University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Message Passing

Advanced Operating Systems

Tutorial 6

Review of Lectured Material

- Message passing systems
  - Limitations of threads and lock-based concurrency
  - Multicore memory models; composition of lock-based code
  - Concepts of message passing systems
    - Interaction models; communication and the type system; naming communications
    - Message handling; immutability; linear types; use of an exchange heap
    - Pattern matching and state machines
    - Error handling; let-it-crash philosophy; supervision hierarchies; case study
  - Scala+Akka, Rust, Singularity, and Erlang as examples

Key Points

- Understand the concepts of actor-based message passing programming languages and systems
- Reflect on the suitability of message passing as a concurrency primitive for future systems
  - Advantages and disadvantages compared to lock-based concurrency with shared mutable state

Discussion

DOI:10.1145/1810891.1810910

The same component isolation that made it effective for large distributed telecom systems makes it effective for multicore CPUs and networked applications.

BY JOE ARMSTRONG

Erlang

ERLANG IS A concurrent programming language designed for programming fault-tolerant distributed systems at Ericsson and has been (since 2000) freely available subject to an open-source license. More recently, we've seen renewed interest in Erlang, as the Erlang way of programming maps naturally to multicore computers. In it the notion of a process is fundamental, with processes created and managed by the Erlang runtime system, not by the underlying operating system. The individual processes, which are programmed in a simple dynamically typed functional programming language, do not share memory and exchange data through message passing, simplifying the programming of multicore computers.

Erlang[2] is used for programming fault-tolerant, distributed, real-time applications. What differentiates it from most other languages is that it's a concurrent programming language; concurrency belongs to the language, not to the operating system. Its programs are collections of parallel processes cooperating to solve a particular problem that can be created quickly and have only limited memory overhead; programmers can create large numbers of Erlang processes yet ignore any preconceived ideas they might have about limiting the number of processes in their solutions.

All Erlang processes are isolated from one another and in principle are "thread safe." When Erlang applications are deployed on multicore computers, the individual Erlang processes are spread over the cores, and programmers do not have to worry about the details. The isolated processes share no data, and polymorphic messages can be sent between processes. In supporting strong isolation between processes and polymorphism, Erlang could be viewed as extremely object-oriented though without the usual mechanisms associated with traditional OO languages.

Erlang has no mutexes, and processes cannot share memory.[a] Even within a process, data is immutable. The sequential Erlang subset that executes within an individual process is a dynamically typed functional programming language with immutable state.[b] Moreover, instead of classes, methods, and inheritance, Erlang has modules that contain functions, as well as higher-order functions. It also includes processes, sophisticated error handling, code-replacement mechanisms, and a large set of libraries.

Here, I outline the key design criteria behind the language, showing how they are reflected in the language itself, as well as in programming language technology used since 1985.

Shared Nothing

The Erlang story began in mid-1985 when I was a new employee at the Ericsson Computer Science Lab in Stock-

[a] The shared memory is hidden from the programmer. Practically all application programmers never use primitives that manipulate shared memory; the primitives are intended for writing special system processes and not normally exposed to the programmer.

[b] This is not strictly true; processes can mutate local data, though such mutation is discouraged and rarely necessary.

- J. Armstrong, "Erlang", Communications of the ACM, 53(9), Sept. 2010, DOI: 10.1145/1810891.1810910
- Does the programming model make sense?
  - Purely functional with immutable data
  - Software isolated processes (i.e., actors) with message passing and no shared mutable state
  - Syntax and type system
  - Relation to Singularity?
  - Independent of Erlang, is the message passing approach a good alternative to the threads-and-locks model of concurrency?
  - Are problems with race conditions, deadlock, etc., solved?
- Does the reliability model ("let it crash") make sense?
  - Move error handling to a separate process
  - Replication and fault tolerance at the process level
  - Do you believe in independent failures of software processes?
  - Is dynamic typing required to enable upgrade of running systems?