15-440
Distributed Systems
Recitation 8

Slides By: Hend Gedawy

& Previous TAs

Announcements

- **PS3** Due Today
- **P2** Due October 24
  - (next Tuesday)

Outline

- **Project 2 Objectives Recap**
- Dining Philosophers & Deadlocks
- Synchronization in Project 2
- Implementing Synchronization in Java

Project 2 Objectives: Reminder

1. **Devise and apply a synchronization algorithm that:**
   - achieves *correctness* while sharing files
   - and ensures *fairness* to clients.

2. **Devise and apply a replication algorithm that:**
   - achieves load-balancing among storage servers
   - and ensures consistency of replicated files.

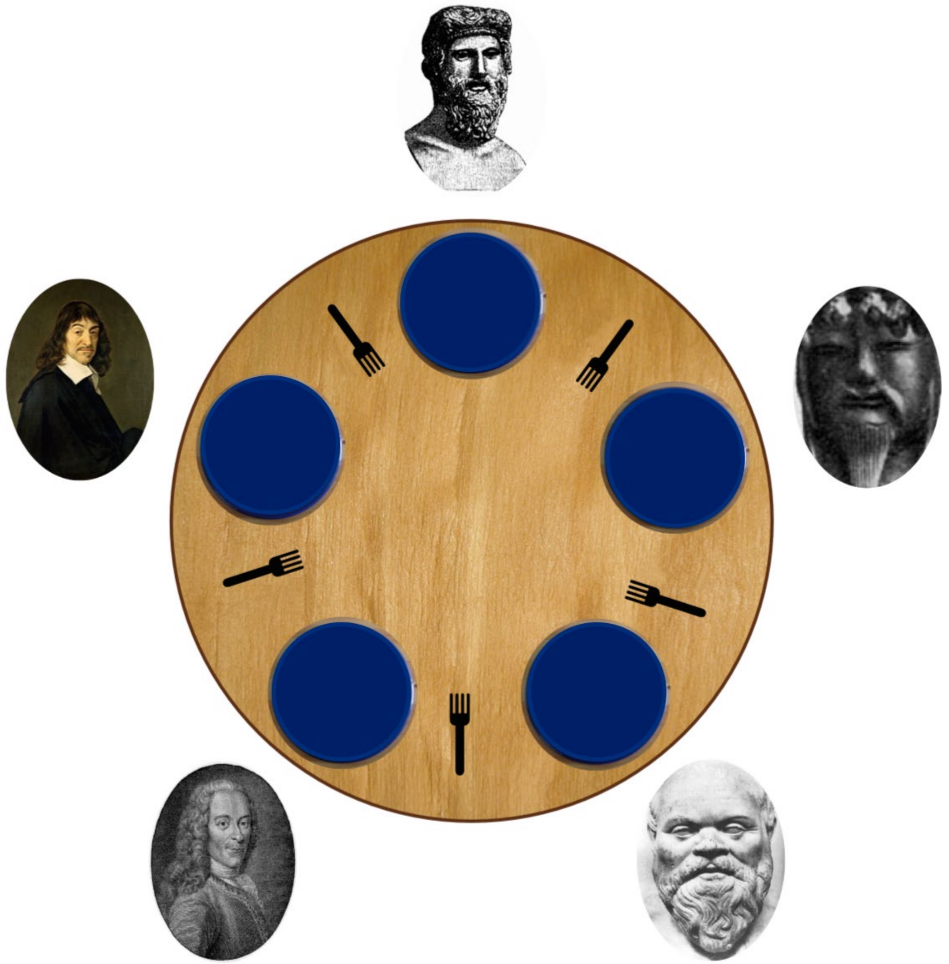
Project 2 Objectives: Reminder

1. **Devise and apply a synchronization algorithm that:**
   - achieves *correctness* while sharing files
   - and ensures *fairness* to clients.

2. **Devise and apply a replication algorithm that:**
   - achieves load-balancing among storage servers
   - and ensures consistency of replicated files.

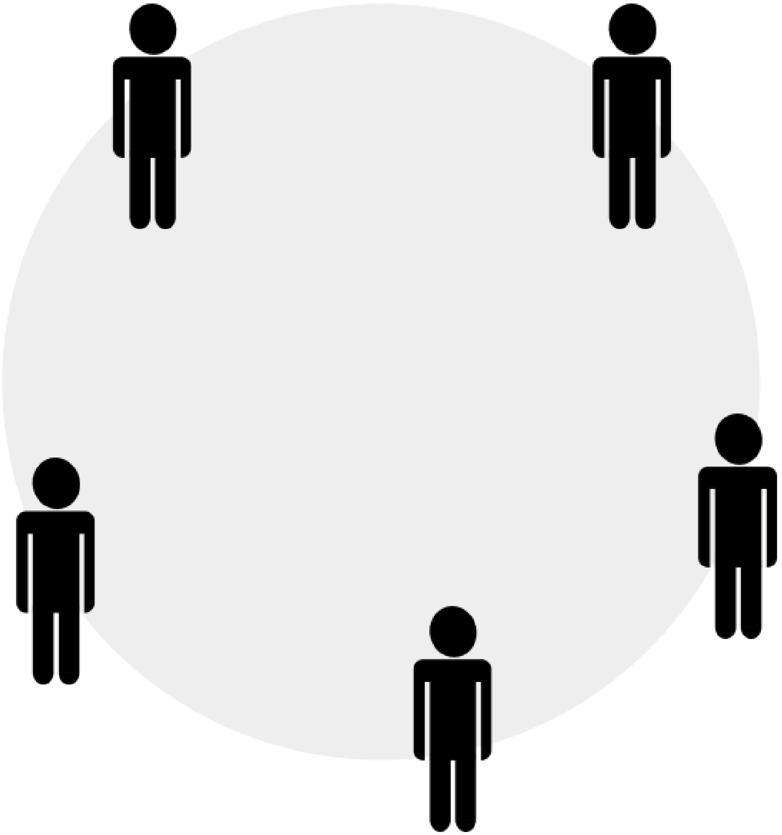
Outline

- Project 2 Objectives Recap
- **Dining Philosophers & Deadlocks**
- Synchronization in Project 2
- Implementing Synchronization in Java



