Robert Virding
Principle Language Expert
Erlang Solutions Ltd.
What IS the BEAM?

- A virtual machine to run Erlang
- Interfaces to the “outside” world
  - Ports and NIFs
- A bunch of built-in “useful” functions
  - BIFs
Properties of the Erlang system

- Lightweight, massive concurrency
- Asynchronous communication
- Process isolation
- Error handling
- Continuous evolution of the system
- Soft real-time
Properties of the Erlang language

- Immutable data
- Pattern matching
- Functional language
So to run Erlang the BEAM needs to support all this.

AT LEAST
We will look at

• Schedulers
• Processes
• Memory management
• Message passing
• Multi-core
• ...

© 1999-2012 Erlang Solutions Ltd.
Schedulers

- Semi-autonomous BEAM VM
- One per VM thread
  - By default one VM thread per core
- Contains its own run-queue
  - Run-queue contains things to be done
- Run as separately as possible
  - Reduce nasties like locks/synchronisation
Schedulers: balancing

- Once every period (20–40k reductions) a new master scheduler is chosen
  - Basically first to reach that count
- Master balances/optimises workloads on schedulers
- Schedules changes on other schedulers run-queues
Schedulers: scheduling processes

- Each scheduler has its own run-queue
- Processes suspend when waiting for messages
  - Not a busy wait
- Suspended processes become runnable when a message arrives
  - Put on the run-queue
- Running processes will not scheduler by
  - Suspending waiting for a message
  - Re-scheduled after 2000 reductions
Memory

4 separate memory areas/types

• Process heaps
• ETS tables
• Atom table
• Large binary space
Memory: Atom table

- All atoms are interned in a global atom table
  - FAST equality comparison
    - NEVER need to use integers as tags for speed

- Atoms are NEVER deleted
  - Create with caution
  - Avoid programs which rampantly creates atoms in an uncontrolled fashion

- Fixed size table
  - System crashes when full
Memory: large binary space

- Large binaries (> 64 bytes) stored in separate area
- Fast message passing as only pointer sent
  - Can save a lot of memory as well

- Can be long delay before being reclaimed by GC
  - All processes which have “seen” the binary must first do a GC
  - Can grow and crash system
Memory: **ETS tables**

- Separate from process heaps
- Not implicitly garbage collected
- But memory reclaimed when table/element deleted
- All access by elements being copied to/from process heaps
  - match/select allows more complex selection without copying
- Can store LARGE amounts of data
Memory: **Process heaps**

- Each process has a separate heap
- All process data local to process
- Can set minimum process heap size
  - Per process and for whole system

- Sending messages means copying data

- This NOT required by Erlang which just specifies process isolation
Isn’t all this data copying terribly inefficient?

Well, yes. Sort of. Maybe.

BUT ...
Process heaps: Garbage collection

Having separate process heaps has some important benefits

- Allows us to collect each process separately
  - Processes small so GC pauses not noticeable
- Garbage collection becomes more efficient
- Garbage collector becomes simpler
- Needs no synchronisation
  - This is a BIG WIN™
  - And it gets bigger the more cores you have
Process heaps: Garbage collection

- Copying collector
- Generational collector
  - 2 spaces, new and old
  - New data is kept in new space for a number of collections before being passed to the old heap
  - Not much data unnecessarily ends up in old heap
  - Eventually old heap must be collected as well
Process heaps: **Tuning**

- Minimum process heap size (`min_heap_size`)
  - Process starts bigger, never gets smaller
  - Be selective or pay the price in memory

- Full sweep in garbage collector (`fullsweep_after`)
  - Black magic, just test and see
  - Forces collections more often, reclaim memory faster
    - Uses less memory, reclams large binaries faster
    - Less efficient collection
Async thread pool

- File i/o is done in the scheduler thread
  - It can take time
  - Blocks the scheduler while waiting

- Using the async threads moves i/o operations out of the scheduler thread
  - Scheduler thread now no longer waits for file i/o

- File i/o will automatically use them if created
- Linked-in port drivers can use them if they exist
- Inet driver never uses them
How to crash the BEAM

- Fill the atom table
- Overflow binary space
- Uncontrolled process heap growth
  - Infinite recursion
  - VERY long message queues
  - A lot of data
- Errors in NIFs and linked-in port drivers!
  - These can really get you
Thank you!

robert.virding@erlang-solutions.com

@rvirding
Lock example
Lock example
Lock example

- Spawns processes which creates timestamps checks if there in order and sends the result to its parent

- Uses `erlang:now/0`