# Assignment 6

## The Sleeping Barber: Elixir and Go

## Overview

In this assignment you will implement the same concurrent system twice: once in Elixir, using its built-in actor model, and once in Go, using idiomatic Go concurrency — goroutines, channels, and select. The system is the classic Sleeping Barber problem.

The goal is not just to produce two working programs. It is to experience firsthand how two different languages model concurrency, and to articulate — in writing — the differences you observe. Both implementations must be instrumented to collect and report runtime statistics, requiring each concurrent process to maintain and update local state across its lifetime (stateful process).

You are welcomed to use AI coding tools to code the Elixir version, like we did in the previous assignment. The Go implementation, the instrumentation logic, and the written comparison are your own work. You may get AI assistance with syntax, semantic details, try ideas, etc. as the AI policy for the class says, but the full solution is yours to produce.

## Learning Objectives

- Implement a non-trivial concurrent system using Elixir's actor model: spawn, send, receive, and stateful process loops.
- Implement the same system in idiomatic Go: goroutines, channels, and select statements.
- Maintain local state inside concurrent processes/goroutines without shared mutable variables.
- Implement request/reply message patterns in both languages.
- Compare the two concurrency models concretely, based on the code instrumentaions.

## The Problem: Sleeping Barber

**To review:**

A barbershop has one barber and a waiting room with a fixed number of seats. The rules are:

1. If the waiting room is full when a customer arrives, the customer leaves immediately;
2. otherwise the customer takes a seat and waits.
3. When the barber finishes a haircut — or wakes up from sleeping — the next waiting customer is called.
4. The barber cuts the customer's hair ( simulated by the process sleeping a random duration – not the same as “sleeping” as a barber ).
5. After the haircut the barber asks the customer for a satisfaction rating (1–5 stars).

6. The customer's rating is influenced by how long they waited — longer waits produce lower ratings.
7. The barber records the rating and the duration, updating running averages; we don't need to keep every individual rating and duration, we just have the barber keep a running average.
8. The simulation ends when a configurable total number of customers have arrived and all in-progress haircuts have completed.

Both implementations must produce identical observable behavior: the same logical process structure, the same message flow, the same statistics collected, and the same closing report format. Of course, since the stats are based on randomness, the actual numbers reported will vary across implementations, and from run to run. But we want the same information collected and reported. You are free to report other information as well, whatever interests you or helps show the processes are working properly (no deadlock, no starvation, etc.).

# Process Specification

Your implementations must include the following four concurrent entities. In Elixir these are actor processes; in Go these are goroutines. The behavior of each is described here in language-neutral terms.

## Customer

One concurrent entity is created per customer. Each has a unique ID and records its arrival time when created.

- Sends an arrive message to the Waiting Room immediately on creation, including its own identity (so the Waiting Room can reply).
- Waits for one of two responses:
  - Turned Away — logs the rejection and exits.
  - Admitted — waits to be called by the Barber.
- When called by the Barber, notes the time (this is the haircut-start time, used to compute total wait time).
- After the haircut, receives a rate request from the Barber, computes a satisfaction score, and sends it back.
- Exits after sending the rating.

Satisfaction score: `score = clamp(1, 5, 5 - floor(wait_seconds / threshold) + jitter)` where threshold is a tunable constant (suggested: 3 seconds) and jitter is a small random value (±1). This means a customer who waited less than 3 seconds is likely to give 5 stars; one who waited 9+ seconds is likely to give 1 star.

## Waiting Room

A single long-lived concurrent entity. Maintains two pieces of local state: the current queue of waiting customers (up to a configurable capacity, suggested default: 5), and a running count of customers turned away.

- Receives arrive messages from Customer entities.
- If at capacity: replies turned_away to the customer, increments its turnaway count.
- Otherwise: adds the customer to its queue, replies admitted to the customer.
- Receives next_customer requests from the Barber.
- If the queue is non-empty: removes the front customer and sends them to the Barber.
- If the queue is empty: notifies the Barber that no one is waiting (the Barber will then sleep).
- When a customer arrives and the Barber is sleeping, the Waiting Room is responsible for waking the Barber.
- Responds to a stats request with its turnaway count and current queue length.
- Shuts down cleanly when instructed.