Dining Philosophers

One of the classic problems used to describe **synchronization issues in accessing shared resources by multiple entities** and illustrate techniques for solving them.

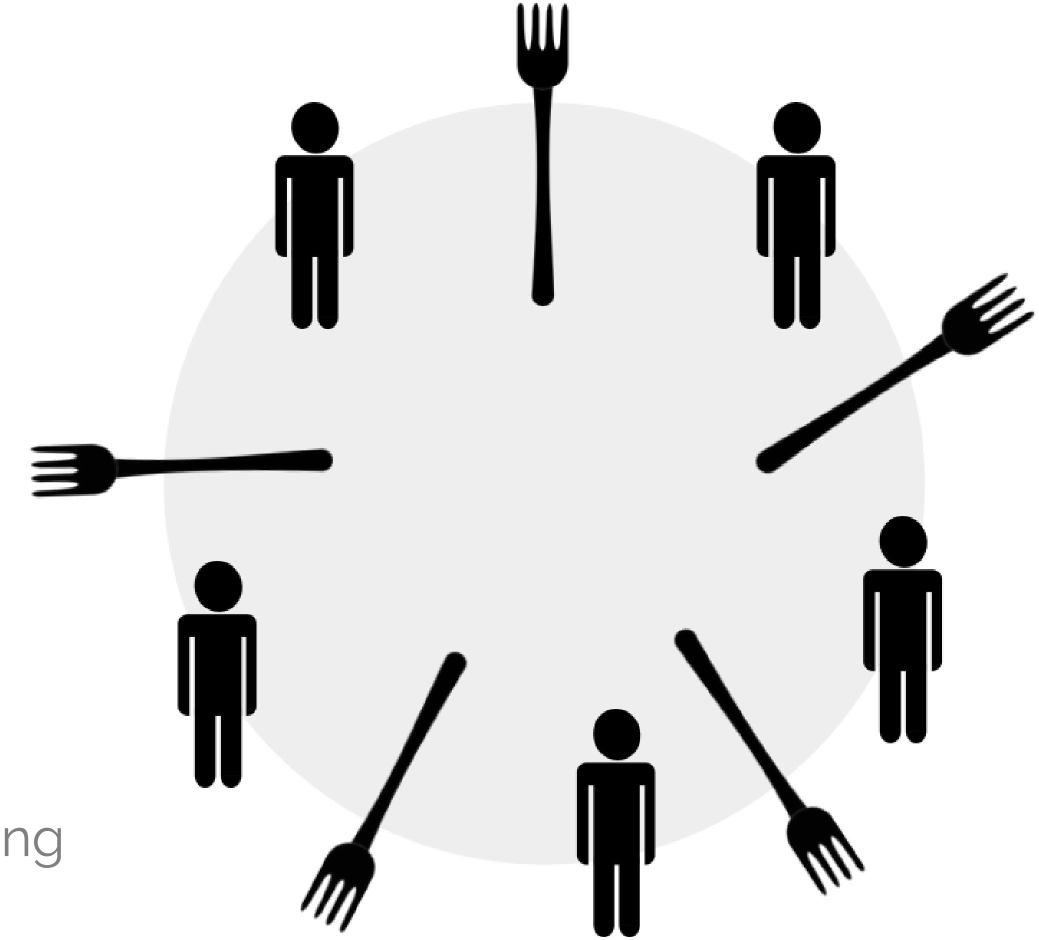


Dining Philosophers

- ***5** Silent **philosophers** (P1 - P5)*
- *Actions: **Thinking and Eating***
- ***5 Forks** to share (F1 - F5)*

- *Each Pi needs a pair of forks*
- *When Pi is done eating, he is back to thinking and puts back his forks*

Goal: come up with a scheme/protocol that helps the philosophers achieve their goal of eating and thinking without getting starved to death



Dining Philosophers

Step 1: think until the left fork is available; when it is, pick up;

Step 2: think until the right fork is available; when it is, pick up;

Step 3: when both fork are held, eat for some time;

Step 4: then, put the right fork down;

Step 5: then, put the left fork down;

Step 6: repeat from the beginning

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same forks at the same time.

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same forks at the same time.

Efficiency

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same forks at the same time.

Efficiency

Philosophers do not wait too long to pick-up forks when they want to eat.

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same forks at the same time.

Efficiency

Philosophers do not wait too long to pick-up forks when they want to eat.

Fairness

Dining Philosophers

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same forks at the same time.

Efficiency

Philosophers do not wait too long to pick-up forks when they want to eat.

