MULTITHREADING AND PARALLEL PROGRAMMING

Chapter 30- Liang

Roadmap

- Overview of multithreading (§30.2).
- Implementing the **Runnable** interface (§30.3).
- The **Thread** class (§30.3).
- Methods in the **Thread** class (§30.4).
- Thread pool (§30.6).
- Synchronize threads to avoid race conditions (§30.7).
- Synchronize threads using locks (§30.8).
- Thread communications using conditions on locks (§30.9–30.10).
- Restricting the number of accesses to a shared resource using semaphores (§30.12).
- Using the resource-ordering technique to avoid deadlocks (§30.13).
- Thread life cycle (§30.14).

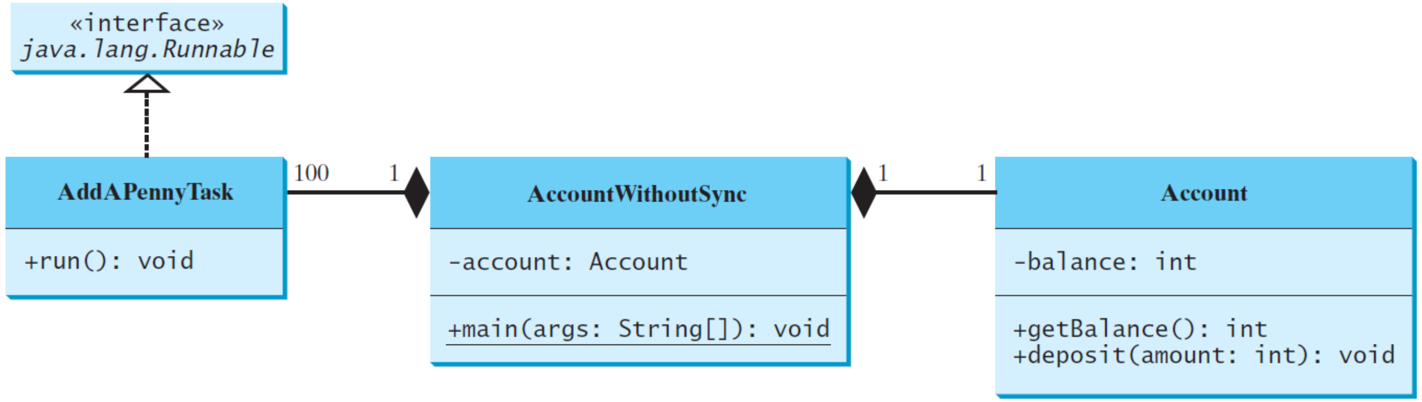
Thread Synchronization

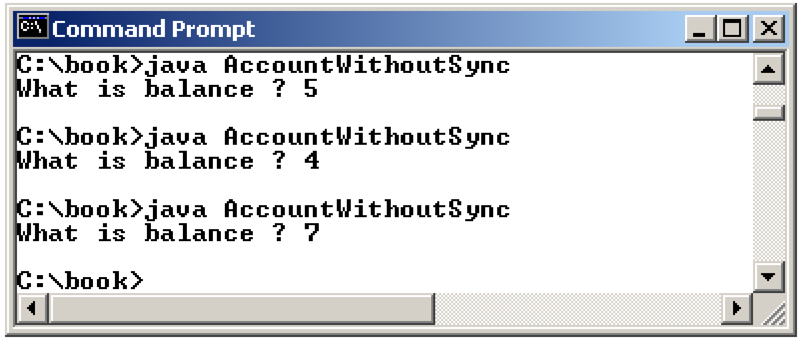
- Thread **synchronization** is to coordinate the execution of the dependent threads.
- A shared resource may be corrupted if it is accessed simultaneously by multiple threads.
  - For example, two unsynchronized threads accessing the same bank account may cause conflict.

Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

Example: Showing Resource Conflict

- Objective: Write a program that demonstrates the problem of resource conflict. Suppose that you create and launch one hundred threads, each of which adds a penny to an account. Assume that the account is initially empty.





