

Chapter 10

Storage Management

[These notes are slightly modified from notes on C storage allocation from the Fall 1991 offering of CS60C. The language used is C, not Java.]

10.1 Classification of storage

In languages like C or Java, the storage used by a program generally comes in three categories.

Static storage. This refers to variables—generally given names by declarations—whose lifetime by definition encompasses the entire program’s execution.

Local storage. Variables—also usually named in declarations—whose lifetimes end after the execution of some function or block.

Dynamic storage. Variables (generally anonymous) whose lifetime begins with the evaluation of a specific statement or expression and ends either at an explicit deallocation statement or at program termination.

For example, in Java, static variables are introduced by as static fields in classes. C and C++ also allow for static variables in functions and outside classes and functions (at the “outer level” where they are in effect static fields in a giant anonymous class). For example,

```
int rand(void)    /* C code */
{
    static int lastValue = 42;
    extern int randomStatistics;

    ...
}
```

Here, there is a single variable `lastValue` and a single variable `randomStatistics` that retain their last values from call to call. It is true that only the function `rand`

is allowed to access `lastValue` by name, but that is an independent question¹.

Local variables in Java and C are simply non-static, non-external variables or parameters declared in a function. They disappear upon exit from the function, which is why the following piece of code, beloved of C beginners, is almost certainly incorrect.

```
int* newIntPtr(int N) /* C code */
/* Return a pointer to an integer initially containing N. */
{
    int X = N;
    return &X;
}
```

In C, one can have pointers to simple containers: `&X` creates a pointer to the container `X`, and `int*` denotes the type pointer-to-`int`. The variable `X` officially disappears immediately after the return. Practically speaking, this means that the compiler is allowed to re-use the storage location that was used to contain `X` at any subsequent time (which will probably be the very next call to any function).

Finally, dynamic variables in Java and C++ are the anonymous objects the programmer creates using `new`, or in C using `calloc` or `malloc`. In C and C++, any deallocation that takes place must be explicit (by use of the `free` function or `delete` operator, respectively). Languages like Java and Lisp have no explicit `free` operation, and instead deallocate storage at some point where the storage is no longer needed. We'll discuss how later in this chapter.

Just to show that hybridization is possible, some C implementations support a function called `alloca`. This takes the same argument as `malloc` and returns a pointer to storage. But the lifetime of the storage ends when the function that called `alloca` exits (one may not `free` storage allocated by `alloca`). The storage is therefore sort of “locally dynamic.” It is useful for functions that create local linked lists (for example) or arrays whose sizes are not known at compilation. Alas, due to the peculiar runtime memory layouts used by some machines and C implementations, it is not a standard function.

10.2 Implementation of storage classes

It is not my purpose to give a comprehensive survey of all the twists employed in implementing the various classes of storage described above. Instead, I'll describe one implementation as representative—that used in most Unix implementations.

Figure 10.1 diagrams the layout of memory from the point of view of a single Unix process². Static storage resides in a fixed, writable area immediately after

¹Rules that determine which parts of a program may name the variable defined by a particular declaration are called *scope rules*. In this section, we discuss rules about how long a variable exists—its *extent* or *lifetime*—regardless of who (if anyone) is allowed to name it. Unfortunately, the term “scope” has been given various meanings in the literature, some of which involve lifetime. Be cautious, therefore, in interpreting the term.

²As you probably know, there are generally numerous Unix processes at any given time, each

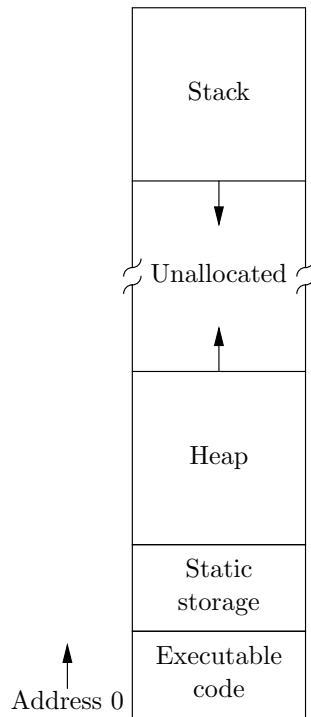


Figure 10.1: An example of run-time storage layout: the Unix C library strategy.

the area containing instructions and constants for the program (which is called the *text segment*). Local storage resides in the *run-time stack*, which grows down toward the static storage area. The area in between is available for the program to request and use as it will. The standard C library uses the beginning of this area for dynamic storage, growing the portion it uses for this purpose toward the stack. By an unfortunate and confusing convention, the dynamic storage area is known as the “heap,” although it has nothing in common with the data structure we have used for priority queues.

10.3 Dynamic storage allocation with explicit freeing

The C language and its standard library present the following features.

