Shared integer service

Informal description: A service consisting of an integer, say `v`, that can be accessed via a function `f(x)`, where `x` is a non-zero integer (positive or negative). Multiple calls (by different threads) can be simultaneously ongoing. The call adds `x` to `v` and returns the new value of `x` only if non-negative, blocking if the value is negative (waiting for another thread to make `v` non-negative).

A blocked thread eventually returns if `v` is continuously non-negative.

Formalization 1

Here is a service program that formalizes the above informal description in a straightforward way.

```
service B1() {
  int v ← 0

  input f(int x):
    // input part
    ic {x≠0}
    // output part
    oc {v+x ≥ 0}
    v ← v+x
    return v

  progress:
    ((thread t at oc) and (v+x ≥ 0)) leads-to
       ((t not at oc) or (v+x < 0))
}
```

This formalization of the informal English description is not conducive to parallelism in implementations. It requires an implementation to funnel all inputs to one location.

Question: Can the update to `v` be done in the input part. If so, would it be the same service?

Formalization 2

We now come up with a service program that allows for more parallelism in implementations. Specifically, we will make use of the notion of serializability (from databases):

- Let the **global history** at any point be the sequence of calls and returns so far.
- For any user, let its **local history** be the sequence of its calls and its returns.
- The global history is **serial** if at most one call is ongoing at any time (i.e., each return is immediately preceded by its call) and each value returned is the sum of all previous call values.
- The global history is **serializable** if it can be reordered to a sequence that is serial and preserves each user's local history. (Equivalently, the global history is a merge of all its local histories.)
- The service can return any value such that the global history is serializable.

Now to cast the above as a service program.

Introduce a variable `gh` that records the sequence of call and return entries. A **call entry** is a tuple `[CALL,x,j]`, where `CALL` is a constant, `x` is the parameter of the call, and `j` is the caller's tid (thread id). A **return entry** is a tuple `[RET,y,j]`, where `RET` is a constant, `y` is the value returned, and `j` is the caller's tid.

```
service B2() {
  gh ← []  // global history
  constants CALL,RET
  type Hstry = “sequence of call entries and return entries”
  // helper functions
  bool serial(Hstry α) {“return true iff α is serial”}
  Seq lh(j, Hstry α) {“return j's local history of α”}
  bool valid1(Hstry α) {“return true iff α is serializable”}

  input f(int x):
    // input part
    ic{x ≠ 0}
    gh.append([CALL,x,mytid])
    // output part
    output(int y)
      oc{valid1(gh ∘ [RET,y,mytid]) and y ≥ 0}
      gh.append([RET,y,mytid])
      return y

  progress:
    // t.oc is the output condition for thread t
    ((thread t at oc) and (t.oc)) leads-to ((t not at oc) or (not t.oc))
}
```

Formalization 3

Service `B2` allows a value to be returned only if all values that are used to make that value have already returned. This makes sense when the operations are database transactions, because until a transaction ends (commits), the service must allow for the possibility that it will abort. (So if transaction p reads from transaction q, then the service cannot end p before ending q; otherwise, q may abort after p's return.)

But in our service, the operations are simple additions; there are no aborts. So it is ok to return a value p even if that value depends on a value q that has not yet been returned provided q will eventually be returned.

Homework 1: Define service `B3()` that allows such parallelism.