MODULE *backpressure*

EXTENDS *FiniteSets*, *Integers*, *Sequences*, *TLC*

$Null \triangleq 0$
$Cowns \triangleq 1 \,..\, 3$ # *TODO*: 4
$MaxMessageCount \triangleq 3$ # *TODO*: 4
$MaxMessageSize \triangleq 3$
$OverloadThreshold \triangleq 2$
$PriorityLevels \triangleq \{-1, 0, 1\}$

$Pick(s) \triangleq$ CHOOSE $x \in s$: TRUE
$Min(s) \triangleq$ CHOOSE $x \in s : \forall\, y \in s \setminus \{x\} : y > x$
$Max(s) \triangleq$ CHOOSE $x \in s : \forall\, y \in s \setminus \{x\} : y < x$
$Range(f) \triangleq \{f[x] : x \in$ DOMAIN $f\}$
$Subsets(s, min, max) \triangleq$
  $\{x \in$ SUBSET $s : (Cardinality(x) \geq min) \land (Cardinality(x) \leq max)\}$
RECURSIVE $Concat(\_)$
$Concat(s) \triangleq$ IF $s = \{\}$ THEN $\langle\rangle$ ELSE LET $x \triangleq Pick(s)$IN $\quad x \circ Concat(s \setminus \{x\})$

VARIABLES *fuel*, *queue*, *scheduled*, *running*, *mutor*, *priority*, *blocker*
$vars \triangleq \langle fuel, queue, scheduled, running, mutor, priority, blocker\rangle$

$Messages \triangleq$ UNION $\{Range(queue[c]) : c \in Cowns\}$

$Normal(c) \triangleq priority[c] = 0$
$Prioritized(c) \triangleq priority[c] = 1$
$Muted(c) \triangleq priority[c] = -1$

$EmptyQueue(c) \triangleq Len(queue[c]) = 0$
$Overloaded(c) \triangleq Len(queue[c]) \geq OverloadThreshold$
$Enqueue(c, m) \triangleq c :> Append(queue[c], m)$
$Dequeue(c) \triangleq c :> Tail(queue[c])$

RECURSIVE $Blockers(\_)$
$Blockers(c) \triangleq$
  IF $blocker[c] = Null$ THEN $\{\}$
  ELSE $\{blocker[c]\} \cup Blockers(blocker[c])$

TODO: apply to all blockers
$Unblock(c) \triangleq c :> (Muted(c) \lor scheduled[c])$

$Running(c) \triangleq \exists\, k \in Cowns : running[k] \land c \in Head(queue[k])$

$AcquiredBy(a, b) \triangleq$
  $\land\, a < b$
  $\land \exists\, m \in Range(queue[b]) \;\; : (a \in m) \land (b \in m)$
$Acquired(c) \triangleq \exists\, k \in Cowns : AcquiredBy(c, k)$

$MutedBy(a,\ b) \triangleq$
$\land\ Muted(a)$
$\land\ \exists\, m \in Range(queue[b]) : (b \notin m) \land (a \in m)$

$Init \triangleq$
$\land\ fuel = MaxMessageCount$
$\land\ queue = [c \in Cowns \mapsto \langle\{c\}\rangle]$
$\land\ scheduled = [c \in Cowns \mapsto \text{TRUE}]$
$\land\ running = [c \in Cowns \mapsto \text{FALSE}]$
$\land\ mutor = [c \in Cowns \mapsto Null]$
$\land\ priority = [c \in Cowns \mapsto 0]$
$\land\ blocker = [c \in Cowns \mapsto Null]$

$Terminating \triangleq$
$\land\ \forall\, c \in Cowns : EmptyQueue(c)$
———
$\land$ UNCHANGED $vars$

$ExternalReceive(cown) \triangleq$
$\land\ fuel > 0$
———
$\land$ UNCHANGED $\langle scheduled,\ running,\ mutor,\ priority,\ blocker\rangle$
$\land\ fuel' = fuel - 1$
\# Receive a message from an external source
$\land\ \exists\, others \in Subsets(\{c \in Cowns : c > cown\},\ 0,\ MaxMessageSize - 1) :$
$queue' = Enqueue(cown,\ \{cown\} \cup others)\ @@\ queue$

