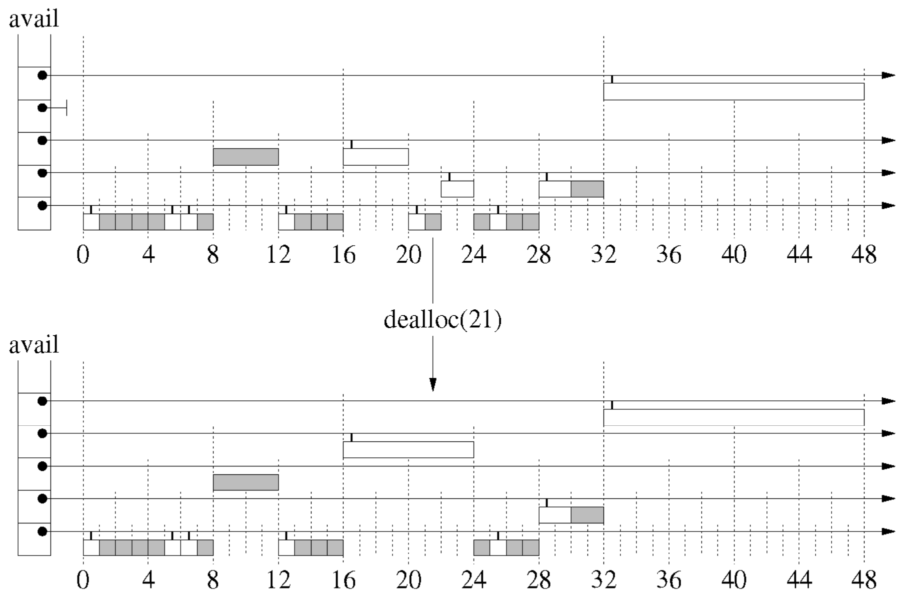

Figure 71: Buddy deallocation.

Lecture 24: Garbage Collection

Reading: Chapter 3 in Samet's notes.

Garbage Collection: In contrast to the explicit deallocation methods discussed in the previous lectures, in some memory management systems such as Java, there is no explicit deallocation of memory. In such systems, when memory is exhausted it must perform *garbage collection* to reclaim storage and sometimes to reorganize memory for better future performance. We will consider some of the issues involved in the implementation of such systems.

Any garbage collection system must do two basic things. First, it must detect which blocks of memory are unreachable, and hence are "garbage". Second, it must reclaim the space used by these objects and make it available for future allocation requests. Garbage detection is typically performed by defining a set of *roots*, e.g., local variables that point to objects in the heap, and then finding everything that is reachable from these roots. An object is *reachable* (or *live*) if there is some path of pointers or references from the roots by which the executing program can access the object. The roots are always accessible to the program. Objects that are not reachable are considered garbage, because they can no longer affect the future course of program execution.

Reference counts: How do we know when a block of storage is able to be deleted? One simple way to do this is to maintain a *reference count* for each block. This is a counter associated with the block. It is set to one when the block is first allocated. Whenever the pointer to this block is assigned to another variable, we increase the reference count. (For example, the compiler can overload the assignment operation to achieve this.) When a variable containing a pointer to the block is modified, deallocated or goes out of scope, we decrease the reference count. If the reference count ever equals 0, then we know that no references to this object remain, and the object can be deallocated.

[1]Copyright, David M. Mount, 2001

Reference counts have two significant shortcomings. First, there is a considerable overhead in maintaining reference counts, since each assignment needs to modify the reference counts. Second, there are situations where the reference count method may fail to recognize unreachable objects. For example, if there is a circular list of objects, for example, X points to Y and Y points to X, then the reference counts of these objects may never go to zero, even though the entire list is unreachable.