Fairness

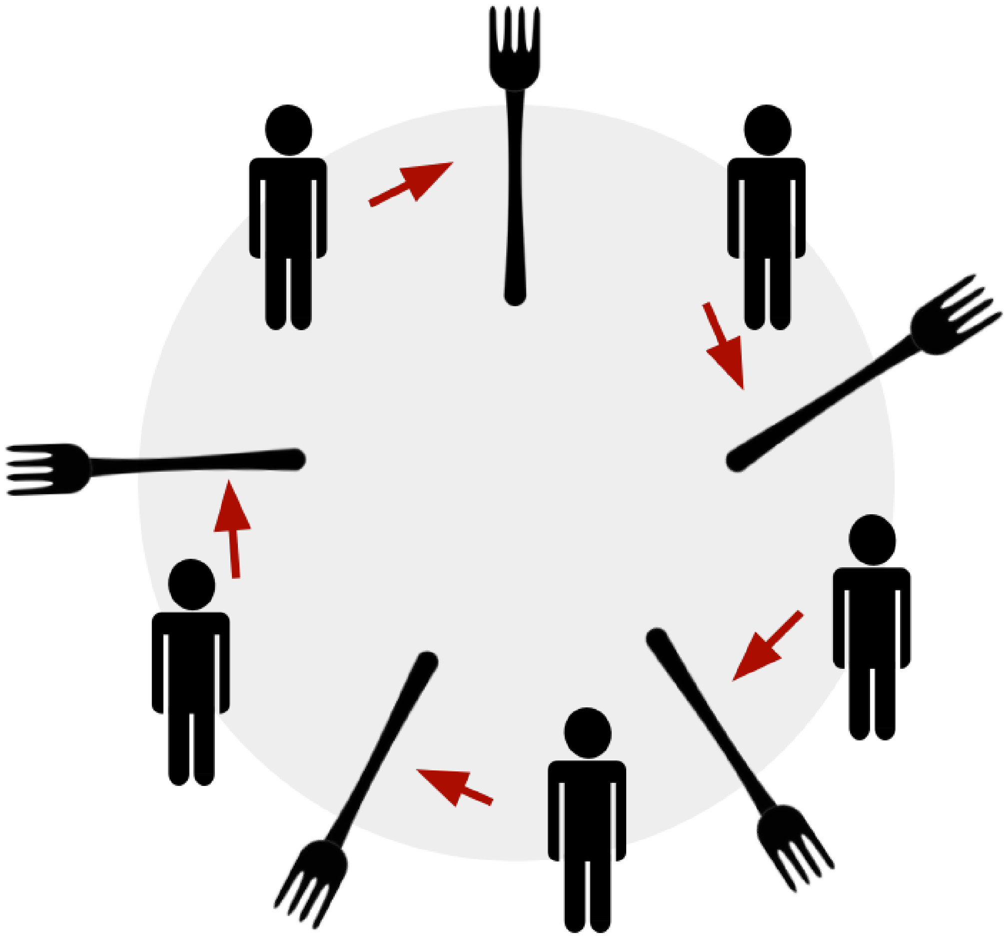
No philosopher should be unable to pick up forks forever and starve

Pseudocode

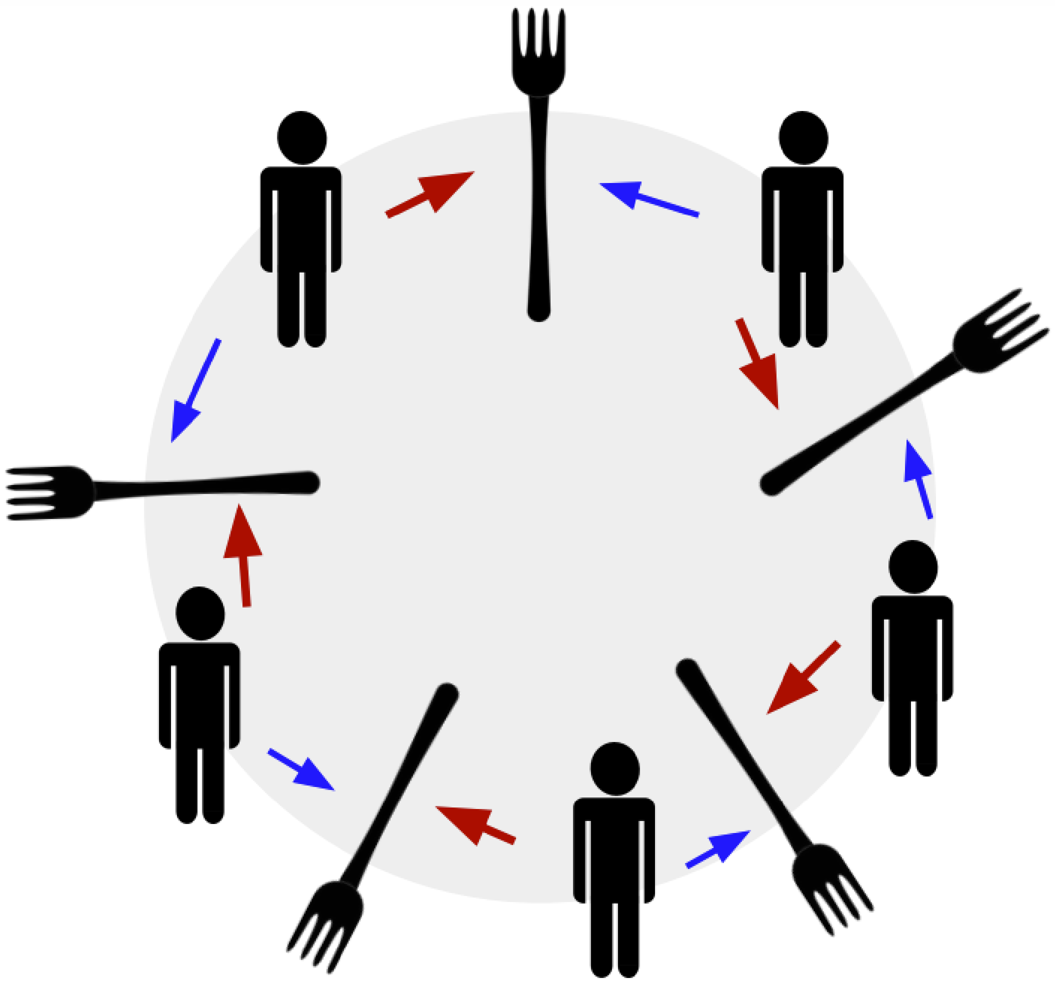
```
while(true) {
    // Initially, thinking about life, universe, and everything
    think();
    // Take a break from thinking, hungry now
    pick_up_left_fork();
    pick_up_right_fork();
    eat();
    put_down_right_fork();
    put_down_left_fork();

    // Not hungry anymore. Back to thinking!
}
```

