Data Types & Messaging

The BEAM for Developers

Erik Stenman

Module Overview

- Module 2:
  Data Types & Messaging
- Process Memory Layout
- Message Passing
- Measuring and Optimizing Message Costs
- Checklist
- Conclusion
- Hands-On Exercises

Module 2: Data Types & Messaging

Goal

- Master BEAM's data representation: atoms, tuples, lists, binaries, maps
- Understand message copying semantics and costs
- Learn how data types affect memory usage and performance
- Build intuition for efficient message design

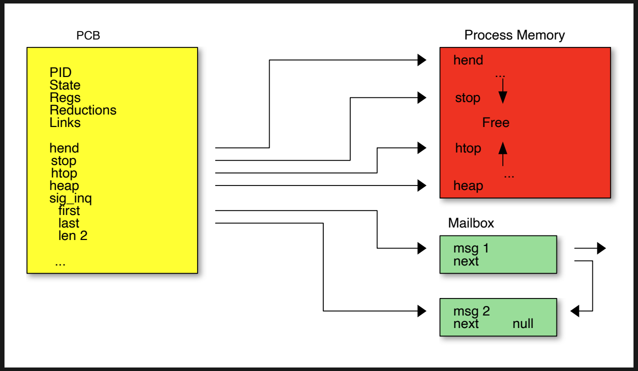
Outcome

- Choose the right data structure for your use case
- Design efficient message payloads
- Understand binary sharing and sub-binary pitfalls
- Predict and measure message costs across processes

Process Memory Layout

Processes are just memory

Each gnome carries a backpack (heap + stack) and has a mailbox.



Speaker notes

Key points about this layout:

PCB (Process Control Block): Holds process metadata and state. PID, current state (runnable, waiting, etc.), reduction count for scheduling, links to other processes, and a pointer to the mailbox.

Stack and Heap: Share the same memory block but grow toward each other. Stack grows down from high addresses (return addresses, local variables). Heap grows up from low addresses (your data structures). When they meet, garbage collection triggers.

Mailbox: Lives outside the main memory area. Messages queue here until the process receives them. When you receive a message, it gets copied from the mailbox into the heap.

This layout explains why processes are:

- Lightweight (small initial memory)
- Isolated (no shared heap between processes)
- Predictable (GC only affects one process at a time)

Now let's see what actually goes in each area.

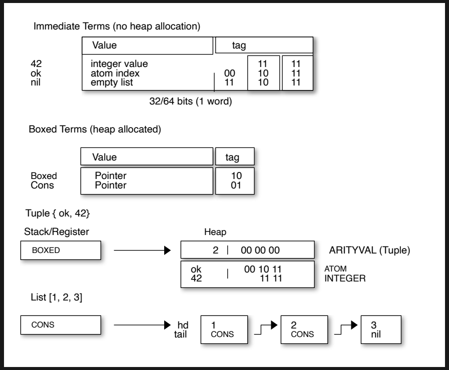
Immediates vs Boxed Terms

- Immediates fit in one word.
- Boxed terms need heap space.

```elixir
# Immediate (1 word, no heap)
:ok
42

# Boxed (allocated on heap)
{:ok, 42}
[1, 2, 3]
```

Immediates vs Boxed Terms (Ilustrated)



Speaker notes

Everything in BEAM is a term. Every term has a tag (type), a value (data), and a location (heap, stack, or literal pool).

Immediate terms fit in one machine word (64 bits on modern systems):

- Small integers (up to 60 bits, ±576 quadrillion)
- Atoms (reference to global atom table)
- PIDs, ports, refs (local ones)
- nil (empty list)

Boxed terms require heap allocation:

- Big integers (beyond 60 bits)
- Floats (always boxed, even though they fit in 64 bits)
- Tuples
- Lists (cons cells)
- Maps
- Binaries

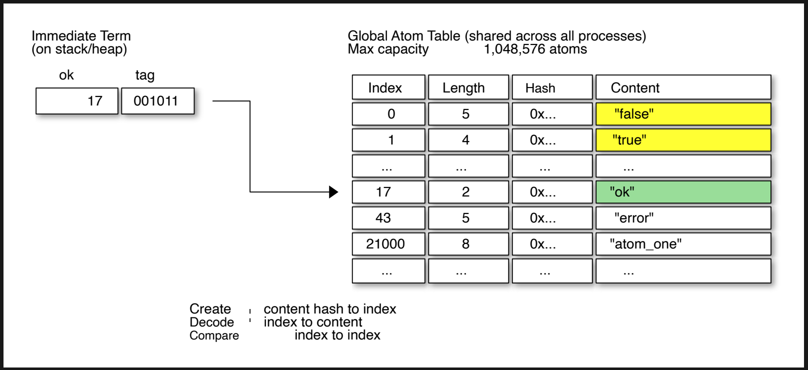
This distinction drives performance: immediates are cheap to copy (just the word), boxed terms must be walked and copied recursively. When you send {:ok, 42}, the atom :ok is immediate but the tuple wrapper requires heap allocation.

