Module 3 – Memory & Garbage Collection

The BEAM for Developers

Erik Stenman

Module Overview

- Course Philosophy –
  The Gnome's Backpack
- The BEAM Memory Model
- Binaries – Storage, Sharing, and Leaks
- Binary Memory Architecture
- Generational Garbage Collection
- **Tools & Techniques**
- Hands-On Exercises

Course Philosophy – The Gnome's Backpack

A process-gnome carries a **backpack** (heap/stack) and a **mailbag** (mailbox).

- The backpack holds its working stuff and gets tidied during GC.
- Very large scrolls (binaries) are kept in the archive and the gnome carries only a **reference**.
- If you tear off a tiny corner of a huge scroll and keep that corner in your notes, you might still keep the whole scroll in the archive.

The BEAM Memory Model

Working Memory - The Workbench

The BEAM is a register machine. The working memory of a process is kept in X and Y registers.

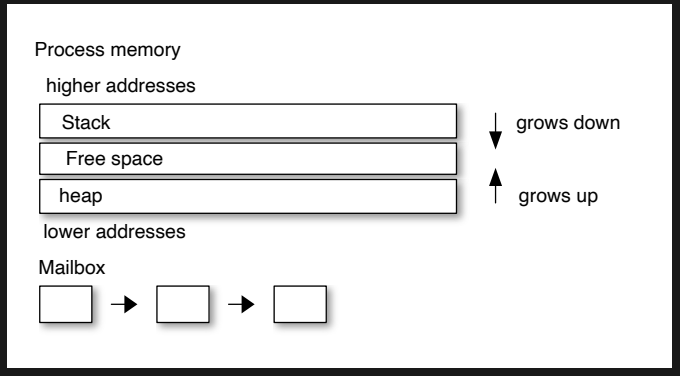
- **X-registers**: transient registers used by BEAM instructions for **arguments and temporaries** (e.g., `X0...Xn`). You don't manipulate these directly in Elixir/Erlang; they show up in disassembly.
- **Y-registers**: locals **spilled to the stack**. In BEAM assembly they're the "Y slots".

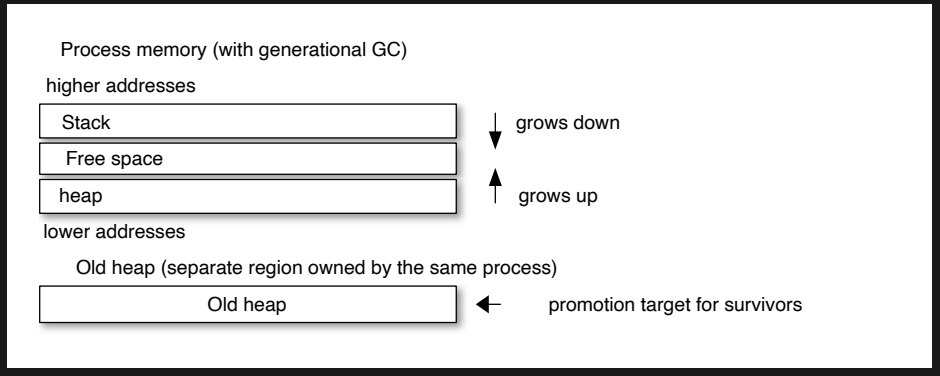
Long lived memory - The Backpack

- **Stack**: holds Y-registers (locals), saved continuation (return address), and frame metadata. Grows **downward**.
- **Heap**: where boxed values, list cells, and most terms live. Grows **upward**.

The mailbox

- **Mailboxes**: incoming messages are queued **outside** the heap; when the process `receive`s, the message is copied into the process heap (or from a message "heap fragment") and later reclaimed by per-process GC. More on this in the next module.


