Debugging & Observability

The BEAM for Developers

Erik Stenman

Module Overview

- **Module 5: Debugging & Observability**
- Process Introspection & Monitoring
- **Lab: Comprehensive System Debugging**
- **Checklist**
- **Conclusion**

Module 5: Debugging & Observability

Goal

- Master BEAM's introspection and debugging capabilities
- Learn to diagnose performance issues and bottlenecks in production
- Build observability into your systems from the ground up
- Understand when and how to use different debugging approaches

Outcome

- Confidently debug process-related issues in production
- Use built-in and third-party tools for system observability
- Implement effective monitoring and alerting strategies
- Troubleshoot memory leaks, scheduler issues, and message bottlenecks

Overview

This module teaches you to see inside the BEAM. Not just what's happening, but why. You'll learn to trace execution, monitor resource usage, and diagnose issues that only show up under load.

Process Introspection & Monitoring

The Process Information Goldmine

Every BEAM process carries a wealth of diagnostic information. The trick is knowing what to look for and when.

```
Pid = self(),
erlang:process_info(Pid).

erlang:process_info(Pid, memory).
erlang:process_info(Pid, message_queue_len).
```

```
erlang:process_info(Pid, reductions).
erlang:process_info(Pid, current_function).
```

Key metrics to monitor:

Memory usage:

- `memory`: Total memory in bytes (heap + stack + mailbox)
- `heap_size`: Current heap size in words
- `total_heap_size`: Total heap including old heap

Key metrics to monitor:

Message handling:

- `message_queue_len`: Messages waiting in mailbox
- `messages`: Actual message list

Possibly expensive.)

Key metrics to monitor:

Execution state:

- `reductions`: CPU units consumed (resets on context switch)
- `current_function`: What function is currently executing
- `status`: running, waiting, suspended, etc.

Process Discovery and Enumeration

```elixir
# Find all processes
all_processes = :erlang.processes()
IO.puts("Total processes: #{length(all_processes)}")
```

Find processes by memory usage

```
process_and_memory() ->
    [{P, erlang:process_info(P, memory)}
    || P <- erlang:processes()].
top_mem(N) ->
  lists:sublist(lists:reverse(lists:keysort(2,
process_and_memory())),N).
```

The Observer GUI

`:observer` is BEAM's built-in graphical debugging tool. Essential for development and safe for production monitoring.

```
# Start observer
:observer.start()
```

Key tabs and their uses:

System Tab:

- Memory usage (total, processes, atoms, binaries, code, ETS)
- CPU utilization per scheduler
- I/O statistics (file descriptors, sockets)
- System limits (process count, port count, atom count)

Key tabs and their uses:

Load Charts:

- Real-time graphs of scheduler utilization
- Memory allocation patterns over time
- Process/port counts trending

Key tabs and their uses:

Memory Allocators:

- Detailed breakdown of memory allocation
- Helps identify memory leaks in specific allocators
- Useful for tuning memory allocator flags

Key tabs and their uses:

Processes Tab:

- Sortable list of all processes with key metrics
- Message queue lengths, memory usage, reductions
- Function call counts and current function
- Direct process introspection and message sending

Key tabs and their uses:

Ports Tab:

- File handles, sockets, and other external resources
- Helps track resource leaks

System Statistics and Metrics Collection

Building system metrics collectors.

See code modules:

- system_metrics.erl - Comprehensive BEAM metrics collection
- metrics_collector.erl - GenServer for periodic metrics gathering

The Recon Library - Production-Ready Diagnostics

Recon by Fred Hebert is the go-to library for production BEAM debugging.

See: production_monitor.erl and code/README.md for installation, usage examples, and Recon quick reference.

Activity:

- Start `:observer` and explore each tab while running different workloads
- Implement a custom metrics dashboard using `SystemMetrics`
- Use Recon to identify the most resource-intensive processes in your system

Understanding BEAM Tracing

BEAM's tracing system lets you observe process execution in real-time. It's powerful but can be overwhelming. The key is knowing what to trace and how to filter the noise.

How Tracing Works: Breakpoint Hooks

BEAM implements tracing through breakpoint mechanisms in the bytecode interpreter and JIT.

Function call tracing: When you enable tracing on a function, BEAM replaces the function's entry point with a breakpoint instruction. Each call triggers the breakpoint, which sends a trace message to the tracer process, executes the original function code, and optionally sends return value traces.

Why some functions can't be traced: Built-in functions (BIFs) like `length/1`, `hd/1`, `tl/1` and Native Implemented Functions (NIFs) execute as native C code outside the BEAM bytecode interpreter. BEAM cannot insert breakpoints into native code.