Understanding this helps predict memory usage and message costs.

Atoms: Global and Permanent

- Never garbage collected.
- Limited to ~1 million.

```elixir
:ok   # Safe
:error   # Safe
String.to_atom(user_input)   # Dangerous!
```



Global Atom Table (shared across all processes)
Max capacity 1,048,576 atoms

Index	Length	Hash	Content
0	5	0x...	"false"
1	4	0x...	"true"
...	...	...	...
17	2	0x...	"ok"
43	5	0x...	"error"
21000	8	0x...	"atom_one"
...	...	...	...

Create	content hash to index
Decode	index to content
Compare	index to index

Speaker notes

Atoms are global constants stored in a VM-wide table. Once created, they never disappear until the VM shuts down.

Max 1,048,576 atoms by default (configurable with +t flag). Exceeding this crashes the VM.

Use atoms for:

- Fixed vocabularies (:ok, :error, :pending)
- Module names
- Function names
- Message tags

If you generate modules and functions you might also run out of atom space, but probably other problems before that.

Never create atoms from user input:

```
# Bad: creates unlimited atoms
String.to_atom(params["status"])

Good: validate against known set
```

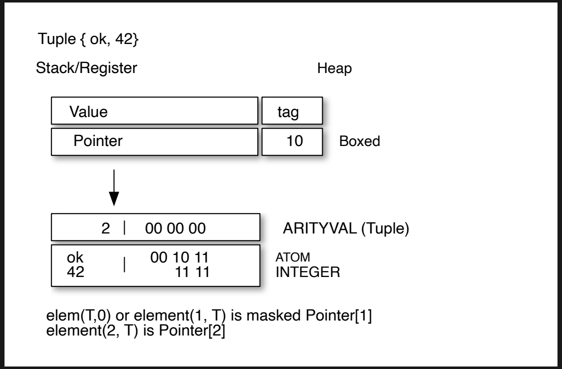
```
case params["status"] do "active" -> :active "pending" -> :pending _ -> :error end
```

Atoms are immediate terms (fit in one word), so they're cheap to pass around. The danger is exhausting the atom table, not performance.

Tuples: Contiguous and Fast

- Fixed size.
- O(1) element access.
- Copied on update.

```
user = {:user, "Alice", 42}
elem(user, 1)  # "Alice" - instant lookup
```



Speaker notes

Tuples are fixed-size, contiguous arrays in memory. Layout:

```
[TUPLE_HEADER | size | element1 | element2 | ... | elementN]
```

Size: header + N elements (N+1 words total).

Access is O(1) because it's just pointer arithmetic. Update is O(N) because the entire tuple must be copied.

Use tuples for:

- Records (before structs existed, still used internally)
- Fixed structures ({:ok, result}, {:error, reason})
- Return values from functions
- Small fixed collections

Avoid tuples for:

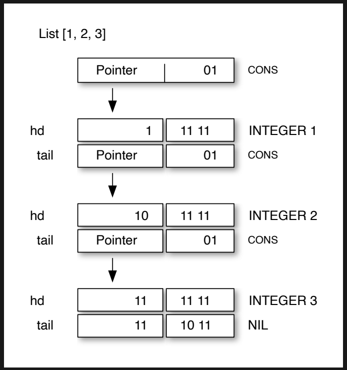
- Variable-length data (use lists)
- Frequent updates (use maps)
- Large collections (copying cost grows)

When you send a tuple in a message, the whole tuple gets copied to the receiver's heap.

Lists: Linked Cells

- Sequential access.
- Build by prepending.
- Each cell is 2 words.

```
[1 | [2 | [3 | []]]]  # How lists really work
[new_head | existing_tail]  # O(1) prepend
```



Speaker notes

Lists are linked cons cells. Each cell (one list element) is 2 words: head pointer and tail pointer.

```
[1, 2, 3] in memory: [CONS | head -> 1 | tail] -> [CONS | head -> 2 | tail] -> [CONS | head -> 3 | tail] -> NIL
```

Size: 2 words per element. [1,2,3] costs 6 words.