Note on the sleeping/waking handshake: the Waiting Room must track whether the Barber is currently sleeping. One clean approach is to flip a "barber is sleeping" flag to true when the Waiting Room sends a none_waiting reply, and back to false when it sends a wakeup. This way the Waiting Room sends wakeup exactly once per sleep cycle — not on every customer arrival.

## Barber

A single long-lived concurrent entity. This is the most stateful entity in the system. The Barber maintains:

- Number of haircuts completed.
- Running average haircut duration (in milliseconds).
- Running average customer satisfaction rating.

*Behavior:*

- On startup, sends a next_customer request to the Waiting Room.
- If none_waiting is received: enters a sleeping state, waiting only for a wakeup message.
- When woken: sends another next_customer request and resumes normal operation.
- When a customer arrives: simulates the haircut by sleeping a random duration (suggested: 1,000–4,000ms drawn uniformly).
- After the haircut: sends a rate request to the customer and waits for the reply.
- Updates all three running statistics.
- Immediately sends another next_customer request to the Waiting Room.
- Responds to a stats request with current cut count, average duration, and average rating.
- Completes any in-progress haircut before processing a shutdown message.

The Barber's state must live entirely inside the Barber's own concurrent context — no external variables, no shared data structures. In Elixir this means carrying state as parameters through the recursive receive loop. In Go this means using local variables inside the goroutine function.

### Shop Owner

A single entity that drives the simulation lifecycle. Maintains a count of customers spawned.

- Spawns Customer entities at random intervals (suggested: 500–2,000ms between arrivals).
- Stops after a configurable total number of customers (suggested default: 20).
- After the last customer is spawned, waits a grace period for in-flight haircuts to complete (suggested: at least 2× the maximum haircut duration).
- Sends stats requests to both the Barber and the Waiting Room and collects the responses.
- Sends shutdown to both the Barber and the Waiting Room.
- Prints the closing report.

## Message Reference

The following table defines all messages. Both implementations must support all of these. Naming and encoding are your choice.

| Message | Direction | Payload / Notes |
|---|---|---|
| arrive | Customer → Waiting Room | Customer identity (PID or channel ref) |
| admitted | Waiting Room → Customer | No payload — customer waits for barber |
| turned_away | Waiting Room → Customer | No payload — customer exits |
| next_customer | Barber → Waiting Room | Barber identity (for wakeup) |
| customer_ready | Waiting Room → Barber | Customer identity |
| none_waiting | Waiting Room → Barber | No payload — barber will sleep |
| wakeup | Waiting Room → Barber | No payload |
| rate_request | Barber → Customer | Barber identity (for reply) |
| rating | Customer → Barber | Integer 1–5 |
| get_stats | Shop Owner → Barber / WR | Requester identity (for reply) |
| stats_reply | Barber / WR → Shop Owner | See Closing Report section |
| shutdown | Shop Owner → Barber / WR | No payload |

## Instrumentation: Logging and Statistics

### Per-event logging

Both implementations must print a log line for each significant event during the simulation. Log lines should include at minimum: a timestamp or elapsed time, the identity of the process generating the event, and a description of the event. Suggested events to log:

- Customer arrives
- Customer admitted to waiting room (with current queue depth)
- Customer turned away (with current queue depth)
- Barber starts haircut for customer N (with planned duration)
- Barber finishes haircut, asks for rating
- Customer gives rating (with their wait time)
- Barber receives rating, logs updated running averages
- Barber goes to sleep / wakes up
- Shop Owner spawns customer N of total
- Shutdown initiated

The purpose of this logging is to make the concurrent interleaving of processes visible and interpretable. A reader should be able to follow the log and understand what every process was doing at every point in the simulation.

### Closing Report

At the end of the simulation both implementations must print a closing report in this format:

```
=== Barbershop Closing Report ===
Total customers arrived:      20
Customers served:             14
Customers turned away:         6
Average haircut duration:   2.47s
Average satisfaction:       3.6 / 5.0
==================================
```

Exact spacing and borders are not required — the fields are. Numbers must be accurate: average duration and rating must reflect only completed haircuts.