1. Storage may be allocated dynamically at any time by a library call.
2. Dynamically-allocated storage may be freed at any time by a library call.

running its own program. All of them seem to have access to all of memory, as if they were each alone on the machine. This trick is accomplished by means of a hardware feature known as *virtual memory*, which allows different processes to have the same address for physically distinct pieces of memory.

3. Programs may *cast* void pointers—which include the pointers returned by dynamic allocation—to and from pointers of any type with a compatible size and alignment. This casting operation may not change the contents of the allocated storage.
4. Programs may cast the pointers returned by dynamic allocation to and from sufficiently large integer types.

As will become clearer when we look at storage management in Lisp and Java, items 3 and 4 above militate against automatic storage de-allocation in C. That is, it is in principle impossible to determine automatically that a particular piece of storage is no longer needed and may be “recycled” for use in future allocations. It is likewise impossible to move dynamically-allocated storage regions around “behind the programmer’s back” to make room, say, for a new, dynamically-allocated object. The C library allocates blocks of storage when requested and never touches them again until it is requested to free them.

The general strategy is to maintain a list of blocks of unallocated storage, called the *free list*. When there is a request to allocate storage, we search the free list for a block of sufficient size, and return the address of an appropriate portion of it, possibly returning leftover storage to the free list. When no block of sufficient size for a request exists on the free list, the library requests a new large block of free storage from the underlying operating system. When there is a request to free storage, we return the block of storage to the free list.

This sketchy description needs some refinement. We must assume that the library can determine sizes of allocated and free blocks. There is also a problem that will arise when a large number of blocks have been freed: storage becomes *fragmented* as small blocks are released. Formerly-large blocks gradually get allocated as many small ones, until requests for large amounts of storage cannot be met. To combat this problem, it is often desirable to *coalesce* adjacent blocks of free storage back into larger blocks. There are numerous ways of filling in the resulting strategy. Here, I’ll describe two concrete methods for explicitly allocating and freeing storage.

Java does not provide the operations needed to implement memory management, so the remainder of this chapter actually uses C, which your instructor can explain as needed. Basically, the additional functionality we need is the ability to change an arbitrary integer number back and forth into an address of an arbitrary kind of object.

10.3.1 Boundary tag method

The first method requires an additional *administrative word* of storage for each free or allocated block, which will immediately precede the block. The free list will be a circular doubly-linked list of blocks. If X is a pointer to a block, then we will assume the existence of the following operations on a block X and its administrative word.

isFree(X) a boolean value that is true iff X is the address of a free block.

precedingIsFree(*X*) a boolean value that is true iff the block of storage immediately preceding *X* is free. This value is normally false if *X* is a free block (that is, adjacent free blocks are generally coalesced rather than being left separate).

blockSize(*X*) the size of block *X*, including its administrative word.

precedingBlock(*X*) is valid only if **precedingFree(*XX*)**. It is the address of the free block adjacent to and preceding *X* in memory.

followingBlock(*X*) is the address of the block immediately following *X* in memory.

freeNext(*X*) is the address of the next free block in the free list. It is valid only if **isFree(*XX*)**.

freePrev(*X*) is the address of the previous free block in the free list. It is valid only if **isFree(*XX*)**.

For convenience, I'll assume these are defined so as to be assignable (so for example, to set **blockSize(*XX*)** to *V*, I'll write **blockSize(*XX*)=V**).

These interfaces are written abstractly just to remind you that different machines may require different implementations. Here, for example, are concrete definitions that will work on Sun Sparc workstations; Figure 10.2 illustrates how the data structures fit together³

```
typedef struct AdminWord AdminWord;

/* The type Address is assumed to be large enough to hold any
 * object's address. We also assume that
 * sizeof(AdminWord) = sizeof(Address). */
typedef long Address;

struct AdminWord {
    unsigned int
        size : 30,          /* The size of this block, including the
                             * administrative word. The size is always
                             * a multiple of 4 and is always at least 12. */
        isFree : 1,
        precedingIsFree : 1;
};

/** The administrative word associated with a block at location X is
 * stored immediately before X. */
```

