Concurrent Programming
19530-V (WS01)

Lecture 5:
Introduction to Concurrency in Java

Dr. Richard S. Hall
`rickhall@inf.fu-berlin.de`

Concurrent programming – November 20, 2001

Operating System Concepts

- Definition of a *process*
  - An executing program
    - Similar to an FSP process (i.e., a set of actions that cause state transformations)
  - A conceptual bookkeeping unit for the OS, is used to keep track of
    - Program counter
      - This keeps track of the next instruction to execute
    - All CPU register contents
    - Call stack
    - Open files
    - Memory (including actual program text/code)
    - Any other resources owned by the process

Operating System Concepts

- Definition of a *process*
  - Paraphrasing the previous slide, a process can be described as a resource container combined with an execution flow
  - Given this view of a process, we can imagine that it might make sense to conceptually break a process into these two orthogonal concepts

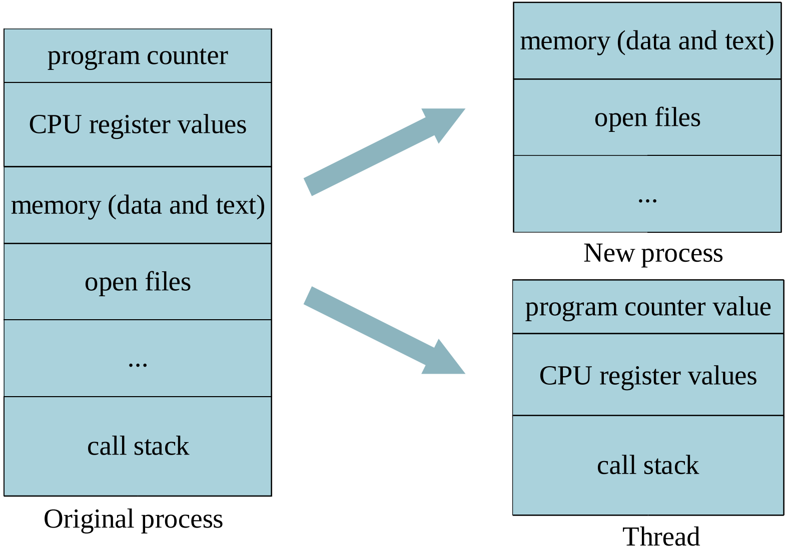
program counter
CPU register values
memory (data and text)
open files
...
call stack

Operating System Concepts

- Assume that we divide a process into two pieces
  - A resource container
  - An execution flow
- After making this division, we name the resource container a *process* and the execution flow a *thread*
  - A *thread cannot exist without a process*, thus a process is also a “container” for threads
  - A process may contain *multiple threads*
  - A process with a single thread is *equivalent to our original definition of a process*

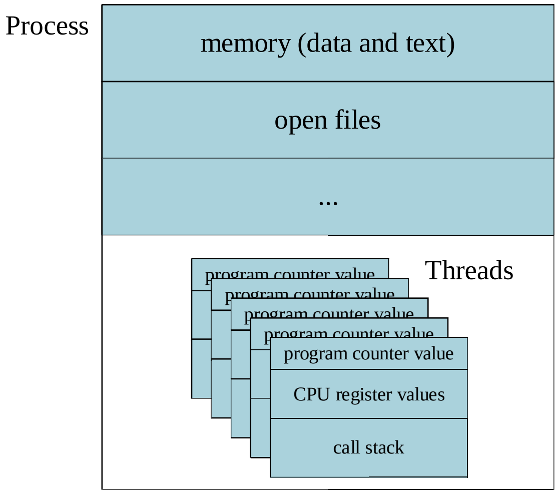
Operating System Concepts

- The new process contains resources shared by all its threads, whereas the thread contains its own private resources

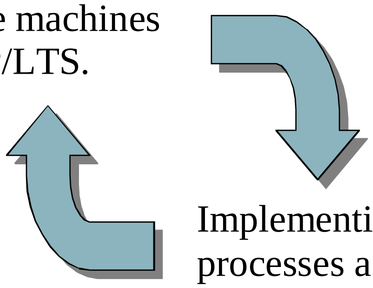


Operating System Concepts

- Process/thread relationship
  - Process contains threads
  - Threads share process resources



Relationship to FSP



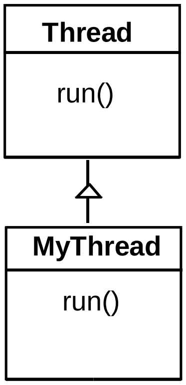
Modeling *processes* as finite state machines using FSP/LTS.

