Module 4 – Scheduling & Concurrency

The BEAM for Developers

Erik Stenman

Module Overview

- Course Philosophy – The Gnome Orchestra
- Conductor / Scheduler
- The Scheduler
- 🧪 **Comprehensive Scheduling Lab**
- Summary
- Hands-On Exercises

Course Philosophy – The Gnome Orchestra

- Extend the gnome metaphor to scheduling: imagine thousands of gnomes (processes) working in a giant workshop
- The **schedulers** are conductors who ensure every gnome gets fair time to work, no gnome monopolizes resources, and work flows smoothly between different sections of the workshop

Conductor / Scheduler

- Each conductor (scheduler) manages their own section but coordinates with other conductors to balance the overall workload
- When a gnome finishes their task or needs to wait, the conductor smoothly switches to another gnome without missing a beat

The Scheduler

The Scheduler Architecture

- The BEAM scheduler is what transforms millions of lightweight processes into manageable execution on real CPU cores.
- Unlike OS threads, BEAM processes are cooperatively scheduled within the VM, giving precise control over fairness and preemption.

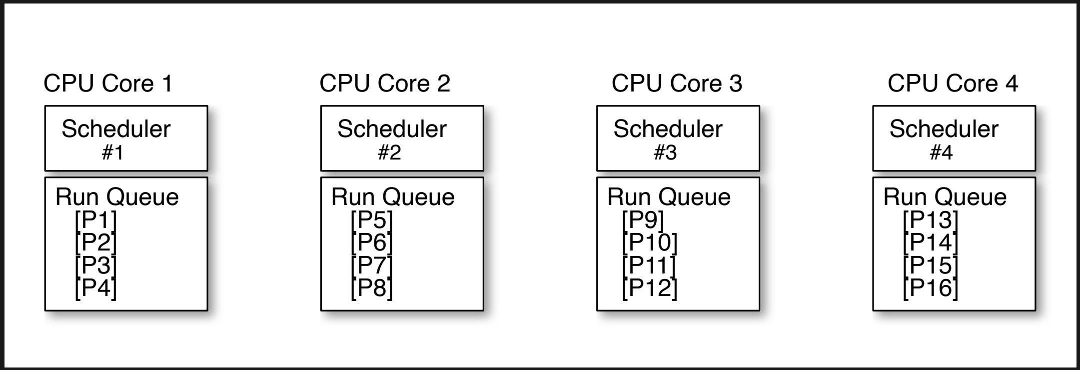
1 Scheduler / Core

BEAM runs one **scheduler thread** per CPU core by default.

Each scheduler manages its own **run queue** of processes ready to execute.

This design eliminates most locks and contention between cores.

System Overview (4-core machine):



Scheduler States and Transitions

Each scheduler operates in different states depending on workload:

- Active: Executing processes from the run queue (code or GC)
- Idle: No processes to run
- Load Balancing: Coordinating with other schedulers

Active

Executing processes from the run queue

- Scheduler is busy running processes
- CPU core is fully utilized
- Processes are being preempted based on reductions

Idle

No processes to run

- Run queue is empty
- Scheduler may sleep to save power
- Wakes up when work arrives

Load Balancing

Coordinating with other schedulers

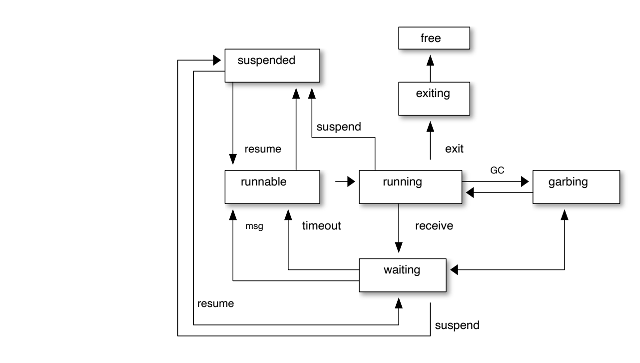
- Checking for work stealing opportunities
- Migrating processes between schedulers
- Balancing system-wide load

Process State Machine

The field status in the PCB contains the process state.
A process can be in one of these states:

- free - Process has exited
- runnable - Ready to run in the ready queue
- waiting - Blocked
- running - Currently executing
- exiting - terminating
- garbing - garbage collecting
- suspended - `erlang:suspend_process/2`

Process State Diagram



Key Behaviors

Normal states: runnable, waiting, and running

- running → runnable: Process uses up all its reductions (4000 reductions)
- running → waiting: Process enters receive with no matching message
- waiting → runnable: Message arrives or timeout occurs
- runnable → running: Scheduler picks process from ready queue

Garbing

Garbage collection: Process temporarily enters garbing state, saves previous state in gcstatus, then returns to previous state when GC completes.