Performance cost: Each traced function call adds overhead through message sends and pattern matching. Production tracing requires rate limits and careful function selection.

Trace Types:

- **Function calls**: When functions are called, what arguments, return values
- **Process events**: spawn, exit, link, unlink, register, unregister
- **Message passing**: send, receive (with optional message content)
- **Scheduling events**: in, out, exiting
- **Garbage collection**: GC start/stop, heap sizes

Basic tracing with `trace`:

```
Tracer = spawn(fun F() ->
    receive M -> io:format("~p~n", [M]), F() end
end).

Session = trace:session_create(my_session, Tracer, []).
trace:process(Session, self(), true, [call]).
trace:function(Session, {lists, seq, 2}, [], []).

lists:seq(1, 5).
lists:seq(10, 15).

trace:session_destroy(Session).
```

Output:

```
{trace,<0.80.0>,call,{lists,seq,[1,5]}}
```

Speaker notes

BIFs like `length/1` cannot be traced. Use `lists`, `string`, `maps`.

The dbg Module

```
dbg:tracer().
dbg:p(self(), [call]).
dbg:tpl(calendar, day_of_the_week, 1, []).

calendar:day_of_the_week({2025, 11, 3}).

dbg:stop_clear().
```

Output:

```
(<0.80.0>) call calendar:day_of_the_week({2
```

Redbug - User-Friendly Tracing

```
redbug:start("lists:sort/1", [{time, 5000}, {msgs, 10}]).

lists:sort([3,1,2]).
```

Output:

```
16:20:15 <0.80.0>({erl_eval,do_apply,6})
lists:sort([3,1,2])
```

Other Profiling Tools

- **fprof** - Function call profiler with time measurements
- **tprof** - Time profiler (OTP 27+)
- **eprof** - Lightweight profiler for process time
- **lcnt** - Lock contention profiler
- **recon** - Production-safe diagnostic toolkit

Activity:

- Use `:dbg` to trace a GenServer's message handling
- Create a production monitoring script using `SafeTracer`
- Profile a computational function and identify bottlenecks with `:fprof`

Production Debugging Strategies

Common Production Issues and Diagnosis

Production debugging is different. You can't restart things freely. You can't enable verbose logging. You need surgical precision and minimal system impact.

Common issue patterns:

Memory Leaks:

- Process heap growing without bounds
- Binary reference leaks (sub-binaries holding large originals)
- ETS tables growing indefinitely
- Atom table exhaustion from dynamic atom generation

Common issue patterns:

Process Bottlenecks:

- Single GenServer handling too much load
- Message queue backlog causing timeout cascades
- Selective receive scanning large mailboxes
- Process registry becoming a bottleneck

Common issue patterns:

Scheduler Issues:

- Unbalanced load across schedulers
- Long-running processes blocking schedulers
- Dirty schedulers overwhelmed by I/O

Common issue patterns:

Let's build diagnostic tools for each:

```elixir
defmodule ProductionDiagnostics do
  @moduledoc """
  Safe diagnostic tools for production environments
  """

  def diagnose_system() do
    %{
      memory_issues: diagnose_memory_issues(),
      process_bottlenecks: diagnose_process_bottlenecks(),
      scheduler_issues: diagnose_scheduler_issues(),
      system_health: get_system_health_summary()
    }
  end
```

When production systems are failing, you need a playbook:

```elixir
defmodule EmergencyResponse do
  @moduledoc """
  Emergency procedures for production incidents
  """

  def emergency_diagnosis() do
    IO.puts("🚨 EMERGENCY SYSTEM DIAGNOSIS")
    IO.puts("=" * 50)

    # Quick system overview
    memory = :erlang.memory()
    total_mb = memory[:total] / (1024 * 1024)
```

```
    proc_count = length(:erlang.processes())
```

The best debugging happens before problems occur:

```
defmodule ResilientMonitoring do
  @moduledoc """
  Comprehensive monitoring system with circuit breakers
and alerting
  """

  defmodule CircuitBreaker do
    defstruct [:name, :failure_threshold, :reset_timeout,
:failures, :state, :last_failure]

    def new(name, failure_threshold \\ 5, reset_timeout \\
60_000) do
```

```
    %__MODULE__{
      name: name,
```

- Set up `ProductionMonitoring` in a test environment and trigger various alerts
- Practice emergency procedures with `EmergencyResponse` module
- Build a custom alert system using `ResilientMonitoring` concepts

Lab: Comprehensive System Debugging

Lab 1: Building a Chat Server Monitoring System

Let's create a realistic chat server and monitor it under load:

```
defmodule ChatServer do
  use GenServer
  require Logger

  defmodule State do
    defstruct [:rooms, :users, :message_count,
:start_time]
  end
```

```elixir
  # Client API
  def start_link(opts \\ []) do
    GenServer.start_link(__MODULE__, opts, name:
__MODULE__)
  end
```