What's wrong with this code

Dining Philosophers

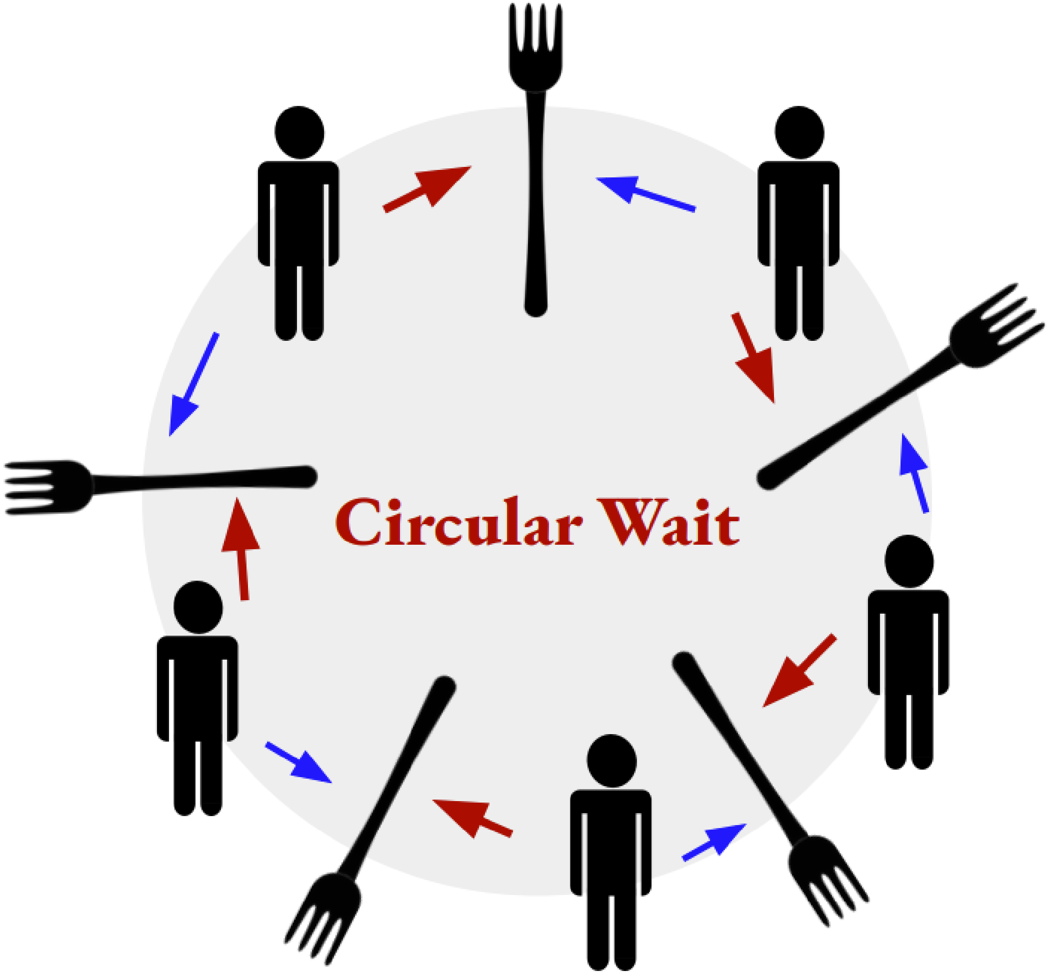


Dining Philosophers



A deadlock is a situation where the **progress of a system is halted** as each process is waiting to acquire a resource held by some other process.

Dining Philosophers



Dining Philosophers

How do we fix this?

A concurrent system with a need for synchronization, should ensure

Correctness

No two philosophers should be using the same chopsticks at the same time.

Efficiency

Philosophers do not wait too long to pick-up chopsticks when they want to eat.

