High Level Design and Detailed Design

1. Overview

This solution provides a shared memory based IPC system that enables no-copy message communication and cover the unexpected stop of processes as well. This library is supposed to run in Linux.

The system is composed by three elements: process proxy, SHM and a monitor.

Process means thread in Linux in this document. The process proxy defines behavior of application processes in communication. Each process is assigned a unique process id in initiation by monitor and it would be the ID of cells hold by this process. Refer to section 2 for details. A process can hold more than one receiving queues but one queue belongs to only one process. The process has no sending queue. All messages are pushed into destination receiving queue directly. No subsystem of message routing descripted in this document. When a process died, all allocated resources are collected and recycled by the monitor.

The SHM contains all memory to be transferred. The memory provided to application processes is managed in cell. And each cell has one owner at any time. When a cell had been allocated, the owner is the process; when a cell had been released, the owner is the monitor; when a cell had been transferred, the ownership is transferred from sender process to receiver process. The owner is the critical for garbage collection.

There is a monitor per OS besides all application processes. The monitor monitors death of processes and executes garbage collection thereafter. The monitor works as a daemon and has the same lifecycle as the whole system.

A process may crash in between the ownership transfer, ownership of some cells are ambiguous. Then a transient layer is introduced in this system. The transient layer includes all the handlers of these suspect cells and is exposed to the monitor so that it is able to discover the real owner by special but costly treatment. The corresponding cases are allocation/release/transfer.

The following figure displays the overview:

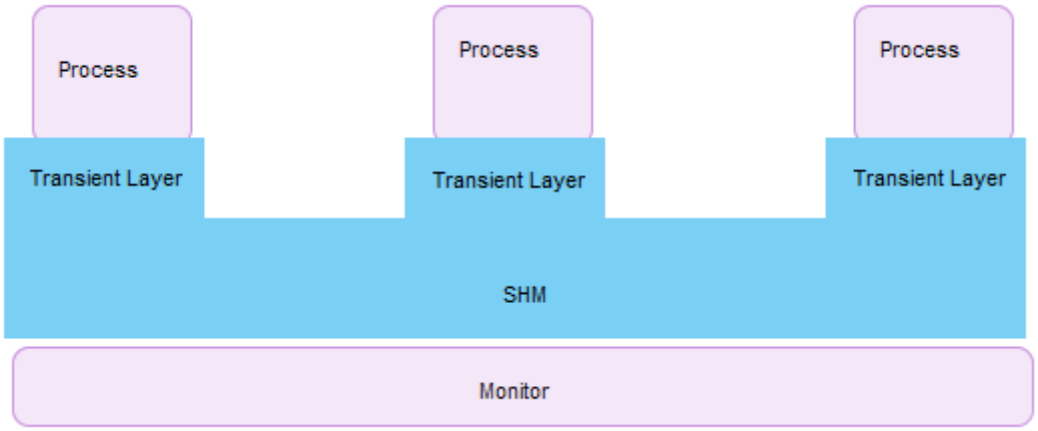

Figure1: Overview

2. Structure

The figure2 displays the structure of the system.