Mark-and-sweep: A more complete alternative to reference counts involves waiting until space runs out, and then scavenging memory for unreachable cells. Then these unreachable regions of memory are returned to available storage. These available blocks can be stored in an available space list using the same method described in the previous lectures for dynamic storage allocation. This method works by finding all immediately accessible pointers, and then traces them down and *marks* these blocks as being accessible. Then we *sweep* through memory adding all unmarked blocks to the available space list. Hence this method is called *mark-and-sweep*. An example is illustrated in the figure below.

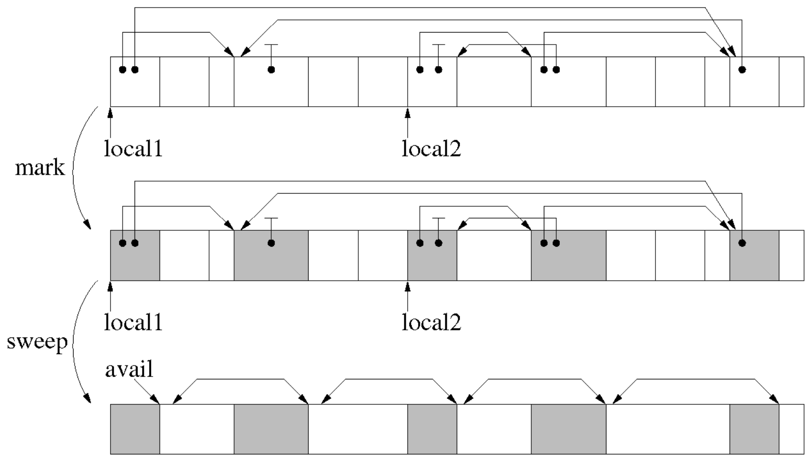

Figure 72: Mark-and-sweep garbage collection.

How do we implement the marking phase? One way is by performing simple *depth-first traversal* of the "directed graph" defined by all the pointers in the program. We start from all the root pointers t (those that are immediately accessible from the program) and invoke the following procedure for each. Let us assume that for each pointer t we know its type and hence we know the number of pointers this object contains, denoted `t.nChild` and these pointers are denoted `t.child[i]`. (Note that although we use the term "child" as if pointers form a tree, this is not the case, since there may be cycles.) We assume that each block has a bit value `t.isMarked` which indicates whether the object has been marked.

Recursive Marking Algorithm

```
mark(Pointer t) {
    if (t == null || t.isMarked) return     // null or already visited
    t.isMarked = true                       // mark t visited
    for (i = 0; i < t.nChild; i++)          // consider t's pointers
        mark(t.child[i])                    // recursively visit each one
}
```

The recursive calls persist in visiting everything that is reachable, and only backing off when we come to a null pointer or something that has already been marked. Note that we do not need to store the `t.nChild` field in each object. It is function of t's type, which presumably the run-time system is aware of (especially in languages like Java that support dynamic casting). However we definitely need to allocate an extra bit in each object to store the mark.

Marking using Link Redirection: There is a significant problem in the above marking algorithm. This procedure is necessarily recursive, implying that we need a stack to keep track of the recursive calls. However, we only invoke this procedure if we have run out of space. So we do not have enough space to allocate a stack. This poses the problem of how can we traverse space without the use of recursion or a stack. There is no method known that is very efficient and does not use a stack. However there is a rather cute idea which allows us to dispense with the stack, provided that we allocate a few extra bits of storage in each object.

The method is called *link redirection* or the *Schorr-Deutsch-Waite method.* Let us think of our objects as being nodes in a multiway tree. (Recall that we have cycles, but because of we never revisit marked nodes, so we can think of pointers to marked nodes as if they are null pointers.) Normally the stack would hold the parent of the current node. Instead, when we traverse the link to the ith child, we redirect this link so that it points to the parent. When we return to a node and want to proceed to its next child, we fix the redirected child link and redirect the next child.

