet?

# Self-deadlock

- What if src==dest?
  - Thread will block forever on second call to lock!
  - Cyclic lock dependency with itself
- Quick fix: need to check this case
- Not always easy to avoid – e.g., library call, recursive functions, may end up locking again
- Reentrant lock – special mutex that permits relocking to succeed if lock is already held by this thread

# Locks summary

- Mutexes provide <u>mut</u>ual <u>ex</u>clusion
- Used as locks to protect critical sections of code that use or modify shared data
- Make critical sections appear atomic relative to each other
- Single big locks are easy, but limit performance, concurrency
- Fine-grained locks allow good concurrency, but are tricky
    - Deadlocks possible when more than 1 lock is held