Race Condition

- What, then, caused the error in the example? Here is a possible scenario:

Step	Balance	Task 1	Task 2
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

- The effect of this scenario is that Task 1 did nothing, because in Step 4 Task 2 overrides Task 1's result.
- Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes conflict.
- This is a common problem known as a ***race condition*** in multithreaded programs.
- A class is said to be *thread-safe* if an **object of the class does not cause a race condition in the presence of multiple threads**.
- <u>The Account class is not thread-safe.</u>

The **`synchronized`** keyword

- To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as critical region/section.
- The critical region is the entire deposit method.
- You can use the `synchronized` keyword to synchronize the method so that only one thread can access the method at a time.
- There are several ways to correct the problem,
  - one approach is to make Account thread-safe by adding the synchronized keyword in the deposit method in Line 45 as follows:

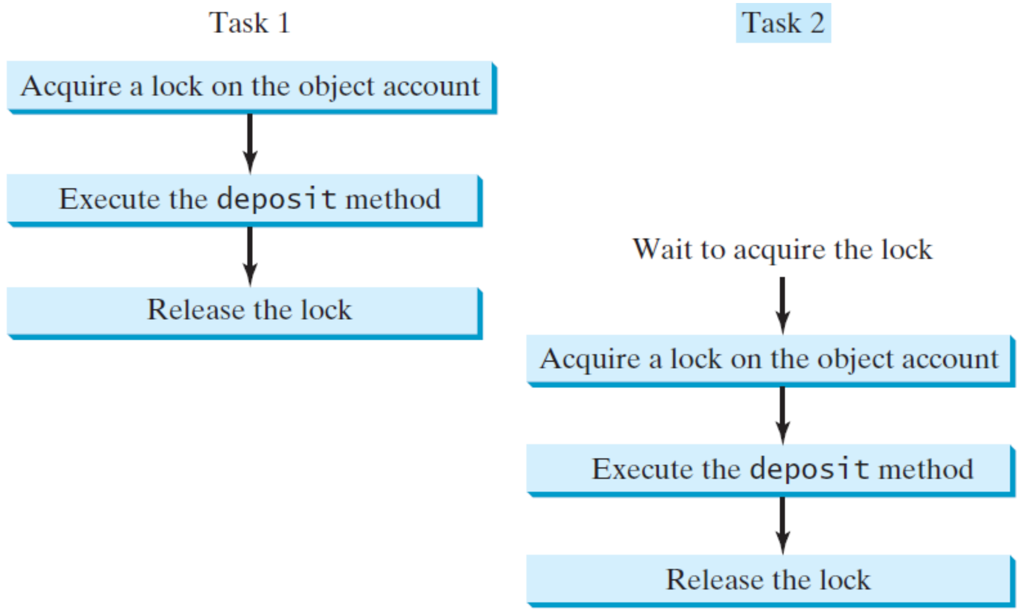
```
public synchronized void deposit(double amount)
```

Synchronizing Instance Methods and Static Methods

- A synchronized method acquires a lock before it executes.
  - A lock is a mechanism for exclusive use of a resource.
- In the case of an instance method, the lock is on the object for which the method was invoked.
  - If one thread invokes a synchronized instance method on an object, the lock of that object is acquired first, then the method is executed, and finally the lock is released.
  - Another thread invoking the same method of that object is blocked until the lock is released.
- In the case of a static method, the lock is on the class.
  - If one thread invokes a static method on an object, the lock of that class is acquired first, then the method is executed, and finally the lock is released.
  - Another thread invoking the same method of that class is blocked until the lock is released.

Synchronizing Tasks

- If the `deposit` method synchronized, the earlier scenario cannot happen.
- If Task 2 starts to enter the method, and Task 1 is already in the method, Task 2 is blocked until Task 1 finishes the method.



Synchronizing Statements

- A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method.
- This block is referred to as a *synchronized block*. The general form of a synchronized statement is as follows:

```
synchronized (expr) {
  statements;
}
```

- The expression expr must evaluate to an object reference.
- If the object is already locked by another thread, the thread is blocked until the lock is released.
- When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronizing Statements vs. Methods

