Comp 411
Principles of Programming Languages
Lecture 30
Garbage Collection II

Corky Cartwright

April 11, 2014

Copying Collectors I

A fundamentally different approach to periodically collecting the live data in the heap is to copy the live data to a new area of memory. Since nearly all heap data can be live in the worst case, the new area must be potentially as large as the collected heap. Hence, copying collections typically divide heap space in half and *only populate half of it with allocated objects*! The two spaces are typically called "old space" and "new space" and the association between names and heap halves flips on each garbage collection. All allocation is performed in "new space".

The classic version of copying collection is called Cheney collection in honor of the computer scientist who invented it (1970). In Cheney collection, GC is triggered when new space becomes full. When collection starts, the identity of new space and old space flips. The Cheney collector copies all of the live data from old space (formerly new space) to a contiguous prefix of new space, Intuitively, Cheney collection simply drags all of the live objects (think of pointers being implemented by cables connecting objects) from the old space to new space.

Copying Collectors II

The Cheney algorithm for performing this copy operation is deceptively simple. Initially, all of the target objects of the root set are copied to new space. In the process, a "front" pointer is maintained to the first copied object and a "rear" pointer to the last copied object. Whenever an object is copied from old space to new space, a forwarding pointer is stored in a designated place (typically the first word of the header) in the old copy of the object (in old space). After the root objects have been copied to new space, the Cheney collector simply performs *breadth-first* search to copy all live objects to new space. The queue of copied but not yet processed objects is the sequentially allocated list of objects between the front and rear pointers inclusive.

While the front pointer is less than or equal to the rear pointer, the collector repeatedly processes the front object in the queue and advances the front pointer to the next object in the queue. An object is processed by examining each embedded pointer and copying the target object to the end of the queue (next open area in new space) unless it has already been copied (which is easily determined by checking for the existence of a forwarding pointer in the old copy of the object). If the target object has already been copied, the value of the embedded pointer is simply updated to the new address.

This loop must terminate because each object is only copied and processed once and there are only a finite number of objects.

Copying Collectors III

Pseudo-code for a Cheney collector:

Flip identity of new and old space.

Copy the target objects of roots to new space, updating the root pointer cells, creating forwarding pointers in old copies of these objects, front and rear pointers to the sequentially allocated queue formed by the copied objects in new space, and a next pointer to the first word following object(rear).

```
while (front <= rear) {
  for each pointer field p in object(front) {
    if (object(p) has a forwarding pointer p') update field p to p';
    else {
      copy object(p) to next;
      update field p to next;
      rear := next;
      advance next;
    }
  advance front;
}
```

Copying Collectors IV

Allocation of a new object in new space: do the following atomically

Save old value of next;
Initialize the allocated object, incrementing next by size of allocated object;
Return the old (saved) value of next;

Pros and cons of Cheney collection:

GC is efficient: runs in time proportional to amount of live data (size of new heap). Amount of work for each byte of copied data is small.

Half of available heap space is lost!

Can we combine advantages of Cheney collection and mark-and-compact?

Yes! Use generational “scavenging” (collection).

Generational Collectors

Idea: partition the heap into two generations based on how long an object has lived. Objects (unless they are too large and must be treated as special cases) are always allocated in the youngest generation heap (the "nursery") and promoted to the main heap only if they survive a specified number of collections (typically a small number) of the nursery (which are called "ephemeral" collections).

Note; the nursery is typically about the size of the L2 cache.

Note: this idea can clearly be generalized to more than two generations.

Key technical problem: how to we determine the roots for the nursery. Some subset of the root set points to the nursery. But what about pointers from the mature heap to the nursery. These must be recorded in a auxiliary table (a hash table that is also a linked collection).

Key observation: auxiliary table is easy to maintain. It is only updated when (I) an assignment to a cell in the mature heap is made and (ii) when GC (ephemeral or global) happens. In a functional language (i) never happens! In mostly functional programs (i) is rare!

Generalization: a generational collector can have more than 2 generations; not clear if additional generations improve performance.