Suspension

Suspension: Processes can be suspended for debugging. Each suspend_process/2 call increases the suspend count (rcount). Process stays suspended until count reaches zero via resume_process/1 calls.

Questions for Reflection

Architecture Understanding

- Why does BEAM use one scheduler per CPU core instead of a global scheduler?
- What problems does per-scheduler run queues solve?
- How does this design eliminate most locking?

Questions for Reflection

Observation and Analysis

- Run the system analysis on your machine. How many schedulers do you have?
- What happens to run queue lengths when you create CPU-intensive processes?
- Can you predict which scheduler a new process will be assigned to?

Questions for Reflection

Performance Implications

- When might you want fewer schedulers than CPU cores?
- What workloads benefit most from BEAM's scheduling approach?
- How does scheduler utilization relate to system responsiveness?

Checklist

Can you:

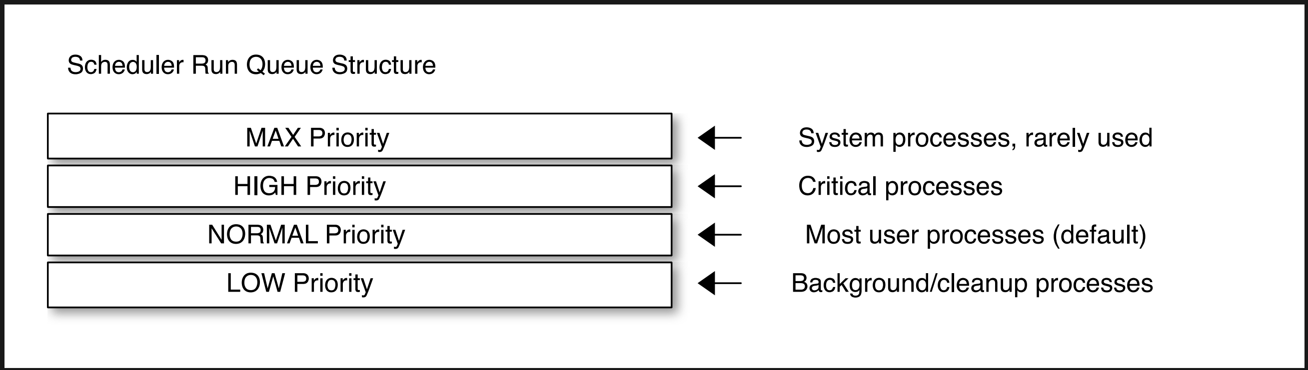
- explain BEAM's per-core scheduler architecture?
- explain the difference between RUNNING, WAITING, and SUSPENDED process states?
- measure scheduler utilization on your system?
- tell what factors affect run queue lengths?
- identify scheduler-related performance issues?

Run Queues

Run queues are the heart of BEAM's scheduling system. Understanding how processes enter, wait in, and leave run queues explains most scheduling behavior you'll observe in production.

Run Queue Structure and Priority

Each scheduler maintains multiple priority levels within its run queue:



Priority scheduling rules

- Higher priority processes always run before lower priority
- Within same priority, processes are scheduled in FIFO order
- A running process can be preempted by higher priority processes
- Priority inversion is avoided through careful queue management

Process Selection Algorithm

When a scheduler selects the next process to run:

- **Check higher priority queues first**
- **Within priority level, use FIFO**
- **Consider process "locality"** (recently run processes may have better cache)
- **Apply fairness constraints** (prevent starvation)

Speaker notes

The low priority does not have its own queue but is picked every X turn in the timing wheel.

Message-Driven Scheduling

Process transitions between states are primarily driven by message passing:

Process becomes runnable when:

- A message arrives in its mailbox
- A timeout expires
- An I/O operation completes
- It's newly spawned

Process becomes suspended when:

- It executes `receive` with no matching messages
- It calls a blocking operation (Send to busy port)
- It explicitly yields

Timing Wheels

How They Work

Timers are handled in the VM by a timing wheel

An array of time slots that wraps around.

Prior to Erlang 18, the timing wheel was a global resource with potential contention for the write lock when many processes inserted timers.

Speaker notes

The timing wheel is a circular buffer mechanism that efficiently handles timeout operations for receive statements and other timer-based operations. For timeouts longer than the wheel size, the count field tracks how many complete rotations are needed before the timer fires.

