History

- **1971: Edsger Dijkstra proposes and solves the dining philosophers problem**
  - Circular table, five people, five plates, and five chopsticks
  - Each philospher thinks until he becomes hungry.
  - Need two chopsticks to eat
  - When a philospher gets two chopsticks, he eats for a while. He then puts the chopsticks down and resumes thinking.
  - They never speak to each other, so there is possibility of deadlock
    - Each person holds the left chopstick and waits for the person to the right to put his down
  - Invented a construct known as the semaphore

Semaphores

- **In a nutshell:**

```
class Semaphore {
        private s;
        public Semaphore(int v) {
                s = v;
        }
        public v() {
                s = s + 1; // Must be atomic
        }
        public p() {
                // Must be atomic once s > 0 is detected.
                wait until s > 0, then s = s – 1;
        }
}
```

- **V and P**
  - V: From Dutch "verhoog" (increase)
    - Sometimes called "signal"
  - P: From Dutch "probeer te verlagen" (try and decrease)
    - Sometimes called "wait"
- **Solving the dining philosophers problem**
  - Initialize a semaphore to 4 (= n – 1)
  - Each philosopher does a P ("wait") before starting to eat.
  - Each philosopher does a V ("signal") after finishing eating.
  - Semaphores can also be used for other "classical" synchronization problems:
    - The sleeping barber
    - Database readers and writers
- **Binary semaphores**
  - Value can only be 0 or 1.
  - Provides mutual exclusion.

- Implicit in Java:

```
Object mutex = new Object();
…
synchronized (mutex) {
      …
}
```

Language features for concurrency

- **Semaphores**
- **Creating and managing processes or threads**
- **Monitors**
  - Data structure guaranteed to be accessed in mutual exclusion.
  - Cooperation must be programmed explicitly.
  - Concurrent Pascal: keywords delay and continue
  - Can be implemented with binary semaphores.
    - So why have them in the langauge?
      - Semaphores are low level, hard to use, and impossible to check mechanically.
- **Rendezvous**
  - Ada
  - Synchronous message passing
    - Receiver must be willing to receive before sender can continue
  - Very different from semaphores and monitors:
    - No memory locations shared between the processes.
    - The processes might as well be on different machines.

Two paradigms: Shared memory and message passing

- **Can you implement one with the other?**
  - Can you implement message passing using shared memory and semaphores? Yes.
  - Can you implement shared memory using message passing? Yes.
- **Recurring theme in programming languages**
  - Different languages and language features may be technically equivalent.
  - Yet very different in practical use, from a software engineering perspective.

Concurrency in Java

- **See assigned reading**

Erlang

- **Properties**
  - A concurrent programming language.
  - Functional
  - Eager evaluation
  - Single assignment: Can bind a variable to a value only once
  - Dynamic typing

- **History**
  - Developed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications.
  - Open source since 1998
- **Primitives for concurrency**
  - Processes
    - Very light weight
      - Not OS processes or OS threads
      - Can have millions of processes
      - Overhead can be as small as 300 bytes
  - Share-nothing asynchronous message passing
    - Each process has a "mailbox"
    - Messages from other processes end up in the mailbox
    - Use the "receive" primitive to retrieve from the mailbox messages that match a certain pattern.
  - Distribution is transparent
    - Processes can be on other computers, it makes no difference.
  - Error handling
    - When a process crasches, it exits, and a message is sent to the controlling process.