Multithreading

• Advantages and disadvantages of threads
• User and kernel threads in general
• Java threads
  ■ Class `Thread`
  ■ Interface `Runnable`
Thread

• **Definition:** A thread is a single sequential flow of control within a program (also called *lightweight process*).

• Each thread acts like its own sequential program
  - Underlying mechanism divides up CPU between multiple threads.

• Two types of multithreaded applications
  - Make many threads that do many tasks in parallel, i.e., no communication between the threads (GUI).
  - Make many threads that do many tasks concurrently, i.e., communication between the threads (data access).
Advantages and disadvantages

• Advantages
  - Responsiveness
  - Resource sharing
  - Economy
  - Utilization of multiprocessor architectures

• Disadvantages
  - More complicated code
  - Deadlocks (very hard to debug logical program errors)
Single and Multithreaded Processes

![Diagram showing single-threaded and multi-threaded processes](image-url)
User and Kernel Threads

- Thread management done by user-level threads library.
  - Examples
    - POSIX Pthreads (e.g., Linux and NT)
    - Mach C-threads (e.g., MacOS and NeXT)
    - Solaris threads

- Supported by the kernel
  - Examples
    - Windows 95/98/NT/2000
    - Solaris
    - TRU64 (Compaq UNIX)
Solaris 2 Threads

user threads

kernel threads

light weight process

task 1

task 2

task 3

kernel

cpu
cpu
cpu

cpu
Java Threads

• Java threads may be created by
  ■ Extending `Thread` class
  ■ Implementing the `Runnable` interface
Class **Thread**

- The simplest way to make a thread
- Treats a thread as an object
- Override the `run()` method, i.e., the thread’s “main”
  - Typically a loop
  - Continues for the life of the thread
- Create **Thread** object, call method `start()`
- Performs initialization, call method `run()`
- Thread terminates when `run()` exits.
class Worker extends Thread {
    public void run() {
        System.out.println("I’m a worker thread");
    }
}

public class First{
    public static void main (String args[]){
        Worker runner = new Worker();
        runner.start();
        System.out.println("I’m the main thread");
    }
}
Extending the **Thread** Class, cont.

```java
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
public class TwoThreadsDemo {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

[Source: java.sun.com]
Sharing Resources

- *Single threaded programming*: you own everything, no problem with sharing

- *Multi-threaded programming*: more than one thread may try to use a shared resource at the same time
  - Add and withdraw from a bank account
  - Speak at the same time, etc.

- Java provides locks, i.e., monitors, for objects, so you can wrap an object around a resource
  - First thread that acquires the lock gains control of the object, and the other threads cannot call synchronized methods for that object.
Locks

• One lock pr. object for the object’s methods.
• One lock pr. class for the class’s static methods.

• Typically data is private, only accessed through methods.

• If a method is synchronized, entering that method acquires the lock.
  ■ No other thread can call any synchronized method for that object until the lock is released.
Sharing Resources, cont.

• Only one synchronized method can be called at any time for a particular object

```java
synchronized void foo() {/*...*/}
synchronized void bar() {/*...*/}
```

• Efficiency
  - Memory: Each object has a lock implemented in `Object`
  - Speed: JavaSoft: 6x method call overhead. Theoretical minimum 4 x overhead
    - Older standard Java libraries used synchronized a lot, did not provide any alternatives.
public class CubbyHole {
    private int contents;
    private boolean available = false;
    public synchronized int get() {
        while (available == false) {
            try { wait(); } ... }
        available = false;
        notifyAll();
        return contents;
    }
    public synchronized void put(int value) {
        while (available == true) {
            try { wait(); ...} }
        contents = value;
        available = true;
        notifyAll();
    }
}
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer "+ this.number + " put: " + i);
            try {Thread.sleep((int)(Math.random() * 100));}
                catch (InterruptedException e) { } }
    }
}
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer "+this.number+" got: "+value);
        }
    }
}
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}

Sharing Resources, cont.
The **Runnable** Interface

- To inherit from an existing object and make it a thread, implement the **Runnable** interface.
- A more classical, function-oriented way to use threads.

**Rule of Thumb:** If your class must subclass some other class (the most common example being **Applet**), you should use **Runnable**.

```java
public interface Runnable{
    public abstract void run();
}
```
The **Runnable** Interface, cont.

class Worker implements Runnable{
    public void run(){
        System.out.println("I\’m a worker thread");
    }
}

public class Second{
    public static void main(String args[]){
        Runnable runner = new Worker();
        Thread thrd = new Thread(runner);
        thrd.start();
        System.out.println("I\’m the main thread");
    }
}

OOP: Multithreading
class SimpleRunnable implements Runnable {
    private String myName;  private Thread t;
    SimpleRunnable (String name) {
        myName = name;  t = new Thread (this);  t.start();
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + myName);
            try {
                t.sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}  
        }
        System.out.println("DONE! " + myName);
    }
}

public class TwoRunnableDemo {
    public static void main (String[] args) {
        SimpleRunnable runner1 = new SimpleRunnable("Jamaica");
        SimpleRunnable runner2 = new SimpleRunnable("Fiji");
    }
}
Java Thread Management

- `suspend()` – suspends execution of the currently running thread.
- `sleep()` – puts the currently running thread to sleep for a specified amount of time.
- `resume()` – resumes execution of a suspended thread.
- `stop()` – stops execution of a thread.
Summary

• *Single-threaded programming*: live by all by your self, own everything, no contention for resources.

• *Multithreading programming*: suddenly ”others” can have collisions and destroy information, get locked up over the use of resources.

• Multithreading is built-into the Java programming language.

• Multithreading makes Java programs complicated
  ■ Multithreading is by nature difficult, e.g., deadlocks.