Synchronization and Liveness Hazards

Shuhao Zhang

Nanyang Technological University

shuhao.zhang@ntu.edu.sg

June 3, 2024

Outline

- Last week: visibility issues
  - Cache coherence
    - Ensures that each processor has consistent view of memory through its local cache
  - Memory consistency
    - Order of memory accesses → opportunity for reducing program execution time
- This Week:
  - race conditions
  - execution ordering
  - deadlocks

Introduction

- Threads cooperate in multithreaded programs
  - Share resources, access shared data structures
  - Coordinate their execution: One thread executes relative to another
- For correctness, we have to **control this cooperation**
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control by default
- Use synchronization
  - Restrict the possible interleaving of thread executions

Shared Resources

- Coordinating access to shared resources
  - Basic problem:
    - If two concurrent threads (processes) are accessing a shared variable, and that variable is read/ modified/ written by those threads, then access to the variable must be controlled to avoid erroneous behavior
  - Mechanisms to control access to shared resources
    - Volatile, Locks, mutexes, semaphores, monitors, condition variables, etc.
    - Still remember ‘volatile’? :)
  - Patterns for coordinating accesses to shared resources
    - Bounded buffer, producer-consumer, etc.

Motivation Example

Implement a function to handle withdrawals from a bank account:

```
public double withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance − amount;
    put_balance(account, balance);
    return balance;
}
```

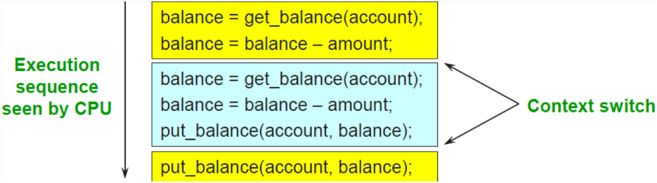
- 2 people share a bank account with a balance of $1000
- Simultaneously withdraw $100 from the account

Motivation Example (cont'd)

- Create a thread for each person to do the withdrawals
- These threads run on the same bank server
- What are the possible problems?

Motivation Example - Issues

Execution of the two threads can be interleaved



Race Condition

- Two concurrent threads (or processes) accessed a shared resource (account) without any synchronization
  - Known as a race condition
- Control access to these shared resources
- Necessary to synchronize access to any shared data structure
  - Buffers, queues, lists, hash tables, etc.

Mutual Exclusion

- Use mutual exclusion to synchronize access to shared resources
  - This allows us to have large atomic blocks
- Code sequence that uses mutual exclusion is called critical section
  - Only one thread at a time can execute in the critical section
  - All other threads have to wait on entry
  - When a thread leaves a critical section, another can enter

Critical Section Requirements

- Mutual exclusion (mutex)
  - If one thread is in the critical section, the no other is
- Progress
  - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
  - A thread in the critical section will eventually leave it (remember to unlock)
- Bounded waiting (no starvation)
  - If some thread T is waiting on the critical section, then T will eventually enter the critical section
- Performance
  - The overhead of entering and existing the critical section is small with respect to the work being done within it

Critical Section Requirements - Details

- Requirements:
  - Safety property: nothing bad happens
    - Mutex
  - Liveness property: something good (eventually) happens
    - Progress, Bounded Waiting
  - Performance requirement
- Properties hold for each run, while performance depends on all the runs
- Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!)

Synchronization Mechanisms Overview

- Atomic Variables
  - Intuitive solution, always consider it first
- Locks
  - Primitive, minimal semantics, used to build others
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Barriers, Latches, and More
  - High-level, requires language support, operations implicit

Atomic Variables

- Most modern programming languages provide **AtomicXXX** data types/constructs.
- For example,
  - *AtomicInteger x = new AtomicInteger(0)*
  - *x.incrementAndGet() //increments x by 1 atomically*

