Process Creation in Java

• Note: Java really doesn’t implement processes per se, but provides a clean interface to spawn processes to run system programs outside of the JVM

• “Process” class in Java
  – `Runtime.getRuntime().exec(""`) method will return a process object (interface) that executes what is in the string argument
  – all process objects support the following methods:
    • `waitFor()`
    • `exitValue()`
    • `destroy()`

```java
try {
    Process child1 = Runtime.getRuntime().exec("/bin/ls");
    InputStream in = child1.getInputStream();
    // echo output of 'ls'
    child1.waitFor();
    System.out.println("Child exited with "+ child1.exitValue());
} catch(IOException e) {
    System.err.println(e);
}
```

New Subtopic: Threading

• Up till now, we’ve viewed processes as
  – unit of resource ownership
  – unit of scheduling (dispatching)

• Consider a Server Process:

![Diagram of Server Process]

“BOTTLENECK”: The Server Process has a single “thread of execution”
Threading Introduction

• Observation: multiple threads within a process?
  – each has its own PC, stack, etc. (register context)
  – schedulable

• Distinguishing Threads & Processes
  – PROCESS: memory space
  – PROCESS: “owns” resources
  – PROCESS: “big PCB”
  – THREAD: shared memory with other threads
  – THREAD: per-thread static storage for local variables
  – THREAD: execution state (READY, RUNNING, etc) & execution stack

• Threads are “lightweight processes”

• Resource-ownership --> Processes

• Scheduling --> Threads

Threading Models

1 Process
1 Thread

Multiple Processes
1 Thread/Process

1 Process
Multiple Threads

Multiple Processes
Multiple Threads/Process
Benefits of Threading

Performance-Oriented Benefits:

Communication-Oriented Benefits:

Design-Oriented Benefits:

Thread Implementation

• User-Level Threading (ULT)
  – Application process does all work in managing threads
  – Application process maintains bookkeeping

• Advantages
  – thread switching doesn’t need mode-switching overhead
  – specific scheduling of threads is possible
  – easy to add as application-level utility library to any OS

• Disadvantages
  – OS calls within ULTs block entire process
  – Application process cannot schedule threads on different CPUs
Thread Implementation

- Kernel-Level Threading (KLT)
  - OS Kernel does all scheduling/management of threads
  - thread switching is privileged instruction
- Advantages
  - OS Kernel can take advantage of multiple CPUs
  - OS Kernel can block threads without blocking processes
- Disadvantages
  - switching overhead
- NEAT IDEA: kernel threads as interrupts

Multithreading Models

- When OS supports BOTH user and kernel threads, there are a few possibilities for implementing multithreading:
  - Many-to-One
    - all user threads map to one kernel thread
    - pros/cons?
  - One-to-One
    - each user thread maps to its own kernel thread
    - pros/cons?
  - Many-to-Many
    - user threads “multiplexed” to <= # of kernel threads
    - pros/cons?
Threading in Java (2 approaches)

- Define new class that extends Thread class
  - default technique assumed by book, and by us

```java
class Worker1 extends Thread {
    public void run() {
        System.out.println("I'm a worker thread");
    }
}
```

- Define class that implements Runnable interface
  - "interface:" a promise by a class to offer support for a collection of methods.
  - this is preferred when we need to make an object of a class that is already an extension of some parent execute inside a thread. (e.g. Applet)

```java
public class First {
    public static void main(String args[]) {
        Worker1 runner = new Worker1();
        runner.start();
        System.out.println("I'm a main thread");
    }
}
```

Another View of Multithreading & PC

```java
public class Printer extends Thread {
    private int extra;
    public Printer(int x) {
        extra = x;
    }
    public void run() {
        int y = extra + 1;
        System.out.println("Total is "+ y);
    }
}
```

```java
public class Test {
    public static void main() {
        (new Printer(1)).start();
        (new Printer(2)).start();
    }
}
```

```
MOV @x,@extra
JMP (exit)
LOD r1,@extra
INC r1,1
MOV r1,@y
CALL (println)
JMP (exit)
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Java Thread Management Methods

- `suspend()`: suspends execution of thread
- `sleep()`: makes thread stop for a specified duration
- `resume()`: resumes a specified suspended thread
- `stop()`: stops a thread -- cannot be resumed or started
- `yield()`: allows thread to give JVM to some other arbitrary runnable thread.

Suspend, Resume, and Stop are deprecated (see Deadlock)

Thread States in Java

- **NEW**: object for thread is in middle of being created
- **RUNNABLE**: thread is eligible to be run by JVM
  - no distinction between the actual thread being executed and those waiting to be executed (but eligible)
- **BLOCKED**: thread has performed a blocking statement such as I/O request, or sleep() or suspend()
- **DEAD**: thread's run() method has terminated or stop() method has been invoked.