High Level Design and Detailed Design

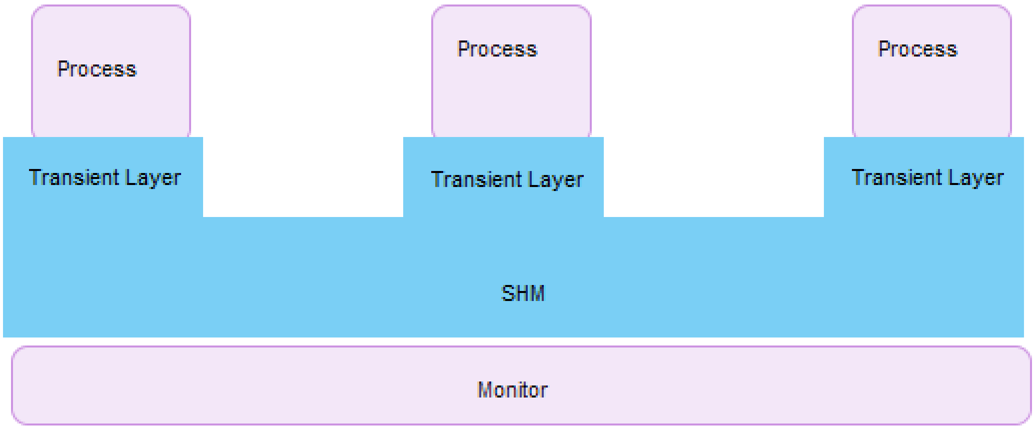
1. Overview

This library is to provide a shared memory based IPC system that enables messages communicated between processes in pointer and cover the unexpected stop of processes at the same time. This library is supposed to run in Linux.

The memory provided to application processes is managed in cell. And each cell has one owner at any time. There is a monitor per OS besides all application processes who also works as garbage collector. 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.

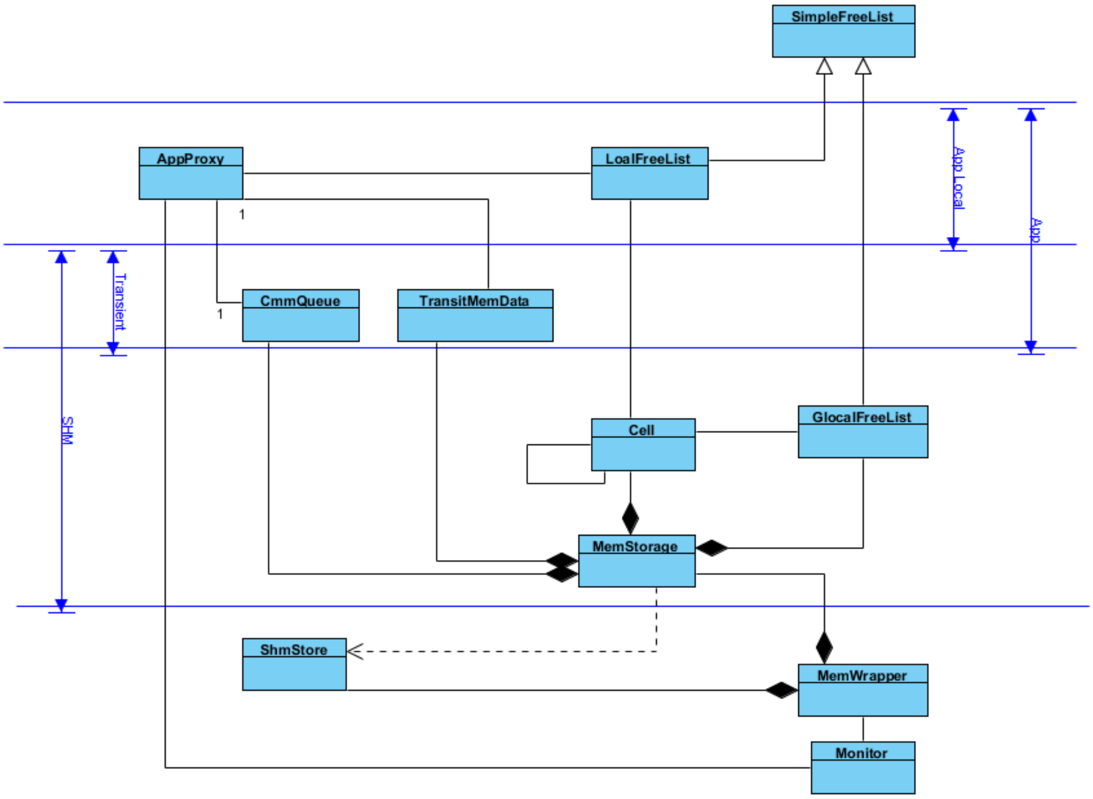
When ownership is ambiguous, that is, when ownership is transferring, the handler of the cell is exposed to the monitor so that it is able to discover the real owner by special but costly treatment. Then, there is a Transient Layer attached to each process that belongs to the SHM. The corresponding cases are in allocation/release/transfer.