- Any synchronized instance method can be converted into a synchronized statement.
- Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {
  // method body
}
```

- This method is equivalent to

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

Example

- Four threads are accessing and modifying a shared Counter object named cntObj. What is the impact of invoking:
  - T4 invoking doubleValue() ?
  - T1, T2 invoking increment() and T3 invoking decrement() ?
  - T3 invoking increment() and T2, T4 invoking getValue() ?
- Address the possibility of race conditions.

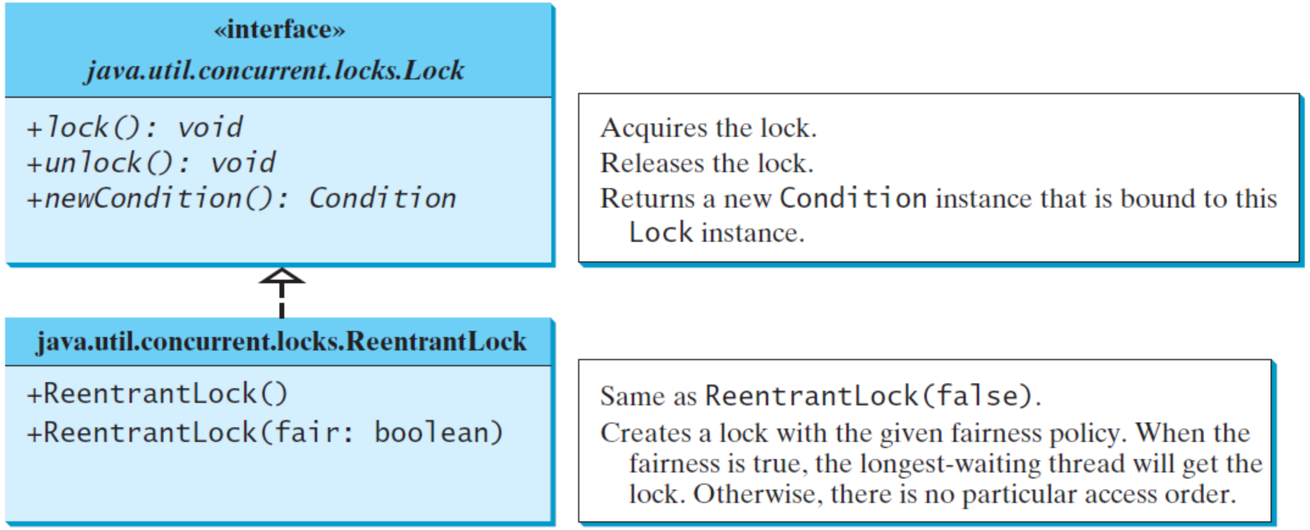
```
public class Counter {
     private int count = 0;
     public static synchronized void doubleValue() {
        count = count * 2;
     }
     public synchronized void increment() {
        count++;
     }
     public synchronized void decrement() {
        count--;
     }
     public void update() {
        count+=3;
     }
     public int getValue() {
          return count;
     }
}
```

Synchronization Using Locks

- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.
- Locks and conditions can be explicitly used to synchronize threads.
  - JDK 1.5 enables you to use locks explicitly.
- The new locking features are flexible and give you more control for coordinating threads.

Synchronization Using Locks

- A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks.
- A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.



Fairness Policy

- `ReentrantLock` is a concrete implementation of Lock for creating mutual exclusive locks.
- You can create a lock with the specified fairness policy.
  - True fairness policies guarantee the longest-wait thread to obtain the lock first.
  - False fairness policies grant a lock to a waiting thread without any access order.
- There are trade-offs between different types of locks used in multithreaded programming.
  - Overall performance: Measured by factors like throughput (requests processed per second) or latency (time taken to acquire the lock).
  - Variance: The spread of values around the average. Here, it refers to the variability in time it takes threads to acquire a lock.
  - Starvation: A situation where a thread waits indefinitely to acquire a lock due to other threads constantly taking it.

Fairness Policy

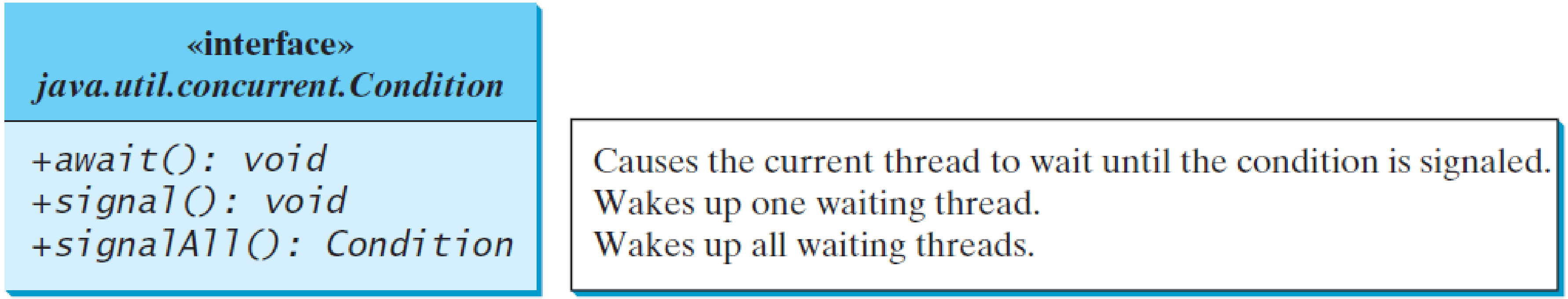
- Programs using fair locks may have poorer overall performance compared to those using the default (usually non-fair) setting.
  - This is because fair locks prioritize serving waiting threads in order, this can lead to situations where a busy thread constantly acquires the lock, delaying other waiting threads and potentially impacting overall throughput.
- Fair locks offer smaller variances in lock acquisition times.
  - This means wait times for threads are more predictable, unlike non-fair locks where a single thread might dominate access, causing some threads to wait significantly longer.
- Fair locks prevent starvation, which is a major concern with non-fair locks.
  - In non-fair scenarios, a thread might never get a chance to acquire the lock if other threads keep taking it, essentially starving it of access. Fair locks guarantee everyone eventually gets a turn, eliminating this risk.

Example: Using Locks

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance ? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

  // An inner class for account
  public static class Account {
    private static Lock lock = new ReentrantLock(); // Create a lock
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      lock.lock(); // Acquire the lock

      try {
        int newBalance = balance + amount;

        // This delay is deliberately added to magnify the
        // data-corruption problem and make it easy to see.
        Thread.sleep(5);

        balance = newBalance;
      }
      catch (InterruptedException ex) {
      }
      finally {
        lock.unlock(); // Release the lock
      }
    }
  }
}
```