³The “*field : length*” notation in C indicates that a given field of a record occupies exactly *length* bits. Consecutive bit fields of this sort are generally packed together. The compiler generates the necessary shifting and masking instructions to extract and set them when called for.

```

#define _ADMIN_WORD(X)    ((AdminWord *) (X))[-1]

/** The minimum size of a free block. */
#define MIN_FREE_BLOCK (3 * sizeof(Address))

/** True iff the block at location X is a free block. */
#define isFree(X) (_ADMIN_WORD(X).isFree)
/** True iff the block just before the block at location X is a free
 * block. */
#define precedingIsFree(X) (_ADMIN_WORD(X).precedingIsFree)
/** The size of the block at X, including the administrative word. */
#define blockSize(X) (_ADMIN_WORD(X).size)
/** A pointer to the block next in memory after the one at X. */
#define followingBlock(X) ((Address) (X) + blockSize(X))

/** If X points to a free block, then the link to the next block in the
 * free list is at location X, and a back link to the previous block
 * in the free list is at the end of the block pointed to by X.
 * If precedingIsFree(X), then the back link for the free block
 * that precedes X in memory is immediately before the
 * administrative block for X. Therefore, one can find the address
 * of the free block that precedes X in memory by the circuitous
 * route of picking up this back link and then following the
 * forward from there. */
#define freeNext(X) \
    ((Address*) (X))[0]
#define precedingBackLink(X) \
    ((Address*) (X))[-2]
#define freePrev(X) \
    precedingBackLink(followingBlock(X))
#define precedingBlock(X) \
    freeNext(precedingBackLink(X))

Address FREE_LIST;

```

Initially, the allocation routines reserve a large, contiguous block of storage, allocating a dummy sentinel block at the high end to prevent the `free` routine from attempting to coalesce a newly-freed block with the storage that follows. The `freeNext` and `freePrev` pointers for the remaining initial free block are initialized to point to the block itself, creating a one-element circular, doubly-linked list.

Allocation. To allocate a block, we use the following procedure (text in *italics* for missing code, which is left to the reader to supply).

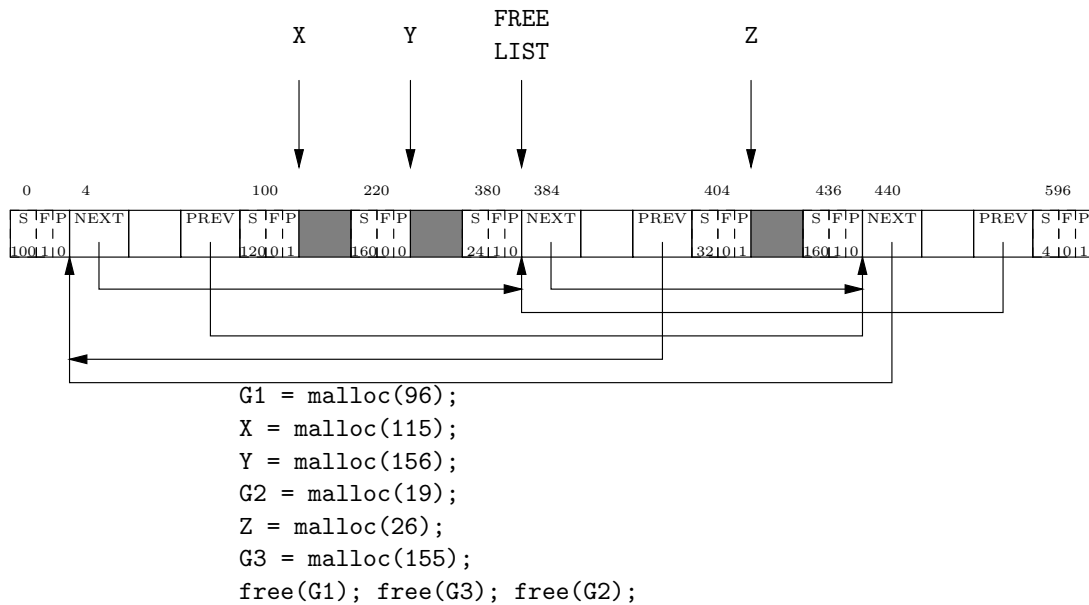


Figure 10.2: The state of the storage allocator after executing the allocations and frees shown above. Shaded areas are being used by the program; unshaded areas are used by the storage allocator. The original block of free space was 600 bytes long. A permanently-allocated 4-byte block at the end is a sentinel that guarantees that all other blocks have a block following them. Memory addresses relative to the beginning of the entire chunk of storage are shown above certain boxes. The quantities used by the storage allocator are labeled ‘S’ for `blockSize`, ‘F’ for `isFree`, ‘P’ for `precedingIsFree`, ‘NEXT’ for `freeNext`, and ‘PREV’ for `freePrev`.

```

Address malloc(unsigned int N)
{
    Address FREE0;
    Address result, next, last;

    if (FREE_LIST == NULL)
        GET_MORE_STORAGE(N, FREE_LIST);

    FREE0 = FREE_LIST;
    loop {
        FREE_LIST = freeNext(FREE_LIST);
        if (blockSize(FREE_LIST) >= N + sizeof(AdminWord))
            break;
        if (FREE_LIST == FREE0) {
            GET_MORE_STORAGE(N, FREE_LIST);
            return malloc(N);
        }
    }

    Round N upward to an even multiple of sizeof(Address) such that
    N + sizeof(AdminWord) ≥ MIN_FREE_BLOCK.

    /* If the remaining free block would be too small, expand the
     * request to eat up the entire free block. */
    if (blockSize(FREE_LIST) - N - sizeof(AdminWord) < MIN_FREE_BLOCK)
        N = blockSize(FREE_LIST) - sizeof(AdminWord);

    result = FREE_LIST;
    Delete current block from free list
    if (blockSize(result) > N + sizeof(AdminWord))
        Add the last blockSize(result) - N - sizeof(AdminWord)
        bytes of the block at result back to FREE_LIST.

    isFree(result) = precedingIsFree(result) = 0;
    blockSize(result) = N + sizeof(AdminWord);

    return result;
}