Configuration

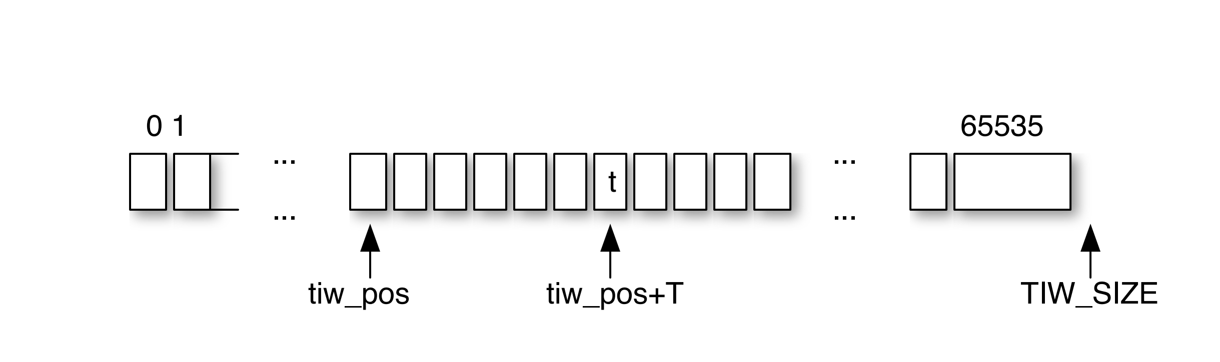
- Default size (TIW_SIZE): 65536 slots (or 8192 slots for small memory footprint builds)
- Current time pointer: tiw_pos - index into the array indicating current time

Timer Insertion Algorithm

When a timer is inserted with timeout T:

- tiw_pos points to current time slot
- Timer is placed at slot: (tiw_pos + T) % TIW_SIZE
- Array wraps around from slot 65535 back to slot 0
- Multiple timers in the same slot are linked together in a linked list

Timing Wheel Diagram



ErlTimer Structure

Each timer in the timing wheel is represented by an ErlTimer struct:

```
typedef struct erl_timer {
    struct erl_timer* next;    /* next entry tiw slot */
    struct erl_timer* prev;    /* prev entry tiw slot */
    Uint slot;                 /* slot in timer wheel */
    Uint count;                /* loops remaining */
    int    active;             /* 1=activ, 0=inactiv */
    void (*timeout)(void*);
    void (*cancel)(void*);
    void* arg;
} ErlTimer;
```

Speaker notes

- next/prev: Linked list pointers for chaining multiple timers in same slot
- slot: Which timing wheel slot contains this timer
- count: For timeouts > TIW_SIZE, this tracks remaining wheel rotations (T/TIW_SIZE)
- active: Whether timer is currently active
- timeout: Function called when timer expires
- cancel: Optional function called when timer is cancelled
- arg: Argument passed to timeout/cancel functions

Run Queue Dynamics and Bottlenecks

Healthy run queue characteristics:

- Most queues empty or with 1-2 processes
- Processes don't stay in queue long
- Even distribution across schedulers

Run Queue Dynamics and Bottlenecks

Problem indicators:

- Consistently long run queues (>10-20 processes)
- Uneven distribution across schedulers
- High priority processes stuck behind normal priority

Questions

Priority and Fairness:

- What happens if you have too many high-priority processes?
- How does BEAM prevent priority inversion?
- When should you use non-normal priorities?

Questions

Run Queue Health:

- What run queue lengths indicate problems?
- How can you detect scheduler imbalance?
- What causes processes to accumulate in run queues?

Questions

Message-Driven Behavior:

- Why do suspended processes not consume scheduler time?
- How does `receive` with timeouts affect scheduling?
- What makes a process transition from SUSPENDED to WAITING?

Checklist

- Do you understand the run queue priority structure?
- Can you explain when processes transition between states?
- Can you monitor and assess run queue health?
- Do you know when to use different process priorities?
- Can you predict how message patterns affect scheduling?

Reductions and Preemption

The reduction system is BEAM's mechanism for fair preemption. It ensures no process can monopolize a scheduler thread, making the system responsive even under heavy computational load.

What Are Reductions?