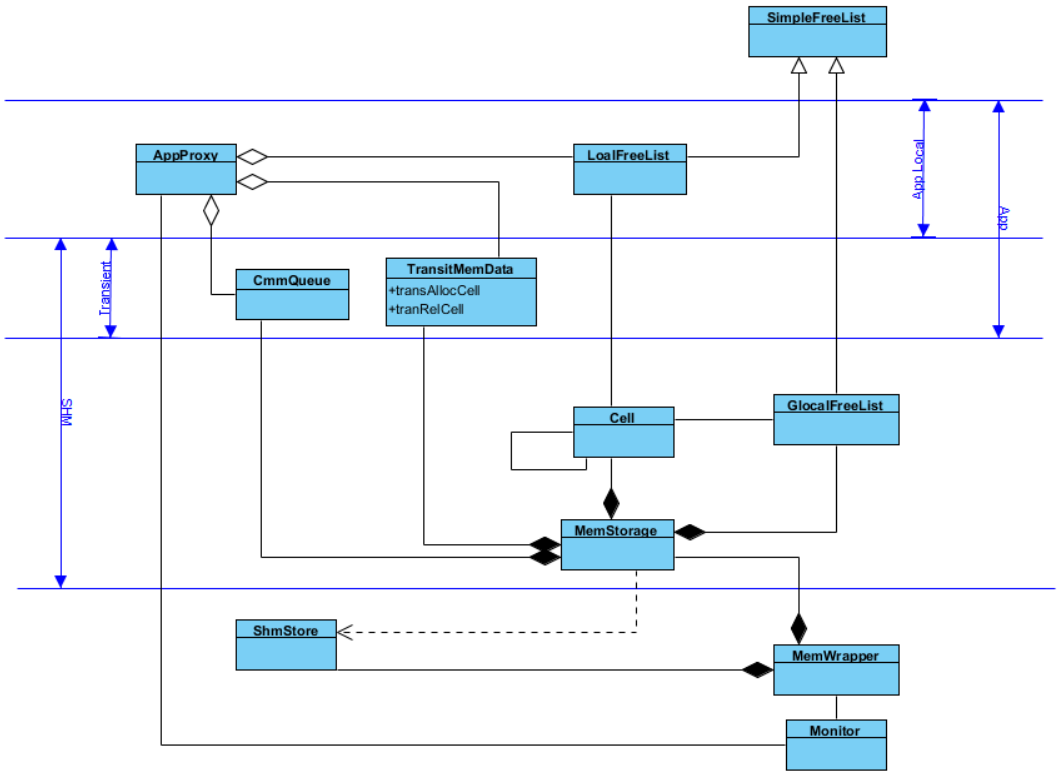

Figure2: Structure

3. Algorithms Overview

In this system, a principle is followed: make the process operations as simple as possible and leave the expensive maintenance to the monitor.

The algorithm is based on following ideas

- Atomic operations, e.g. those provided by GLIBC library [1].
- Non-blocking queue [2].
- Cell ownership.
- Transient layer.
- Garbage collector.
- Cells are managed in list of arrays to reduce conflict on global free list. The children and the leader of one Block are guaranteed to have the same owner.

Following sections will give detailed description of the implementation.

4. Allocation and Release

The figure3 gives layout of memory in a snapshot view:

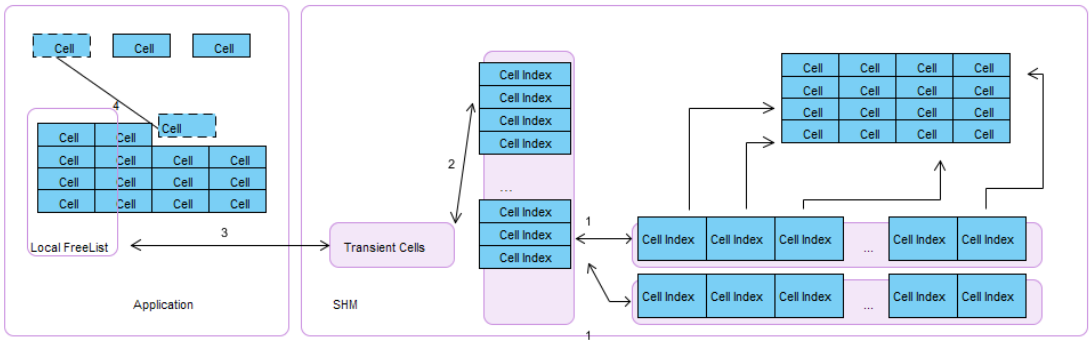

Figure3: Memory Layout

1) The global free list has been initialized as list of arrays. The array, named Block, has fixed size. Indexes of all cells had been pushed into the global free list. The owner is the Monitor (FREE).
2) In allocation and release, cells are transmitted into transient cells by unit of block. The ownership is in transfer.
3) When the transient cell pushed/popped successfully, they are assigned to local free list. Local free list also managed in unit of block. The owner is the application.
4) In application allocation and release, memory is managed in unit of cell. The owner is the application.