Lab 2: Memory Leak Detection and Analysis

Create a memory leak scenario and detect it:

```
defmodule MemoryLeakDemo do
  use GenServer

  # Simulates various types of memory leaks

  def start_link() do
    GenServer.start_link(__MODULE__, [], name: __MODULE__)
  end

  def start_binary_leak() do
```

```elixir
  GenServer.cast(__MODULE__, :start_binary_leak)
end

def start_process_leak() do
```

Lab 3: Production Incident Simulation

Simulate and respond to production incidents:

```
defmodule IncidentSimulator do
  def run_incident_response_drill() do
    IO.puts("🚨 STARTING PRODUCTION INCIDENT DRILL")
    IO.puts("=" * 60)

    # Setup monitoring
    {:ok, _} =
ResilientMonitoring.start_link(check_interval: 5_000)
    {:ok, _} =
ProductionMonitoring.start_link(check_interval: 5_000)
```

```
    # Create normal baseline load
    create_baseline_load()
    time.sleep(10.000)
```

- How do the tracing techniques from Session 3 help with the debugging strategies in Session 4?
- What patterns emerge when combining process monitoring (Session 1) with system-wide observability (Session 2)?
- How would you adapt these debugging tools for a distributed system across multiple nodes?

Interactive Exercises

Open the LiveBook:

`module-5-exercises.livemd`

20 minutes of hands-on practice with:

- Process introspection and monitoring techniques
- System-wide observability with :observer and recon
- Tracing and profiling production systems
- Building custom diagnostic tools

Speaker notes

The exercises are in an interactive LiveBook notebook. Students should open `module-5-exercises.livemd` in LiveBook to work through the hands-on exercises.

Each exercise has runnable code cells and discussion questions.

Checklist

Conclusion

What's Next?

Module 6: Patterns & Anti-Patterns—learn to recognize common design patterns that leverage BEAM's strengths and avoid the anti-patterns that fight against them. We'll explore supervision strategies, process communication patterns, and system architecture principles that create resilient, maintainable applications.

Question 1

You notice a process with 10,000 messages in its mailbox. What can happen?

- The VM will automatically throttle senders
- The process will use more memory to store queued messages
- Messages will be dropped after reaching a limit
- The process may become slow due to selective receive scanning

Question 2

What does the `reductions` metric indicate?

- Number of messages sent
- Heap size in bytes
- Amount of CPU work performed
- Number of process crashes

Question 3

A process is using 500MB of memory. Which tools help identify what it's storing?

- :erlang.process_info(pid, :memory)
- :recon.proc_count(:memory, 10)
- :observer.start() (GUI tool)
- :dbg.tracer()

Question 4

Scheduler utilization shows: Scheduler 1 at 95%, Scheduler 2 at 10%. What does this indicate?

- The system is perfectly balanced
- Workload is not evenly distributed across schedulers
- Scheduler 2 has failed
- Some processes may be pinned to specific schedulers

Question 5

When should you use `:dbg` for tracing in production?

- Always, it has no performance impact
- Only during scheduled maintenance windows
- Carefully, with specific filters to limit trace volume
- Never, use recon_trace instead for safety

Question 6

A binary memory leak occurs when:

- Processes send too many messages
- Sub-binaries keep references to large parent binaries
- ETS tables grow without bounds
- Schedulers become imbalanced

Question 7

What happens when you call :erlang.garbage_collect/1 on a process?

- Forces immediate garbage collection on that process
- Terminates the process and frees its memory
- Clears the process mailbox
- Resets the process reduction counter

Question 8

How can you safely identify the top CPU-consuming processes in production?

- Use :erlang.processes() and check each one manually
- Use :recon.proc_count(:reductions, 10)
- Enable full :dbg tracing on all processes
- Monitor :erlang.statistics(:scheduler_wall_time)

Question 9

A process dies during `:erlang.process_info/2`.
What does the function return?

- An error tuple {:error, :dead}
- Crashes your shell process
- Returns `undefined`
- Waits indefinitely for the process to restart

Question 10

Circuit breakers in monitoring systems help by:

- Preventing cascading failures when checks fail repeatedly
- Physically disconnecting hardware in emergencies
- Temporarily disabling failing health checks to allow recovery
- Killing problematic processes automatically

Activity

Content type: activity

Activity

Content type: activity

Activity

Content type: activity

Navigation

← Previous Module: Scheduling & Concurrency | Back to Course Structure | Next Module: BEAM Patterns & Anti-Patterns →