Fairness

No philosopher should be unable to pick up chopsticks forever and starve

Dining Philosophers – Handling the Deadlock of Circular Waits

Initial Protocol

Philosopher (Object firstForkToPick, Object SecondForkTpPick)

```
for (int i = 0; i < philosophers.length; i++) {
    Object leftFork = forks[i];
    Object rightFork = forks[(i+1) % forks.length];
    philosophers[i] = new Philosopher(leftFork, rightFork);
    Thread t = new Thread(philosophers[i], "Philosopher " + (i+1));
    t.start();
}
```

Dining Philosophers – Handling the Deadlock of Circular Waits

Breaking the Waiting Circle

Philosopher (Object firstForkToPick, Object SecondForkTpPick)

```
for (int i = 0; i < philosophers.length; i++) {
            Object leftFork = forks[i];
            Object rightFork = forks[(i + 1) % forks.length];
            if (i == philosophers.length - 1) {
                // The last philosopher picks up the right fork first
                philosophers[i] = new Philosopher(rightFork, leftFork);
            } else {
                philosophers[i] = new Philosopher(leftFork, rightFork);
            }

            Thread t = new Thread(philosophers[i], "Philosopher " + (i + 1));
            t.start();
        }
```

Deadlocks-
Account Transfer Example

```
public boolean transfer(Account from, Account to, int val) {
    synchronized(from) {
        if (from.getbal() > val)
            from.post(-val);
        else
            return false;
        synchronized(to) {
            to.post(val);
        }
        return true;
    }
}
```

Time

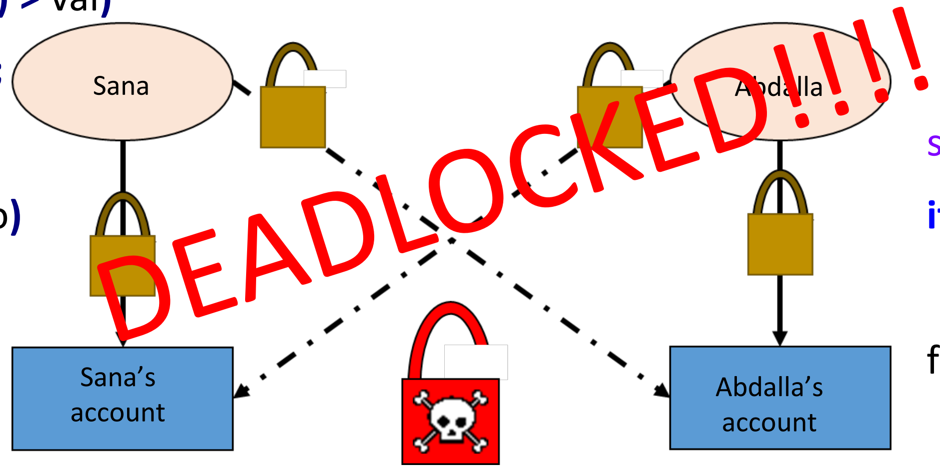
Sana -> Abdalla

synchronized(from) {

if (from.getbal() > val)

from.post(-val);

synchronized(to)



Abdalla -> Sana

synchronized(from) {

if (from.getbal() > val)

from.post(-val);

synchronized(to)

How to fix?

Sana wants to transfer 10 riyals to Abdalla
Abdalla wants to transfer 20 riyals to Sana
Will our code always work?

Deadlocks-
Account Transfer Example Resolution

```java
public boolean transfer(Account2 from, Account2 to, int val) {
    Account2 first = (from.rank > to.rank)? from : to;
    Account2 second = (from.rank > to.rank)? to: from;
    synchronized(first) {
        synchronized(second) {
            if (from.getbal() > val)
                from.post(-val);
            else {
                return false;
            }
            to.post(val);
            return true;
        }
    }
}
```

Time

Sana -> Abdalla

synchronized(SanaAccount)

synchronized(AbdallAccount)

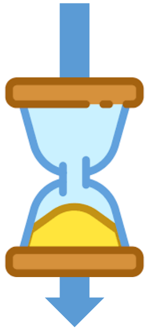
if (SanaAccount.getbal() > val)

SanaAccount.post(-val)

AbdallaAccount.post(val)

Abdalla -> Sana

Synchronized(SanaAccount)



synchronized(AbdallaAccount)
if (AbdallaAccount.getbal() > val)
AbdallaAccount.post(-val)
SanaAccount.post(val)

Fix: Apply Ranking to shared resources and locks should be acquired in order based on rank

Suppose Sana's account has higher rank

Sana wants to transfer 10 riyals to Abdalla
Abdalla wants to transfer 20 riyals to Sana

Outline

- Project 2 Objectives Recap
- Dining Philosophers & Deadlocks
- **Synchronization in Project 2**
- Implementing Synchronization in Java