Cooperation Among Threads

- The conditions can be used to facilitate communications among threads.
- A thread can specify what to do under a certain condition.
- Conditions are objects created by invoking the newCondition() method on a Lock object.
- Once a condition is created, you can use the methods `await()`, `signal()`, and `signalAll()` methods for thread communications

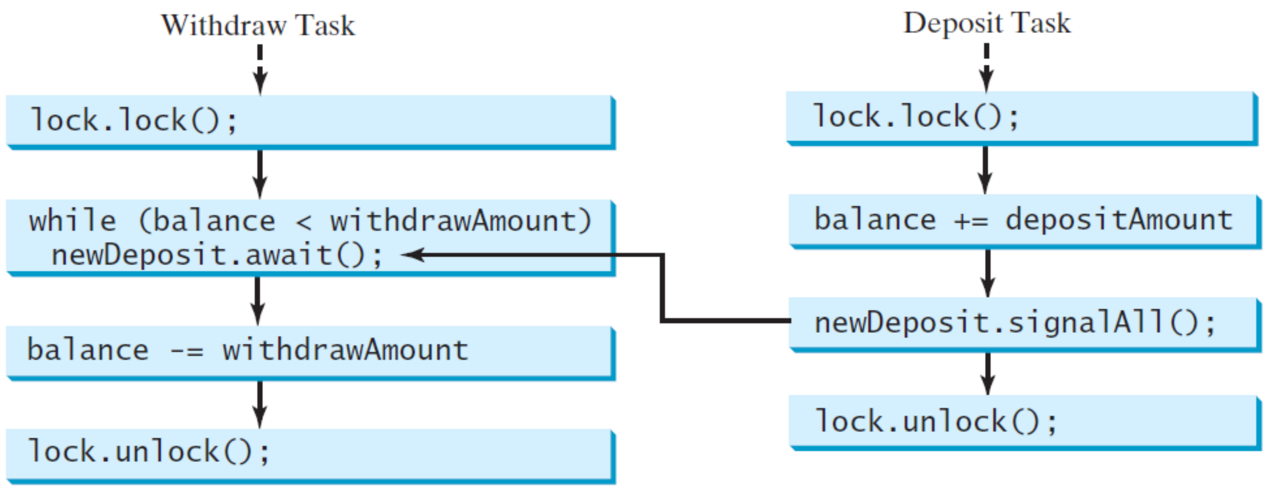


Cooperation Among Threads

- **How condition locks work?**
  1. A thread acquires the lock associated with a shared resource.
  2. The thread checks for the desired condition to be true (e.g., data available, task completed).
  3. If the condition is not true, the thread uses `await()` to release the lock and voluntarily wait on the condition variable associated with the lock.
  4. Another thread fulfills the condition by modifying the shared resource and calling `signal()` or `signalAll()` on the condition variable.
  5. One or all waiting threads are woken up based on the `signal()`/`signalAll()` implementation and compete to reacquire the lock.
  6. The awakened thread re-checks the condition and proceeds if it's true or waits again if not.

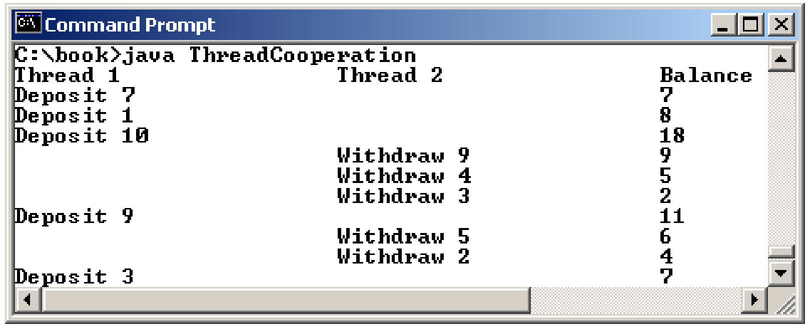
Cooperation Among Threads

- To **synchronize** the operations, use a lock with a condition: `newDeposit` (i.e., new deposit added to the account).
- If the `balance` is less than the amount to be withdrawn, the withdraw task will wait for the `newDeposit` condition.
- When the `deposit` task adds money to the account, the task signals the waiting withdraw task to try again.



Example: Thread Cooperation

- Write a program that demonstrates thread cooperation.
  - Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account.
  - The second thread must wait if the amount to be withdrawn is more than the current balance in the account.
  - Whenever new fund is deposited to the account, the first thread notifies the second thread to resume.
  - If the amount is still not enough for a withdrawal, the second thread must continue to wait for more fund in the account.
  - Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.



