1. Distributed "laxlock" service

Informal description: A "laxlock" is like a lock except that it can be held simultaneously by upto `N` threads, where `N` is a positive integer parameter. So you can view it as a collection of `N` "tokens". A thread calls `acq()` to acquire a token and `rel()` to release the token it holds. For convenience, we say a thread is "hungry" if it is in `acq`), "eating" if it holds a permit, and "thinking" otherwise.

A distributed laxlock is one that can be accessed at different addresses. A thread can acquire a token from one address and release it at another.

Service program: Define a distributed laxlock service with addresses `0` and `1`, At each address `j`, there are input functions v_j`.acq()` and v_j`.rel()`, where v_j is the sid of the local access system. A skeleton is provided below:

```
service Laxlock(int N) {
  ic {N ≥ 1}
  ...
  Map v ← map([0,sid()], [1,sid()])
  return v

  v[0..1 j].acq() {
    ic {...}
    ...
    oc {...}
    ...
  }

  v[0..1 j].rel() {
    ic {...}
    ...
    oc {...}
    ...
  }

  progress assumption {...}         // should imply weak fairness at each address, but using only leads-to assertions
}
```

To do:

Supply the missing parts (indicated by "...").

2. A distributed laxlock implementation attempt

The program below is an attempt at implementing a distributed laxlock. Roughly speaking, the available tokens are divided between the addresses. A local thread attempts to balance the numbers of tokens across addresses.

The goal of this exercise is to get you to do an assertional proof of safety. The program has a global "auxiliary" variable, `eating`, indicating the set of eating threads. We would not need this if we had the desired-service program.

For your analysis, use the effective atomicity indicated by the •'s (it is the same as that provided by the awaits).

```
program LaxDist(N) {
  Bag eating ← []
  Map v
  v[0] ← startSystem(Lax(0))
  v[1] ← startSystem(Lax(1))
  w ← startSystem(Adjuster(v[0],v[1]))
} // LaxDist
```

```
program Adjuster(Sid v0,Sid v1) {
  int bal ← 0
  int y ← 0
  t ← startThread(f())

  function f():
    while (true)
      [bal,y] ← v0.adjust(bal,y)
      [bal,y] ← v1.adjust(bal,y)
      // sleep a bit

  atomicity assumption { }
  progress assumption {wfair for all threads}
} // Adjuster
```

```
program Lax(0..1 i) {
  int x ← if (i=0) N else 0
  return mysid

  input mysid.acq():
    ia {mytid not in eating}
• await (x>0)
      x−−
      eating.add(mytid)
      return

  input mysid.rel():
    ia {mytid in eating}
• await (true)
      x++
      eating.remove(mytid)
      return

  input mysid.adjust(int bal, int y):
• await (true)
      x ← x+bal
      if (x≥y+2)
        tmp ← (x−y)/2  // integer division
        x ← x−tmp
        return [tmp,x]
      else
        return [0,x]

  atomicity assumption {await}
  progress assumption {wfair for all threads}
} // Lax
```

To do:

Does `LaxDist(N)` satisfy *Inv* P, where

P : `eating.size ≤ N`

If you answer yes, give a predicate, say B, such that

- B is established by the initial step.
- B is unconditionally preserved by every other atomic step.
- $B \Rightarrow P$ holds.

If you answer no, give a finite allowed evolution ending in a state where P does not hold.

Don't give any other explanations.