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 (until another thread makes \( v \) non-negative).

Regarding progress, a blocked thread eventually returns if \( v \) is continuously non-negative.

**Service program B1**

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

```plaintext
service B1() {
    int v ← 0
    return mysid

    input f(int x):
        // input part
        ic {x ≠ 0} // abort if ic does not hold
        // output part
        output (y) // return any y that satisfies the oc
        oc {y = v+x ≥ 0}
        v ← v+x
        return y

    progress assumption:
    ((thread t at oc) and (v + t.x ≥ 0)) leads-to
    ((t not at oc) or (v + t.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?
Service program B2

We now come up with a service program that perhaps allows implementations with more parallelism. Specifically, we will adopt the notion of serializability (from database literature):

- 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.
- A global history is **serial** if each return is immediately preceded by its call and each value returned is the sum of all previous call values. (Note: this allows a suffix of ongoing calls.)
- 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 global history variable \( gh \) that is a sequence of call and return entries. A **call entry** is a tuple \([\text{CALL}, x, j]\), where \( \text{CALL} \) 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 \([\text{RET}, y, j]\), where \( \text{RET} \) is a constant, \( y \) is the value returned, and \( j \) is the caller’s tid.

```plaintext
service B2() {
    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 H(Tid j, Hstry α) (“return j’s local history of α”)
    bool valid1(Hstry α) (“return true iff α is serializable”)
    // variable: global history
    Hstry gh ← []
    return mysid

    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} // ◦: concatenation
        gh.append([RET, y, mytid])
    return y

    progress assumption:
        // 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))
    }
```

**Question:** Is B2 more general than B1? Can you come up with an io sequence that B2 can generate but B1 cannot?
**Service program B3**

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 (without any help from the environment). This is accomodated in the following service program.

```plaintext
service B3() {
    // as in B2()
    CALL, RET, Hstry, serial(...) [...], lh(...) [...]
    Hstry gh ← []
    return mysid

    bool valid2(Hstry α) {
        "return true iff α can be extended with returns to a sequence that is serializable"
        
        input f(int x):
            // input part
            ic {x ≠ 0}
            gh.append([CALL, x, mytid])
        
        output(int y)
            oc {valid2(gh ◦ [[RET, y, mytid]]) and y ≥ 0}
            gh.append([RET, y, mytid])
            return y

    progress assumption:
        // 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))
    }
```
Implementation A1

Here is a program A1() intended to implement B1, B2 and/or B3.

```
program A1() {
    int v ← 0
    return mysid

    input f(int x):
    a1: await (v + x ⩾ 0)
        v ← v+x
        return v

    progress assumption:
        weak fairness for all threads
}
```

Note: \texttt{await} (B) S means do S only if B holds, and do it atomically with the evaluation of B. Here, B is a predicate (no side effect), and S is a non-blocking update.
Does $A_1$ implement $B_1$

Because $A_1$ and $B_1$ are almost identical, this obviously holds, but we’ll go through the steps anyway.

In terms of evolutions, $A_1$ implements $B_1$ if

- **(Safety)** for every finite evolution $x$ of $A_1$ that is safe wrt $B_1$
  - if $A_1$ can output $f$ at the end of $x$ then $x \circ [f]$ is safe wrt $B_1$
  - any step that $A_1$ can do at the end of $x$ is fault-free
  - for any input $f$, if $x \circ [f]$ is safe wrt $B_1$ then $A_1$ can accept $f$

- **(Progress)** for every evolution $x$ of $A_1$ that is safe wrt $B_1$
  - if $x$ satisfies $A_1$’s progress assumption, $x$ is complete wrt $B_1$

To state this in terms of programs $A_1$ and $B_1$, first define $B_1$’s inverse, say $\overline{B_1}$.

```
service $\overline{B_1}$(Sid p) {
  int v ← 0
  return mysid
}
```

```
output doF(int x):
  oc {x ≠ 0} // create a thread to execute output part when oc holds
  y ← p.f(x) // output part ends at ‘’, input part begins there
  ic {y = v+x ≥ 0} // thread aborts if not ic, else executes input part and ends
  v ← v+x
```

Progress condition: ‘‘same as $B_1$ progress assumption’’

Next define a program, say $Z$, of an $A_1$ system and a $B_1$ system concurrently executing.