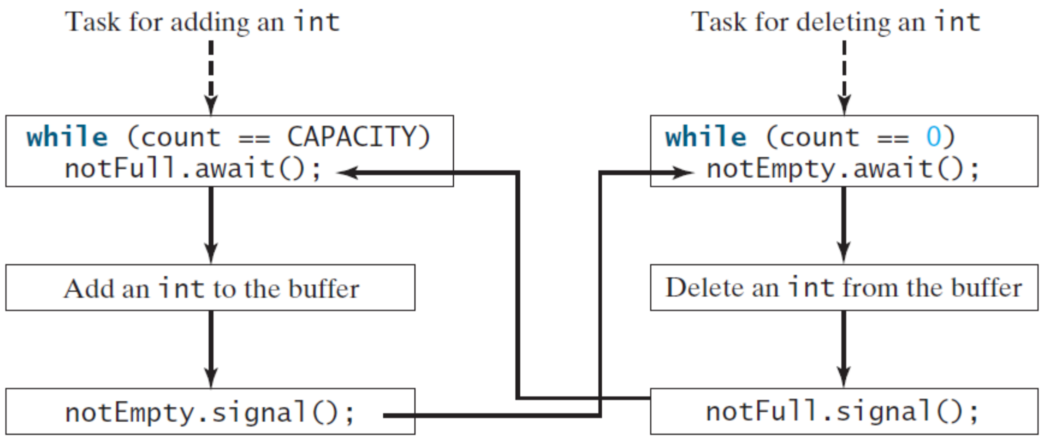
Example: Thread Cooperation

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class ThreadCooperation {
    private static Account account = new Account();
    public static void main(String[] args) {
        // Create a thread pool with two threads
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(new DepositTask());
        executor.execute(new WithdrawTask());
        executor.shutdown();
        System.out.println("Thread 1\t\tThread 2\t\tBalance");
    }
    public static class DepositTask implements Runnable {
        @Override // Keep adding an amount to the account
        public void run() {
            try { // Purposely delay it to let the withdraw method proceed
                while (true) {
                    account.deposit((int) (Math.random() * 10) + 1);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
    public static class WithdrawTask implements Runnable {
        @Override // Keep subtracting an amount from the account
        public void run() {
            while (true) {
                account.withdraw((int) (Math.random() * 10) + 1);
            }
        }
    }
    private static class Account {// An inner class for account
        private static Lock lock = new ReentrantLock();// Create a new lock
        // Create a condition
        private static Condition newDeposit = lock.newCondition();
        private int balance = 0;
        public int getBalance() {
            return balance;
        }
        public void withdraw(int amount) {
            lock.lock(); // Acquire the lock
            try {
                while (balance < amount) {
                    System.out.println("\t\t\tWait for a deposit");
                    newDeposit.await();
                }
                balance -= amount;
                System.out.println("\t\t\tWithdraw " + amount + "\t\t" + getBalance
());
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            } finally {
                lock.unlock(); // Release the lock
            }
        }
        public void deposit(int amount) {
            lock.lock(); // Acquire the lock
            try {
                balance += amount;
                System.out.println("Deposit " + amount + "\t\t\t\t\t" + getBalance
());
                // Signal thread waiting on the condition
                newDeposit.signalAll();
            } finally {
                lock.unlock(); // Release the lock
            }
        }
    }
}
```

Case Study: Producer/Consumer

self-study

- Consider the classic Consumer/Producer example.
- Suppose you use a buffer to store integers.