Example of Using Atomic Variables

```
class A extends Thread {
    Random r = new Random();
    public void run() {
        try {
            Thread.sleep(r.nextInt(10));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        FirstError.count++;
    }
}
```

```
public class FirstError {
    public static int count = 0;
    public static void main(String args[]) {
        int numberofThreads = 10000;
        A[] threads = new A[numberofThreads];
        for (int i = 0; i < numberofThreads; i++) {
            threads[i] = new A();
            threads[i].start();
        }
        try {
            for (int i = 0; i < numberofThreads; i++) {
                threads[i].join();
            }
        } catch (InterruptedException e) {
            System.out.println("some thread is not finished");
        }
            System.out.println(count);
    }
}
```

The results are somehow randomized due to the race condition. ☹

Notes

See FirstError.java

Example of Using Atomic Variables (cont'd)

```java
public static int count = 0;
```

```java
public static AtomicInteger count = new
AtomicInteger(0);
```

Notes

See FirstErrorFixed.java

Compound Actions

When atomic variables are helpless...

Recall: Example

Implement a function to handle withdrawals from a bank account:

```
public double withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance − amount;
    put_balance(account, balance);
    return balance;
}
```

- 2 people share a bank account with a balance of $1000
- Simultaneously withdraw $100 from the account

Recall: Example

Amount is a shared variable and concurrent access to it can lead to race condition. So,

```java
public static AtomicInteger amount = new AtomicInteger(5000);
```

- It still leads to random results due to race condition.
- Checkout the "SecondError.java" for reference.

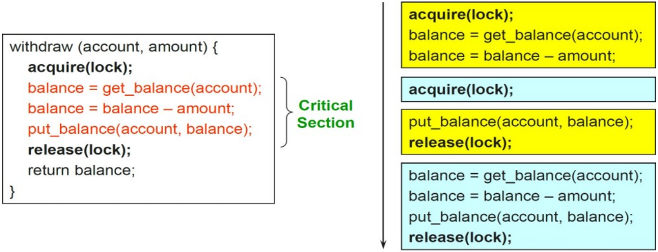
⚠

What to rescue? Locks!

Locks

- Two operations:
  - acquire(): to enter a critical section
  - release(): to leave a critical section
- Pair calls to acquire and release
  - Between acquire/release, the thread holds the lock
  - Acquire does not return until any previous holder releases
  - What can happen if the calls are not paired?
- Locks can spin (a spinlock) or block (a mutex)

Using Locks



- What happens when blue tries to acquire the lock?
- Why is the “return” outside the critical section? Is this ok?
- What happens when a third thread calls acquire?

Using Locks in Java: intrinsic lock

- Every Java object can implicitly act as a lock for purposes of synchronization.

```
synchronized (lock) {
  //Access shared state guarded by lock
}
```

- **Intrinsic locks** act as mutexes (mutual exclusion locks), i.e., at most one thread may own the lock.
- Since only one thread at a time can execute a block of code guarded by a given lock, the synchronized blocks guarded by the same lock execute atomically with respect to one another.

Notes

We will see how to fix the second error with intrinsic lock in tutorial.

Implementing Locks

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock→held);
    lock→held = 1;
}
void release (lock) {
    lock→held = 0;
}
```

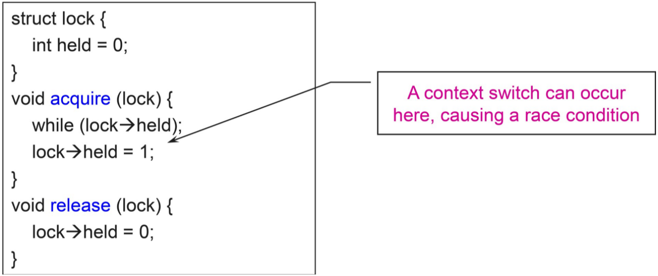
busy-wait (spin-wait) for lock to be released

Figure: First attempt to implement the lock

- This is called a spinlock because a thread spins waiting for the lock to be released
- Does this work?

Implementing Locks (cont'd)

- No. Two independent threads may both notice that a lock has been released at the same time and thereby acquire it.



Implementing Locks (cont'd)

- The problem: implementation of locks requires critical sections, too
- And, we can use "locks" to guarantee critical section... And, those "locks" again requiring critical sections...
  - How do we stop the recursion?
- The key idea: the implementation of acquire/release action must be atomic by itself
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes "all or nothing"
- Need help from hardware
  - Atomic instructions (e.g., test-and-set)
  - Disable/enable interrupts (prevents context switches)

Atomic Instructions: Test-and-set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to indicate available
  - Return the old value
- Hardware executes it atomically!