Process memory
higher addresses
Stack
Free space
heap
grows down
grows up
lower addresses
Mailbox


Process memory (with generational GC)
higher addresses
Stack
grows down
Free space
heap
grows up
lower addresses
Old heap (separate region owned by the same process)
Old heap
promotion target for survivors

Binaries – Storage, Sharing, and Leaks

- **Building on Module 2:** You learned that binaries come in two types: heap binaries (<64 bytes) and refc binaries (≥64 bytes)
- Now we'll see exactly how BEAM manages these and why binary leaks are the most common memory issue in production systems

Binary Memory Architecture

Heap Binaries (≤ 64 bytes)

Small binaries live directly on the process heap like any other term:

```
Process Heap ┌──────────────────────────────┐ | [HEAP_BIN][Size][Data...
```

Properties:

- Copied when sent between processes (like any heap data)
- Garbage collected with the process
- No sharing between processes
- Fast access, no indirection

Refc Binaries (≥ 64 bytes)

Large binaries use a two-level structure: a small ProcBin on the process heap points to shared payload in the binary allocator:

The payload lives off-heap but is shared. Each ProcBin is small (8 words) and lives on its process heap.

Sub-binaries: Views Without Copying

When you slice a binary, BEAM creates a sub-binary that points to the original:

```
Original: <<1,2,3,4,5,6,7,8,9,10>>  (refc binary, 1MB payload)
Slice:    <<_:24, 4,5,6, _:rest>>   (3 bytes wanted)

Process Heap                    Binary Allocator
┌──────────────────┐            ┌──────────────────────┐
│ [SUB_BIN]        │            │                      │
│ - size: 3        │            │ Original 1MB         │
│ - offset: 3      │ ──────────►│ payload...           │
```

The sub-binary keeps the entire 1MB alive! This is the classic binary leak pattern.

Match Context: Pattern Matching Optimization

During binary pattern matching, BEAM creates a match context to efficiently step through the binary:

```
# This creates a match context internally
<<len:32, data:len/binary, rest::binary>> = big_binary
```

The match context tracks the current position without creating intermediate sub-binaries for each step.

The Four Internal Binary Types

- **Heap Binary**: ≤ 64 bytes, stored directly on process
- Layout: `[HEAP_BIN_TAG][Size][Byte0][Byte`
- Copied on message send, GC'd with process
- **Refc Binary**: ≥ 64 bytes, payload off-heap with refere
- Process heap:
  `[PROCBIN_TAG][Size][Offset][BinPointer]`
- Off-heap: payload with reference counter
- **Sub-binary**: View into another binary (offset + lengt
- Layout:
  `[SUB_BIN_TAG][Size][Offset][OriginalBin`

How the binary allocator behaves

Best-fit placement to reduce fragmentation.

Ref counting keeps big payloads alive until no process references them.

GC interaction: a process only releases its references after it GC’s. A relay loop that never GC’s can keep gigabytes alive by accident.

Practical patterns

- When to copy

If you keep only a tiny piece of a huge binary in long-lived state, call a copy.

```
binary:copy(Slice)
```

If you split a big payload into parts and store them, consider copying each part you keep.

- When not to copy

If you process and drop quickly in the same function or worker, keep sub-binaries. Faster and no extra

allocation.

Design to avoid leaks

Keep big payloads inside short-lived workers. Let them die, and memory drops with them.

For relayers and routers, force periodic GC or hibernate when idle to drop references.

Sub-binaries and matching

By default, binary matching creates sub-binaries:

```
<<len::32, rest::binary>> = big
```

Rest is a view into big, no copy. Keep the view only if you drop it soon. If it will live in state, make it independent:

```
safe = :binary.copy(rest)
```

Split example

Erlang:

```
Parts0 = binary:split(Big, <<"\n">>, [global]),
Parts  = [binary:copy(P) || P <- Parts0].  % copy only if
you will keep Parts
```

ANTI-PATTERNS TO WATCH FOR

Keeping a tiny token that is a slice of a huge payload in a long-lived GenServer.

A relay loop that never GC’s but keeps touching different binaries.