$Acquire(cown) \triangleq$
$\land\ scheduled[cown]$
$\land\ \neg running[cown]$
$\land\ \neg EmptyQueue(cown)$
$\land\ cown \in Head(queue[cown])$
$\land\ cown < Max(Head(queue[cown]))$
———
$\land$ UNCHANGED $\langle fuel,\ running,\ mutor\rangle$
\# *Unschedule* and forward the message to the next *cown*.
$\land$ LET
$msg \triangleq Head(queue[cown])$
$next \triangleq Min(\{c \in msg : c > cown\})$
IN
$\land\ queue' = Enqueue(next,\ msg)\ @@\ Dequeue(cown)\ @@\ queue$
$\land\ blocker' = (cown :> next)\ @@\ blocker$
\# *Prioritize* this *cown* and next if either are prioritized. *Unmute* next.
$\land$ IF $\exists\, c \in \{cown,\ next\} : Prioritized(c)$ THEN
$\land\ priority' = (next :> 1)\ @@\ priority$
$\land\ scheduled' = Unblock(next)\ @@\ (cown :> \text{FALSE})\ @@\ scheduled$

```
        ELSE
        ∧ UNCHANGED ⟨priority⟩
        ∧ scheduled' = (cown :> FALSE) @@ scheduled

Unmute(cown) ≜
  ∧ scheduled[cown]
  ∧ ¬running[cown]
  ∧ ¬EmptyQueue(cown)
  ∧ cown ∉ Head(queue[cown])
   ———
  ∧ UNCHANGED ⟨fuel, running, mutor, blocker⟩
   # Remove message from queue.
  ∧ queue' = Dequeue(cown) @@ queue
   # Reschedule muted cowns.
  ∧ LET muted ≜ {c ∈ Head(queue[cown]) : Muted(c)} IN
     ∧ priority' = [c ∈ muted ↦ 0] @@ priority
     ∧ scheduled' = [c ∈ muted ↦ TRUE] @@ scheduled

PreRun(cown) ≜
  ∧ scheduled[cown]
  ∧ ¬running[cown]
  ∧ ¬EmptyQueue(cown)
  ∧ cown = Max(Head(queue[cown]))
   ———
  ∧ UNCHANGED ⟨fuel, queue, scheduled, mutor, priority⟩
   # Set max cown in current message to running
  ∧ running' = (cown :> TRUE) @@ running
  ∧ blocker' = [c ∈ Head(queue[cown]) ↦ Null] @@ blocker

Send(cown) ≜
  ∧ running[cown]
  ∧ fuel > 0
   # ——
  ∧ UNCHANGED ⟨running, blocker⟩
  ∧ fuel' = fuel − 1
  ∧ ∃ receivers ∈ Subsets(Cowns, 1, MaxMessageSize) :
     LET
       next ≜ Min(receivers)
       senders ≜ Head(queue[cown])
       mutors ≜ {c ∈ receivers : Overloaded(c)}
     IN
      # Place message for receivers in the first receiver's queue.
      ∧ queue' = Enqueue(next, receivers) @@ queue
      ∧ IF (∃ c ∈ receivers : Prioritized(c)) ∨ Overloaded(next) THEN
          # Prioritize next.
          ∧ priority' = (next :> 1) @@ priority
```

```
        # Reschdeule next if it was muted.
        ∧ scheduled' = Unblock(next) @@ scheduled
        # Set mutor if any receiver is overloaded and there are no receivers in the set of senders.
        ∧ IF
            ∧ mutors ≠ {}
            ∧ mutor[cown] = Null
            ∧ (senders ∩ receivers) = {}
          THEN mutor' = (cown :> Min(mutors)) @@ mutor
          ELSE UNCHANGED ⟨mutor⟩
        ELSE
        ∧ UNCHANGED ⟨scheduled, mutor, priority⟩

PostRun(cown) ≜
  ∧ running[cown]
  ———
  ∧ UNCHANGED ⟨fuel, blocker⟩
  ∧ running' = (cown :> FALSE) @@ running
  ∧ mutor' = (cown :> Null) @@ mutor
  ∧ LET msg ≜ Head(queue[cown]) IN
      # Mute if mutor is set and no running cowns are overloaded.
    ∧ IF (mutor[cown] ≠ Null) ∧ (∀ c ∈ msg : ¬Overloaded(c)) THEN