Project 2: Synchronization

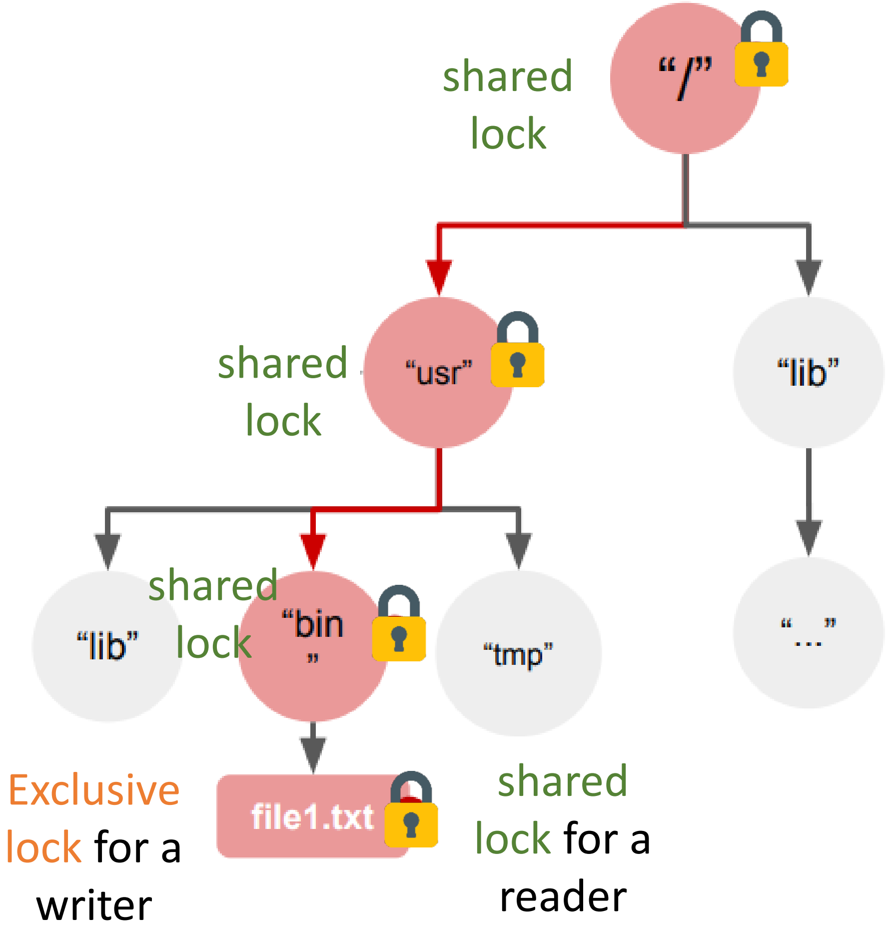
- **Reader & Writer clients acquire lock before invoking the method, and release the lock after they are done**

1. **Reader:**
   - Reader first requests a **read/non-exclusive/shared lock**
   - **Multiple readers can acquire a read lock** simultaneously
2. **Writer:**
   - Writer first requests a **write/exclusive lock**
   - Only **one writer can acquire a write lock** at a time
3. **Order:**
   - Readers and writers are queued and served in the **FIFO** order

Project 2: Synchronization

Naming Server **grants a reader or writer read/shared locks** on all the directories in the path to prevent modifications

Naming Server **then grants the requester a shared lock if it is a reader or an exclusive lock if it is a writer** to the file



Outline

- Project 2 Objectives Recap
- Dining Philosophers & Deadlocks
- Synchronization in Project 2
- **Implementing Synchronization in Java**

Thread Synchronization in Java

- Synchronized Block
  - Using synchronized keyword to define a critical section
- Lock APIs
  - Using Lock interface in the *java.util.concurrent.lock* package
- Semaphores
  - Using Semaphore class in the *java.util.concurrent.Semaphore* package

Thread Synchronization in Java

- Synchronized Block
  - Using synchronized keyword to define a critical section
- Lock APIs
  - Using Lock interface in the *java.util.concurrent.lock* package
- Semaphores
  - Using Semaphore class in the *java.util.concurrent.Semaphore* package

Synchronized Block

```
public boolean transfer(Account2 from, Account2 to, int val) {
    Account2 first = (from.rank > to.rank)? from : to;
    Account2 second = (from.rank > to.rank)? to: from;
    synchronized(first) {
        synchronized(second) {
            if (from.getbal() > val)
                from.post(-val);
            else {
                return false;
            }
            to.post(val);
            return true;
        }
    }
}
```

Thread Synchronization in Java

- Synchronized Block
  - Using synchronized keyword to define a critical section
- Lock APIs
  - Using Lock interface in the *java.util.concurrent.lock* package
- Semaphores
  - Using Semaphore class in the *java.util.concurrent.Semaphore* package

Locks– Lock Usage

```
Lock lock = ...;

lock.lock();
try {
	// manipulate protected state
} finally {
	lock.unlock();
}
```

```
Lock lock = ...;

if (lock.tryLock()) {
	try {
	// manipulate protected state
	} finally {
		lock.unlock();
	}
} else {
 // perform alternative actions
}
```

The thread that calls Lock first becomes the owner, and it is the only thread that can release the lock

Locks vs. Synchronized blocks

Synchronized Blocks	Locks
Fully contained **within a method**	Can have **lock() and unlock()** operation in **separate methods**
Rigid, any thread can acquire the lock once released, **no preference** can be specified	Flexible; we **can prioritize** waiting threads for example
A **thread** always gets **blocked** if it can't get an access to the synchronized block	The Lock API provides **tryLock() non-blocking** method. The thread acquires lock only if it's available and not held by any other thread.
A **thread** which is in "**waiting**" state to acquire the access to synchronized block, **can't be interrupted**	The Lock API provides a method lockInterruptibly() which **can** be used to **interrupt the thread when it's waiting** for the lock

Locks– Lock API

Method	Description
`void lock()`	Acquire the lock if it's available; if the lock isn't available a thread gets blocked until the lock is released
`void lockInterruptibly()`	similar to the *lock()*, but it allows the blocked thread to be interrupted and resume the execution through a thrown *java.lang.InterruptedException*
`boolean tryLock()`	non-blocking version of *lock()* method; it attempts to acquire the lock immediately. It returns true if locking succeeds; false otherwise.
`boolean tryLock(long timeout, TimeUnit timeUnit)`	similar to *tryLock()*, except it waits up the given timeout before giving up trying to acquire the *Lock*
`void unlock()`	unlocks the *Lock* instance

Locks– Read/Write Locks

The rules for acquiring the *ReadLock* or *WriteLock* by a thread:

•**Read Lock** (Shared)– If no thread acquired the write lock or requested for it, multiple threads can acquire the read lock.