Access patterns:

- Head: O(1) - just dereference
- Nth element: O(N) - must walk the list
- Prepend: O(1) - create new cell pointing to old list
- Append: O(N) - must copy entire list

This is why you build lists backwards:

```
# Good: O(N)
Enum.reduce(1..1000, [], fn x, acc -> [x | acc] end)

Bad: O(N²)
```

```
Enum.reduce(1..1000, [], fn x, acc -> acc ++ [x] end)
```

When you send a list in a message, every cons cell gets copied. A 1000-element list copies 2000 words.

Use lists for:

- Sequential processing
- Building incrementally (prepending)
- Pattern matching [head | tail]

Binaries: Small vs Large

- < 64 bytes: on heap.
- ≥ 64 bytes: reference-counted off-heap.

```elixir
"hello"  # heap binary (5 bytes)
:crypto.strong_rand_bytes(1024)  # refc binary
```

Speaker notes

Binaries have two storage modes:

Heap binaries (<64 bytes):

- Stored directly in process heap
- Copied when sent as messages
- Fast for small data
- Example: "short string", <<1,2,3>>

Refc binaries (≥64 bytes):

- Payload lives in shared off-heap memory
- Process heap holds small ProcBin wrapper (6-7 words)
- Reference counted across processes
- Shared when sent as messages (only ProcBin copied)
- Example: file contents, large JSON payloads

This is why large binaries are efficient for messaging:

```
big_data = File.read!("10MB.json")  # refc binary
send(worker1, big_data)  # only ~40 bytes copied
send(worker2, big_data)  # only ~40 bytes copied
# All three processes share the same 10MB payload
```

Threshold: 64 bytes. At 63 bytes, the binary lives on heap. At 64 bytes, it moves off-heap.

Binaries are the most efficient way to pass large data between processes.

Maps: Small vs Large

- ≤ 32 keys: flatmap.
- ≥ 33 keys: HAMT.
- Both immutable.

```
small = %{a: 1, b: 2}  # flatmap
large = Map.new(1..33, &{&1, &1})  # HAMT
```

Speaker notes

Maps have two internal representations:

Flatmap (≤32 keys):

- Keys stored in sorted tuple
- Values in array
- Updates copy the map
- Overhead: ~5 words + keys + values

HAMT (≥33 keys):

- Hash Array Mapped Trie
- Near O(1) lookups
- Structural sharing on updates
- Higher overhead (~53 words)

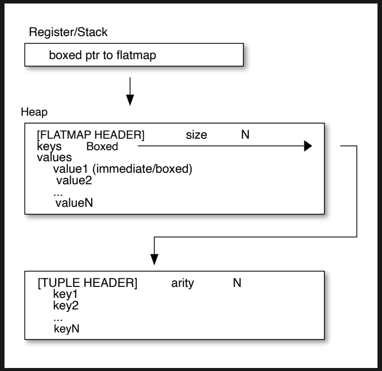
The transition happens at 33 keys. You can measure:

```
:erts_debug.size(Map.new(1..32, &{&1, &1}))  # flatmap size
:erts_debug.size(Map.new(1..33, &{&1, &1}))  # HAMT size (bigger jump)
```

Map Memory Layout

Flatmap structure:

```
Map: [FLATMAP][size][keys_ptr][value1][value2]...[valueN] Keys: [TUPLE]
```

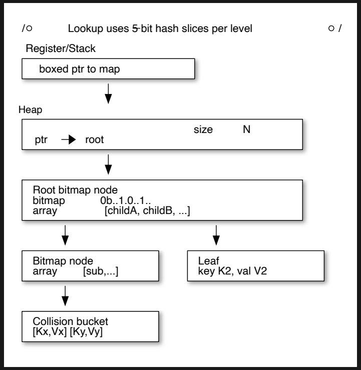


Speaker notes

The flatmap layout shows how small maps store their data. The map header contains a size field and a pointer to the keys tuple. Values are stored inline in the map structure after the header. The keys are stored separately in a sorted tuple for efficient lookup.

HAMT

- HAMT (Hash Array Mapped Trie) share structure when maps are updated
- Better performance for large maps and frequent updates.



