COMPARISION OF CHENEY'S AND MARK-COMPACT ALGORITHM

DONE BY,
SANTHIYA.S,
18 31 048,
MSc. Software Systems,
Department of Software Systems,
Coimbatore Institute Of Technology.

CHENEY’S GARBAGE COLLECTION ALGORITHM

- In Cheney’s algorithm, Memory space is divided in two:
  - → From-Space
  - → To-Space.
- New arrays are allocated in From-Space.
- When From-Space becomes full, program execution stops, and all live data (non-garbage) is copied to To-Space.
- From-Space and To-Space are flipped and program execution continues.

Notations Used

Assuming that h is the header cell (the first cell) of an array:

→ **visited(h)** - A bit recording whether the array has been previously visited

→ **size(h)** - If visited bit is zero, the size of the array

→ **forwardingAddress(h)** - If visited bit is one, the location in to-space that the array was copied to.

→ **pointer?(x)** - True if the value x is a pointer to an array, and false otherwise

→ **roots** - The set of all live variables (which point to arrays) in the program at the time the garbage collector is invoked.

→ **#heap** - The size of the heap

Algorithm

Step – 1 => Copy the array beginning at location from in from-space to the next free locations in to-space. Variable free points to the next free location in to-space.

```
copyArray(from)
    let header = fromSpace[from]
    n ← size(header)
    foreach i in 0..n
        toSpace[free] ← fromSpace[from+i]
        free ← free + 1
```

Step – 2 => If the array has not been visited, copy it to to-space and set the forwarding address. Return the tospace location of array.

```
copy(from)
    let header = fromSpace[from]
    if visited(header)
        return forwardingAddress(header)
    else
        addr ← free
        copyArray(from)
        visited(header) ← true
        forwardingAddress(header) ← addr
    return addr
```

Step – 3 => Finally perform the cheney's algorithm

```
cheney()
    scan ← 1
    free ← 1
    foreach r in roots
        r ← copy(r)
    while scan < free
        let x = toSpace[scan]
        if pointer?(x)
            toSpace[scan] ← copy(x)
        scan ← scan + 1
    fromSpace, toSpace ← toSpace, fromSpace
```

MARK-COMPACT GARBAGE COLLECTION ALGORITHM

- Mark-compact algorithms can be regarded as a combination of the mark-sweep algorithm and Cheney's copying algorithm
- After marking the live objects in the heap in the same fashion as the mark-sweep algorithm, the heap will often be fragmented. The goal of mark-compact algorithms is to shift the live objects in memory together so the fragmentation is eliminated.
- The challenge is to correctly update all pointers to the moved objects, most of which will have new memory addresses after the compaction. The issue of handling pointer updates is handled in different ways.
- It recursively mark all arrays reachable from the roots as visited.
- For each visited array, compute a new address for the array in the compacted heap.
- Update pointers to point to the new addresses.
- Relocate each array to its new address.
- **Larger header cells** : Each header cell now needs to store the size of the array and a forwarding address at the same time.

Algorithm

Step – 1 => Recursively mark all arrays that are reachable from the array beginning at address addr.

```
mark(addr)
    let header = heap[addr]
    if visited(header) == false then
        visited(header) ← true
        n ← size(header)
        foreach i in 1 .. n
            child ← heap[addr+i]
            if pointer?(child) then
                mark(child)

markCompact()
    foreach r in roots
        mark(r)
    computeAddresses()
    updatePointers()
    relocate()
```

Step – 2 => For each visited array, compute a new address for the array in the compacted heap.

```
computeAddresses()
    scan ← 1
    new ← 1
    while scan <= #heap
        let header = heap[scan]
        n ← size(header)
        if visited(header) == true then
                forwardingAddress(header)
← new
            new ← new + 1 + n
        scan ← scan + 1 + n
```

Step – 3 => Update pointers to point to the new addresses.

```
updatePointers()
    scan ← 1
    while scan <= #heap
        let header = heap[scan]
        n ← size(header)
        if visited(header) == true then
            foreach i in 1 .. n
                let x = heap[scan+i]
                if pointer?(x) then
                    x ← forwardingAddress(heap[x])
        scan ← scan + 1 + n
```

Step – 4 => Relocate each array to its new address.

```
relocate()
    scan ← 1
    while scan <= #heap
        let header = heap[scan] and n ← size(header)
        if visited(header) then
            visited(header) ← false
            dest ← forwardingAddress(header)
            foreach i in 0 .. n
                heap[dest+i] ← heap[scan+i]
            scan ← scan + 1 + n
```

COMPARISION OF CHENEY'S AND MARK-COMPACT GARBAGE COLLECTION ALGORITHM

- Cheney's garbage collection algorithm requires the total heap capacity to be halved to implement the two semi-spaces.
- Mark and compact garbage collection algorithm requires an extra pointer sized field to be stored in every array header.
- Mark and compact garbage collection algorithm needs several traversals of the heap whereas Cheney's garbage collection algorithm needs a single traversal of the live heap.
- Cheney's garbage collection algorithm is a simple iterative algorithm whereas Mark and compact garbage collection algorithm requires a stack or a pointer-reversal approach.

CONCLUSION

Garbage collection automatically reclaims unused memory which is the One less thing for the programmer to worry about (usually!). Compacting collectors support very fast memory allocation. Cheney's and Mark-Compact garbage collection algorithm are viable approaches illustrating the space-time trade-off. And both Cheney's and Mark-Compact garbage collection algorithm could be easily employed in the Tower implementation.