Building large binaries repeatedly with <> in tight loops. Prefer iolists.

Iolist pattern

Build output without copying intermediate binaries, then send iolist directly to sockets/files or convert once.

```
io = [prefix, big, separator, small_tail]
bin = IO.iodata_to_binary(io)
```

Hands-on

- Simulate a refc binary leak, then fix it

```
defmodule Leaky do
  use GenServer
  def start_link(_), do: GenServer.start_link(__MODULE__,
:ok, name: __MODULE__)
  def init(:ok) do
    big = :crypto.strong_rand_bytes(10_000_000)    # ~10 MB
    tiny_view = binary_part(big, 0, 10)            # 10-
byte view
    {:ok, %{token: tiny_view}}                      # Keeps
10 MB alive by reference
  end
end
```

```
defmodule Fixed do
```

Measure before and after in :observer and with:

```
:erlang.process_info(pid, [:binary, :memory, :heap_size,
:total_heap_size])
```

- Relay loop that leaks by holding refs, then fix

```
loop(Workers, N) ->
  receive
    WorkItem ->
      Worker = lists:nth(N+1, Workers),
      Worker ! WorkItem,
```

```erlang
        %% Fix 1: periodic GC to release ProcBins
        erlang:garbage_collect(),
        %% Fix 2: or hibernate occasionally
        %% after 0 -> erlang:hibernate(?MODULE, loop,
[Workers, N]);
        loop(Workers, (N+1) rem length(Workers))
    end.
```

GenServer idea, After a burst, hibernate to drop refs and shrink, `{:noreply, state, :hibernate}`, Split and copy only what you keep.

Load a large file once.

:binary.split/3 on newline.

Keep only the first line in process state. Compare memory with and without :binary.copy/1.

- Binary allocator snapshots

Run a workload that streams large files through a router.

Take `:recon_alloc.snapshot(:current)` before and after.

Inspect growth in the binary allocator pools.

Decision checklist

- Do I keep a small slice for a while? -> Copy it.
- Do I process and drop quickly in a short-lived worker? -> Keep sub-binaries.
- Is a router touching many large messages? -> Add periodic GC or hibernate.
- Do I concatenate a lot? -> Use iolists and convert once at the boundary.
- Are binaries growing node-wide? -> Check :erlang.memory(:binary) and :recon_alloc.

Mini-quiz

What does a ProcBin hold, and where is the payload stored?

Sharing Expands on Send

Inside one process, the VM can reuse subterms. When you send a message, the term is copied and that sharing disappears. The receiver gets independent copies of repeated substructures. Large binaries are the exception—they are off-heap and shared by reference. Same rule applies when passing spawn arguments and when writing to ETS.

```
# One list, shared three times in this process
l = Enum.to_list(1..100_000)
t = {l, l, l}
```

```elixir
# After send, the receiver holds three separate lists
send(pid, {:t, t})
```

Why this matters: A term that uses 800KB in the sender (one list + three references) might expand to 2.4MB in the receiver (three full copies). This can cause surprising memory growth in message-heavy systems.

Build Option: Sharing-Preserving Copy

ERTS can be built to preserve sharing when copying terms for intra-node messages and exit signals. Configure with:

```
./configure --enable-sharing-preserving make -j sudo make install
```

This adds a pointer-table pass that preserves identical subterms, so `{L, L, L}` becomes one copied list referenced three times on the receiver. Trade-off: the copy step is more expensive when there is little or no

sharing. The option has been marked experimental historically.