Implementing processes as *threads* in Java.

To avoid confusion, the term ***process*** is used when referring to models, and ***thread*** when referring to an implementation in Java.

Threads in Java

A **`Thread`** class manages a single sequential thread of control. Threads may be created and deleted dynamically.

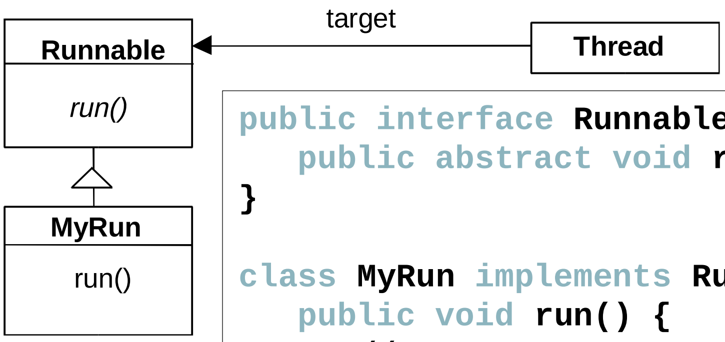


```
class MyThread extends Thread {
    public void run() {
        //...
    }
}
```

The **`Thread`** class executes instructions from its method **`run()`**. The actual code executed depends on the implementation provided for **`run()`** in a derived class.

Threads in Java

Since Java does not permit multiple inheritance, we often implement the **`run()`** method in a class not derived from **`Thread`** but from the interface **`Runnable`**.



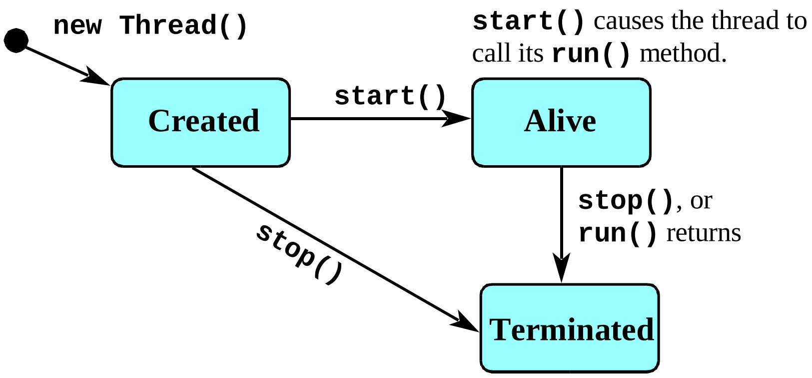
```
public interface Runnable {
   public abstract void run();
}

class MyRun implements Runnable {
   public void run() {
      //…
   }
}
```

Thread Subclass versus Runnable

- A **`Runnable`** is potentially more re-usable since other pieces of the Java class library also use the **`Runnable`** interface
- A **`Runnable`** does not "waste" inheritance
- Although it is arguable, a **`Runnable`** may promote better design
  - Separation of concerns – a thread is a flow of execution that executes some set of actions
    - This creates a **`Thread`**/**`Runnable`** distinction where the **`Thread`** is the flow of execution and the **`Runnable`** is the set of actions

Thread Life Cycle in Java

An overview of the life cycle of a thread as state transitions



`new Thread()`

`start()` causes the thread to call its `run()` method.