  - The buffer size is limited.
- The buffer provides the method `write(int)` to add an int value to the buffer and the method `read()` to read and delete an int value from the buffer.
- To synchronize the operations, use a lock with two conditions: `notEmpty` (i.e., buffer is not empty) and `notFull` (i.e., buffer is not full).
- When a task adds an int to the buffer, if the buffer is full, the task will wait for the `notFull` condition.
- When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the `notEmpty` condition.



Case Study: Producer/Consumer

self-study

- Listing 30.8 presents the complete program. The program contains the Buffer class (lines 43-89) and two tasks for repeatedly producing and consuming numbers to and from the buffer (lines 15-41). The write(int) method (line 58) adds an integer to the buffer. The read() method (line 75) deletes and returns an integer from the buffer.

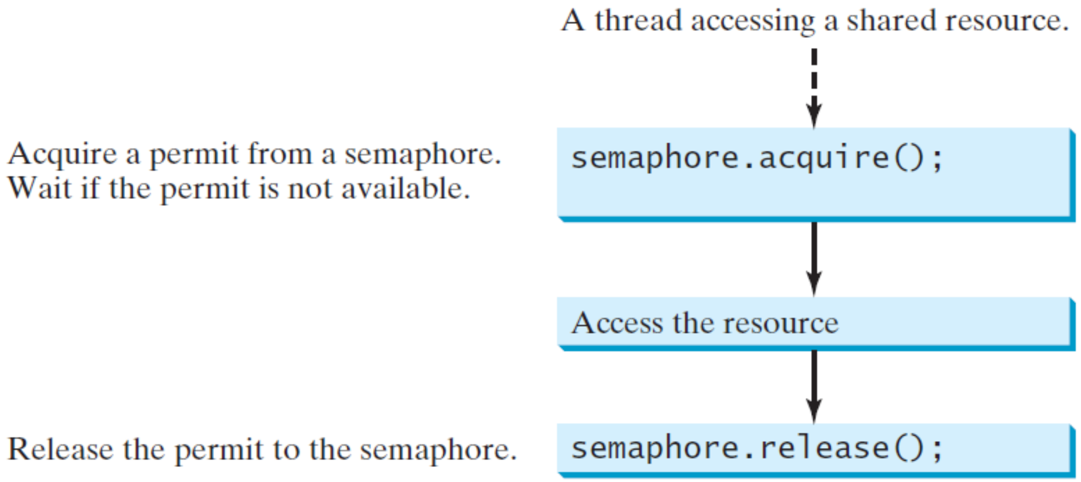
- For simplicity, the buffer is implemented using a linked list (lines 48-49). Two conditions notEmpty and notFull on the lock are created in lines 55-56. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the wait() and notify() methods to rewrite this example, you have to designate two objects as monitors.

Remarks

- Once a thread invokes `await()` on a condition, the thread waits for a signal to resume.
- If you forget to call `signal()` or `signalAll()` on the condition, the thread will wait forever.
- A condition is created from a Lock object.
- To invoke the method (e.g., `await()`, `signal()`, and `signalAll()`), you must first own the lock.
- If you invoke these methods without acquiring the lock, an exception will be thrown.

Semaphores

- Semaphores can be used to restrict the number of threads that access a shared resource.
  - Before accessing the resource, a thread must **acquire a permit** from the semaphore.
  - After finishing with the resource, the thread must **return the permit** back to the semaphore.



Creating Semaphores

- To create a semaphore, you must specify the number of permits with an optional fairness policy.
- A task acquires a permit by invoking the semaphore's `acquire()` method and releases the permit by invoking the semaphore's `release()` method.
- Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1.
- Once a permit is released, the total number of available permits in a semaphore is increased by 1.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

Creating Semaphores

- The code shows a semaphore with just one permit can be used to simulate a mutually exclusive lock.