```
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

Implementing Locks with Test-and-Set

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock→held));
}
void release (lock) {
    lock→held = 0;
}
```

Problems with Spinlocks

- Spinlocks are wasteful
  - If a thread is spinning on a lock, then the thread holding the lock cannot make progress (on a uniprocessor)
  - **When using "synchronized", the thread which wants to obtain a lock will have to keep waiting if the lock is being held by other thread.**
- How did the lock holder give up the CPU in the first place?
  - Lock holder calls yield or sleep: wait() and notify() in Java.
  - Involuntary context switch

Monitor: Wait and Notify/Signal

- A **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false
- **Wait:** calling wait() forces the current thread to wait until some other thread invokes notify() or notifyAll() on the same *object's monitor*
- **Notify/signal:** use the notify() method for waking up threads that are waiting for an access to *this object's monitor*

Remark

Synchronizers are implemented based on monitors in various ways. Monitor is explicitly defined as conditional variable in C++, while is implicitly defined in Java (i.e., every object has a monitor).

Synchronizers

- A synchronizer is an object that coordinates the control flow of threads based on its state.
- We will study four important synchronizers in the following.
  - Semaphore
  - CyclicBarrier
  - CountDownLatch
  - Phaser
- They are all available in JDK, and some are available in C++ standards. You may also find the corresponding synchronizers as a third-party lib for other programming language, e.g., Python.

Why Synchronizers?

- Compare to Locks, Synchronizers are more flexible and powerful.
- Synchronizers are widely used for parallel program correctness and performance testing.
- They can be also used to simplify the development of parallel program, but use with care.

Powerful Synchronizers

- A synchronizer is an object that coordinates the control flow of threads based on its state
  - **Semaphores**
  - **CyclicBarrier**
  - **CountDownLatch**
  - **Phaser**

Attention 1

They are more flexible and powerful than intrinsic locks and ReentrantLock but use them with care, e.g., don't get into the trouble of liveness hazards.

Attention 2

All provided as standard API in Java. Only Semaphore is provided as standard API in C++/Python but nothing stop you from implementing the rest by your own!

Recall Monitor: Wait and Notify/Signal

- A **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false
- **Wait:** calling wait() forces the current thread to wait until some other thread invokes notify() or notifyAll() on the same *object's monitor*
- **Notify/signal:** use the notify() method for waking up threads that are waiting for an access to *this object's monitor*

Remark

Synchronizers are implemented based on monitors in various ways. Monitor is explicitly defined as conditional variable in C++, while is implicitly defined in Java (i.e., every object has a monitor).

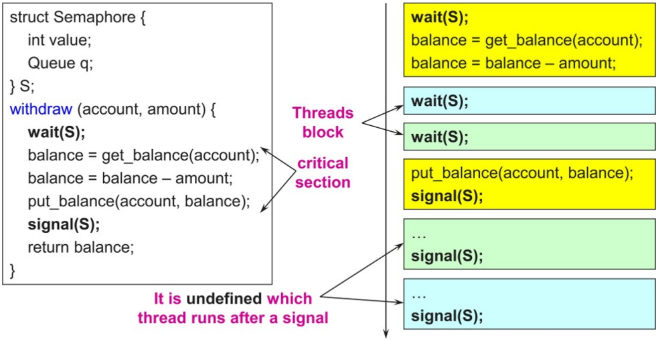
Semaphores

- Semaphores are an abstract data type that provide mutual exclusion through atomic counters
  - Described by Dijkstra in the “THE” system in 1968
- Semaphores are “integers” that support two operations:
  - Semaphore::Wait(): decrement, block until semaphore is open
  - Semaphore::Signal: increment, allow another thread to enter
  - Semaphore safety property: the semaphore value is always greater than or equal to 0

Semaphore Types

- Mutex semaphore (or binary semaphore)
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore (or general semaphore)
  - Multiple threads can pass the semaphore
  - Number of threads determined by the semaphore "count"
    - mutex has count = 1, counting has count = N

Example


struct Semaphore {
int value;
Queue q;
} S;
withdraw (account, amount) {
wait(S);
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
signal(S);
return balance;
}
Threads block
critical section
It is undefined which thread runs after a signal
wait(S);
balance = get_balance(account);
balance = balance – amount;
wait(S);
wait(S);
put_balance(account, balance);
signal(S);
...
signal(S);
...
signal(S);

How to use Semaphores

Take a look at SemaphoreExample.java, learn how it works.

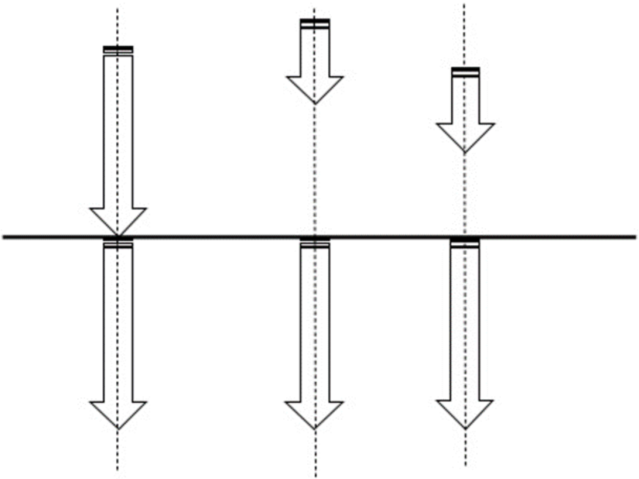
```
private final Semaphore roomOrganizer = new Semaphore(3, true); //create