The following figure displays the overview:



2. Structure

The following figure displays the structure of the system.



3. Algorithms Overview

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

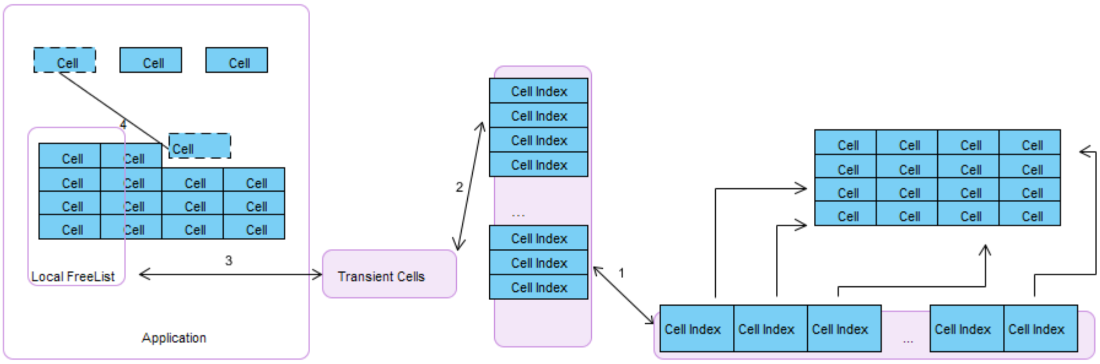
The algorithm is based on following ideas

- Atomic operations provided by GLIBC library.
- Lock-less queue. Refer to [1] for detailed description of the algorithm. Or there is another one written by myself before [1] was found.
- Transient layer mentioned in section 1.
- Monitor holds all the information and has the same life cycle as system.
- Cell ownership.
- Cells are managed in list of arrays.

Following sections will give detailed description of the implementation.

4. Allocation and Release

The following figure gives layout of memory in static view:

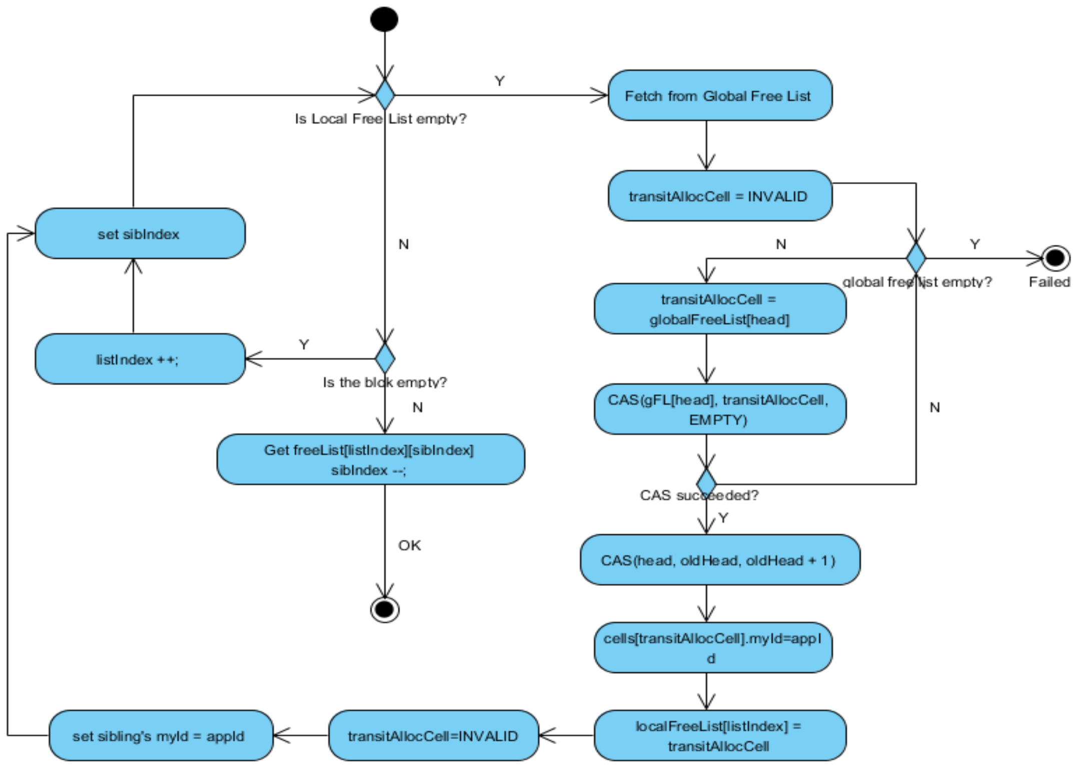


Step1: The global free list has been initialized as list of arrays. The array has fixed size, named Block. Indexes of all cells had been pushed into the global free lit. The own is the Monitor.
Step2: In allocation and release, cells are transmitted into transient cells by unit of a block. The ownership is in transfer.
Step3: When the transient cell pushed/popped successfully, they are assigned to local free list. Local free list also managed in unit of block. The own is the application.
Step4: In application allocation and release, memory are 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, that is, list of arrays. And to guarantee that the sibling and the leader cell have the same ownership, the global list follow 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

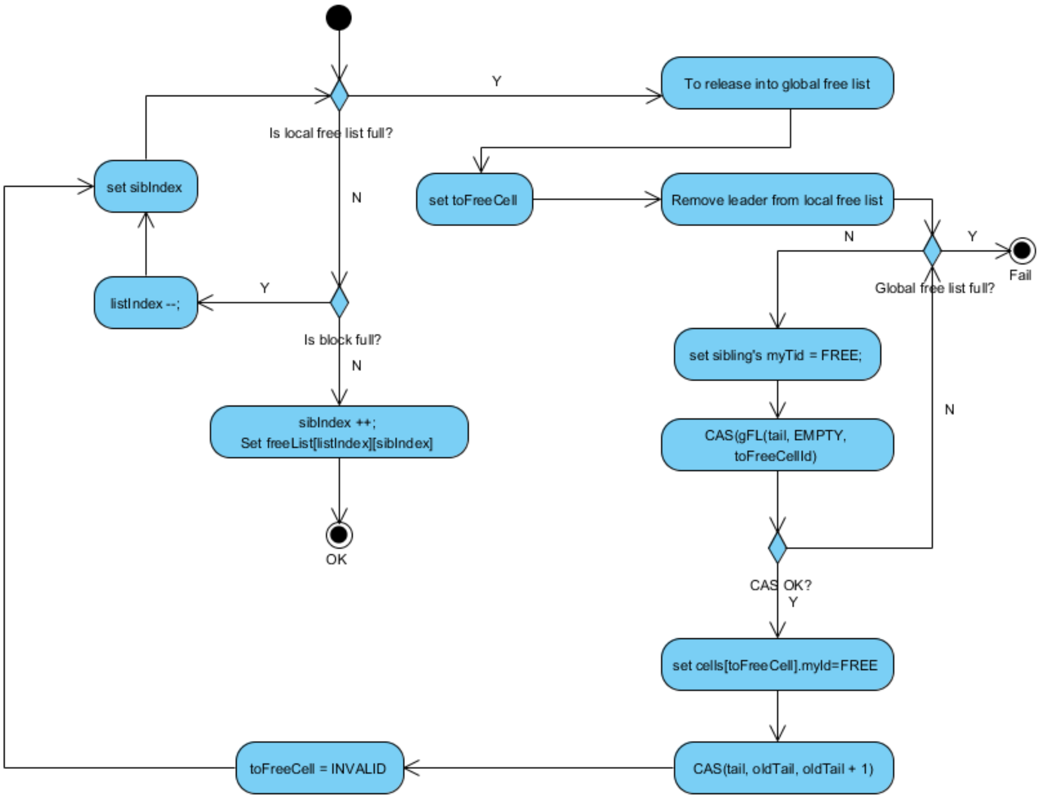
The following figure shows details of allocation algorithm.