Created

`start()`

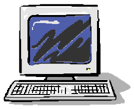
Alive

`stop()`, or `run()` returns

`stop()`

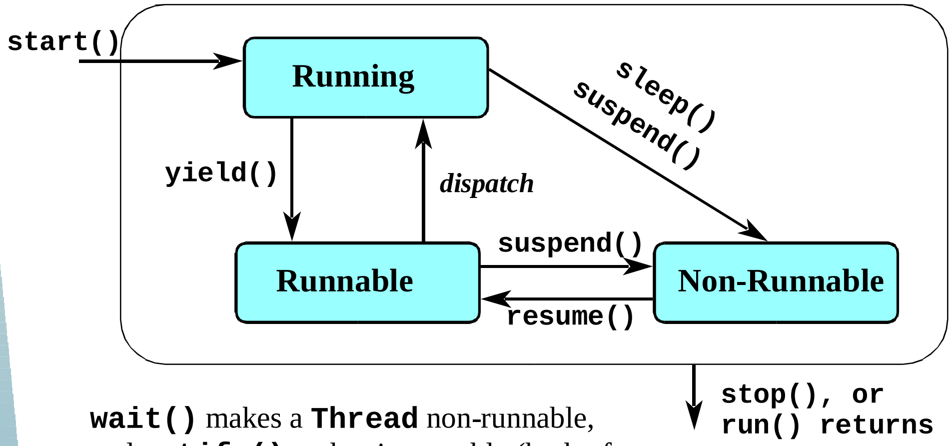
Terminated

The predicate **`isAlive()`** is used to test if a thread has been started but not terminated. Once terminated, a thread cannot be restarted.



Thread Life Cycle in Java

An “alive” thread has a number of sub-states



`start()`

Running

`sleep()`
`suspend()`

`yield()`

dispatch

Runnable

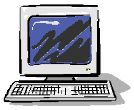
`suspend()`

Non-Runnable

`resume()`

`stop(), or run() returns`

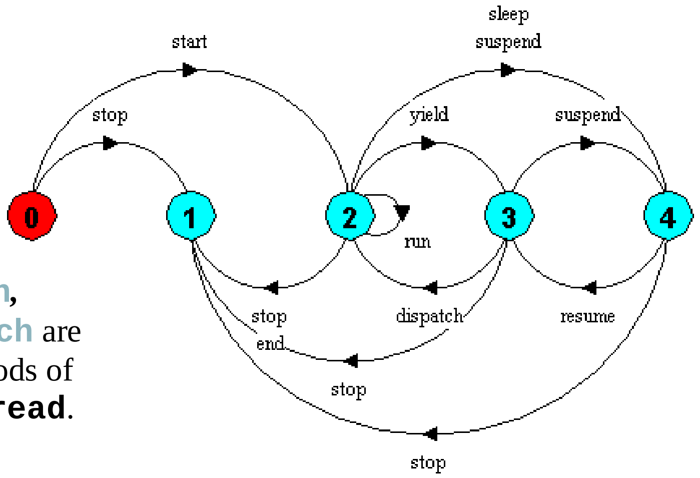
`wait()` makes a **`Thread`** non-runnable, and **`notify()`** makes it runnable (both of these methods are discussed later).



Java Thread Life Cycle in FSP

```
THREAD       = CREATED,
CREATED      = (start           ->RUNNING
               |stop            ->TERMINATED),
RUNNING      = ({suspend,sleep}->NON_RUNNABLE
               |yield           ->RUNNABLE
               |{stop,end}      ->TERMINATED
               |run             ->RUNNING),
RUNNABLE     = (suspend         ->NON_RUNNABLE
               |dispatch        ->RUNNING
               |stop            ->TERMINATED),
NON_RUNNABLE = (resume          ->RUNNABLE
               |stop            ->TERMINATED),
TERMINATED   = STOP.
```

Java Thread Life Cycle in LTS

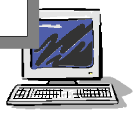


`end`, `run`, `dispatch` are not methods of class **`Thread`**.

States 0 to 4 correspond to **`CREATED`**, **`TERMINATED`**, **`RUNNING`**, **`RUNNABLE`**, and **`NON_RUNNABLE`** respectively.

Java Thread Life Cycle in FSP

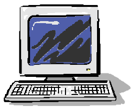
```
public class Hello implements Runnable {
  public void run() {
    System.out.println(“Hello”);
    try {
      Thread.sleep(1000);
    } catch (Exception ex) { }
  }

  public static void main(String[] argv) {
    new Thread(new Hello()).start();
    System.out.println(“World”);
  }
}
```



Countdown Timer Example in Java

```
COUNTDOWN (N=3)   = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
        (when(i>0) tick->COUNTDOWN[i-1]
        |when(i==0) beep->STOP
        |stop->STOP).
```

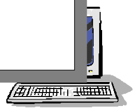
How do we implement this in Java?