Start the emulator and inspect the banner or system_version; builds with this option typically include the tag [sharing-preserving]:

```
iex> :erlang.system_info(:system_version)
"ERTS ... [sharing-preserving] ..."
```

- **Preserves intra-node sharing** on message send and exit signals
- **Also interacts with literals:** OTP 20+ does not copy module literals on send, regardless of this flag. They are copied only during module purge
- **ETS still copies** terms in and out. Literals and `persistent_term` values are copied when inserted into ETS
- **Distribution always serializes**. There is no pointer sharing across nodes
- You pass large terms that repeat big subparts, for

example fan-out of parsed schemas, ASTs, or big maps reused inside one message

- You saw memory blow-ups from expansion after send and the hot path is message copying rather than construction. In these cases the extra copy work can be offset by less heap growth on the receiver
- **CPU overhead** on send increases in common cases with little sharing. Benchmark on your workload
- **Past bugs** affected "magic references" with sharing-preserving builds; fixed in recent releases. Keep current

Practical Guidance

Prefer one large binary when broadcasting the same payload to many processes. Off-heap sharing wins:

```
# Good - binary shared by reference
big_payload = :erlang.term_to_binary(complex_data)
for pid <- subscribers, do: send(pid, {:broadcast,
big_payload})

# Less efficient - term structure copied and expanded
for pid <- subscribers, do: send(pid, {:broadcast,
complex_data})
```

Keep repeated subterms out of messages when possible. Send an identifier or index and let the receiver fetch shared data locally:

```
# Store once, reference many
:persistent_term.put({:schema, version}, parsed_schema)

# Send reference instead of full schema
for worker <- workers do
  send(worker, {:process_with_schema, version, data})
end
```

If you build with sharing-preserving, re-measure mailbox latency, GC, and binary memory before and

after. Use `msacc` and memory stats to see where time and bytes go.

Benchmark: Sharing Impact

Here's a simple test to see the memory difference:

```elixir
defmodule SharingTest do
  def test_expansion do
    # Create a large list that will be shared
    big_list = Enum.to_list(1..50_000)
    shared_tuple = {big_list, big_list, big_list}

    # Measure in sender
    sender_size = :erts_debug.size(shared_tuple)
    IO.puts("Sender sees: #{sender_size} words")

    # Send and measure in receiver
    parent = self()
    spawn(fn ->
```

```
receive do
```

Key insight: Message copying is not just about bandwidth—it's about memory multiplication on the receiving side.