Is Local Free List empty?
Y
Fetch from Global Free List
transitAllocCell = INVALID
N
Y
global free list empty?
Failed
set sibIndex
listIndex ++;
Y
Is the block empty?
N
Get freeList[listIndex][sibIndex]
sibIndex --;
OK
transitAllocCell = globalFreeList[head]
CAS(gFL[head], transitAllocCell, EMPTY)
N
CAS succeeded?
Y
CAS(head, oldHead, oldHead + 1)
cells[transitAllocCell].myId=appId
localFreeList[listIndex] = transitAllocCell
transitAllocCell=INVALID
set sibling's myId = appId

Application try to fetch a cell from local free list firstly, and then turn to global free list if 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.

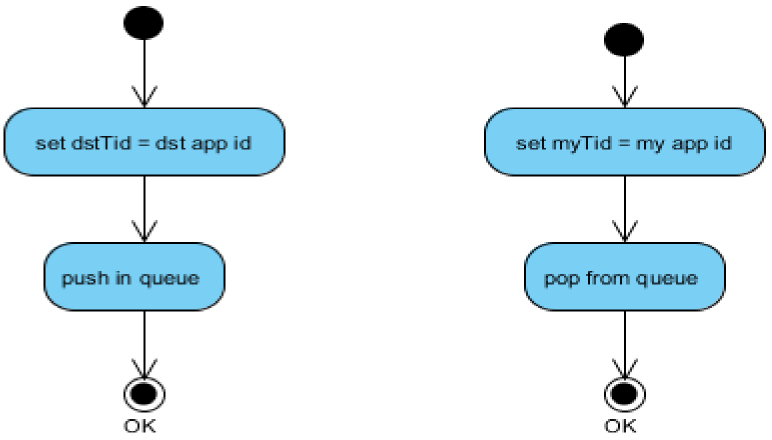
4.2Release

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.


Y
To release into global free list
Is local free list full?
set sibIndex
N
set toFreeCell
Remove leader from local free list
N
Y
Fail
Global free list full?
listIndex --;
Y
Is block full?
N
set sibling's myTid = FREE;
N
sibIndex ++;
Set freeList[listIndex][sibIndex]
CAS(gFL(tail, EMPTY, toFreeCellId)
OK
CAS OK?
Y
set cells[toFreeCell].myId=FREE
toFreeCell = INVALID
CAS(tail, oldTail, oldTail + 1)

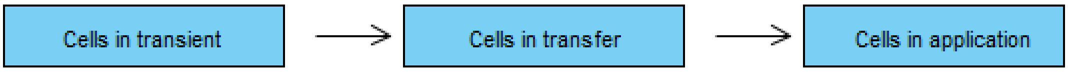
5. Communication

For a cell in sending, both my TID and destination TID are set, to indicate it's a flying cell. For a cell in receiving, with assumption of single-threaded receiver, the my TID is set before pop. The following figure shows the details.



6. Garbage Collector

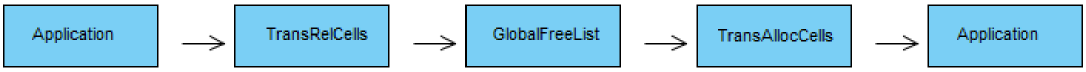
When a process died, the garbage is collected by Monitor. The monitor would never been restarted when there were living processes. That is, when monitor dead, the sub-system dead.
The monitor would be notified with the application ID when a process dead. The sequence of collection as following figure shows. The cells in transient are collected in first to remove ambiguous cases. Cells in queue are collected later. On the one hand, monitor removes messy before collection in whole global pool; on the other hand, monitor gives some time for receiver to reach the en-queued cells. And the global pool is collected at last. Those are cells held by application.



