Concurrent Reference Counting

RCGC is naturally incremental, how about making it concurrent …

Review Incremental mark-sweep

- Steele's multiprocessing, compactifying collector
- Dijkstra's on-the-fly collector
- Kung and Song improved four-color collector
- Yuasa sequential collector
  - Uses snapshot-at-the-beginning write-barrier
- Compared using these metrics
  - Operation of write-barrier
  - Treatment of new objects
  - Cost of initialization & termination of each GC cycle

Initialization of GC

- In sequential algorithm
  - When request for more memory cannot be satisfied
- In serial incremental MM systems
  - When free memory falls below a certain threshold
  - Yuasa suggests heap space headroom ~ 22%
- How to initiate GC
  - Simple method:
    - push pointers in registers, system stack, & global variables on marking stack (color them grey)
    - Root set may be large
    - If suspending mutator, pause may be unbounded

Bounding initiation pause

- Kung & Song:
  - Push roots on double-ended mark queue one at a time
  - Incremental: mutator's computation is unrestricted
- Yuasa:
  - Copy entire program stack to saved_stack using a fast copy method (e.g. UNIX *memcpy*)
  - Entries in saved_stack transferred to mark stack k2 at a time
  - Reduces fragmentation

Marking in concurrent system

- Concurrent system:
  - Multiple processes or thread execute at the same time and potentially interact with each other
  - Collector locks mark stack while examining it

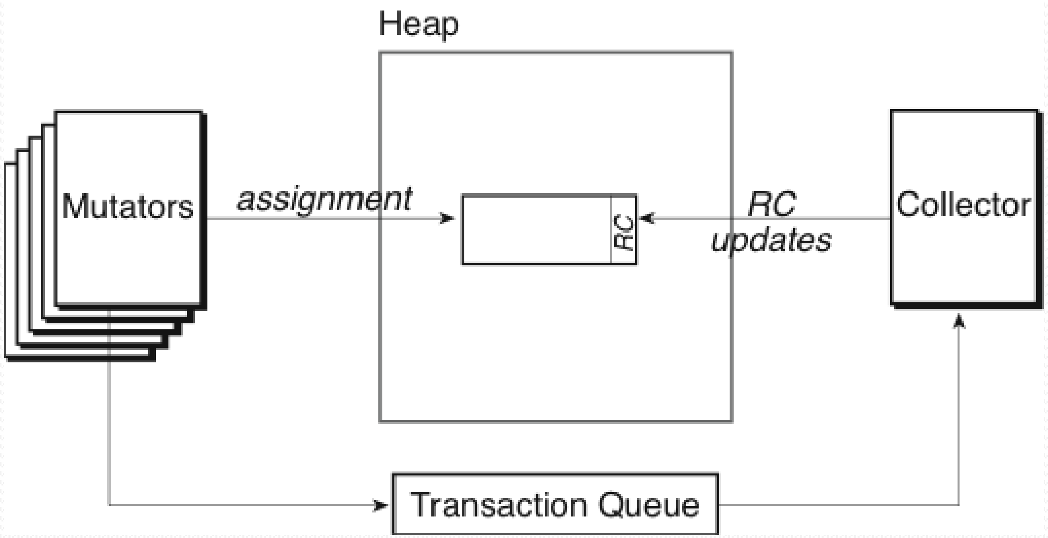
Termination of GC

- Mark phase completes when no grey object left in heap
  - Dijkstra determines this by scanning for grey objects
  - Restarts marking from any grey objects encountered
  - Marking terminates when no grey object found
- Can marking and sweeping be pipelined?
  - Quiennec says yes
    - Use two color fields
    - Start mark phase of collection cycle n+1 while sweeping in cycle n
    - Odd collections use one color field while even collections uses the other

Concurrent reference counting

- Updating a RC must be atomic to avoid race conditions between threads
  - Can lead to premature collection of objects
- Atomicity requires locking shared objects
  - Increases cost of pointer assignment
- Increase mutator performance
  - Run collector in separate thread
  - Make collector responsible for updating RC fields
  - Mutators no longer update RC but log assignments in a block of a transaction queue (figure on next slide)

Modula-2+ RC architecture



Jones and Lin: Diagram 8.7

Modula-2+ RC

- Mutator and collector communicates through a transaction queue
- When current block is full (~ 16, 384 assignments) or aout 40 KB of data allocated
  - Mutator notifies collector
  - Mutator gets new empty block
- Lock required to prevent simultaneous assignment to same shared variable

Reducing RC cost

- Distinguish assignments to local variables from assignments to global variables and heap data
- Only reference count shared-pointer-valued-variables
- RC is only lower bound of refs to object from local & shared variables

Mutator code: shared references

```
update(A, C){
    LOCK mutex
        insert (A, C, tq)                // insert in transaction queue
        if( tq is full){
            notify_collector(tq)         // send block to collector
            tq = gt_next_block()
        }
        *A = C
}
```

Modula-2+ RC algorithm

- TQ Block holds details up to some time t_0
- Collector interrupts threads one at a time to scan its state
- Collector locks mutex to stop a thread
- Any ref in thread's state to heap object is saved for later use
- All thread states scanned at time t_1
- Collector adjust RC of pair of variables inserted in tq
- If RC == 0, object added to Zero-Count-List (ZCL)
  - Object deleted if shared RC== 0 at t_0 and local RC == 0 and not on RHS of assignment

Collector code: shared references

```
collector(){
    while (; ;){
        tq = wait_next_block()
        for each thread th {
            LOCK mutex{
                suspend(th)
                scan_thread(th)
                restart(th)
            }
        }
        adjust_counts(tq)
        free_block(tq)
        adjust_shared_copunts()
        process_ZCL()
}
```

Processing ZCL

- If object's shared RC no longer 0, it is removed from ZCL
- If object is found in a thread state, it is left in ZCL
  - It may be freed in future collection
- Otherwise object is removed from ZCL and recursively freed
- Note: Can reduce cost of assignment
  - Use per thread transaction queue