Speaker notes

Key points

Small maps start as flatmap. Large maps promote to HAMT at a size threshold.

Internal nodes use a 32-bit bitmap. Array stores only present children.

Index = popcount(bitmap & mask(hash_slice)). Leaves store key and value.

Collision bucket handles true hash collisions.

Presenter notes (brief)

Mention 5-bit slices per level.

Explain bitmap compaction and popcount index.

Tie back to performance: O(1) expected lookups, stable updates, low memory waste.

Collision example +--------------------------------------+ | Leaf Collision Bucket | | hash prefix equal at this level | | entries: | | [Kx, Vx], [Ky, Vy], [Kz, Vz] | +--------------------------------------+

Legend +--------------------------------------+ | Bitmap node | | - Holds: 32-bit bitmap | | - Array stores only present slots | | - Child selection: popcount trick | +--------------------------------------+ | Leaf | | - Stores key and value | +--------------------------------------+ | Collision bucket | | - For rare full-hash collisions | +--------------------------------------+

Hash slice routing +---+ | Key hash split into 5-bit slices per level | | h = h0 h1 h2 h3 h4 h5 ... | | Level 0 picks h0, Level 1 picks h1, etc. | | Index within node = popcount(bitmap & mask(hN)) | +---+

When you send a map in a message, the structure gets copied. For large maps, this can be expensive. Consider using ETS or persistent_term for shared config data instead.

Message Passing

Messages are copied

- Messages are copied to the receiver's heap.
- Isolation costs.

```elixir
data = [1, 2, 3]
send(pid, {:work, data})
# List structure copied to pid's heap
```

Speaker notes

When you send a message, BEAM walks the entire term structure and copies it to the receiver's heap. This is how processes stay isolated: no shared memory.

Always copied:

- Lists (every cons cell)
- Tuples (entire structure)
- Maps (keys and values)
- Small binaries (<64 bytes)
- Nested structures (recursively)

The copying happens when the receiver processes the signal and moves the message into its mailbox. The cost is proportional to the term size.

Why copy? Safety. If sender and receiver shared the same memory, mutations would break isolation. Copying ensures the receiver gets a snapshot that won't change.

Trade-off: safety over speed. For small messages (a few hundred words), copying is negligible. For large messages, use binaries.

What Gets Copied

- Everything living in the stack or heap.
- Shared, Not copied:
    - Large binaries
    - literals
    - persistent terms

```
big = :crypto.strong_rand_bytes(1024)  # refc binary
send(pid, {:file, big})  # only ProcBin copied
```

Speaker notes

Exceptions to copying:

Large binaries (≥64 bytes):

- Payload lives in shared off-heap memory
- Process heap holds small ProcBin struct (~40 bytes)
- Reference counted across processes
- When sent, only the ProcBin gets copied
- All processes share the same payload

Literals from compiled code:

- Constants in your modules
- Stored in literal area, not copied
- Example: `@default_config %{...}` module attribute

Persistent terms (OTP 21.2+):

- Explicitly shared read-only data
- `:persistent_term.put(:config, big_map)`
- Looked up by key, never copied
- Use for large config data shared across many processes

Cross-node messages: everything gets serialized with `:erlang.term_to_binary/1`, even large binaries.

This is why binaries are the most efficient way to pass large data: send a 10MB binary to 100 processes, only ~4KB gets copied (100 ProcBins).

Message Cost Examples

- Small terms are cheap.
- Large terms or deep nesting cost more.

```elixir
:erts_debug.size({:ok, 42})  # 3 words
:erts_debug.size(Enum.to_list(1..1000))  # 2000 words
```

Speaker notes

Measure before optimizing. Use `:erts_debug.size/1` to see term size in words (1 word = 8 bytes on 64-bit):

```
# Atoms and small ints: immediate, 1 word each
:erts_debug.size(:ok)  # 0 (immediate, no heap)
:erts_debug.size(42)  # 0

Tuples: header + elements
:erts_debug.size({:ok, 42})  # 3 words
Lists: 2 words per element
:erts_debug.size([1, 2, 3])  # 6 words
Maps: overhead + keys + values
:erts_debug.size(%{a: 1, b: 2})  # ~10 words
Binaries: depends on size
```

`:erts_debug.size("hello") # heap binary, ~3 words :erts_debug.size(:crypto.strong_rand_bytes(1024)) # ProcBin, ~7 words (payload off-heap)`

Copying cost is roughly linear in term size. Sending 1000-word message takes ~1000x longer than sending 1-word message.