A **reduction** is BEAM's unit of work measurement. Every operation that can potentially consume significant time costs reductions:

- Function calls
- Message sends
- Pattern matching
- Arithmetic operations
- Memory allocation
- Built-in function calls (BIFs)

The Preemption Mechanism

Each process gets a **reduction budget** when scheduled (typically 2000 reductions). When the budget is exhausted:

- **Process is preempted** (removed from CPU)
- **Process goes back to run queue** (if still runnable)
- **Scheduler picks next process**
- **Process gets new budget when rescheduled**

Reduction Counting in Practice

```erlang
measure(F) ->
    {reductions, B} = process_info(self(), reductions),
    {_, A} = process_info(self(), reductions),
    Owerhead = A - B,
    {_, Before} = process_info(self(), reductions),
    Result = F(),
    {_, After} = process_info(self(), reductions),

    io:format("Reductions used: ~p~n",
     [(After - Before) - Owerhead]),
    Result.
```

Reduction Counting in Practice

```erlang
%% Simple arithmetic
measure(fun() -> 1 + 1 + 1 + 1 + 1 end).
%% Function calls
measure(fun() -> lists:sum([1, 2, 3, 4, 5]) end).
%% List processing
measure(fun() ->
    lists:map(fun(X) -> X end, lists:seq(1, 100))
end).
%% Message sending
measure(fun() -> self() ! test end).
%% Pattern matching
measure(fun() ->
    case {1, 2, 3} of {A, B, C} -> A end
end).
```

Preemption and Latency

Understanding preemption helps predict system latency:

```
defmodule LatencyAnalysis do
  def analyze_preemption_impact do
    # Simulate competing processes with different characteristics

    # CPU-intensive process (uses full reduction budget)
    cpu_intensive = spawn(fn ->
      cpu_heavy_work()
    end)

    # Interactive process (frequent yields)
```

```elixir
interactive = spawn(fn ->
  interactive_work()
end)
```

Questions on Reductions

- **Reduction Understanding**:
- Why does BEAM count reductions instead of using time-based preemption?
- Which operations are "expensive" in terms of reductions?
- How do reductions relate to actual CPU time?

Questions on Reductions

- **Preemption Behavior**:
- When exactly does preemption occur?
- Can a process be preempted in the middle of a function call?
- How does preemption affect message processing latency?

Questions on Reductions

- **Performance Implications:**
- When should you manually yield control?
- How do reduction costs affect system design?
- What happens if processes never yield?

Checklist

- Do you understand what reductions measure?
- Can you predict which operations will be expensive?
- Do you know when processes get preempted?
- Can you measure reduction usage in your code?
- Do you understand when to yield manually?

SMP and Load Balancing

BEAM's Symmetric Multi-Processing (SMP) support enables true parallelism across CPU cores.

Key principles

- Each scheduler operates independently
- Minimal locking between schedulers
- Work stealing for load balancing
- Process migration for optimization

Work Stealing and Migration

When schedulers become imbalanced, BEAM uses **work stealing** to redistribute load:

Work stealing triggers:

- One scheduler has empty run queue while others are loaded
- Significant imbalance detected during load checks
- Scheduler utilization falls below threshold

Migration strategies:

- **Steal from heavily loaded schedulers**
- **Prefer recently spawned processes** (less cache pollution)
- **Respect process affinity when possible**
- **Avoid migrating message-heavy processes**

Process Affinity and Locality

BEAM tries to maintain **process affinity** to improve cache performance:

Affinity benefits:

- Better CPU cache utilization
- Reduced memory latency
- Less cross-core communication

When affinity is broken:

- Work stealing for load balancing
- Process migration for resource access
- Explicit scheduler assignment

VM flags for scheduler tuning:

```
# Set number of schedulers
erl +S 4:4          # 4 schedulers, 4 online

# Scheduler binding
erl +sbt db         # Default binding (recommended)
erl +sbt u          # Unbound
erl +sbt ns         # No spread

erl +swt very_low|low|medium|high|very_high -
```

Speaker notes

Does not work on OsX

Questions on Run Queues

- **SMP Architecture**:
- Why does BEAM use separate run queues per scheduler?
- What are the trade-offs of work stealing?
- How does process migration affect performance?

Questions on Run Queues

- **Load Balancing**:
- When does work stealing occur?
- What factors determine if a process should be migrated?
- How can you detect scheduler imbalance?

Questions on Run Queues

- **Configuration and Tuning**:
- When might you want fewer schedulers than CPU cores?
- How does scheduler binding affect performance?
- What workloads benefit from SMP scheduler tuning?

Checklist

- Do you understand BEAM's SMP architecture?
- Can you explain work stealing and migration?
- Do you know how to measure scheduler utilization?
- Can you identify load balancing issues?
- Do you understand scheduler configuration options?

