Concurrency: Definitions

- **Concurrency**: two or more execution contexts that may be *active* at the same time

- **Parallelism**: concurrent execution contexts that are *actually running* at the same time
  - This distinguishes between multiprocessor systems where multiple CPUs are working at the same time, and a single-CPU system with pre-emptive multitasking
  - Programming techniques are the same for any system with concurrency, regardless of whether the concurrency is truly parallel or not

- **Thread**: specific term for the aforementioned execution contexts
- **Process**: an operating system-level execution context, which may correspond to one or more language-level threads

More Definitions

- **Heavyweight process**: a process with its own address space
- **Lightweight process**: a process which shares an address space with other process
  - Note that these are in an operating systems context; the mapping between a language abstraction for a task and the corresponding operating system resources is a language implementation issue

- **Task**: a well-defined unit of work that a program needs to perform, typically in one thread
  - A single thread can perform multiple tasks
  - Multiple threads can share a “bag of tasks” — no strict mapping between a task and a thread; it’s just a matter of who is available to do more work

- Different systems may have different definitions!
First Cut: Coroutines

- Coroutines have some of the feel of concurrency, but are semantically quite distinct
- Coroutines are multiple execution contexts — typically packaged as subroutines — across which a program can switch
  - Not concurrent because only one of them can be considered “active” at any given time
  - The trick is that coroutines can transfer control to each other, picking up where they left off
  - Think of coroutines as subroutines that transfer instead of return

- Implemented in Simula and Modula-2
- Typical applications of coroutines:
  - Implementation of iterators
  - Discrete event simulation

On to Concurrency

- Concurrent programs remove the explicit notion of a coroutine transfer — you program under the assumption that multiple tasks are truly occurring at the same time

- More definitions:
  - Race condition: a situation where multiple tasks affect the same resources (variables, memory, files) such that a program’s result will change depending on which task gets to the resource first
    - Race conditions aren’t always bad, but it’s good to know when they are happening
  - Deadlock: a situation where multiple tasks need to wait for the same resources to be available, thus resulting in none of these tasks being able to proceed
Key Concurrency Issues

- The issues of race conditions and deadlocks motivates the two primary issues in concurrent programming:
  - *Communication*: how do multiple threads send/receive information to/from each other?
  - *Synchronization*: when dependencies across threads occur (e.g. race conditions), how does a program specify the relative order in which threads should do their work?

- Two forms of communication in use today:
  - *Shared memory*: multiple threads can “see” the same variables/memory/data, and read/write as needed
  - *Message passing*: multiple threads have completely isolated state, and must explicitly communicate state to each other (through “messages”)

Concurrency and Architecture

- While concurrent programming and computer system architecture are distinct topics, they share a certain “symbiosis”
  - Some concurrent programming implementation issues are influenced significantly by the target architecture
  - Theoretically, you can still implement any concurrent programming approach on any architecture, but in reality some approaches and architectures are just “made for each other”

- Architectures of note:
  - Vector architectures: hardware that can operate on huge amounts of data at the same time (pioneered by Cray, evident in PCs as “Pentium MMX” and “PowerPC Velocity Engine” buzzwords
  - The Flynn classification of multiprocessors: single/multiple instruction streams, single/multiple data stores (SISD, SIMD, MISD, MIMD)
SISD

- This is the classic Von Neumann architecture

SIMD

- Each processor has its own data set.
MISD

- No MISD machines have been constructed thus far.

MIMD

- Current model for parallel architectures — note how it is little different from a network of computers
Centralized MIMD = Shared Memory

Distributed MIMD = Message Passing
Implementing Concurrency in a Language

• Language vs. library

• Language approach: provides compiler support, better integration with other language concepts such as type checking, scoping, exceptions

• Library approach: allows addition of concurrency control to existing languages; must be in the context of language constructs
  – Posix pthreads library
  – Java Thread class (there is internal JVM support, but the presentation is as a class with methods — no concurrency-specific syntax)
  – Remote procedure call: wrapping messages inside stub subroutines

Creating Threads

• co-begin: multiple parallel statements define threads — SR, Algol 68, Occam

• parallel loops: parallel execution of loop iterations (one thread per loop iteration) — SR, Occam, some Fortran dialects

• launch-at-elaboration: subroutine-like syntax executes a thread upon declaration — SR, Ada, others

• fork/join: explicit thread “launching” at any time; join waits for a previously forked thread — SR, Ada, Modula-3, Java

• implicit receipt: automatic thread creation in response to a message from another thread — SR, RPC-capable systems

• early reply: created thread “returns” an initial result, but continues execution — SR, Java (separation of Thread creation from execution)
Shared Memory

- Threads can independently read/write a common resource
- Watch out for cached memory! — implementation issue

- Synchronization is a key issue: when multiple threads depend on the same object, when is the “right” time to access that object?
  - Mutual exclusion: define a critical section in the code and ensure that only one thread is running that section at a time
    - Used in Java at the language level — synchronized keyword
  - Condition synchronization: threads wait for a condition to be true (e.g., a variable gets a value) before proceeding
    - Common for I/O or network-related activity

Message Passing

- Threads must explicitly communicate with each other

- Language support
  - Abstraction for messages
  - Abstractions for send and receive points (e.g., ports)

- Explicit communication
  - Resource management, error handling, return parameters
  - Synchronization/blocking semantics
  - Buffering (particularly when receiving long messages)

- Remote procedure calls
  - Tries to make message passing transparent — “looks like a subroutine”