Design messages to be small:

- Use binaries for bulk data
- Send references or keys instead of full structs
- Tag messages with atoms, not strings
- Avoid deeply nested structures

If you're sending megabytes per message, rethink your architecture. Consider ETS, shared binaries, or streaming protocols.

Selective Receive and Mailbox Scanning

- Mailbox is scanned linearly.
- Tag messages for fast matching.

```elixir
ref = make_ref()
send(server, {:request, ref, query})
receive do
  {:response, ^ref, result} -> result
end
```

Speaker notes

When you call `receive`, BEAM scans the mailbox from oldest to newest, trying to pattern-match each message.

Linear scan means:

- First message tried first
- If no match, try second message
- Continue until match or end of mailbox
- Matched message removed, others stay

Why this matters:

```
# Bad: complex pattern deep in mailbox
receive do
  {_, _, {:nested, %{deep: value}}} -> value
end

Good: simple tag at top level
```

```
receive do {:response, data} -> data end
```

Best practice: use unique references to avoid head-of-line blocking:

```
ref = make_ref()  # Globally unique
send(server, {:request, ref, self(), query})

receive do
  {:response, ^ref, result} -> result  # Only matches this specific reply
after
```

```
  5_000 -> {:error, :timeout}
end
```

This ensures your receive matches exactly one message, even if the mailbox has thousands of unrelated messages. The ref acts as a correlation ID.

Libraries like GenServer use this pattern internally for call/reply.

```
# All of these are copied to the receiver's heap
small_tuple = {:ok, 123}
small_list = [1, 2, 3]
small_binary = "hello"                  # < 64 bytes
small_map = %{a: 1, b: 2}
```

```
send(pid, small_tuple) # Tuple header + contents copied send(pid, small_list) # All cons cells copied send(pid, small_binary) # Binary data copied send(pid,
small_map) # Map structure + values copied
```

Large binary sharing:

```
# Large binaries (≥64 bytes) use reference counting
large_binary = :crypto.strong_rand_bytes(1024)  # refc binary
send(pid, {:data, large_binary})                 # Only ProcBin copied, payload shared

# The receiving process gets a ProcBin pointing to shared payload
```

Message size impact on performance:

```
# Measure copying costs
defmodule MessageCostDemo do
```

```
  def measure_send_cost(payload) do
    parent = self()

    {time, _} = :timer.tc(fn ->
      receiver = spawn(fn ->
        receive do
          _msg -> send(parent, :done)
        end
      end)

      send(receiver, payload)
      receive do :done -> :ok end
    end)

    {payload_size, copy_time} = {:erts_debug.size(payload), time}
    IO.puts("#{payload_size} words -> #{copy_time}μs")
  end
end

# Test different payload types
MessageCostDemo.measure_send_cost({:small, 123})
MessageCostDemo.measure_send_cost(Tuple.duplicate(0, 1000))
MessageCostDemo.measure_send_cost(Enum.to_list(1..1000))
MessageCostDemo.measure_send_cost(:crypto.strong_rand_bytes(1024))
```

Selective receive and mailbox scanning:

The BEAM must scan the mailbox linearly to find matching messages. Message tags help:

```
# Efficient: direct match on message tag
receive do
  {:response, data} -> data
  {:error, reason} -> {:error, reason}
end

Less efficient: complex pattern matching deep in mailbox
```

```
receive do {_, _, {:complex, %{nested: %{pattern: value}}}} -> value end
```

Reference-tagged messages for mailbox efficiency:

```
# Use unique references to avoid head-of-line blocking
ref = make_ref()
send(server, {:request, ref, self(), query})

receive do
  {:response, ^ref, result} -> result  # Matches only this specific response
after
  5_000 -> {:error, :timeout}
end
```

Cross-node message costs:

```
# Local node: only large binaries shared
send(local_pid, {data, large_binary})    # ProcBin copied, payload shared

# Remote node: everything serialized with :erlang.term_to_binary/1
send({pid, :remote@node}, {data, large_binary})  # Entire payload sent over network
```

