PPC Project

The Circle of Life

Razmig Kéchichian, Hugo Reymond and Areski Himeur

1. Goal

The goal of this programming project is to design and implement a multi-process simulation in Python. This document defines the functionalities that you must implement at a minimum. Some questions are deliberately left open for you to propose your own solution. Any extensions and developments will act in your favor in the evaluation. Be rigorous and creative!

2. Presentation

2.1 User view

Consider a simple ecosystem composed of carnivorous predators, herbivorous preys and grass. Predators and preys have a liveliness indicator, called the energy, which decreases regularly during the simulation independently of the state of the individual. At any given moment, an individual is in one of two states: active or passive. Active predators and active preys feed on preys and grass, respectively, if their energy is below a given threshold *H*. Only active preys can be predated. An individual can change its state in a deterministic manner (i.e. active as soon as energy < *H*, passive once energy > *H*) or arbitrarily. Feeding decreases the populations of preys and grass. An individual may choose to reproduce if its energy is above a given threshold *R*, increasing its population. An individual dies if its energy becomes negative. Outside of intermittent drought episodes, grass grows regularly. The simulation should allow observing the development of predator and prey populations given predefined or arbitrary population sizes, attributes of individuals and environmental settings. The operator might also modify the settings of the simulation during its execution.

2.2 Technical specifications

Your implementation should involve at least 4 independent process types:

- `env`: manages the simulation environment, keeping track of populations and climate conditions,
- `predator`: simulates a predator, its attributes and behavior, several `predator` processes might be executing during the simulation with each corresponding to a distinct individual,
- `prey`: simulates a prey, its attributes and behavior, several `prey` processes might be executing during the simulation with each corresponding to a distinct individual,
- `display`: allows the operator to observe and control[1] the simulation in real-time.

Inter-process communication: The `env` process keeps track of populations in a shared memory, accessible to `predator` and `prey` processes only. The `env` process also listens on a socket, allowing `predator` and `prey` processes to join the simulation. The `display` process communicates with process `env` through a message queue. The advent of a drought episode is notified to the `env` process by a signal[2].

1 If necessary, refer to this Stack Overflow thread for non-blocking console input solutions in Python.

2 In addition to the usual mechanism of signal dispatch from independent processes, a process can schedule timers for itself at regular or arbitrary intervals. Standard Python provides two mechanisms: signal-based timers via `signal.settimer` and thread-based ones via `threading.Timer`.

3. Implementation

3.1. Design

Start with a diagram to better visualize the interactions between processes/threads. State machine diagrams can be very helpful during this stage. The following points need to be addressed and justified:

- relationships between processes (parent-child or unrelated), define a parent-child relationship only if absolutely necessary,
- data structures stored in shared memory, their types and how they are accessed,
- synchronization primitives to protect access to shared resources or to count resources,
- sockets used in communication between processes along with exchanged data, their content and order of exchange,
- message queues used in communication and messages exchanged between processes along with their types, content and order of exchange,
- signals exchanged between processes along with their types and associated handlers,
- tubes involved in pairwise process communication, if any, and their types.

Write down a Python-like pseudo-code of main algorithms and data structures for each process/thread as a guideline for your implementation.

3.2. Implementation plan

Plan your implementation and test steps. We strongly encourage you to first implement and test every process/thread separately on hard-coded data in the beginning. Then you can implement and test each pair of communicating processes/threads, again on hard-coded data in the beginning, if necessary. Only after implementing and testing each inter-process communication you can put all processes together and test the simulation. Take care of the startup and the proper shutdown of the simulation, freeing all resources. Identify possible failure scenarios and think about recovery or termination strategies. Lastly, make sure that you are able to demonstrate your implementation during a short presentation, both from the viewpoint of a user and that of a systems engineer.

4. Deadlines

17/12/2025	project is published on Moodle
29/01/2026 - 23:59	submission of project code (including a README explaining how to execute it) and report archive on Moodle, refer to organization slides for the content of the report
30/01/2026	15-minute demonstration per project with the tutor in charge of your group