Countdown Timer Example in Java

High-level implementation approach

```
public class CountDown implements Runnable {
  int i = 0;
  final static int N = 10;
  boolean stopping = false;
  AudioClip beepSound = null, tickSound = null;
  NumberCanvas display = null;

  public void start() {...}
  public void stop()  {...}
  public void run()   {...} // Runnable
  private void tick() {...}
  private void beep() {...}
}
```



Countdown Timer Example in Java

```
public void start() {
  i = N;
  stopping = false;
  new Thread(this).start();
}

public void stop() {
  stopping = true;
}

public void run() {
  while(true) {
    if (stopping) return;
    if (i>0)  { tick(); --i; }
    if (i==0) { beep(); return;}
  }
}
```

`start` action

`stop` action

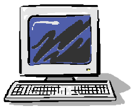
`COUNTDOWN[i]` process **recursion** as a **while** loop

`STOP`

`when(i>0) tick->CD[i-1]`

`when(i==0) beep->STOP`

`STOP` occurs when `run()` returns

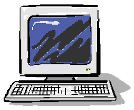


Countdown Applet

```
public class CountDownApplet extends Applet {
  CountDown counter = null;
  NumberCanvas display = null;

  public void init() {
    add(display=new NumberCanvas("CountDown"));
    display.setSize(150,100);
    counter = new CountDown(display);
  }

  public void start() {
    counter.start();
  }

  public void stop() {
    counter.stop();
  }
}
```

Multiple Threads in Java

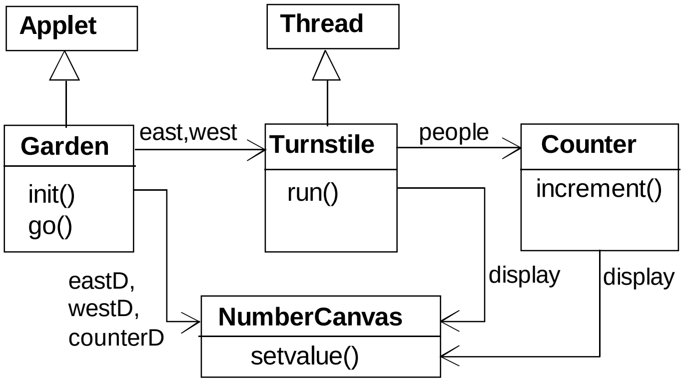
- In the previous Java thread examples, we only created one thread, Java programs can create many threads
- Actually, every Java program has multiple threads
  - A program starts with its own thread that calls **`main()`**
  - Various background threads exist, like the garbage collector
- Creating multiple threads is as simple as creating multiple **`Thread`** instances using **`new`**



Recall the Ornamental Garden

```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

VAR      = VAR[0],
VAR[u:T] = (read[u]->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go->RUN),
RUN       = (arrive->INCREMENT
            |end->TURNSTILE),
INCREMENT = (value.read[x:T]
             ->value.write[x+1]->RUN
            )+VarAlpha.

||GARDEN = (east:TURNSTILE
           ||west:TURNSTILE
           ||{east,west,display}::value:VAR)
            /{go/{east,west}.go,
              end/{east,west}.end}.
```

Ornamental Garden in Java



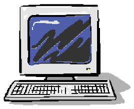
The **`Turnstile`** thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the **`increment()`** method of the counter object.

Ornamental Garden in Java

The **`Counter`** object and **`Turnstile`** threads are created by the **`go()`** method of the Garden applet:

```
private void go() {
  counter = new Counter(counterD);
  west = new Turnstile(westD,counter);
  east = new Turnstile(eastD,counter);
  west.start();
  east.start();
}
```

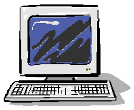
Note that **`counterD`**, **`westD`** and **`eastD`** are objects of **`NumberCanvas`** used in **`CountDown`** example.



Ornamental Garden in Java

```
class Turnstile extends Thread {
  NumberCanvas display;
  Counter people;

  Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

  public void run() {
    try{
      display.setvalue(0);
      for (int i=1;i<=Garden.MAX;i++){
        Thread.sleep(500); // sleep
        display.setvalue(i);
        people.increment();
      }
    } catch (InterruptedException e) {}
  }
}
```

The **`run()`** method exits and the thread terminates after **`Garden.MAX`** visitors have entered.



