CS 455: Introduction To Distributed Systems [Threads]

Threads: Reap What You Sow

Care to use more than a core?
Let threads come to the fore

Maximize your utilizations they will
Spurn them at your throughputs' peril

Shrideep Pallickara
Computer Science
Colorado State University

1

Topics covered in this lecture

- Threads
  - Creation and Management
  - Lifecycle

2

Many hands make light work. John Heywood (1546)

THREADS

3

Why should you care about threads?

- CPU clock rates have tapered off
  - Days when you could count on "free" speed-up are long gone
- Manufacturers have transitioned to multicore processors
  - Each with multiple hardware execution pipelines
- A single threaded process can utilize only one of these execution pipelines
  - Reduced throughput
- But more importantly, threads are awesome!

4

What we will look at

- Threads and its relation to processes
- Thread lifecycle
- Contrasting approaches to writing threads
- Data synchronization and visibility
  - Avoiding race conditions
- Thread safety
- Sharing objects and confinement
- Locking strategies
- Writing thread-safe classes

What are threads?

- Miniprocesses or lightweight processes
- Why would anyone want to have a *kind of process* **within** a process?

The main reason for using threads

- In many applications *multiple activities* are going on at once
  - Some of these may block from time to time
- Decompose application into multiple sequential threads
  - Running **concurrently**

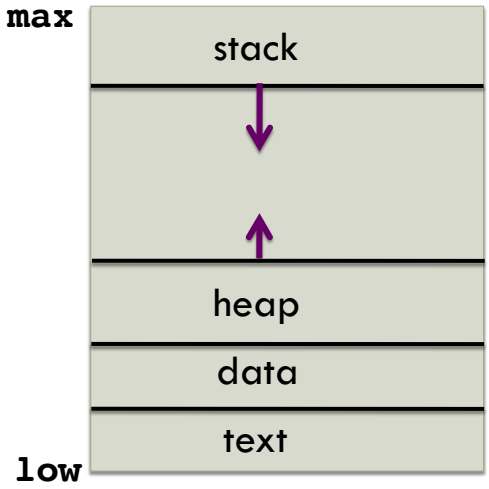
Isn't this precisely the argument for processes?

- Yes, *but* there is a new dimension …
- Threads have the ability to **share the address space** (and <u>all</u> of its data) among themselves
- For several applications
  - Processes (with their *separate* address spaces) don't work

Contrasting items unique & shared across threads

Per process items {Shared by threads with a process}	Per thread items {Items unique to a thread}
Address space	Program Counter
Global variables	Registers
Open files	Stack
Child Processes	State
Pending alarms	
Signals and signal handlers	
Accounting Information	

A process in memory



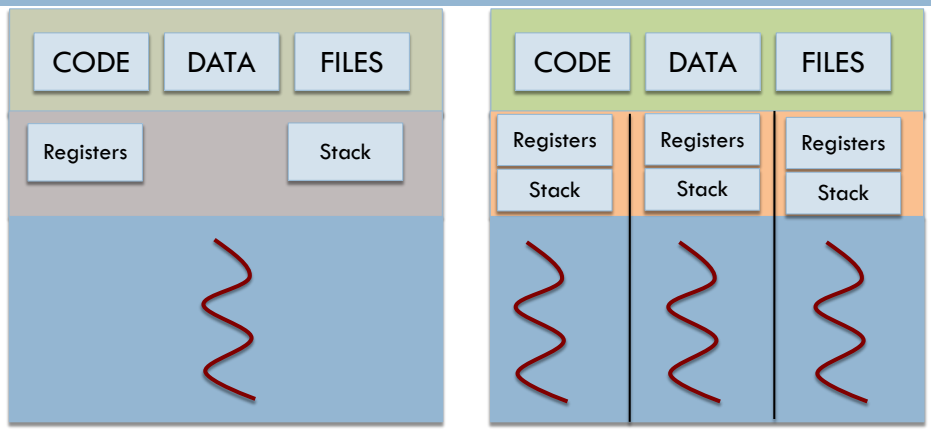
{Function parameters, return addresses, and local variables}

{Memory allocated dynamically during runtime}

{Global variables}

{Program code}

A process with multiple threads of control can perform more than 1 task at a time