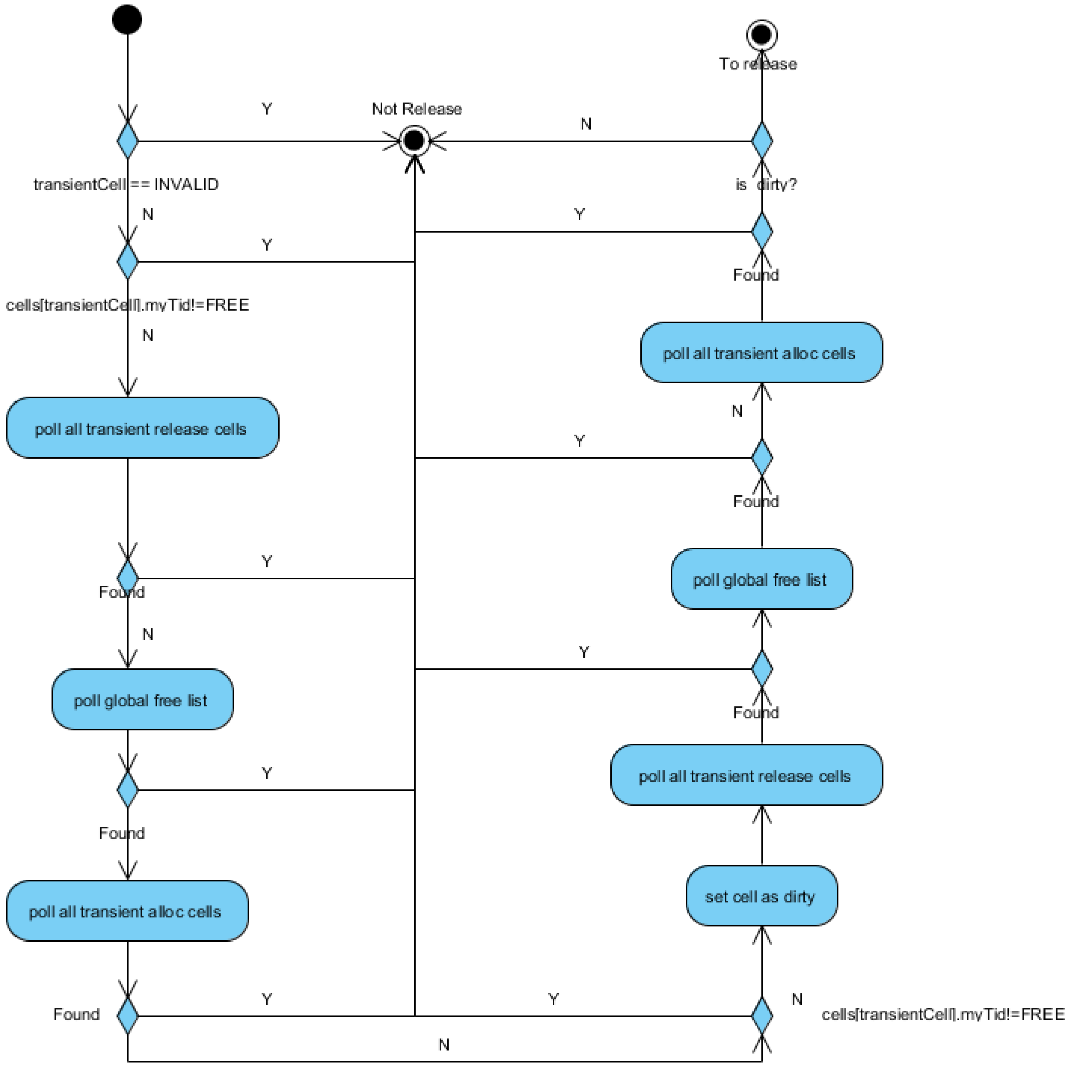
6.1Transient Cells Collection

6.1.1 Double Poll Check

Cells are flying as the following figure shows.



The monitor traverses in the flow as the above figure shows. If the monitor could not find the suspected cell in the way, and the cell had been reused; then the cell must move faster than the monitor. That is, the cell had been held by application or released behind the monitor. After set the dirty bit and traverse the system again, and the cell still had not been found and also had bit set, then the cell could be released by the monitor. That is because that when the second traverse could not found the suspected cell, the cell must have been cycled with dirty bit at least once; and the gates, that is, release and allocation, would have removed the bit set; so if the dirty bit was still set, the cell must have not been put into recycle routine. Then, to release by the monitor is correct.