try {
   roomOrganizer.acquire(); //lock
} catch (InterruptedException e) {
   System.out.println("received InterruptedException");
   return;
}
System.out.println("Thread " + this.getId() + " starts to use the room");
try {
   sleep(1000);
} catch (InterruptedException e) {
}
System.out.println("Thread " + this.getId() + " leaves the room\n");
roomOrganizer.release(); //unlock
```

Semaphores Summary

- Semaphores can be used as a mutex
- However, they have some drawbacks
  - They are essentially shared global variables
    - Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
- Sometimes hard to use and prone to bugs

Barrier



Cyclic Barriers

- Allows a set of threads to all wait for each other to reach a **common barrier point**.
- The barrier is often called **cyclic** because it can be re-used after the waiting threads are released.

Cyclic Barriers

- Two powerful features of CB:
  - A CyclicBarrier supports an **optional runnable command** that is run once per barrier point, after the last thread arrives, but before any threads are released.
  - CyclicBarrier is **auto-reset** and can be immediately reused once the last thread arrives.

Remark

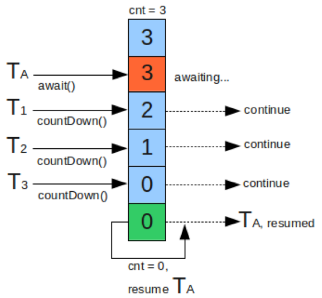
Since C++20 (December 2020), c++ supports barrier officially. `https://en.cppreference.com/w/cpp/thread/barrier`. Find out the difference compared to cyclic barrier in Java.

How to use CyclicBarrier

Take a look at CyclicBarrierExample.java, learn how it works.

CountDownLatch

- A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
  - Key difference: CDL can be count down by the same thread more than one time!
  - e.g., one thread has multiple operations to be synchronized with others.



How to use CountDownLatch

Take a look at CountDownLatchExample.java, learn how it works.

Phaser

Phaser (introduced in Java 7)

- A reusable synchronization barrier, similar in functionality to CyclicBarrier and CountDownLatch **but supporting even more flexible usage**.
  - register(), bulkRegister(int parties) to control the number participants at runtime.
  - while cyclicbarrier and countdownlatch have fixed participants

How to use phaser

Take a look at PhaserExample.java, learn how it works.

Barrier vs Latch vs Phaser

- CountDownLatch:
  - Created with a fixed number of threads
  - Cannot be reset
  - Allow threads to wait (method await) or continue with its execution (method countdown())
- Cyclic Barrier:
  - Can be reset.
  - Does not provide a method for the threads to advance. The threads have to wait till all the threads arrive.
  - Created with fixed number of threads.
- Phaser:
  - Number of threads need not be known at Phaser creation time. They can be added dynamically.
  - Can be reset and hence is, reusable.