Traditional Heavy weight process **Process with multiple threads**

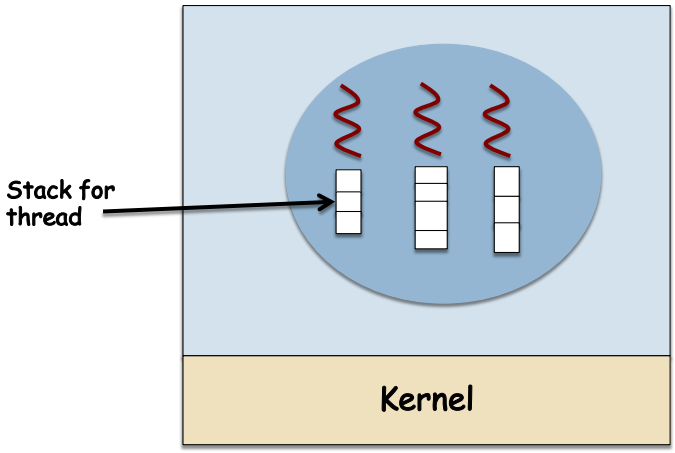
Why each thread needs its own stack [1/2]

- Stack contains one **frame** for each procedure *called but not returned from*
- Frame contains
  - Local variables
  - Procedure's return address

Why each thread needs its own stack [2/2]

- Procedure **X** calls procedure **Y**, **Y** then calls **Z**
  - When **Z** *is executing*?
    - Frames for **X**, **Y** and **Z** will be on the stack
- Each thread calls *different* procedures
  - So has a *different execution* history

Each thread has its own stack



Almost impossible to write programs in Java without threads

- We use multiple threads without even realizing it

Blocking I/O: Reading data from a socket

- Program blocks *until data is available* to satisfy the `read()` method
- Problems:
  - Data may not be available
  - Data may be delayed (*in transit*)
  - The other endpoint sends data sporadically
- If program **block**s when it tries to read from socket?
  - Unable to do anything else *until data is actually available*

Three techniques to handle such such situations

- **I/O multiplexing**
  - Take all input sources and use system call, `select()`, to notify data availability on any of them
- **Polling**
  - Test if data is available from a particular source
    - System call such as `poll()` is used
    - In JDK 1.4, `available()` on the `FilterInputStream`
- **Signals**
  - File descriptor representing signal is set
  - *Asynchronous* signal delivered to program when data is available
  - Java does not support this

Writing to a socket may also block

- If there is a **backlog** getting data onto the network
  - Does not happen in fast LAN settings
  - But if it's over the Internet? Possible.
- So, often handling TCP connections requires both a sender and receiver thread

Writing programs that do I/O in Java?

- Use multiple threads
  - Handle traditional, blocking I/O
- Use the NIO library
- Or both

We are trained to think linearly

- Often don't see *concurrent paths* our programs may take
- No reason why processes that we conventionally think of as single-threaded should remain so

Thread Abstraction

- A **thread** is a *single execution sequence* that represents a separately schedulable task
  - **Single execution sequence**
    - Each thread executes sequence of instructions – assignments, conditionals, loops, procedures, etc. – just as the sequential programming model
  - **Separately schedulable task**
    - The OS can run, suspend, or resume a thread at any time

THREAD CREATION & MANAGEMENT

Computing the factorial of a number

```
public class Factorial {

   public static void main(String[] args)  {
      int n = Integer.parseInt(args[0]);

      int factorial = 1;
      while (n>1) {
         factorial *=n;
          n--;
      }
      System.out.println(factorial);
   }
}
```


Behind the scenes …

- Instructions are executed as machine-level assembly instructions
  - Each logical step requires many machine instructions to execute
- Applications are executed as a series of instructions
  - The *execution path* of these instructions?
    - **Thread**

Every program has at least one thread

- Thread executes the body of the application
  - In Java, this is called the **main thread**
    - Begins executing statements starting with the first statement of the `main()` method
- In Java every program has more than 1 thread
  - E.g. threads that do *garbage collection*, *compile bytecodes* into machine-level instructions, etc.
  - Programs are highly threaded
    - You may add additional application threads to this

Let's add another task to our program

- Say, computing the square-root of a number
- What if we wrote these as separate threads?
  - JVM has two distinct lists of instructions to execute
- Threads can be thought of *as tasks that we execute at roughly the same time*
- But in that case, why not just write multiple applications?