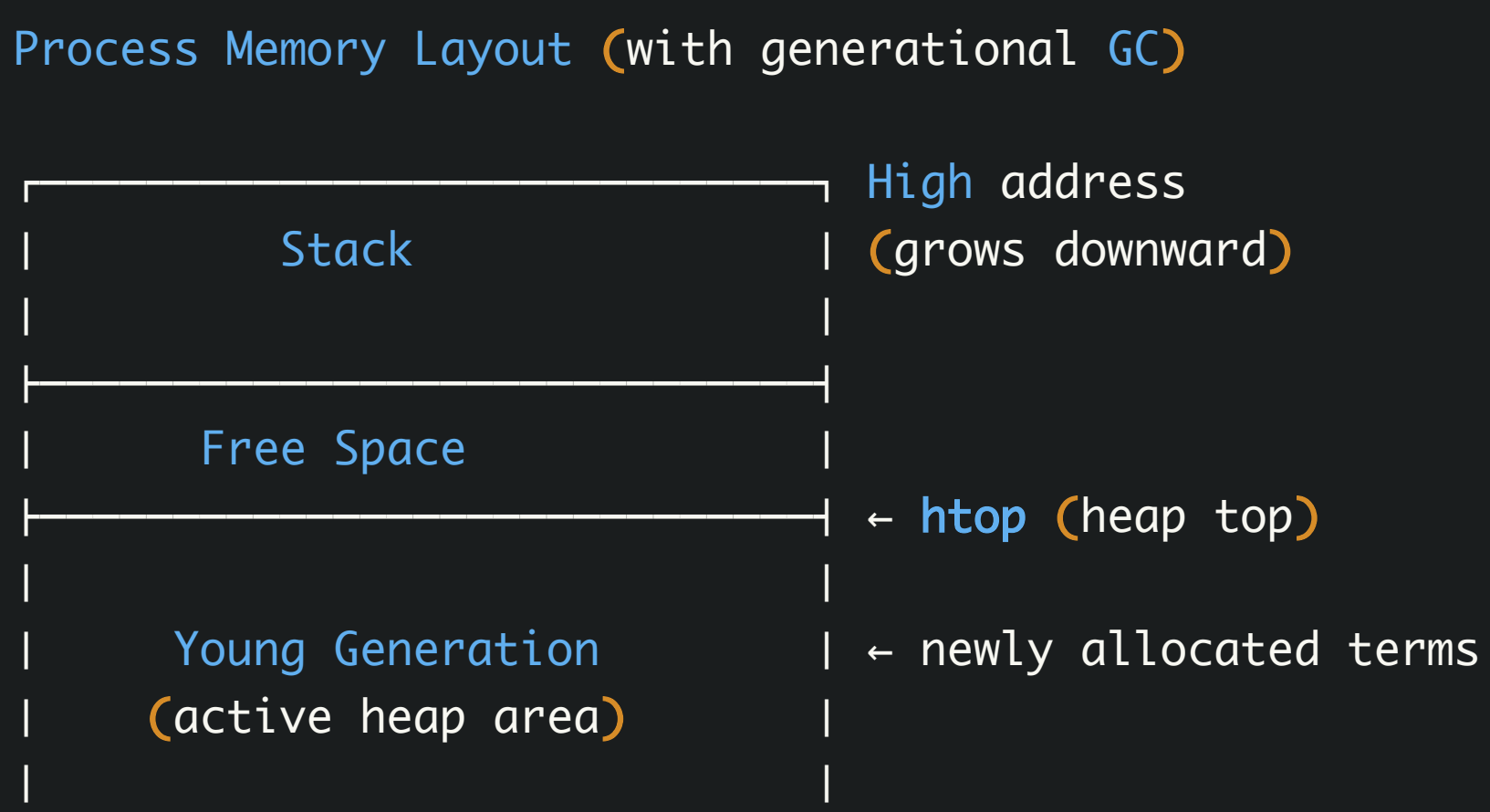
Generational Garbage Collection

The Generational Hypothesis

Most allocated objects die young. This insight drives BEAM's two-generation collector:

- **Young generation**: newly allocated terms, collected frequently
- **Old generation**: long-lived terms that survived multiple minor collections

Process Memory with Generational GC




├──────────────────────────────────┤ ← mature (promotion
boundary)

Minor GC (Young Generation Only)

Minor GC is BEAM's workhorse. It runs frequently and handles most garbage:

Triggers:

- Young heap approaches its limit
- Message receive with large payload
- Explicit call to `:erlang.garbage_collect/0`

Process:

- **Mark roots**: Stack variables, registers, persistent terms
- **Copy live young objects**: Depth-first traversal from roots
- **Promote survivors**: Objects that survived multiple minors → old generation
- **Update pointers**: Fix all references to moved objects
- **Reclaim**: Young generation space is now free

```
Before Minor GC: Young: [A][B][C][dead][D][dead][E] Old: [X][Y] After M
```

Major GC (Full Collection)

Major GC collects both young and old generations. More expensive but thorough:

Triggers:

- Too many minor GCs (controlled by `fullsweep_after`)
- Old generation getting full
- Process has accumulated many off-heap binaries
- Explicit `:erlang.garbage_collect/1` with full sweep

Process:

- **Collect everything**: Both young and old generations
- **Defragment**: Compact old generation, eliminating internal fragmentation
- **Binary sweep**: Release off-heap binary references
- **Heap consolidation**: May merge or resize heap areas

GC Triggers in Detail

Understanding when GC runs helps predict and tune performance:

Heap limit reached:

```
# Heap grows beyond current limit
process_flag(:min_heap_size, 1024)  # words
# Forces larger initial heap, fewer early minor GCs
```