Pseudocode for this procedure is given below. As before, in the initial call the argument t is a root pointer. In general t is the current node. The variable p points to t's parent. We have a new field `t.currChild` which is the index of the current child of t that we are visiting. Whenever the search ascends to t from one of its children, we increment `t.currChild` and visit this next child. When `t.currChild == t.nChild` then we are done with t and ascend to its parent. We include two utilities for pointer redirection. The call `descend(p, t, t.c)` moves us from t to its child $t.c$ and saves p in the pointer field containing $t.c$. The call `ascend(p, t, p.c)` moves us from t to its parent p and restores the contents of the p's child $p.c$. Each are implemented by a performing a cyclic rotation of these three quantities. (Note that the arguments are reference arguments.) An example is shown in the figure below, in which we print the state after each descend or ascend.

$$\text{descend}(p,t,t.c) : \begin{pmatrix} p \\ t \\ t.c \end{pmatrix} \longleftarrow \begin{pmatrix} t \\ t.c \\ p \end{pmatrix} \qquad \text{ascend}(p,t,p.c) : \begin{pmatrix} p \\ t \\ p.c \end{pmatrix} \longleftarrow \begin{pmatrix} p.c \\ p \\ t \end{pmatrix}.$$

Marking using Link Redirection

```
markByRedirect(Pointer t) {
    p = null                                    // parent pointer
    while (true) {
        if (t != null && !t.isMarked) {         // first time at t?
            t.isMarked = true                   // mark as visited
            if (t.nChild > 0) {                 // t has children
                t.currChild = 0                 // start with child 0
                descend(p, t, t.child[0])       // descend via child 0
            }
        }
        else if (p != null) {                   // returning to t
            j = p.currChild                     // parent's current child
            ascend(p, t, p.child[j])            // ascend via child j
            j = ++t.currChild                   // next child
            if (j < t.nChild) {                 // more children left
```

```
                    descend(p, t, t.child[j])          // descend via child j
                }
            }
            else return                                // no parent?  we're done
        }
    }
```

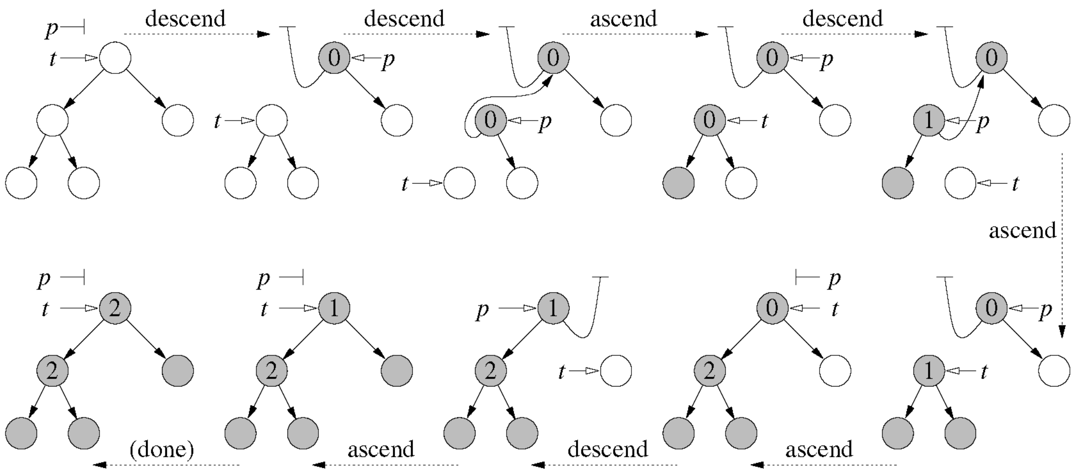

Figure 73: Marking using link redirection.

As we have described it, this method requires that we reserve enough spare bits in each object to be able to keep track of which child we are visiting in order to store the `t.currChild` value. If an object has k children, then $\lceil \lg k \rceil$ bits would be needed (in addition to the mark bit). Since most objects have a fairly small number of pointers, this is a small number of bits. Since we only need to use these bits for the elements along the current search path, rather than storing them in the object, they could instead be packed together in the form of a stack. Because we only need a few bits per object, this stack would require much less storage than the one needed for the recursive version of mark.