  - Allows threads to wait (method arriveAndAwaitAdvance()) or continue with its execution(method arrive()).
  - Supports multiple Phases

Classic Synchronization Problems

- Producer-consumer
  - Infinite buffer
  - Finite buffer
- Readers-writers
- Dining philosophers

Note

There are many interesting classic synchronization problems in the textbook.

Producer-consumer

- Producers create items of some kind and add them to a data structure
- Consumers remove the items and process them
- Variables:
  - mutex = Semaphore (1)
  - items = Semaphore (0)

Producer-consumer

Producer

- event = waitForEvent ()
- mutex.wait()
  - buffer.add (event)
  - items.signal ()
- mutex.signal ()

Consumer

- items.wait ()
- mutex.wait ()
  - event = buffer.get ()
- mutex.signal ()
- event.process ()

⚠

Can you draw a "sequence diagram" to illustrate their interactions, i.e., "producer", "mutex", "item", and "consumer".

Improved Producer-consumer

Producer

- event = waitForEvent ()
- mutex.wait()
    - buffer.add (event)
- mutex.signal ()
- *items.signal ()*

Consumer

- items.wait ()
- mutex.wait ()
    - event = buffer.get ()
- mutex.signal ()
- event.process ()

⚠

Can you draw a "sequence diagram" to illustrate why this becomes a better version?

Broken Producer-consumer

Producer

- event = waitForEvent ()
- mutex.wait()
  - buffer.add (event)
- mutex.signal ()
- items.signal ()

Consumer

- mutex.wait ()
  - items.wait ()
  - event = buffer.get ()
- mutex.signal ()
- event.process ()

⚠

Can you draw a “sequence diagram” to illustrate why this won’t work?

Producer-consumer with Finite Buffer

Producer

- event = waitForEvent ()
- **spaces.wait()**
- mutex.wait ()
  - buffer.add (event)
- mutex.signal ()
- items.signal ()

Consumer

- items.wait ()
- mutex.wait ()
  - event = buffer.get ()
- mutex.signal ()
- **spaces.signal ()**
- event.process ()

⚠

You may notice that, the buffer may become infinite large. Can you draw a "sequence diagram" to illustrate their interactions, i.e., "producer", "mutex", "spaces", and "consumer" of this version?

Readers-writers problem

Requirements:

- Any number of readers can be in the critical section simultaneously
- Writers must have exclusive access to the critical section
- Variables:
    - int readers = 0
    - mutex = Semaphore (1)
    - roomEmpty = Semaphore (1)

Readers-writers

Writers

- roomEmpty.wait ()
    - #critical section for writers
- roomEmpty.signal ()

Readers

- mutex.wait ()
    - readers += 1
    - if readers == 1:
        - roomEmpty.wait () // first in locks
- mutex.signal ()
- # critical section for readers
- mutex.wait ()
    - readers -= 1
    - if readers == 0:
        - roomEmpty.signal () // last out unlocks
- mutex.signal ()

Lightswitch Definition

class Lightswitch :

