Three Dogs And a Garden
Tournament of Three

Alessandro Monticelli, Jóhannes Helgi Tómasson, Jonas Stahl April 3, 2026

1 Description of the Solution

The proposed solution generalizes Kessels' two-process mutual exclusion algorithm to three processes, here represented with the names Alice, Bob and Charlie (A, B, C).

1.1 General idea

The general idea is to make the three processes compete for obtaining access to the critical section (i.e. *the garden*). The competition happens in the form of a tournament of three, A and B must compete to pass to the second level, where the winner will compete with C to gain access to the critical section.

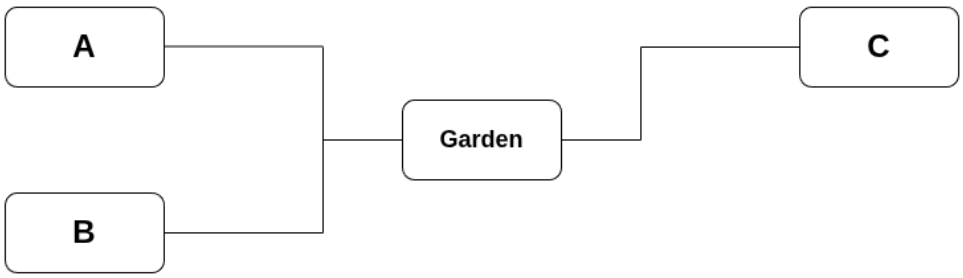

Figure 1: Tournament of three representation

1.2 Implementation

The competition is structured on two levels. At each level Kessels' algorithm determines which of the two competitor process will go to the next level, determining who will get access to *the garden*.

The processes are divided in two categories:

- The challengers: Alice and Bob
- The gatekeeper: Charlie

Flags

We define three arrays of boolean flags: `alices_flags[4]`, `bobs_flags[4]`, and `charlies_flags[3]`, to determine who gets priority for accessing the critical section. Each flag represents either the

active intent to participate in a specific round, or the will to yield priority to the opponent when a conflict occurs.

Index	Alice's Usage	Bob's Usage
0	**R1 Intent**: Signals entry into the qualifying round against Bob.	**R1 Intent**: Signals entry into the qualifying round against Alice.
1	**R1 Courtesy (Copier)**: Copies Bob's state to yield priority if they arrive together.	**R1 Courtesy (Inverter)**: Inverts Alice's state to yield priority if they arrive together.
2	**R2 Intent**: Signals victory in R1 and entry into the final round against Charlie.	**R2 Intent**: Signals victory in R1 and entry into the final round against Charlie.
3	**R2 Courtesy**: Copies Charlie's state to yield priority in the final round.	**R2 Courtesy**: Copies Charlie's state to yield priority in the final round.

11.5

Table 1: Flag definitions for the tournament challengers.

Index	Charlie's Usage
0	**Intent**: Signals desire to enter the critical section (Garden).
1	**Courtesy vs. Alice (Inverter)**: Inverts Alice's R2 flag to resolve contention if she wins Round 1.
2	**Courtesy vs. Bob (Inverter)**: Inverts Bob's R2 flag to resolve contention if he wins Round 1.

Table 2: Flag definitions for the tournament gatekeeper.

The flags are stored in shared arrays readable by all processes, but the algorithm enforces a single-writer logic. Although the memory is technically accessible to all processes, the control flow dictates that each flag index is modified solely by its assigned owner process, based on the value of its competitors' flags.

2 Proposed Pseudocode

Listing 1: Alice

```
func Alice() {
    for {
entry:
        // Round 1 Alice vs Bob
        alices_flag[0] = true // A wants to enter round 1
        alices_flag[1] = bobs_flag[1] // Same as 2 dogs 1 garden solution.

        Await(!bobs_flag[0] || alices_flag[1] != bobs_flag[1])

        // Alice wins vs Bob.
        // Round 2 Alice vs Charlie
        alices_flag[2] = true
        alices_flag[3] = !charlies_flag[1]

        Await(!charlies_flag[0] || alices_flag[3] == charlies_flag[1])
critical:
        // Alice's dog in the yard
exit:
        // Reset rounds access
        alices_flag[2] = false
        alices_flag[0] = false
    }
}
```

Listing 2: Bob

```
func Bob() {
    for {
entry:
        // Round 1 Bob vs Alice
        bobs_flag[0] = true
        bobs_flag[1] = !alices_flag[1] // Same as 2 dogs 1 garden solution.

        Await(!alices_flag[0] || bobs_flag[1] == alices_flag[1])

        // Bob wins vs Alice.
        // Round 2 Bob vs Charlie
        bobs_flag[2] = true
        bobs_flag[3] = !charlies_flag[2]

        Await(!charlies_flag[0] || bobs_flag[3] == charlies_flag[2])
critical:
        // Bob's dog is in the garden
exit:
        // Reset rounds
        bobs_flag[2] = false
        bobs_flag[0] = false
    }
}
```

Listing 3: Charlie

```
func Charlie() {
    for {
entry:
        charlies_flag[0] = true
        charlies_flag[1] = alices_flag[3]
        charlies_flag[2] = bobs_flag[3]

        Await((!alices_flag[2] || charlies_flag[1] != alices_flag[3]) &&
              (!bobs_flag[2] || charlies_flag[2] != bobs_flag[3]))
critical:
        // Charlie's dog is in the garden
exit:
        charlies_flag[0] = false
    }
}
```

3 Argument of Correctness

3.1 Mutual Exclusion

Mutual exclusion is guaranteed by the hierarchical tournament structure. The algorithm acts as a logical filter at each node of the tree:

- **Round 1:** This level prevents Alice and Bob from progressing simultaneously. The complementary logic of the wait predicates ($\neq$ for Alice, $==$ for Bob) ensures that for any static valuation of the priority flags, it is logically impossible for both conditions to be false simultaneously. Thus, at most one process can pass.
- **Round 2:** This acts as the final gate. Even if the winner of Round 1 arrives at the same time as Charlie, the same exclusion logic applies.

Since a process must win both its specific path nodes to enter the Critical Section (Garden), and each node strictly allows only one winner, mutual exclusion is maintained.

3.2 Absence of Deadlock

Deadlock is impossible because the wait conditions at every node are contradictory. For any pair of competing processes P_i and P_j at a specific node, their wait conditions depend on the comparison of their priority flags f_i and f_j:

- One process waits if $f_i = f_j$.
- The other waits if $f_i \neq f_j$.

Since f_i and f_j are boolean values, it is impossible for both conditions to be true simultaneously. Therefore, it is impossible for both processes to be stuck waiting for each other at the same time.

3.3 Absence of Starvation

Starvation is prevented by the "courtesy" mechanism (the priority flags) which enforces a strict turn-taking policy. When a process P waits, it has already signaled its intent and set its courtesy flag to yield to the opponent Q. While P waits, its state does not change. If P is waiting, Q is in the critical secton, or winning the round. When Q exits, it resets its intent flag (e.g. `alices_flag[0]=false`), immediately releasing P from the wait loop. At last, if Q tries to immediately re-enter, it will have to set its flags based on P state, since P was waiting, its flags didn't change, and the change of state will force Q to yield priority to P Thus, a process can be overtaken at most once per round per competitor, guaranteeing eventual entry.