Stop-and-Copy: The alternative strategy to mark-and-sweep is called *stop-and-copy.* This method achieves much lower memory allocation, but provides for very efficient allocation. Stop-and-copy divides memory into two large *banks* of equal size. One of these banks is *active* and contains all the allocated blocks, and the other is entirely unused, or *dormant.* Rather than maintaining an available space list, the allocated cells are packed contiguously, one after the other at the fron of the current bank. When the current bank is full, we *stop* and determine which blocks in the current bank are reachable or *alive.* For each such live block we find, we *copy* it from the active bank to the dormant bank (and mark it so it is not copied again). The copying process packs these live blocks one after next, so that memory is compacted in the process, and hence there is no fragmentation. Once all the reachable blocks have been copied, the roles of the two banks are swapped, and then control is returned to the program.

Because storage is always compacted, there is no need to maintain an available space list. Free space consists of one large contiguous chunk in the current bank. Another nice feature of this method is that it only accesses reachable blocks, that is, it does not need to touch the garbage. (In mark-and-sweep the sweeping process needs to run through all of memory.) However, one shortcoming of the method is that only half of the available memory is usable at any given time, and hence memory utilization cannot exceed one half.

The trickiest issue in stop-and-copy is how to deal with pointers. When a block is copied from one bank to the other, there may be pointers to this object, which would need to be redirected. In order to handle this, whenever a block is copied, we store a *forwarding link* in the first word of the old block, which points to the new location of the block. Then, whenever we detect a pointer in the current object that is pointing into the bank of memory that is about to become dormant, we redirect this link by accessing the forwarding link. An example is shown in the figure below. The forwarding links are shown as broken lines.

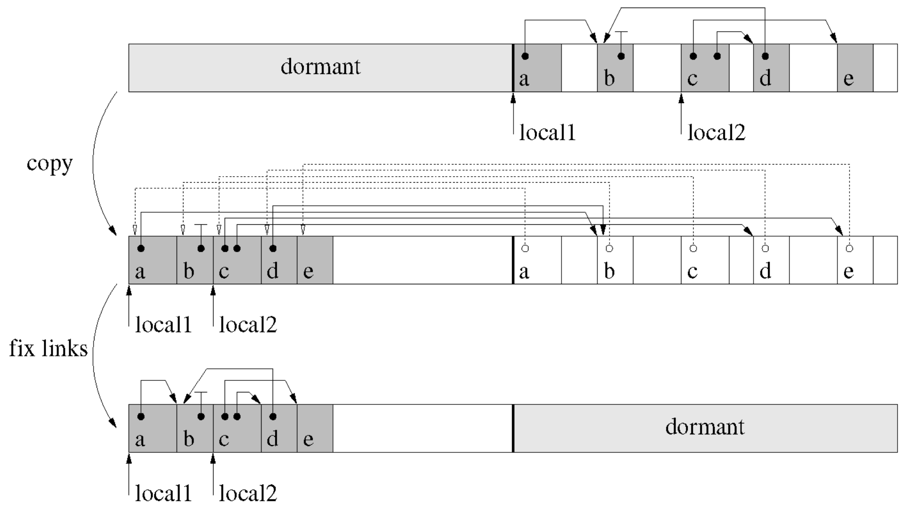

Figure 74: Stop-and-copy and pointer redirection.

Which is better, mark-and-sweep or stop-and-copy? There is no consensus as to which is best in all circumstances. The stop-and-copy method seems to be popular in systems where it is easy to determine which words are pointers and which are not (as in Java). In languages such as C++ where a pointer can be cast to an integer and then back to a pointer, it may be very hard to determine what really is a pointer and what is not, and so mark-and-sweep is more popular here. Stop-and-copy suffers from lots of data movement. To save time needed for copying long-lived objects (say those that have survived 3 or more garbage collections), we may declare them to be *immortal* and assign them to a special area of memory that is never garbage collected (unless we are really in dire need of space).