```
def__init__( self ):
    self.counter = 0
    self.mutex = Semaphore (1)
def lock (self , semaphore ):
    self.mutex.wait ()
        self.counter += 1
        if self.counter == 1:
            semaphore.wait ()
        self.mutex.signal ()
def unlock (self , semaphore ):
    self.mutex.wait ()
        self.counter -= 1
        if self.counter == 0:
            semaphore.signal ()
    self.mutex.signal ()
```

Readers-writers with Lightswitch

Writers

- roomEmpty.wait ()
  - #critical section for writers
- roomEmpty.signal ()

Readers

- readLightSwitch.lock (roomEmpty)
  - # critical section
- readLightSwitch.unlock (roomEmpty)

Note

starving writers...
Use a turnstile = Semaphore (1)

No-starve Readers-writers

Writers

- turnstile.wait ()
  - roomEmpty.wait ()
    - # critical section for writers
- turnstile.signal ()
- roomEmpty.signal ()

Readers

- turnstile.wait ()
- turnstile.signal ()
- readLightSwitch.lock (roomEmpty)
  - # critical section
- readLightSwitch.unlock (roomEmpty)

⚠

Can you draw a “sequence diagram” to illustrate their interactions, i.e., “writer 1”, “writer 2”, “reader1”, “reader2”, “turnstile”, “readLightSwitch”, and “roomEmptyMutex” of this version?

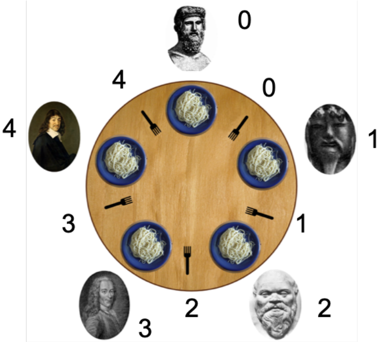
Definition: Deadlock

- Deadlock definition
  - Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set
- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized

Condition for Deadlock

- Deadlock can exist if and only if the following four conditions hold simultaneously:
  - Mutual exclusion – At least one resource must be held in a non-sharable mode
  - Hold and wait – There must be one process holding one resource and waiting for another resource
  - No preemption – Resources cannot be preempted (critical sections cannot be aborted externally)
  - Circular wait – There must exist a set of processes [P1, P2, P3,. . . ,Pn] such that P1 is waiting for P2, P2 for P3, etc.

Classical Deadlock Problem: Dining Philosophers Problem



- Each philosopher needs two forks to eat.
- Each philosopher picks the one on the left first.

Classical Deadlock Problem: Dining Philosophers Problem

Recall: Deadlock is the situation when two or more threads are both waiting for the others to complete, forever.

Notes

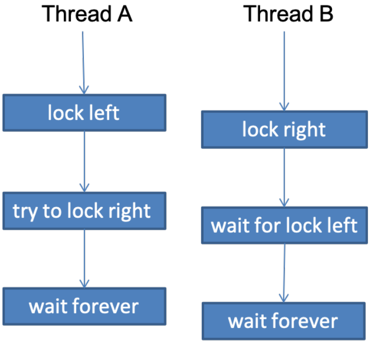
Take DiningPhilDemo.java as a reference

Why a deadlock may occur?