Activity: Measure message copying impact

```
defmodule CopyBenchmark do
  def run() do
    # Test payloads
    payloads = [
      {:small_atom, :ok},
      {:small_tuple, {1, 2, 3}},
      {:large_tuple, Tuple.duplicate(0, 1000)},
      {:small_list, [1, 2, 3]},
```

```elixir
      {:large_list, Enum.to_list(1..1000)},
      {:small_binary, "hello world"},
      {:large_binary, :crypto.strong_rand_bytes(1024)},
      {:nested_map, %{users: %{active: Enum.to_list(1..100)}}}
    ]

    for {name, payload} <- payloads do
      term_size = :erts_debug.size(payload)

      # Measure message send + receive time
      {time, _} = :timer.tc(fn ->
        parent = self()
        spawn(fn ->
          receive do _msg -> send(parent, :done) end
        end) |> send(payload)
        receive do :done -> :ok end
      end)

      IO.puts("#{name}: #{term_size} words, #{time}μs")
    end
  end
end

CopyBenchmark.run()
```

Key BEAM internals

- Message copying uses process heap space - affects GC timing
- Large binaries bypass copying through reference counting
- Mailbox scanning is linear - message order and tagging matter
- Cross-node messaging serializes everything, even large binaries

Activity:

```elixir
# Compare message costs
small_list = Enum.to_list(1..10)
big_list = Enum.to_list(1..10_000)
small_binary = "hello"
big_binary = :crypto.strong_rand_bytes(1024)

# Measure and send each, observe memory changes
```

Signals and Message Queues

Speaker notes

Building on the basics: Now that you understand what gets copied vs shared, let's explore how BEAM actually implements message passing under the hood and the modern optimizations available.

- All inter-process communication uses **signals**.
- Messages are one kind of signal.
- When you use `send/2` (`!`), here's what actually happens:

1. Signal enqueueing
2. Message processing
3. Receive scanning

Ordering guarantee

- Messages from sender A to receiver B arrive in the order sent.
- Messages from different senders can interleave.

```
# Process A sends: msg1, msg2, msg3 -> Process C
# Process B sends: msgX, msgY -> Process C
# Process C might receive: msg1, msgX, msg2, msgY, msg3
```

On-Heap vs Off-Heap Message Queues

You control where queued message data lives:

- on_heap (default for most processes)
- off_heap

on_heap

- Messages sit on the process heap
- GC must walk the entire message queue
- Simple, works well for low-volume processes

off_heap

- Messages sit outside the heap until matched by `receive`
- Regular GCs can skip unmatched messages
- Better for high-volume receivers

Can be set

Set per process at spwan

```
:erlang.process_flag(:message_queue_data, :off_heap)
```

Or set node default at startup

```
erl +hmqd off_heap
```

Speaker notes

When to use off_heap: Routers, aggregators, and any process that receives bursts of messages.

Modern Optimizations (OTP 25+)

Parallel Reception for Off-Heap Queues (OTP 25+)

- off_heap processes can accept signals from multiple senders **in parallel**.
- The runtime shards the outer signal queue so concurrent senders contend less.

```
# Many processes sending to one hot receiver
for i <- 1..1000 do
  spawn(fn ->
    send(hot_receiver, {:work, i})  # These can arrive in
parallel
```

```
    end)
end
```

Speaker notes

Benefits: Shows up when many processes send to a single receiver. Does not change per-sender ordering.

Priority Messages (OTP 28+)

A process can opt in to receive priority messages using a **priority alias:**

```
# Enable priority message reception
alias_ref = :erlang.alias([priority])

# Urgent signals bypass ordinary queue position
send(alias_ref, {:urgent, :shutdown})
```

Speaker notes

Use sparingly: Only for truly urgent control signals where long queues would cause unacceptable delays.

Non-blocking Distributed Sends (OTP 25.3+)

Enable fully asynchronous distributed signaling:

```elixir
# Per process
:erlang.process_flag(:async_dist, true)

# Or node-wide default: erl +pad true
```

Speaker notes

Trade-off: With `async_dist=true`, sending over distribution never blocks the sender, but you must implement your own flow control or memory can grow without bound.

Here if not before we should mention that distributed send is blocking.