To reduce conflict on global free list, local free list is managed by 2 layers, i.e., list of arrays. And to guarantee that the sibling and the leader cell have the same ownership. The global list follows the same layout of local free list. To be mentioned that, the local free list does not lay in SHM. That is, local free list is made local to reduce the interference between application processes.

4.2Allocation

The figure4 shows details of allocation algorithm.

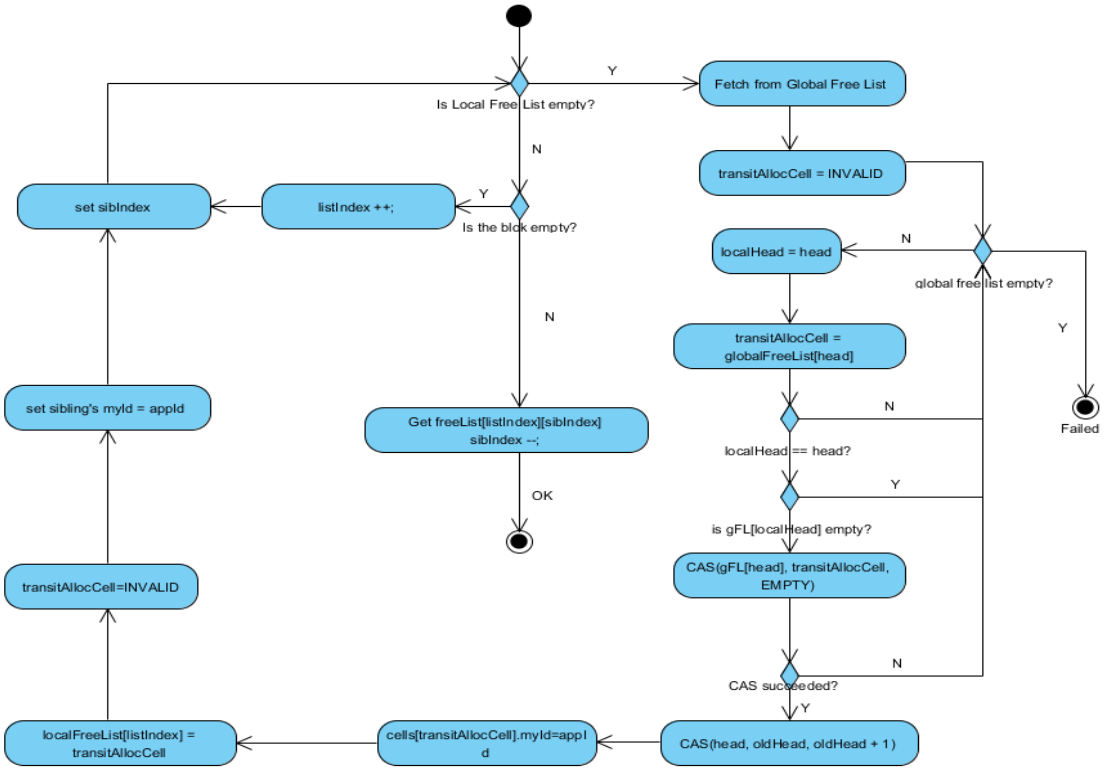

Figure4: Allocation

Application try to fetch a cell from local free list firstly, and then turn to global free list when the local pool is empty. As declared above, ownership transfer happens in transient allocate cell. If the application crashed in global free list allocation, it has been left to Monitor to distinguish if the cell had been removed from the list or not.

4.2Release

Release is similar to allocation. Set the transient free cell before remove the leader from local free list. Set the sibling's ownership before globally release as the sibling's ownership is always align to the leader. Then execute release on transient release cell. Set the transient release cell as invalid to indicate the release has been finished successfully.