## Configuration

All tunable parameters must be centralized — in a Config module (Elixir) or a config struct / named constants block (Go). No magic numbers scattered through the code. Required parameters:

| Parameter | Suggested Default |
|---|---|
| Total customers to arrive | 20 |
| Waiting room capacity (seats) | 5 |

| | |
|---|---|
| Customer arrival interval | 500 – 2,000 ms (random uniform) |
| Haircut duration range | 1,000 – 4,000 ms (random uniform) |
| Satisfaction wait threshold | 3.0 seconds per star lost |
| Shutdown grace period | 8,000 ms after last customer spawned |

# Constraints

## Both implementations

- No shared mutable state. All state lives inside the concurrent entity that owns it.
- No busy-waiting. Processes block on message receipt; goroutines block on channel reads or select.
- Graceful shutdown: no goroutine leaks, no deadlocks on exit, log lines from shutdown must appear before the report.
- All configurable parameters in one place.

## Elixir-specific

- Each process must be a plain spawned function — no GenServer, no OTP behaviours. The point is to see raw actor mechanics.
- State must be carried as parameters through recursive receive loops, not stored in module attributes or ETS.
- You may use AI tools for Elixir syntax and idioms. You must understand and be able to explain every line you submit.

## Go-specific

- One goroutine per logical process (Customer, Waiting Room, Barber, Shop Owner). Do not collapse multiple logical processes into one goroutine.
- Communication between goroutines exclusively via channels. No mutexes, no sync primitives, no shared variables.
- Use select where a goroutine must be ready to receive from more than one channel (e.g., the Barber waiting for either a customer or a wakeup, or a shutdown signal).
- Define explicit message types — structs with a type tag and payload fields. Do not pass raw primitives on channels.

# Go Implementation Guidance

This section offers some structural guidance for the Go implementation. This is not prescriptive — your design may differ — but it should help you get started.

## Message types

Define a message type that can carry any message in the system. A common Go pattern:

```
type MsgKind int

const (
    MsgArrive MsgKind = iota
    MsgAdmitted
    MsgTurnedAway
    MsgNextCustomer
    MsgCustomerReady
    MsgNoneWaiting
    MsgWakeUp
    MsgRateRequest
    MsgRating
    MsgGetStats
    MsgStatsReply
    MsgShutdown
)

type Message struct {
    Kind      MsgKind
    From      chan Message  // reply-to channel
    CustomerID int
    Value     int           // rating, or other integer payload
    ArrivalMs int64         // arrival timestamp
}
```

## Process identity

In Go, a goroutine has no built-in identity. Use its input channel as its identity — passing a channel reference is equivalent to passing a PID in Elixir. A goroutine that needs to receive replies creates its own channel and passes it in the From field of outgoing messages.

## The Barber's select

The Barber is the most complex goroutine. It needs to handle incoming messages differently depending on whether it is sleeping or cutting. One clean approach is two separate receive paths:

```
// sleeping state — only wakeup or shutdown accepted
func (b *BarberState) sleepLoop(mailbox chan Message) {
    for {
        msg := <-mailbox
        switch msg.Kind {
        case MsgWakeUp:
            return  // exit sleep loop, re-enter main loop
        case MsgShutdown:
            return
        }
    }
}
```

### The waiting room's wakeup logic

The Waiting Room needs to know whether the Barber is sleeping so it can send exactly one wakeup per sleep cycle. A simple boolean field on the Waiting Room's state struct, flipped to true when none_waiting is sent and back to false when wakeup is sent, is sufficient.

### Graceful shutdown

Shutdown ordering matters. The Shop Owner should:

9. Wait for the grace period.
10. Send get_stats to Barber and Waiting Room and collect both replies before sending shutdown.
11. Send shutdown to Barber, then Waiting Room.
12. Use a small additional sleep (200ms) to allow shutdown log lines to print before the report.