Y
Not Release
N
To release
transientCell == INVALID
N
Y
is dirty?
Y
Found
cells[transientCell].myTid!=FREE
N
poll all transient alloc cells
poll all transient release cells
N
Y
Found
Y
Found
N
poll global free list
Y
poll global free list
Found
Y
poll all transient release cells
Found
poll all transient alloc cells
set cell as dirty
Found
Y
Y
N
cells[transientCell].myTid!=FREE
N

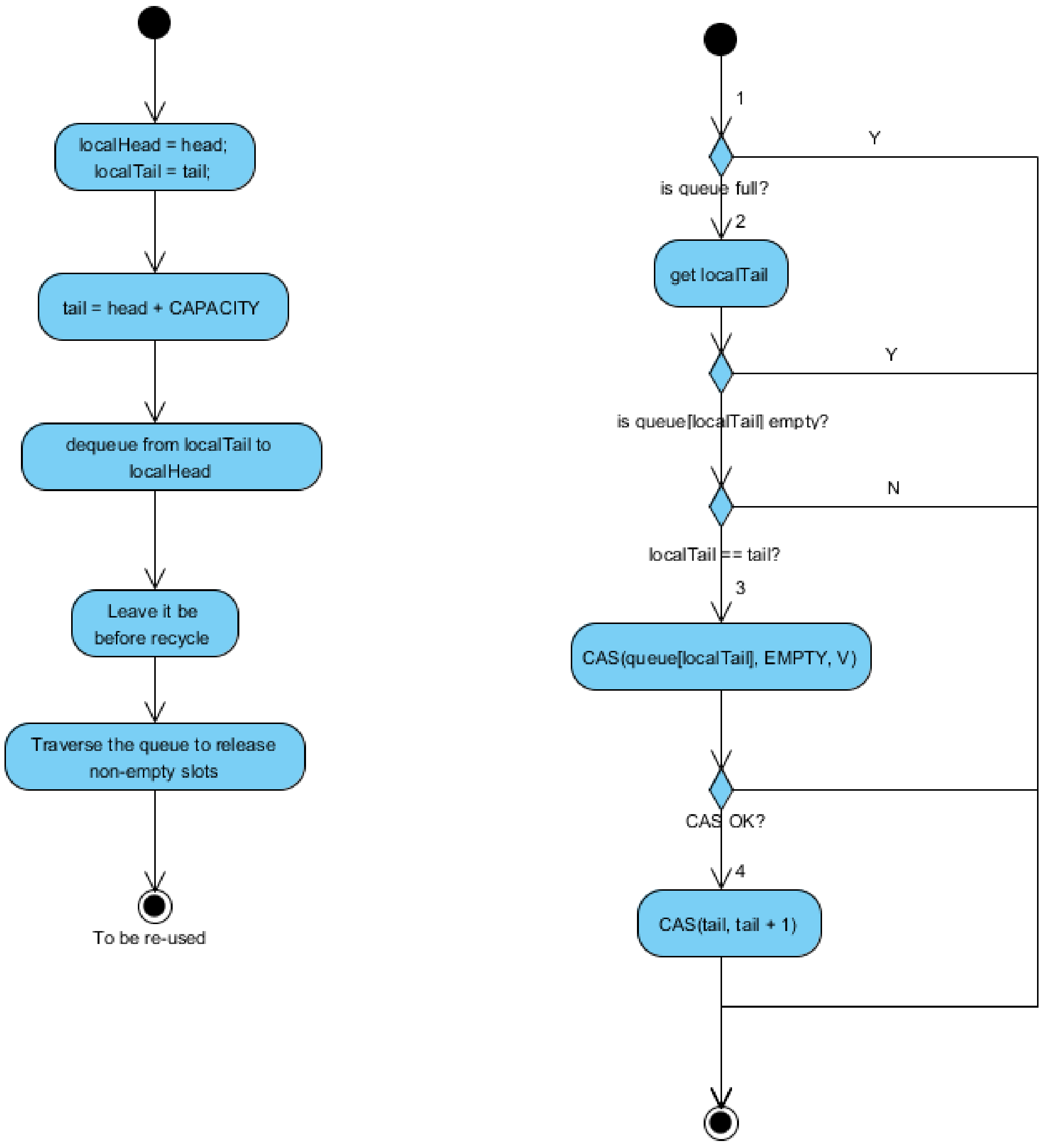
6.2Flying Cells Collection

6.2.1 Receiving Queue

The receiving queue is just drained. While when to poll out all the received cells is a subtle issue. It is impossible for all the sending processes being informed and stop sending before monitor deputizing receiving all remained cells as it a problem of censorship in asynchronous system.