```
// An inner class for Account
private static class Account {
  // Create a semaphore
  private static Semaphore semaphore = new Semaphore(1);
  private int balance = 0;

  public int getBalance() {
    return balance;
  }

  public void deposit(int amount) {
    try {
      semaphore.acquire(); // Acquire a permit
      int newBalance = balance + amount;

      // This delay is deliberately added to magnify the
      // data-corruption problem and make it easy to see
      Thread.sleep(5);

      balance = newBalance;
    }
    catch (InterruptedException ex) {
    }
    finally {
      semaphore.release(); // Release a permit
    }
  }
}
```

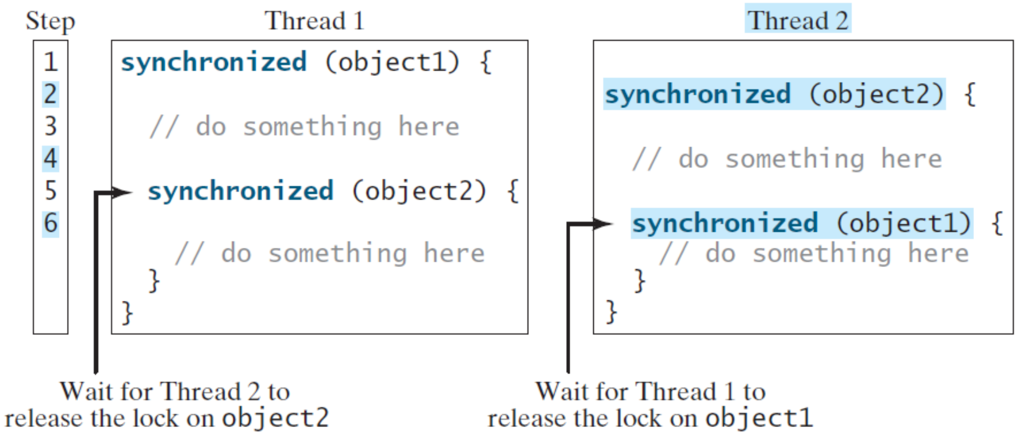
create a semaphore

acquire a permit

release a permit

Deadlock

- Sometimes two or more threads need to acquire the locks on several shared objects.
- This could cause deadlock, in which each thread has the lock on one of the objects and is waiting for the lock on the other object.
- Consider the scenario with two threads and two objects.
  - Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2.
  - Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1.
  - The two threads wait for each other to release the in order to get the lock, and neither can continue to run.

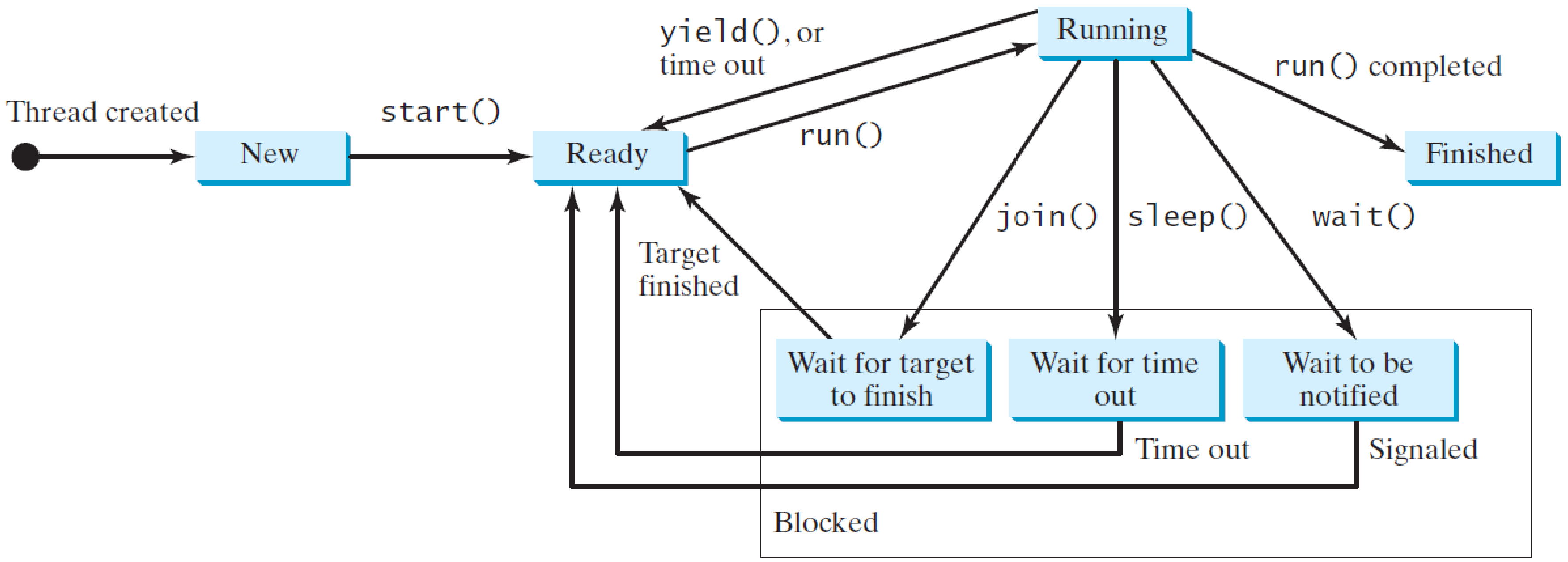


Preventing Deadlock

- Deadlock can be easily avoided by using a simple technique known as resource ordering.
- With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order.
- For the example, suppose the objects are ordered as object1 and object2.
  - Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2.
  - Once Thread 1 acquired a lock on object1, Thread 2 must wait for a lock on object1.
  - Thread 1 will be able to acquire a lock on object2 and no deadlock would occur.

Thread States

- A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.



Thread States

- A thread is in the NEW state after its creation using the Thread constructor but before calling its `start()` method. In this state, the thread hasn't allocated any resources and isn't eligible to run.