If you find the Barber is sometimes mid-haircut when stats are collected, increase the grace period or add an explicit "I am idle" acknowledgment from the Barber.

## Deliverables

Submit a single ZIP archive with the following structure:

```
assignment3/
  elixir/
    sleeping_barber.exs     # single-file Elixir script
    README.md               # how to run
  go/
    main.go                 # or a small Go module
    README.md               # how to run
  reflection.md            # written comparison (see below)
```

Both programs must run with a single command from their respective directories:

```
# Elixir
elixir sleeping_barber.exs

# Go
go run main.go
```

### Reflection (reflection.md)

Write 600–900 words addressing the following. Be specific — reference actual code, line numbers, or design decisions. Generic answers will receive partial credit.

13. Process identity. In Elixir, processes have PIDs. In Go, goroutines have no built-in identity. How did you solve the identity/addressing problem in Go? What did you pass around to let goroutines find each other?
14. State management. In Elixir, a process carries state as parameters through a recursive receive loop. In Go, state lives in local variables inside a goroutine function. How similar or different did these feel to implement? Did one feel more natural for any particular process?
15. The sleeping barber handshake. The barber must sleep when no customers are waiting and wake up when one arrives. How did you implement this in each language? What mechanisms did each language offer, and how did your solutions differ?
16. Message types. Elixir pattern-matches on atoms and tuples with no boilerplate. Go requires explicit struct definitions and a type tag. What was the practical impact of this difference on your code?
17. select vs. receive. Go's select lets a goroutine wait on multiple channels simultaneously. Elixir's receive matches against a single mailbox. Did this difference affect your design for any process? If so, which one and how?
18. AI tool usage. Describe how you used AI tools for the Elixir implementation. What did you prompt it to do? What did the generated code get right? What did you have to fix, restructure, or explain back to the tool?

## Grading

| Component | Points | Key criteria |
|---|---|---|
| Elixir — correctness and completeness | 20 | All four processes, all messages, correct behavior |
| Elixir — stateful barber (averages) | 5 | Averages computed inside process loop, no external state |
| Go — correctness and completeness | 25 | Correct behavior, no races, no goroutine leaks |
| Go — stateful barber (averages) | 5 | State in goroutine local vars, no shared variables |
| Go — message type definitions | 5 | Explicit structs, no raw primitives on channels |
| Go — select used appropriately | 5 | Barber sleep/wake and any multi-channel waits |
| Instrumentation and closing report | 10 | Logging is useful; report fields are correct |
| Reflection — depth and specificity | 25 | Addresses all six questions with code references |

**Total: 100 points.**

Note on the no-shared-state constraint: any use of mutexes, sync.Mutex, sync.WaitGroup for state protection (as opposed to shutdown synchronization), or global variables holding process state will result in a deduction of up to 15 points from the Go section, regardless of whether the program produces correct output.

# Tips

## Start with Elixir

Even if Go feels more familiar, implement Elixir first. The actor model is cleaner in Elixir and getting the message flow right there will clarify the structure for the Go version. If your Elixir implementation is confused about message flow, your Go version will be too.

## Test with small parameters

Set total customers to 5 and waiting room seats to 2 while developing. You want to be able to read the full log output and verify every event manually before scaling up.

## The wakeup race

The most common bug in this problem is the barber missing a wakeup signal. This happens when: (1) the Barber asks for the next customer and gets none_waiting, (2) a customer arrives before the Barber has entered its sleeping state, (3) the Waiting Room sends wakeup to a Barber that isn't sleeping yet, (4) the Barber enters its sleeping state and waits forever. In Elixir this is largely avoided by the mailbox buffering messages. In Go, using a buffered channel for the Barber's mailbox (capacity ≥ 1) prevents the Waiting Room from blocking on the wakeup send, which sidesteps the race.

## On AI tools and Elixir

AI tools produce reasonable Elixir for straightforward patterns. They are less reliable on the sleeping/waking handshake and on correct request/reply patterns (making sure the right PID is passed in the right message). Treat generated code as a draft to be read and understood, not a final submission.

---

*Questions? Post to the course discussion board, email them, or bring them to office hours.*