Ornamental Garden in Java

```
class Counter {
  int value=0;
  NumberCanvas display;

  Counter(NumberCanvas n) {
    display=n;
    display.setvalue(value);
  }

  void increment() {
    int temp = value; // read
    Simulate.HWinterrupt();
    value=temp+1; // write
    display.setvalue(value);
  }
}
```

Hardware interrupts can occur at **arbitrary** times.

The **`counter`** simulates a hardware interrupt during an **`increment()`**, between reading and writing to the shared counter **`value`**. Interrupt randomly calls **`Thread.yield()`** to force a thread switch.

Ornamental Garden in Java

Remember that we found a bug in our FSP implementation of the Ornamental Garden; the same bug exists in our Java implementation

This bug results from the threads interfering with each other...how do we solve this problem in Java?

Avoiding Interference in Programs

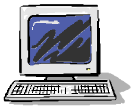
Naïve Solution

This will not work, even if setting “lock” is atomic, due to arbitrary instruction interleaving -- a thread may jump ahead of current one

```
// global space
Value v = new Value(0);
boolean lock = false;
```

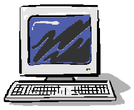
```
// Thread 1
while (lock) { } // wait
lock = true; // lock
x = v.read();
v.write(x + 1);
System.out.println(v.read());
lock = false;
…
```

```
// Thread 2
while (lock) { }  // wait
lock = true; // lock
x = v.read();
v.write(x + 1);
System.out.println(v.read());
lock = false;
…
```



Avoiding Interference in Programs

- What is the programmatic solution?
  - It is not generally possible using conventional programming language constructs
  - Must have special support for synchronization
    - Either language support or library support
    - Actually, solution is provided through hardware support
      - Atomic “test-and-set” instruction (or similar)
    - This “special” support provides a means to guarantee that some minimal instruction can be done atomically
    - From this minimal atomic instruction, it is possible to build higher level synchronization concepts

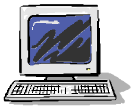


Mutual Exclusion in Java

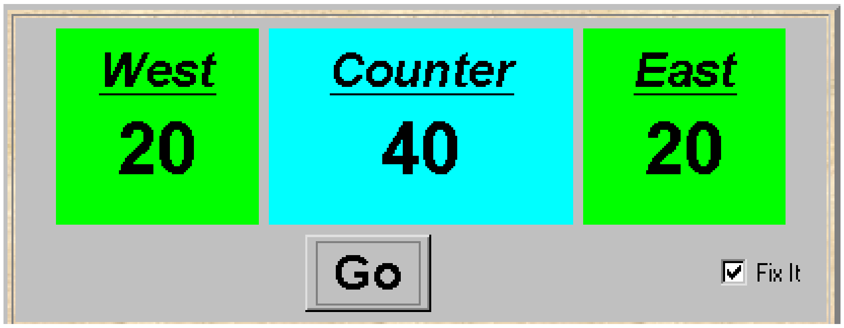
Java associates a ***lock*** with each object. Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword **`synchronized`**.

We correct **`COUNTER`** class by deriving a class from it and making the increment method `synchronized`

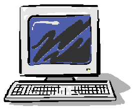
```
class SynchronizedCounter extends Counter {

  SynchronizedCounter(NumberCanvas n)
     {super(n);}

   synchronized void increment() {
        super.increment();
   }
}
```



Mutual Exclusion in Java



The Java compiler inserts code to acquire the lock before executing the body of the `synchronized` method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.

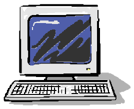


Synchronized Methods are Recursive

Once a thread has access to a `synchronized` method, it can enter the method again and again (since it already has the lock), for example:

```
public synchronized void increment(int n) {
   if (n > 0) {
      value++;
      increment(n - 1);
   }
}
```

Conceptually, each entry into the `synchronized` method increments a lock counter and each exit decrements the lock counter. When the conceptual lock counter reaches zero then the lock is once again free.



Java Synchronized Statement

Access to an object may also be made mutually exclusive by using the `synchronized` statement

```
synchronized (object) { statements }
```

A less elegant way to correct the garden example would be to modify the `Turnstile.run()` method

```
synchronized (counter) {counter.increment();}
```