•**Write Lock** (Exclusive)– If no threads are reading or writing, only one thread can acquire the write lock.

Locks–
Read/Write Locks

ReadWriteLock Interface

Modifier and Type	Method and Description
Lock	readLock() Returns the lock used for reading.
Lock	writeLock() Returns the lock used for writing.

Lock Interface

Modifier and Type	Method and Description
void	lock() Acquires the lock.
void	lockInterruptibly() Acquires the lock unless the current thread is **interrupted**.
Condition	newCondition() Returns a new Condition instance that is bound to this Lock instance.
boolean	tryLock() Acquires the lock only if it is free at the time of invocation.
boolean	tryLock(long time, TimeUnit unit) Acquires the lock if it is free within the given waiting time and the current thread has not been **interrupted**.
void	unlock() Releases the lock.

ReentrantReadWriteLock Class

Method and Description
getOwner() Returns the thread that currently owns the write lock, or null if not owned.
getQueuedReaderThreads() Returns a collection containing threads that may be waiting to acquire the read lock.
getQueuedThreads() Returns a collection containing threads that may be waiting to acquire either the read or write lock.
getQueuedWriterThreads() Returns a collection containing threads that may be waiting to acquire the write lock.
getQueueLength() Returns an estimate of the number of threads waiting to acquire either the read or write lock.
getReadHoldCount() Queries the number of reentrant read holds on this lock by the current thread.
getReadLockCount() Queries the number of read locks held for this lock.
getWaitingThreads(Condition condition) Returns a collection containing those threads that may be waiting on the given condition associated with the write lock.
getWaitQueueLength(Condition condition) Returns an estimate of the number of threads waiting on the given condition associated with the write lock.
getWriteHoldCount() Queries the number of reentrant write holds on this lock by the current thread.
hasQueuedThread(Thread thread) Queries whether the given thread is waiting to acquire either the read or write lock.
hasQueuedThreads() Queries whether any threads are waiting to acquire the read or write lock.
hasWaiters(Condition condition) Queries whether any threads are waiting on the given condition associated with the write lock.
isFair() Returns true if this lock has fairness set true.
isWriteLocked() Queries if the write lock is held by any thread.
isWriteLockedByCurrentThread() Queries if the write lock is held by the current thread.
readLock() Returns the lock used for reading.
toString() Returns a string identifying this lock, as well as its lock state.
writeLock() Returns the lock used for writing.

Locks– Using ReentrantReadWriteLock Class

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();

    // multiple readers can enter this section
    // if not locked for writing,
    // and not writers waiting to lock for writing.

readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();

    // only one writer can enter this section,
    // and only if no threads are currently reading.

readWriteLock.writeLock().unlock();
```

Locks– ReentrantReadWriteLock Class Example

```
public class SynchronizedHashMapWithReadWriteLock {

    Map<String,String> syncHashMap = new HashMap<>();
    ReadWriteLock lock = new ReentrantReadWriteLock();

    Lock writeLock = lock.writeLock();

    Lock readLock = lock.readLock();

    //...
```

```
public void put(String key, String value) {
        try {
            writeLock.lock();
            syncHashMap.put(key, value);
        } finally {
            writeLock.unlock();
        }
    }
```

Locks– ReentrantReadWriteLock Class Example

```
public String remove(String key){
    try {
        writeLock.lock();
        return syncHashMap.remove(key);
    } finally {
        writeLock.unlock();
    }
}
```

```
public String get(String key){
    try {
        readLock.lock();
        return syncHashMap.get(key);
    } finally {
        readLock.unlock();
    }
}
```

Locks– Locks with Conditions

- The [*Condition* class](#) provides the ability for a *thread* to *wait for some condition to occur while executing the critical section.*
- This can occur when a thread *acquires the access to the critical section but doesn't have the necessary condition to perform its* operation

Example?

- Traditionally Java provides *wait(), notify() and notifyAll()* methods for thread intercommunication.
    - *Conditions* have similar mechanisms, but in addition, we can specify multiple conditions

Locks– Locks with Conditions

Modifier and Type	Method and Description
void	await() Causes the current thread to wait until it is signalled or **interrupted**.
boolean	await(long time, TimeUnit unit) Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
long	awaitNanos(long nanosTimeout) Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
void	awaitUninterruptibly() Causes the current thread to wait until it is signalled.
boolean	awaitUntil(Date deadline) Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.
void	signal() Wakes up one waiting thread.
void	signalAll() Wakes up all waiting threads.

Locks– Locks with Conditions Example

```
public class ReentrantLockWithCondition {

    Stack<String> stack = new Stack<>();
    int CAPACITY = 5;

    ReentrantLock lock = new ReentrantLock();
    Condition stackEmptyCondition = lock.newCondition();
    Condition stackFullCondition = lock.newCondition();
```

Locks– Locks with Conditions Example

```
public void pushToStack(String item){
       try {
           lock.lock();
           while(stack.size() == CAPACITY) {
               stackFullCondition.await(); //wait for a signal that the stack isn't full
           }
           stack.push(item);
           stackEmptyCondition.signalAll(); //Send a signal that the stack isn't empty
       } finally {
           lock.unlock();
       }
   }
```

Locks– Locks with Conditions Example

```
public String popFromStack() {
        try {
            lock.lock();
            while(stack.size() == 0) {
                stackEmptyCondition.await(); //wait for a signal that the stack isn't empty
            }
            return stack.pop();
        } finally {
            stackFullCondition.signalAll();  //Send a signal that the stack isn't full
            lock.unlock();
        }