Message receive:

```
# Large message triggers GC before copying to heap
# Ensures space is available for the message
send(pid, large_binary)  # May trigger minor GC in
receiver
```

Binary threshold:

```
# Process holds many off-heap binary references
process_flag(:min_bin_vheap_size, 512)  # words
# Trigger GC when binary ref size exceeds threshold
```

Fullsweep counter:

```
process_flag(:fullsweep_after, 10)  # default
# Major GC after 10 minor GCs
```

GC Performance Characteristics

Minor GC cost: O(live young data)

- Fast: only scans and copies young objects
- Predictable: time proportional to young generation size
- Frequent: keeps young generation small

Major GC cost: O(total live data)

- Slower: scans and copies entire heap
- Thorough: reclaims maximum memory
- Less frequent: every N minor GCs

Memory efficiency vs pause time trade-off:

- Small heaps: frequent minor GCs, low memory usage, short pauses
- Large heaps: infrequent GCs, higher memory usage, longer pauses when GC runs

Process Lifecycle Patterns

Short-lived workers (spawn, work, exit):

```
Task.start(fn ->
  data = expensive_computation()
  send(parent, {:result, data})
  # Process exits, entire heap reclaimed instantly
end)
```

- Benefit: no GC needed, heap reclaimed on exit
- Pattern: prefer for one-shot tasks

Long-lived servers (GenServers, persistent processes):

```elixir
# Accumulates state over time
def handle_call(req, _from, state) do
  new_state = Map.put(state, req.key, req.value)
  {:reply, :ok, new_state}  # State grows in old
generation
end
```

- Challenge: old generation grows, major GCs become expensive
- Solution: limit state size, use external storage (ETS, databases)

Observing GC in Practice

GC tracing:

```
# Enable GC tracing for a process
:erlang.trace(self(), true, [:garbage_collection])

# Generate some garbage
big_list = Enum.to_list(1..10_000)
processed = Enum.map(big_list, &(&1 * 2))

# Check trace messages
flush()
```

Process memory monitoring:

```elixir
# Before allocation
before = :erlang.process_info(self(), [:memory,
:total_heap_size,
                                       :heap_size,
:minor_gcs, :major_gcs])

# Do work that allocates
data = for _ <- 1..1000, do: {:some, :data, "string"}

# After allocation
after = :erlang.process_info(self(), [:memory,
:total_heap_size,
                                      :heap_size,
:minor_gcs, :major_gcs])
```

Heap growth patterns:

```elixir
defmodule HeapWatcher do
  def start_monitoring(pid) do
    spawn(fn -> monitor_loop(pid, []) end)
  end

  defp monitor_loop(pid, history) do
    info = :erlang.process_info(pid, [:heap_size,
:total_heap_size,
                                      :minor_gcs,
:major_gcs])
    new_history = [info | Enum.take(history, 10)]

    :timer.sleep(1000)
    monitor_loop(pid, new_history)
  end
```

Activity: GC Behavior Exploration

Experiment 1: Minor vs Major GC

```elixir
# Force different GC types and measure
defmodule GCExperiment do
  def minor_gc_test do
    # Allocate and discard repeatedly (minor GCs)
    for _ <- 1..100 do
      _temp = Enum.to_list(1..1000)
    end
  end

  def major_gc_test do
    # Accumulate long-lived data (triggers major GCs)
    Enum.reduce(1..100, [], fn i, acc ->
      [Enum.to_list(1..1000) | acc]  # Keeps growing
```

```
end)
```

Experiment 2: Heap Size Impact

```
# Small heap - frequent GCs
pid1 = spawn_opt(fn -> work_loop() end, [{:min_heap_size,
100}])

# Large heap - infrequent GCs
pid2 = spawn_opt(fn -> work_loop() end, [{:min_heap_size,
10000}])

# Compare GC frequency and pause times
```

Tuning Guidelines

For short-lived processes:

- Use small initial heap sizes
- Let minor GCs run frequently
- Prefer process exit over cleanup