Local Sends

- Local `!` does not block on I/O
- May still yield to scheduler
- Use `:erlang.send_nosuspend/2` if you cannot tolerate suspension

```elixir
case :erlang.send_nosuspend(pid, message) do
  true -> :ok
  false -> {:error, :would_suspend}
end
```

Distributed Sends

- Default: flow control can suspend sender when channel is busy
- Alternative 1: Use `send_nosuspend` for best-effort dropping
- Alternative 2: Enable `async_dist` and build back-pressure at application level
- Tuning: `+zdbbl` raises the busy limit for more buffering

Use Off-Heap for Fan-In Receivers

```elixir
defmodule MessageRouter do
  use GenServer

  def init(_) do
    # This router will receive many messages
    :erlang.process_flag(:message_queue_data, :off_heap)
    {:ok, %{}}
  end
end
```

Shape Payloads with Binaries

For broadcast or large payloads, pack into binaries for sharing:

```elixir
# GOOD - binary shared across processes
payload = :erlang.term_to_binary(large_data)
for pid <- subscribers do
  send(pid, {:broadcast, payload})
end

# LESS EFFICIENT - term structure copied each time
for pid <- subscribers do
  send(pid, {:broadcast, large_data})
end
```

Keep Selective Receive Tight

Tag requests with unique references:

```elixir
def call_server(pid, request) do
  ref = make_ref()
  send(pid, {:request, ref, self(), request})

  receive do
    {:response, ^ref, result} -> result
  after 5000 ->
    {:error, :timeout}
  end
end
```

Use Aliases for Droppable Replies

Hand out a reply alias and deactivate it on timeout:

```
def request_with_timeout(pid, request, timeout) do
  alias_ref = :erlang.alias()
  send(pid, {:request, alias_ref, request})

  receive do
    {:response, result} -> result
  after timeout ->
    :erlang.unalias(alias_ref)  # Late replies will be
dropped
    {:error, :timeout}
```

```
        end
         .
```

Configuration and Measurement

At boot:

```
# Default off-heap queues and larger distribution buffer erl +hmqd off_
```

Runtime verification:

```
:erlang.system_info(:message_queue_data)      # Check
default
:erlang.system_info(:dist_buf_busy_limit)     # Check dist
buffer
```

What to measure

- Mailbox length during load: `:erlang.process_info(pid, :message_queu`
- Binary memory growth: `:erlang.memory(:bina`
- GC time for hot servers: Compare `on_heap` vs `off_heap`
- Latency under fan-in workloads

Lab: Message Queue Optimization

```elixir
defmodule QueueBenchmark do
  def test_queue_types do
    # Test on_heap vs off_heap under load
    on_heap_pid = spawn(fn -> receiver_loop(:on_heap) end)

    off_heap_pid = spawn(fn ->
      :erlang.process_flag(:message_queue_data, :off_heap)
      receiver_loop(:off_heap)
    end)

    # Send burst of messages to both
    for i <- 1..10_000 do
```

```
      send(on_heap_pid, {:work, i})
      send(off_heap_pid, {:work, i})
```

Binary Construction Optimization

```elixir
# Inefficient: multiple allocations
def build_packet(data) do
  header = <<1, 2, 3>>
  length = <<byte_size(data)::32>>
  header <> length <> data
end

# Efficient: single allocation
def build_packet(data) do
```

Pitfall 1: Sub-binary retention

```elixir
# Problem: keeps entire file in memory
{:ok, file_content} = File.read("large_file.txt")
first_line = hd(String.split(file_content, "\n", parts:
2))
# first_line references the entire file!

# Solution: copy the sub-binary
first_line = :binary.copy(first_line)
```

Pitfall 2: Accumulating binary garbage

```elixir
# Problem: creates many intermediate binaries
Enum.reduce(list, <<>>, fn item, acc ->
  acc <> encode(item)  # new binary each iteration
end)

# Solution: use iolist
iolist = Enum.map(list, &encode/1)
:erlang.iolist_to_binary(iolist)  # single allocation
```

- Create a large binary and extract sub-binaries
- Monitor process memory with `:erlang.process_info(self(), :memory)`
- Force binary copying and observe the difference

Measuring and Optimizing Message Costs

Measurement Tools

```elixir
# Term size in words
:erts_debug.size(term)

# Flat (shallow) size
:erts_debug.flat_size(term)

# Binary representation size
byte_size(:erlang.term_to_binary(term))

# Process memory before/after
before = :erlang.process_info(self(), :memory)
# ... do something ...
after = :erlang.process_info(self(), :memory)
```

Cross-Process Communication Patterns

Local node messaging:

```
# Cost: copy entire term (except large binaries)
send(local_pid, {:data, big_map})
```

Cross-node messaging:

```
# Cost: serialize + network + deserialize
send({:process, :remote@node}, {:data, big_map})
```

```
# Everything gets copied, even large binaries!
```