        ∧ priority' = [c ∈ msg ↦ −1] @@ priority
        ∧ scheduled' = [c ∈ msg ↦ FALSE] @@ scheduled
        # Send unmute message to mutor
        ∧ queue' = Enqueue(mutor[cown], msg) @@ Dequeue(cown) @@ queue
        ELSE
        ∧ UNCHANGED ⟨priority⟩
        ∧ scheduled' = [c ∈ msg ↦ TRUE] @@ scheduled
        ∧ queue' = Dequeue(cown) @@ queue

RunStep(cown) ≜
    ∨ ExternalReceive(cown) \ *# Very expensive check
  ∨ Acquire(cown)
  ∨ Unmute(cown)
  ∨ PreRun(cown)
  ∨ Send(cown)
  ∨ PostRun(cown)

Next ≜ ∃ c ∈ Cowns : RunStep(c)

Spec ≜
  ∧ Init
  ∧ □[Next ∨ Terminating]_vars
  ∧ ∀ c ∈ Cowns : WF_vars(RunStep(c))

 # Properties
```

```
 # Ensure that the termination condition is reached by the model.
Termination ≜ ◇□(∀ c ∈ Cowns : EmptyQueue(c))

 # Invariants

 # Ensure that the model produces finite messages.
MessageLimit ≜ Cardinality(Messages) ≤ (Cardinality(Cowns) + MaxMessageCount)

 # Cowns are acquired by one running message at a time.
UniqueAcquisition ≜
  LET msgs ≜ Concat({⟨Head(queue[c])⟩ : c ∈ {k ∈ Cowns : running[k]}})
  IN  Cardinality(Range(msgs)) = Len(msgs)

 # Each queue has at most one token message.
LoneToken ≜ ∀ c ∈ Cowns : Len(SelectSeq(queue[c], LAMBDA m : m = {})) ≤ 1

 # A running cown must be scheduled and be the max cown in the message at the head of its queue.
RunningImplication ≜ ∀ c ∈ Cowns : running[c] ⇒
  ∧ scheduled[c]
  ∧ c = Max(Head(queue[c]))
  ∧ ∀ k ∈ Head(queue[c]) : (k < c) ⇒ AcquiredBy(k, c)

 # An acquired cown is not scheduled.
AcquiredImplication ≜ ∀ c ∈ Cowns : Acquired(c) ⇒
  ∧ ¬scheduled[c]

 # A muted cown is not scheduled or running.
MutedImplication ≜ ∀ c ∈ Cowns : Muted(c) ≡
  ∧ ∃ k ∈ Cowns : MutedBy(c, k)
  ∧ ¬scheduled[c]
  ∧ ¬Running(c)

 # A muted cown exists in an unmute message in the queue of at least one mutor.
MutedInUnmuteMsg ≜
  ∀ m ∈ {c ∈ Cowns : Muted(c)} :
    Cardinality({c ∈ Cowns : MutedBy(m, c)}) > 0

 # A cown may be acqiured by at most one message.
AcquiredOnce ≜
  ∀ a ∈ {c ∈ Cowns : Acquired(c)} :
    Cardinality({c ∈ Cowns : AcquiredBy(a, c)}) = 1

 # An acquired cown is either acquired by a cown in its blocker set or it is running.
AcquiredByBlocker ≜ ∀ ⟨a, b⟩ ∈ Cowns × Cowns :
  AcquiredBy(a, b) ⇒ b ∈ Blockers(a) ∨ Running(a)

 # A prioritized cown is not acquired by a muted cown.
PrioritizedNotAcquiredByMuted ≜ ∀ ⟨o, m⟩ ∈ Cowns × Cowns :
```

$$Prioritized(o) \land Muted(m) \Rightarrow \neg AcquiredBy(o,\ m)$$

```
(*
\* https://github.com/tlaplus/Examples/blob/master/specifications/TransitiveClosure/TransitiveClosure.tla#L114

TC(R) ≜
 LET
   S ≜ {r[1] : r ∈ R} ∪ {r[2] : r ∈ R}
   RECURSIVE TCR(_)
   TCR(T) ≜
    IF T = {} THEN R
     ELSE
      LET
        r ≜ CHOOSE s ∈ T: TRUE
        RR ≜ TCR(T \ {r})
      IN
        RR ∪ {⟨s, t⟩ ∈ S × S : ⟨s, r⟩ ∈ RR ∧ ⟨r, t⟩ ∈ RR}
 IN
   TCR(S)

CylcicTransitiveClosure(R(_, _)) ≜
 LET s ≜ {⟨a, b⟩ ∈ Cowns × Cowns : R(a, b)}
 IN  ∃ c ∈ Cowns: ⟨c, c⟩ ∈ TC(s)
*)
```