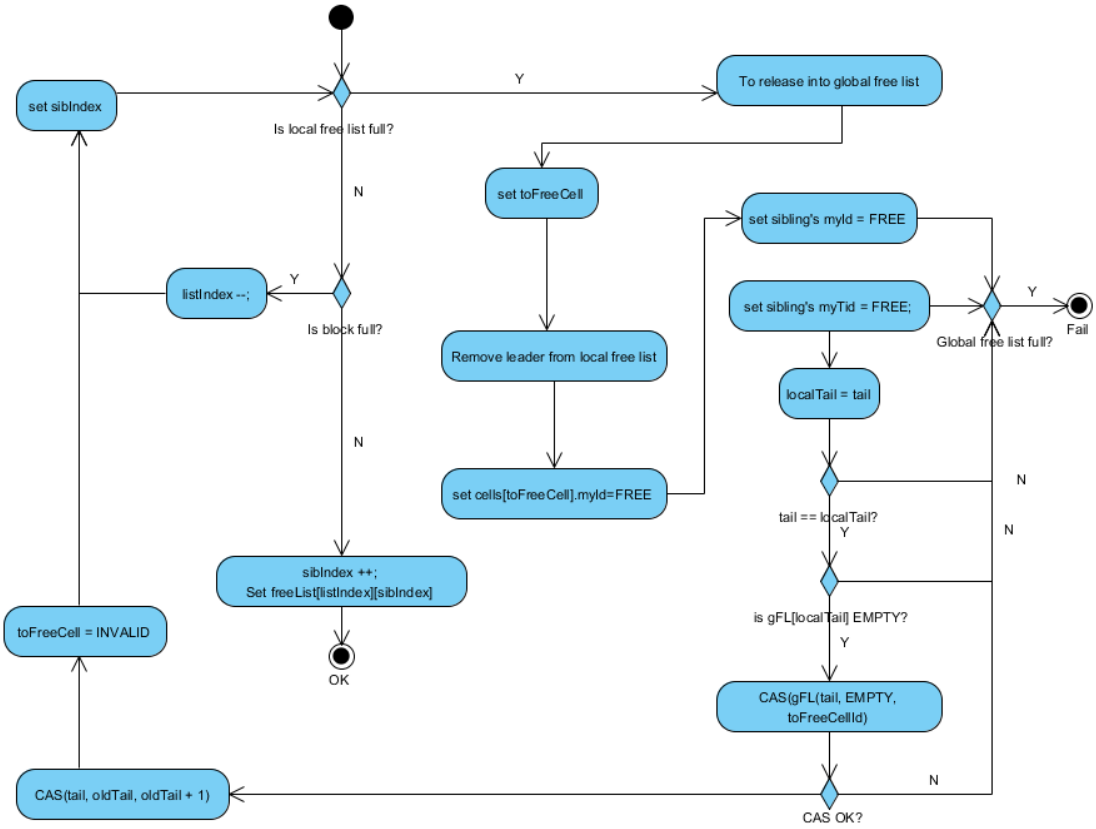

Figure5: Release

5. Communication

For a cell in sending, both self-ID and destination ID has been set before pushed into the destination queue to indicate it's a flying cell. For a cell in receiving, with assumption of single-threaded receiver, the self-ID is set before pop. The following figure shows the details.