```
public class LeftRightDeadlock {
   private final Object left = new Object ();
   private final Object right = new Object ();

   public void leftRight () {
      synchronized (left) {
         synchronized (right) {
            //doSomething();
         }
      }
   }

   public void rightLeft () {
      synchronized (right) {
         synchronized (left) {
            //doSomethingElse();
         }
      }
   }
}
```



Dealing with Deadlock

There are four approaches for dealing with deadlock:

- Ignore it – how lucky do you feel?
- Prevention – make it impossible for deadlock to happen
- Avoidance – control allocation of resources
- Detection and Recovery – look for a cycle in dependencies

Remedy for Deadlock: Single Lock

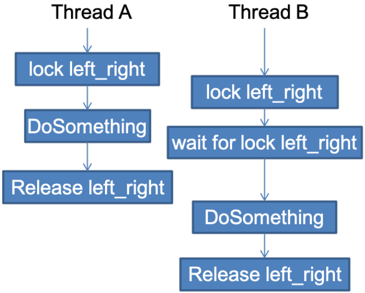
- A program that never acquires more than one lock at a time will not experience lock-ordering deadlocks.
- A simple strategy is to combine two locks into one, i.e., must acquire two locks at the same time.

Remedy Deadlock: Single Lock (Example)

```
public class LeftRightDeadlock {
    private final Object left_right = new
Object ();

    public void leftRight () {
        synchronized (left_right ) {
            doSomething();
        }
    }

    public void rightLeft () {
        synchronized (left_right ) {
            doSomethingElse();
        }
    }
}
```



Remark

Deadlock is prevented, but..

Remedy for Deadlock: global sequence

A program will be free of lock-ordering deadlocks **if all threads acquire the locks they need in a fixed global order.**

- Is this deadlocking? Thread A locks *a*, *b*, *c*, *d*, *e* in the sequence and thread B locks *c*, *f*, *e*.
- Is this deadlocking? Thread A locks *a*, *b*, *c*, *d*, *e* in the sequence and thread B locks *e*, *f*, *c*.

Remedy for Deadlock: global sequence (Example)

```
public void transferMoney(Account from, Account to, int amount) {
    synchronized (from) {
        synchronized (to) {
            if (from.getBalance() < amount) {
                System.out.println("Insufficient Fund");
            } else {
                from.debit(amount);
                to.credit(amount);
            }
        }
    }
}
```

- When can transferMoney deadlock?
  - Thread A: transferMoney(myAccount, yourAccount, 1)
  - Thread B: transferMoney(yourAccount, myAccount, 1)

Notes

Take ‘TransferExample.java‘ as reference

Remedy for Deadlock: global sequence (Fixed example)

```
public void transferMoney(Account fromAccount, Account toAccount, int amount) throws Exception {
    int fromHash = System.identityHashCode(fromAccount);
    int toHash = System.identityHashCode(toAccount);

    if (fromHash < toHash) {
        synchronized (fromAccount) {
            synchronized (toAccount) {
                transfer(fromAccount, toAccount, amount);
            }
        }
    } else if (fromHash > toHash) {
        synchronized (toAccount) {
            synchronized (fromAccount) {
                transfer(fromAccount, toAccount, amount);
            }
        }
    } else {
        synchronized (tieLock) {
            synchronized (fromAccount) {
                synchronized (toAccount) {
                    transfer(fromAccount, toAccount, amount);
                }
            }
        }
    }
}
```

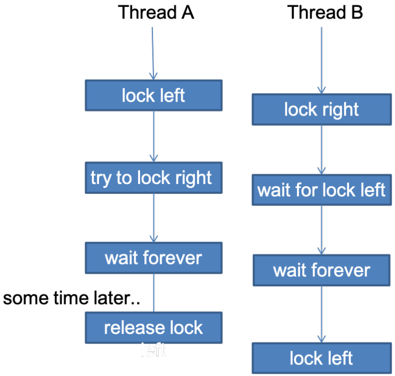
always lock the account with smaller hash value first.

Notes

Take ‘TransferExampleFixed.java‘ as reference

Remedy for Deadlock: Explicitly Break Deadlocks

When deadlock happens, we can break it by releasing the locks.



Remedy for Deadlock: Explicitly Break Deadlocks (Example)

- Use the timed tryLock feature of the explicit Lock class instead of intrinsic locking.
  - *ReentrantLock* in Java API
  - What's the counterpart in C/C++/Python?

```
final ReentrantLock reentrantLock = new ReentrantLock();
boolean flag = reentrantLock.tryLock(1000,
TimeUnit.MILLISECONDS);
```

Notes

Take 'ReentrantLockExample.java' as reference

Other Liveness Hazards

- **Starvation** is a situation where a process is prevented from making progress because some other process has the resource it requires
- Starvation is a side effect of the scheduling algorithm
  - OS: A high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock

Other Liveness Hazards (cont'd)

- Poor responsiveness
  - may be caused by poor lock management
- Livelock: a thread, while not blocked, still cannot make progress because it keeps retrying an operation that will always fail
  - e.g., when two overly polite people are walking in the opposite direction in a hallway. Both continuously give away the current position