Contents

1. Classes and Objects
2. Inheritance
3. Interfaces
4. Exceptions and Error Handling
5. Intro to Concurrency
6. Concurrency in Java
7. Graphics and Animation
8. Applets
Chapter 5: Intro To Concurrency

After this chapter you will be able to:

- Compare threads versus processes
- Understand the race condition problem
- Describe three approaches to avoiding race conditions
Terms

• **Concurrent** tasks are independent and can proceed without waiting for one another

• Tasks that execute in **parallel** execute simultaneously
  – multi-processors
  – distributed systems
History

• Concurrency first existing in operating systems
  – multi-user support (timesharing)
  – IBM System/360, 1965
  – Unix, 1969

• Motivation is to keep CPU busy
Architecture

- **Scheduler** decides which task to run based on various methods
  - priorities
  - round-robin

![Ready Queue Diagram]
Operating System Process

• A **Process** is an executing program and it’s environment (heavy weight)

  – PC (program counter)
  – run-time stack
  – open file info
  – user info
  – shell environment variables
Programming with Concurrency

- Previously programmers created concurrency by making external calls to the OS.

- Later concurrency was integrated into programming languages
  - advantage: portability
Threads

• A thread is a (light weight) concurrent task
  – run-time stack
  – PC

• Threads exist in both programming languages
  – languages: Ada83, Concurrent Pascal, Java

• And operating systems:
  – Solaris, POSIX, NT
Motivation for Concurrency

- Useful for programming applications with inherent concurrency
  - animation
  - image processing
  - web server

- User interaction

- Performance: multi-processors and distributed systems
The Problem

- **Race condition** exists when tasks executing the same code may leave memory in an invalid state

\[ x := x + 1 \]

- Two operations: read and write x
The solution

• Only one task is allowed inside a critical section

• Three different ways to program a critical section
  – semaphores (1965) due to Dijkstra
  – message passing (channel communication)
  – monitors (1970’s) due to British computer scientist Hoare
Semaphore Critical Section

- The semaphore can be in two states (on/off)
- $P$ operation turns semaphore on
- $V$ operation turns semaphore off
- $P$ and $V$ are atomic operations

```plaintext
P()
x := x + 1
V()
```
Message Passing Critical Section

• Let one concurrent task (middle man) execute the code

\[ x := x + 1 \]

• All other tasks must *talk* to the middle man to request an update
Monitor Critical Section

- Monitor is a set of procedures
- Only one concurrent task can be inside the monitor

```plaintext
Begin Monitor
    procedure addOne ()
    begin
        x := x + 1
    end
    
    procedure decOne ()
    begin
        x := x - 1
    end
End Monitor
```
Java

• In Java each object has a single monitor

• **Question:** Java does not have semaphores or message passing. What do we do if we need them?
Answer

• **No problem!** Can use monitors to implement semaphores or message passing

• **Fact:** Can use any one of semaphores, monitors, or message passing to implement the other two.
Summary

• Concurrent programming is hard because these issues must be handled properly:
  – race conditions
  – deadlock: happens when all threads are waiting and no progress is being made
  – starvation: occurs when program logic allows a thread to wait indefinitely by not queueing waiting threads fairly