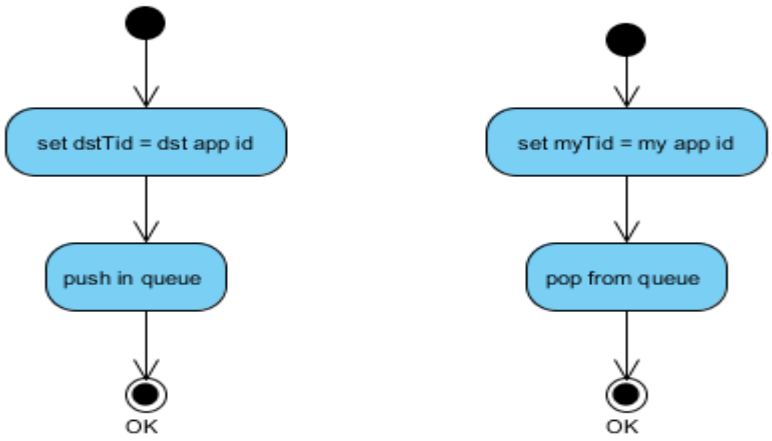

Figure6: Communication

6. Garbage Collector

When a process died, the garbage is collected by monitor. The monitor would never been restarted when there were living processes. That is, when monitor dead, the system dead.

The monitor would be notified with the application ID when a process dead. The sequence of collection as following figure shows. Flying cells are collected at first.

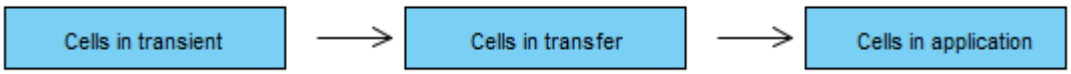

Figure7: Garbage Collection Sequence

6.1Transient Cells Collection

To find out if a transient cell longs to application or monitor before crash, one bit of the self TID is extracted to indicate if the cell is suspect, named dirty bit. Figure8 shows the structure:

Figure8: Dirty Bit

Cells are flying as the figure9 shows.

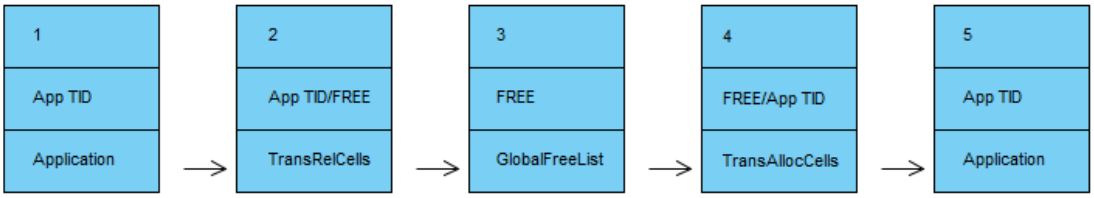

Figure9: Cell ID Transfer

To clarify if the cell had been recycled, the monitor set the self TID as dirty at first. Then it traverses the transient layer and global free list in sequence showed in above figure. When the ID is set dirty, the cell was in the

1) Then monitor can find that self TID is not FREE. Or the cell has moved to 2.
2) The monitor can find the cell in traverse. Or the cell has moved into 3.
3) The monitor can find the cell in traverse. Or the cell has moved into 4.
4) The monitor can find the cell in traverse. Or the cell has moved into 5.
5) The monitor can find the self TID is not FREE.

If the cell had been recycled more than once, that is, across step2 or step4 completely at least once after dirty bit set, its ID would be set as pure ID without dirty bit set. Then monitor can detect this case by assertion on dirty bit. In general, in this case, the cell must be faster than monitor in the way of recycle.

Above cases are all for leader cell. While for siblings, it is impossible that leader had been recycled while the siblings had not. After one round of recycle, the leader and the siblings may be no longer bound. If one sibling has more cycles than the leader, the leader must be found in the

second poll. If leader is quicker, the dirty bit search can detect it.

Figure10 shows the detailed algorithm:

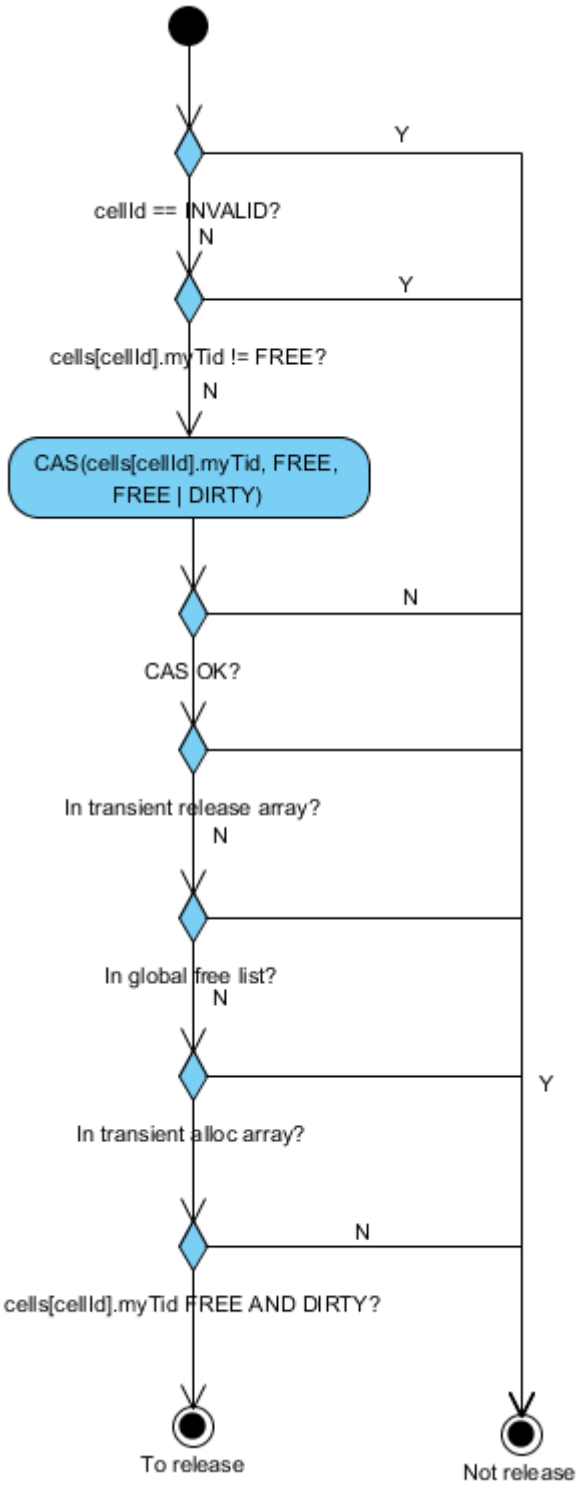

Figure10: Transient Cell Collection

6.2Flying Cells Collection

6.2.1 Receiving Queue

The receiving queue is just drained in cleanup. Figure11 shows the details algorithm:

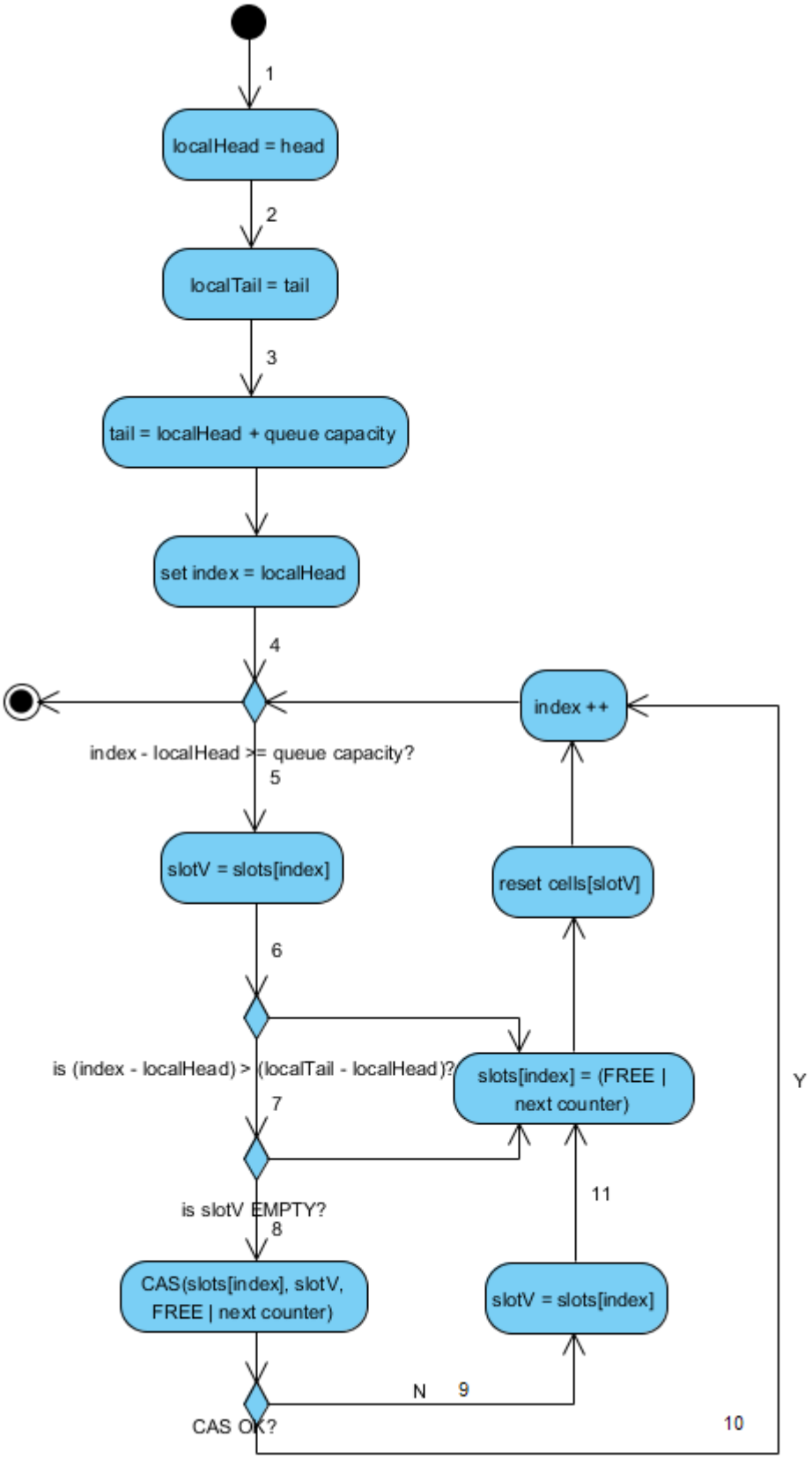

Figure11: Receiving Queue Collection

1) Get local head before tail as head can't change.
2) Get local tail. Tail may change after step2.
3) Set the queue as full.

4) As head is frozen after receiving application died, at most items of size of capacity of queue could be pushed.
5) Get local value in slot.
6) Slots between local tail and local head had been filled and won't be changed by producers.
7) Slots after local tail may also have been pushed after local tail read.
8) Try to push by FREE and new counter.
9) If CAS failed, a producer must have been pushed succeeded. When the producer tries to update tail, it failed or pushed tail exceeding capacity of the queue. In both cases, the full queue assertion is OK. Then read the pushed cell.
10) If CAS succeeded, producing application must fail to push into this slot as the producer either encounters the new counter and failed in value CAS or finds the full queue in retry.
11) Follow the rule of pop to update the slot's counter. It is optional: if the producer is before full assertion, it will fail; if the producer is before tail assertion, the assertion will fail; if the producer is before CAS value and empty assertion, the slot must have been pushed and popped at least once and the producer will fail in CAS in counter mismatch.

The head and tail will be reset when it is reused to expect that all hung push had finished to avoid mess.

6.2.2 Sending-to Queue

The following figure shows process of garbage collection of sending cells. It in fact is integrated in the global pool collection.

When a sending cell can't be found in destination queue, the cell may not have been pushed in queue or have been popped by receiver. If the cell had been received, the cell must have the different TID than sending application's TID. So, if the cell had the TID untouched, it could be released by monitor. There is a special case that the cell had been returned to the sender, while in this case, the ownership of the cell also belongs to the sender and monitor also has the right to release and the all these cells can be released just once.