For long-lived processes:

- Size heap appropriately for working set
- Tune `fullsweep_after` based on allocation patterns
- Consider hibernation to shrink heap during idle periods

For binary-heavy processes:

- Monitor off-heap binary references
- Tune `min_bin_vheap_size` to control binary-triggered GCs
- Copy sub-binaries when retaining small portions of large binaries

Tools & Techniques

Lab Exercise Plan

- **Run each lab individually** and observe the output
- **Use :observer** to visualize process memory during experiments
- **Correlate** Module 2's term size predictions with actual memory usage
- **Experiment with tuning parameters** (heap sizes, GC triggers)
- **Create your own variations** with different data structures

Expected Learning Outcomes

- See how Module 2's data type knowledge translates to memory behavior
- Understand when and why different GC types trigger
- Gain intuition for binary memory management
- Learn to use profiling tools effectively
- Connect theoretical knowledge to practical debugging skills

Interactive Exercises

Open the LiveBook:

`module-3-exercises.livemd`

20 minutes of hands-on practice with:

- Term size and memory layout exploration
- Binary storage and leak prevention
- Generational GC behavior observation
- Memory tuning and optimization

module-3-exercises.livemd

Checklist

- Can you explain where small terms, large binaries, and messages live?
- Can you detect and fix a sub-binary leak with `:binary.copy/1`?
- Do you know when and why a process runs minor vs major GC?
- Can you read `:observer` and `:erlang.memory/0` to find memory hotspots?
- Do you have a plan for backpressure to prevent mailbox growth?
- Do long-lived servers keep their live set small and bounded?

Conclusion

Memory on the BEAM is predictable once you know the rules. Keep long-lived state small, copy binary slices on purpose, and let per-process GC do its job. Measure first, then tune.

Hands-On Exercises

Question 1

Where does a refc binary's payload live?

- On the process heap
- Off-heap in the binary allocator
- In the process stack
- In ETS

Question 2

What causes a sub-binary leak?

- Keeping a small slice of a large binary in long-lived state
- Creating too many small binaries
- The sub-binary reference keeps the entire original binary alive
- Binary concatenation in loops

Question 3

When does a minor GC run?

- When the young generation heap approaches its limit
- Every 65535 reductions
- When receiving a large message
- Only when explicitly called

Question 4

What is the main benefit of generational GC?

- Minor GCs only scan young objects, making them fast
- It eliminates the need for major GCs
- Memory usage is reduced by 50%
- Long-lived data in old generation isn't scanned on every GC

Question 5

A process has 10,000 messages in its mailbox. What happens to that memory?

- Messages are automatically discarded after 1000
- Memory remains allocated until messages are received and GC runs
- The process automatically hibernates
- Messages are moved to ETS

Question 6

How do you fix a sub-binary leak?

- Use `binary:copy/1` to create an independent copy of the slice
- Force garbage collection with `erlang:garbage_collect/0`
- Set the process to off-heap message queue mode
- Copy the slice when you intend to keep it in long-lived state

Question 7

What does hibernation do to a process?

- Compacts heap and releases old generation memory
- Shrinks memory footprint aggressively
- Pauses the process permanently
- Moves the process to a different scheduler

Question 8

Why might a relay process accumulate binary memory even though it doesn't store binaries?

- ProcBin references stay alive until the process runs GC
- Binaries are automatically stored in process dictionary
- Touching binaries creates ProcBin wrappers on the heap
- The binary allocator never frees memory

Question 9

At what byte size does a binary switch from heap storage to refc storage?

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

Question 10

What triggers a major (full sweep) GC?

- After N minor GCs (controlled by fullsweep_after)
- Old generation getting full
- Every time a large message is received
- When calling `erlang:garbage_collect/0` (triggers minor by default)

Navigation

← Previous Module: Data Types & Message Costs | Back to Course Structure | Next Module: Scheduling & Concurrency →