Threads that run within the same application process

- **Share the memory space** of the process
  - Information sharing is seamless
- Two diverse applications within the same machine may not communicate so well
  - For e.g. mail client and music application

In a multi-process environment data is separated by default

- This is fine for **dissimilar programs**
- Not OK for certain types of programs; e.g. a network server sends stock quotes to clients
  - Discrete task: Sending quote to client
    - Could be done in a separate thread
  - Data sent to the clients is the same
    - *No point having a separate server for each client* and …
    - *Replicating data* held by the network server

Threads and sharing

- Threads within a process can access and share any object on the **heap**
  - Each thread has space for its own local variables (stack)
- A thread is a discrete task that operates on data **shared** with other threads

Thread Abstraction

- A **thread** is a *single execution sequence* that represents a separately schedulable task
  - **Single execution sequence**
    - Each thread executes sequence of instructions – assignments, conditionals, loops, procedures, etc. – just as the sequential programming model
  - **Separately schedulable task**
    - The OS can run, suspend, or resume a thread at any time

THREAD CREATION

31

Thread creation

- Using the **`Thread`** class
- Using the **`Runnable`** interface

32

The Thread class

```
package java.lang;

public class Thread implements Runnable {
   public Thread();
   public Thread(Runnable target);
   public Thread(ThreadGroup group, Runnable target);
   public Thread(String name);
   public Thread(ThreadGroup group, String name);
   public Thread(Runnable target, String name);
   public Thread(ThreadGroup group, Runnable target,
                 String name);
   public Thread(ThreadGroup group, Runnable target,
                 String name, long stackSize);

   public void start();
   public void run();

}
```


Threads require 4 pieces of information

- Thread **name**
  - Default is `Thread-N`; N is a unique number
- **Runnable target**
  - *List of instructions* that the thread executes
  - Default: `run()` method of the thread itself
- Thread **group**
  - A thread is assigned to the thread group of the thread that calls the constructor
- **Stack size**
  - Store temporary variables during method execution
  - Platform-dependent: range of legal values, optimal value, etc.

A simple thread

```
public class RandomGen extends Thread {
   private Random random;
   private int nextNumber;
   public RandomGen() {random = new Random();}

   public void run() {
     for (;;) {
       nextNumber = random.nextInt();
       try {

       } catch (InterruptedException ie) {
            ... return;
       }
     }
   }
}
```


About the code snippet

- Extends the `Thread` class
- Actual instructions we want to execute is in the `run()` method
  - Standard method of the `Thread` class
    - Place where `Thread` begins execution

Contrasting the `run()` and `main()` methods

- `main()` method
  - This is where the *first thread starts executing*
  - The **main thread**
- The `run()` method
  - *Subsequent threads* start executing with this method

THREAD LIFECYCLE

Lifecycle of a thread

- Creation
- Starting
- Terminating
- Pausing, suspending, and resuming

Thread: Methods that impact the thread's lifecycle

```
public class Thread implements Runnable {
   public void start();
   public void run();
   public void stop();
   public void resume();
   public void suspend();
   public static void sleep(long millis);
   public boolean isAlive();
   public void interrupt();
   public boolean isInterrupted();
   public static boolean interrupted();
   public void join();
}
```

Deprecated, do not use

Thread creation

- Threads are represented by instances of the `Thread` class
- When you extend the `Thread` class?
  - Your instances are also `Threads`
- We looked at the 4 constructor arguments in the `Thread` class

Starting a thread [1/2]

- Thread exists once it's been constructed
  - But it is *not executing* … it's in a **waiting** state
- In the waiting state, other threads can *interact* with the existing **thread object**
  - Object state may be changed by other threads
    - Via method invocations

Starting a thread [2/2]

- When we're ready for a thread to begin executing code
  - Call the **`start()`** method
  - `start()` performs internal house-keeping and *then calls* the **`run()`** method
- When the `start()` method returns?
  - **Two threads** are executing in parallel
    1. The original thread which just returned from calling **`start()`**
    2. The newly started thread that is executing its **`run()`** method

After a thread's `start()` method is called

- The new thread is said to be **alive**
- The `isAlive()` method tells you about the state
  - `true`: Thread has been started and *is executing* its `run()` method
  - `false`: Thread may *not be started* yet or may be *terminated*

The contents of this slide-set are based on the following references

- *Computer Networks: A Systems Approach. Larry Peterson and Bruce Davie. 4th edition. Morgan Kaufmann. ISBN: 978-0-12-370548-8.* Chapters [4, 5]
- *https://en.wikipedia.org/wiki/Maximum_segment_size*
- *Java Threads. Scott Oaks and Henry Wong. . 3rd Edition. O'Reilly Press. ISBN: 0-596-00782-5/978-0-596-00782-9.* [Chapters 1, 2]
- *Andrew S Tanenbaum. Modern Operating Systems. 3rd Edition, 2007. Prentice Hall. ISBN: 0136006639/978-0136006633. [Chapter 2]*