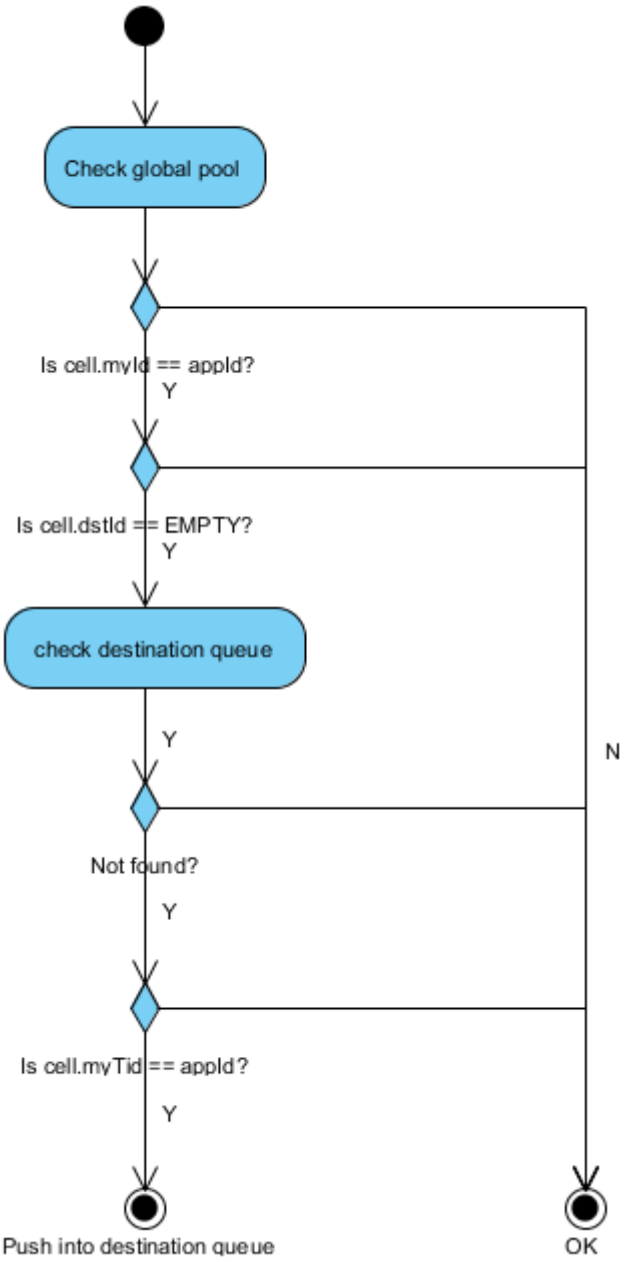

Figure12: Sending Queue Collection

6.3Global Pool Collection

When all the trivial issues had been cleared, the monitor just traverses the global pool and release all cells belonged to dead application.

7. Scenario

The garbage collection is a rather complex process and expected to be time consuming. So the solution is suit to system that not expecting much death concurrently. There will be some monitor mechanism to trigger the restart of the system when there were load of garbage

collector is exceeding. Besides, the solution does not expect too many anticipates either.

On the other hand, as the cells of one process would spread all over the global pool, absence of cache acceleration would negate performance gained by no-copy mechanism in SMP. Performance may be better in NUMA architecture.

A demo at: https://bitbucket.org/fulltopic/test5

8. Further Optimization

Make process clean its own garbage if it died elegantly, while it may be a complicated solution just as that in kernel memory management.

9. Performance

It should be better than POSIX message queue. It may not be better than network with special accelerates.

10. Reference

[1]. https://sourceware.org/glibc/wiki/Concurrency
[2]. Chien-Hua Shann, Ting-Lu Huang, Cheng Chen. *A practical nonblocking queue algorithm using compare-and-swap*. Proceedings Seventh International Conference on Parallel and Distributed Systems