The solution includes two steps.

- Set the queue as full to prevent further pushing.
- Provide a pool of queues that large enough so that the sender is able to detect the failure of death of the receiver or to en-queued successfully before the queue was recycled. While leaving all en-queued cells to be polled in initiation may impose detectable impact on performance of it. So the job would be done in both garbage collection and initiation and expect there are minimum number of leftover in initiation.

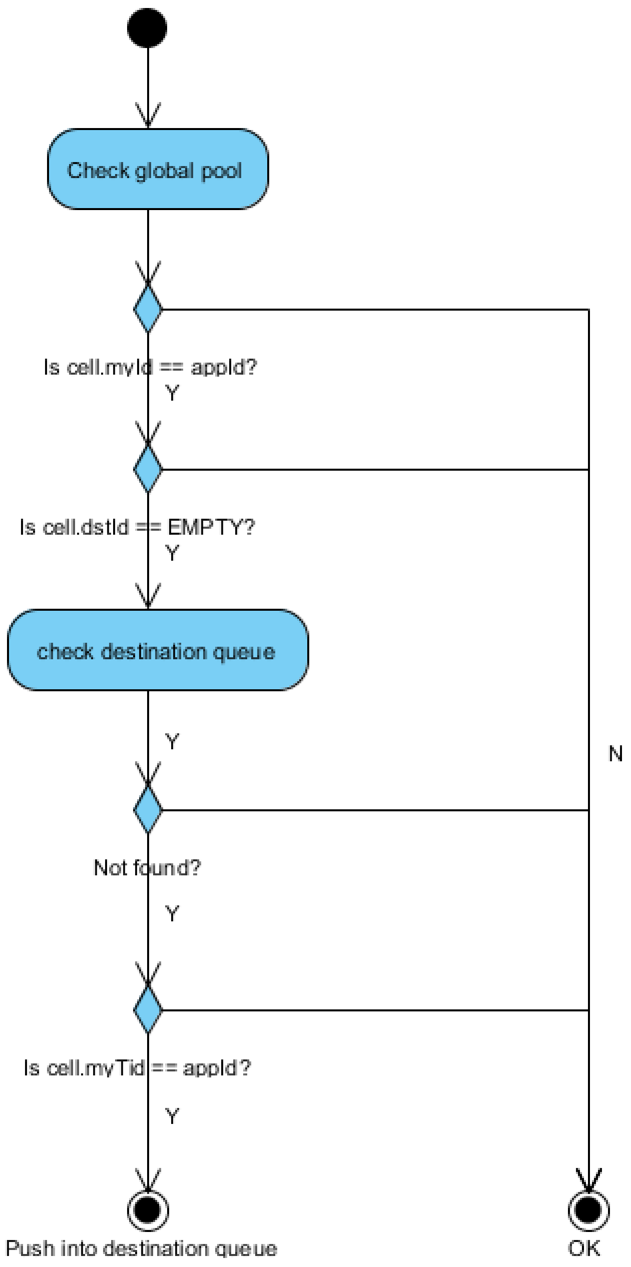


localHead = head;
localTail = tail;
tail = head + CAPACITY
dequeue from localTail to localHead
Leave it be before recycle
Traverse the queue to release non-empty slots
To be re-used
1
Y
is queue full?
2
get localTail
Y
is queue[localTail] empty?
N
localTail == tail?
3
CAS(queue[localTail], EMPTY, V)
CAS OK?
4
CAS(tail, tail + 1)

The above figure shows the details. If set queue full happens at the point:
Case 1: Sending failed as detected full queue.
Case 2: Sending failed at point 3.
Case 3: Sending succeeded. Leave it to be cleaned up in recycle.
Case 4: Sending succeeded. Leave it to be cleaned up in recycle. And tail > (head + CAPACITY + 1), queue is still full.

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.



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.

8. Performance

9. Potential Risk