}
```

Thread Synchronization in Java

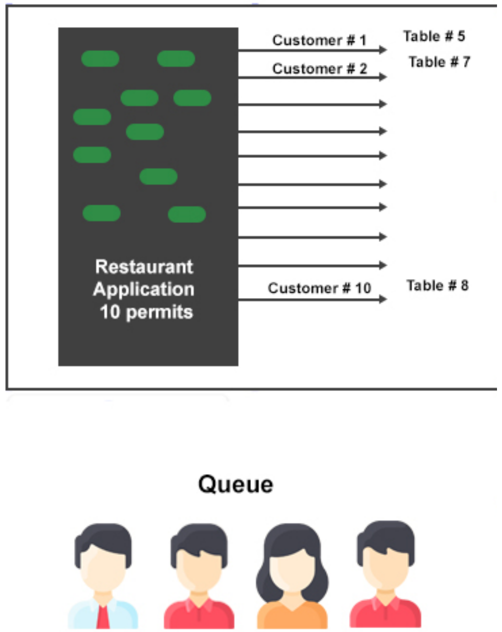
- Synchronized Block
  - Using synchronized keyword to define a critical section
- Lock APIs
  - Using Lock interface in the *java.util.concurrent.lock* package
- Semaphores
  - Using Semaphore class in the *java.util.concurrent.Semaphore* package

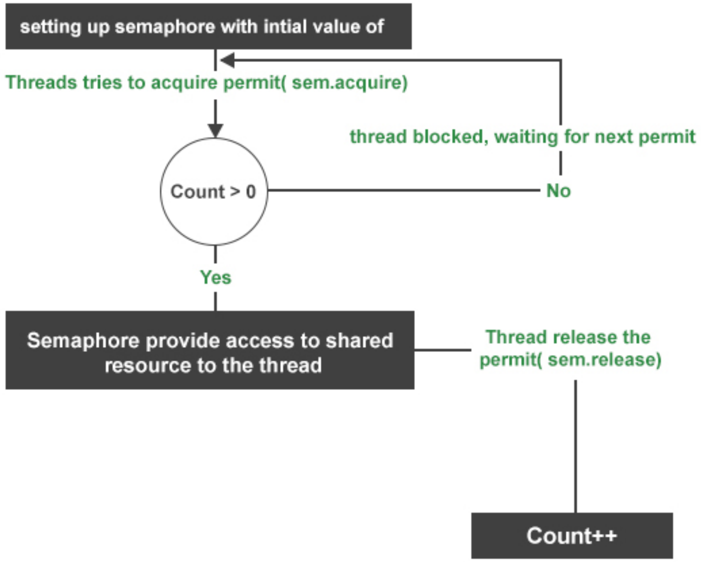
جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Semaphores

- Work on the **concept of permits**
- A semaphore is **initialized** with a **certain *number of permits***, which
  - depends on the problem at hand
  - usually set to the number of resources available
- When a thread wants to access a shared resource, it acquires a permit and releases it when it is done
- Threads that couldn't acquire permits are **queued**

There are 10 tables in a restaurant, and you are managing access to these tables





Semaphores - API

ensures the **order** in which the **queued requesting threads** acquire **permits** (based on their waiting time)

Method/Constructor		Description
Semaphore(int permits, **boolean fair**)		Creates a Semaphore with the given number of permits and the given fairness setting
acquire()	**Blocking**	Acquires a permit; blocks until one is available
acquire(int permits)		Acquires the given number of permits from this semaphore, blocking until all are available
tryAcquire()	**Non-Blocking**	Return true if a permit is available immediately and acquire it; otherwise return false
availablePermits()		Return number of current permits available
drainPermits()		Acquires and returns all permits that are immediately available

BinarySemaphores- Mutex

Mutex acts as a binary semaphore (i.e. only one permission at a time),

We can use it to implement **mutual exclusion.**

```
Semaphore mutex = new Semaphore(1);
try {
 mutex.acquire();
   assertEquals(0, mutex.availablePermits());
} catch (InterruptedException e) {
   e.printStackTrace();
} finally {
 mutex.release();
   assertEquals(1, mutex.availablePermits());
}
```

BinarySemaphores vs. Locks

- **Is a type of signaling mechanism**,
- **provides a non-ownership-based signaling mechanism for mutual exclusion**.
- Any thread can call Acquire or Release
- Therefore, any thread can release the permit for a deadlock recovery of a binary semaphore.
- **a higher-level synchronization mechanism** by allowing a custom implementation of a locking mechanism and deadlock recovery
- A Semaphore can be used as a queue of blocked threads that are waiting for a condition to be true.

- is a **locking mechanism.**
- Provides **a reentrant mutual exclusion with owner-based locking capabilities** and is useful as a simple mutex.
  - The thread who has the lock calls unlock
- On the contrary, **deadlock recovery is difficult** to achieve in the case of a reentrant lock. For instance, **if the owner thread of a reentrant lock goes into sleep or infinite wait, it won't be possible to release the resource**, and a deadlock situation will result.
- **a low-level synchronization with a fixed locking mechanism**.

Credits

This recitation was inspired by multiple Baeldung tutorials:

Readers-writers problem

The Dining Philosophers Problem

Locks in Java

Semaphores in Java

Semaphores in Java (2)

Mutex

https://crystal.uta.edu/~ylei/cse6324/data/semaphore.pdf