```
program Z() {
  Sid p ← startSystem(A1())
  Sid q ← startSystem($\overline{B_1}$(p))
}
```

$A_1$ implements $B_1$ if (every evolution of) $Z$ satisfies the following

- **(Safety)** if a thread is at doF.ic then doF.ic (i.e., its predicate) holds
- **(Progress)** $q$’s progress condition

To express the above in terms of assertions, first identify code chunks of $Z$ that can be treated as atomic.

- initial step: $A_1$.main; $B_1$.main.
- call step: from $q$.doF(x) (including thread creation) to $p$.a1
- return step: from $p$.a1 to end $p$.doF(x) (including thread termination)

Then it suffices if $Z$ satisfies the following assertions:

$P_1 : Inv ((thrd t at p.a1) and (p.v+t.x ≥ 0)) \Rightarrow (q.v+t.x ≥ 0)$

$P_2 : ((thrd t at p.a1) and (q.v+t.x ≥ 0)) \ leads-to ((thrd t not at p.a1) or (q.v+t.x < 0))$

For an assertional proof of $P_1$, we need to come up with a predicate that implies $P_1$’s predicate, is established by the initial step, and unconditionally preserved by every other step (i.e., call step and return step). Here is such a predicate:

$Q_1 : p.v = q.v$

$P_2$ follows from $Inv Q_1$ and $p$’s progress assumption (weak fairness). (We will see proof rules for progress later.)
Does A1 implement B3

We proceed as before, defining programs B3 (B3’s inverse) and Z.

```
service B3(Sid p) {
  // as in B3()
  CALL, RET, Hstry, serial(.) [...], lh(.) [...]
  Hstry gh ← []
  return mysid

  output doF(int x):
    oc {x ≠ 0} // create a thread to execute output part when oc holds
    y ← p.f(x) // output part ends at ‘←’, input part begins there
    ic {valid2(gh ◦ [[RET, y, mytid]]) and y ≥ 0}
    gh.append([[RET, y, mytid]])

    progress condition: ‘same as B3 progress assumption’
}

program Z() {
  Sid p ← startSystem(A1())
  Sid q ← startSystem(B3(p))
}
```

Atomic steps:
- call step: from q.doF(x) (including thread creation) to p.a1
- return step: from p.a1 to end p.doF(x) (including thread termination)

It suffices if Z satisfies the following assertions:

```
P1 : Inv ((thrd t at p.a1) and (y = p.v+t.x ≥ 0)) ⇒
  (valid2(q.gh ◦ [[RET, y, t]]) and y ≥ 0)
```

```
P2 : ((t at p.a1) and (p.v+t.x ≥ 0) and valid2(q.gh ◦ [[RET, p.v+t.x, t]]) leads-to
  ((t not at p.a1) or not ((p.v+t.x ≥ 0) and valid2(q.gh ◦ [[RET, p.v+t.x, t]])))
```

Proving P1

The key to proving P1 is to identify a serial order that A1 enforces. The natural candidate is the order in which A1 updates v. Augment A1 with auxiliary variable sh and function ongng as follows:

```
• sh: serial history
  - initialize sh to empty
  - do “sh ← sh ◦ [[CALL, x, mytid], [RET, v, mytid]]” just before “return v”
• ongng: subsequence of call entries in gh whose calls are still ongoing
```

For an assertional proof of P1, the conjunction of the following is an “adequate” predicate.

```
Q1 : serial(sh ◦ ongng)
Q2 : for all (Tid t: 1h(t, sh ◦ ongng) = 1h(t, gh))
Q3 : sh = [] or sh.last.value = v ≥ 0
```

Note: Q1–Q2 imply valid2(gh)
Here are more details about how $Q_1$–$Q_3$ is adequate.

Initial step:
- empties gh and sh, and zeroes $v$
- establishes $Q_1$, $Q_2$, $Q_3$

Call step $doF(x)$ by thread $t$:
- no change to sh
- appends $[CALL,x,t]$ to gh (and hence to output of ongng)
- establishes $Q_1$ given $Q_1$, i.e. unconditionally preserves $Q_1$  // Hoare-triple: $\{Q_1\}$ call step $\{Q_1\}$
- unconditionally preserves $Q_2$
- unconditionally preserves $Q_3$  // special case: nothing in $Q_3$ changes

Return step $doF(x)$ by thread $t$:
- appends $[CALL,x,t],[RET,v+t,x,t]$ to sh
- adds $x$ to $v$; appends $[RET,v,t]$ to gh (which removes $[CALL,x,t]$ from ongng)
- establishes $Q_1$ given $Q_1$ and $Q_3$ (why is $Q_3$ needed?)  // $\{Q_1,Q_3\}$ return step $\{Q_1\}$
- unconditionally preserves $Q_2$  // step affects $Q_2$ only for thread $t$
- unconditionally establishes $Q_3$  // $\{true\}$ return step $\{Q_3\}$

Proving $P_2$

Does it suffice to prove that $Z$ satisfies the following?

$P_3 : ((t \text{ at } p.a_1) \land (p.v + t.x \geq 0) \text{ leads-to}
((t \text{ not at } p.a_1) \lor \neg ((p.v + t.x \geq 0))))$

If so, we are done because $P_3$ follows from $A_1$’s progress assumption.