- After calling `start()`, the thread enters the RUNNABLE state. It's now eligible to run on a CPU core, waiting for its turn in the scheduling queue. Multiple threads can be in the RUNNABLE state simultaneously.
  - A ready thread is runnable but may not be running yet. The operating system must allocate CPU time to it.
- When the thread scheduler assigns a CPU core to a RUNNABLE thread, it transitions to the RUNNING state. It actively executes its code and consumes CPU resources. Only one thread can be in the RUNNING state on a single CPU core at a time.
  - A running thread can enter the Ready state if its CPU time expires, or its `yield()` method is called.
- A thread can enter the Blocked state (i.e., become inactive) for several reasons.
  - It may have invoked the `join()`, `sleep()`, or `wait()` method.
  - It may be waiting for an I/O operation to finish.

Thread States

- A thread enters the BLOCKED state when it encounters an event that prevents it from further execution, such as:
  - Waiting for I/O operations to complete (e.g., reading from a file).
  - Waiting for timeout.
  - Waiting for a notification from another thread using wait() or join().
- A BLOCKED thread may be reactivated when the action caused the inactivation is reversed.
- The `isAlive()` method is used to find out the state of a thread.
  - It returns true if a thread is in the Ready, Blocked, or Running state.
  - It returns false if a thread is New and has not started or if it is Finished.
- The `interrupt()` method interrupts a thread in the following way:
  - If a thread is currently in the Ready or Running state, its interrupted flag is set.
  - If a thread is currently Blocked, it is awakened and enters the Ready state, and a java.lang.InterruptedException is thrown.
- A thread reaches the FINISHED state when it finishes executing its run method or explicitly throws an uncaught exception. It has released all its resources and can't be restarted.

ANY QUESTIONS?