ETS as shared memory:

```
# Write once, read many - no copying between readers
:ets.insert(:cache, {:key, large_data})
:ets.lookup(:cache, :key)  # no copy if same node
```

Message Payload Optimization Strategies

- Prefer atoms for fixed vocabularies
- Use binaries for string data
- Consider iolist for concatenation
- Be careful with sub-binaries
- Measure before optimizing

Lab: Message Cost Analysis

Build a simple benchmark comparing different message patterns:

```elixir
defmodule MessageBench do
  def benchmark do
    data_types = [
      {:small_tuple, {1, 2, 3}},
      {:large_tuple, Tuple.duplicate(0, 1000)},
      {:small_list, [1, 2, 3]},
      {:large_list, Enum.to_list(1..1000)},
      {:small_binary, "hello"},
      {:large_binary, :crypto.strong_rand_bytes(1024)},
      {:nested_map, %{a: %{b: %{c: %{d: 1}}}}}
    ]
```

```
for {name, data} <- data_types do
  size = :erts_debug.size(data)
```

Interactive Exercises

Open the LiveBook:

`module-2-exercises.livemd`

20 minutes of hands-on practice with:

- Term size measurement and memory footprints
- Message copying costs and performance impact
- Binary storage types and sub-binary retention
- Payload optimization strategies

Speaker notes

The exercises are in an interactive LiveBook notebook. Students should open `module-2-exercises.livemd` in LiveBook to work through the hands-on exercises.

Each exercise has runnable code cells and discussion questions.

Checklist

- Can you predict the memory size of common data structures?
- Do you understand what gets copied vs shared in message passing?
- Can you identify binary retention problems?
- Do you know when to use lists vs binaries vs iolists?
- Can you measure and optimize message costs?
- Do you understand the trade-offs between data types?

Conclusion

Data size = cost: immediates cheap, boxed terms copy.

Messages copy; large binaries are shared by reference.

Keep payloads small, tag with atoms/refs, use binaries or iolists.

Measure with :erts_debug.size/1 and mailbox stats.

Hands-On Exercises

Question 1

What is the threshold size at which a binary switches from heap storage to off-heap reference-counted storage?

- 32 bytes
- 48 bytes
- 64 bytes
- 128 bytes

Question 2

When you send a 1000-element list as a message to another process, what gets copied?

- Only a pointer to the list
- Every cons cell (2000 words total)
- Only the list header
- Nothing, lists are always shared

Question 3

Which data types are "immediate" and require no heap allocation?

- Small integers
- Atoms
- Floats
- Empty list (nil)
- Tuples

Question 4

You send a 10MB binary to 100 processes.
Approximately how much memory gets copied?

- 1000 MB (10MB × 100 processes)
- About 4-7 KB (only ProcBin wrappers)
- 10 MB (once, then shared)
- 0 bytes (binaries are never copied)

Question 5

What happens when a small map grows from 32 keys to 33 keys?

- It gets stored off-heap
- It transitions from flatmap to HAMT representation
- It gets automatically compressed
- It becomes immutable

Question 6

Why should you use `make_ref()` for request-response patterns?

- It creates a globally unique reference for precise matching
- It's faster than using atoms
- It prevents mailbox overflow
- It allows selective receive to skip unrelated messages efficiently
- It automatically times out old requests

Question 7

What is the main benefit of setting `message_queue_data` to `off_heap` for a process?

- Messages are processed faster
- Garbage collection doesn't need to scan unmatched messages
- Messages use less memory
- The mailbox has unlimited capacity

Question 8

When building binary data, which approach is most efficient?

- Concatenating binaries with `<>` or `<<B1/binary, B2/binary>>` in a loop
- Single binary comprehension: `<<header, size::32, data/binary>>`
- Building an iolist and converting once with `iolist_to_binary/1`
- Using `binary:copy/1` repeatedly

Question 9

A list `[1, 2, 3]` allocates how many words on the heap?

- 3 words (one per element)
- 4 words (header + 3 elements)
- 6 words (2 words per cons cell: head + tail)
- 0 words (small integers are immediate)

Question 10

What happens to large binaries when sending a message to a process on a remote node?

- The binary payload is serialized and sent over the network
- Only a reference is sent; payload stays local
- Large binaries are shared via reference counting across nodes
- Unlike local sends, refc binaries are fully copied for remote sends

Navigation

← Previous Module: Thinking in Processes | Back to Course Structure | Next Module: Memory & Garbage Collection →