🧪 Comprehensive Scheduling Lab

Lab 1: Complete Scheduler Analysis

Build a comprehensive scheduler monitoring tool:

```
defmodule SchedulerAnalyzer do
  def full_system_analysis(duration_seconds \\ 30) do
    IO.puts("=== Complete Scheduler Analysis (#{duration_seconds}s) ===")

    # Initialize monitoring
    :erlang.system_flag(:scheduler_wall_time, true)
    :erlang.statistics(:scheduler_wall_time)  # Reset baseline
```

```
    # Start workload
    workload_pids = create_comprehensive_workload()

    # Collect data over time
```

Lab 2: Scheduler Tuning Experiment

Test different scheduler configurations:

```
defmodule SchedulerTuningLab do
  def test_different_configurations do
    IO.puts("=== Scheduler Configuration Testing ===")

    # Test 1: Priority impact
    test_priority_effects()

    :timer.sleep(2000)

    # Test 2: Process affinity
```

```
    test_process_affinity()

    :timer.sleep(2000)
```

- **System Design**:
- What scheduler metrics would you monitor in production?
- How do you balance throughput vs latency in process design?

- **Performance Analysis**:
- A process is using 10x more reductions than expected. How do you investigate?
- Your schedulers show 90% utilization but response times are poor. Why?
- How do you detect if work stealing is happening too frequently?

- **Tuning Scenarios**:
- When would you use fewer schedulers than CPU cores?
- How do you optimize for batch processing vs interactive workloads?
- What scheduler flags would you tune for a low-latency trading system?

Summary

Key Takeaways

- **Scheduler Architecture**: BEAM's per-core design eliminates most locking
- **Fair Scheduling**: Reduction-based preemption ensures responsiveness
- **Load Balancing**: Work stealing maintains balance across cores
- **Configurability**: Multiple tuning options for different workload patterns

Production Considerations

Monitoring: Track scheduler utilization, run queue lengths, reduction rates

Alerting: Set up alerts for persistent queue imbalances or high utilization

Capacity Planning: Understand how your workload scales with scheduler count

Interactive Exercises

Open the LiveBook:

`module-4-exercises.livemd`

20 minutes of hands-on practice with:

- Scheduler architecture and configuration analysis
- Process state transitions and priority effects
- Reduction measurement and preemption behavior
- SMP load balancing and work stealing

Speaker notes

The exercises are in an interactive LiveBook notebook. Students should open `module-4-exercises.livemd` in LiveBook to work through the hands-on exercises.

Each exercise has runnable code cells and discussion questions.

Final Checklist

Can you explain:

- BEAM's scheduling architecture end-to-end?
- how reductions drive preemption?
- hoe to analyze scheduler utilization and load balance?
- how to tune scheduler behavior?
- the trade-offs between different configurations?

Hands-On Exercises

Question 1

How many scheduler threads does BEAM create by default?

- One per logical CPU (including hyperthreading)
- One per physical CPU core only
- Always 4 schedulers regardless of hardware
- As many as needed based on process count

Question 2

What happens when a process exhausts its reduction budget?

- It is preempted and moved to the back of its run queue
- It crashes with a reduction_limit error
- It continues running until it calls receive
- The scheduler picks the next process to run

Question 3

Which process states do NOT consume scheduler time?

- SUSPENDED (blocked in receive)
- RUNNING
- WAITING (in run queue but not executing)
- GARBING (performing GC)

Question 4

What is work stealing in BEAM's scheduler?

- Processes stealing CPU time from other processes
- Idle schedulers taking processes from busy schedulers' run queues
- High priority processes preempting low priority ones
- Load balancing mechanism to distribute work across cores

Question 5

Why does BEAM use reduction-based scheduling instead of time-based?

- Ensures processes yield at safe, known execution points
- Avoids complexity of interrupting arbitrary VM state
- Time-based scheduling is impossible in a VM
- Reductions are faster to count than time

Question 6

What is process priority's effect on scheduling?

- Higher priority processes run before lower priority ones
- Priority determines reduction budget size
- Within same priority, FIFO order is used
- Priority only affects process spawn order

Question 7

When should you manually yield with `timer:sleep(0)`?

- When processing large batches to allow other processes to run
- Never, the scheduler handles everything automatically
- In long-running loops that don't hit receive
- After every function call for fairness

Question 8

What does a consistently high run queue length indicate?

- Efficient use of CPU resources
- More runnable processes than scheduler capacity
- Possible performance bottleneck
- Normal operation under any load

Question 9

How does process migration affect performance?

- Reduces CPU cache locality when process moves to new scheduler
- Always improves performance through better load balance
- Trades cache performance for load balancing benefits
- Has no performance impact

Question 10

What operations automatically yield control before exhausting reductions?

- `receive` with no matching messages
- Long-running BIFs like file I/O
- Arithmetic operations
- Function calls

Navigation

← Previous Module: Memory & Garbage Collection | Back to Course Structure | Next Module: Debugging & Observability →