```

The statement `GET_MORE_STORAGE` is intended to obtain a new large area of storage from the operating system (at least enough for N bytes plus an administrative word) and link it into the free list, causing `malloc` to return a null pointer if this is not possible.

The strategy used above for finding a free area of sufficient size is known as *first-fit*; it finds the first large-enough free block and carves the necessary storage out

of that. At each new allocation, however, the search starts where the previous one left off, rather than at a fixed beginning. This turns out to be extremely important to obtaining good performance. If the search always starts at the same location, the beginning of the free list soon becomes cluttered with chopped-up blocks that don't meet the demands of most requests, but must be skipped over to get to bigger blocks. The rotating free list pointer overcomes this problem.

Another possible strategy is *best-fit*: find the closest fit to the requested size. It is by now well-known, however, that this strategy is expensive (in a simple implementation, one must look at all free blocks) and in fact harmful, leading to many small free blocks.

Freeing. To free a block, we coalesce it with any adjacent free blocks and add it to the free list.

```
void free(Address X)
{
    if (X == NULL || isFree(X))
        return;

    if (isFree(followingBlock(X))) {
        remove followingBlock(X) from FREE_LIST;
        blockSize(X) += blockSize(followingBlock(X));
    }
    if (precedingIsFree(X)) {
        Address previous = precedingBlock(X);
        remove previous from FREE_LIST;
        blockSize(previous) += blockSize(X);
        X = previous;
    }
    /* NOTE: At this point, X is not adjacent to any free block,
     * either before or after it in memory. */

    isFree(X) = 1; precedingIsFree(followingBlock(X)) = 1;
    Link X into FREE_LIST.
}
```

Ordered free lists. The minimum-sized block in this scheme contains two pointers and an administrative word—12 bytes on a Sun-3, for example, corresponding to an allocation of 8 bytes. On that same machine, the real C library versions of `malloc` and `free` get away with blocks containing only one pointer plus the administrative word, single-linking the free list. In order to allow coalescing, they search the free list for adjacent blocks, and speed this up by *ordering* the free list by memory address. Since the search implicitly finds all free blocks, it is unnecessary to have flags indicating that a block or its neighbor is free. The price, of course, is a slower `free` procedure.

10.3.2 Buddy system method

When there is a single free list to search, the time required to perform allocation cannot easily be bounded. In some applications, this may be a problem. The *buddy system* provides for allocation and freeing of storage in time $O(\lg N)$, where N is the size of storage. It allocates storage in units of 2^k storage units (bytes, words, whatever) for $k \geq k_0$, where 2^{k_0} storage units is the minimum needed to hold forward and backward pointers for a free list (this information appears only in free blocks).

The idea is to treat the allocatable storage area as an array of storage units, indexed 0 through $2^m - 1$. A block (free or allocated) of size 2^k will always start at an index in this array that is evenly divisible by 2^k . Free blocks are only coalesced with other free blocks of the same size, and only in such a way as to preserve the property that each free block starts at an index position that is divisible by its size.

For example, suppose that a block of size 16 becomes free and that it starts at index position 48 in the storage array. This block may be merged with a block of size 16 that starts in position 32. It may *not* be merged with a block of size 16 that starts in position 64, because the resulting block would be of size 32, and such blocks may only start at positions divisible by 32; merging our block at 48 with one at 64 would result in a block of size 32 that started at position 48, which is not allowed. We say that the blocks of size 16 at positions 32 and 48 are *buddies*, while those at 48 and 64 are not.

Thus, the rule is that a free block may only be coalesced with its buddy (and only if that block is free). The calculation of one's buddy's index is quite easy, if a bit obscure. The buddy of a block of size 2^k at an index X begins at index $X \oplus 2^k$, where ' \oplus ' computes the exclusive or of the binary representations of its operands (the '`string^`' operator in C).

Each free block contains forward and backward links for inclusion in a free list. The system maintains four arrays.

MEMORY is the actual allocatable storage (containing 2^m `StorageUnits`, where the type `StorageUnit` is typically something like `char`).

FREE_LIST is an array of `FreeBlocks` with `FREE_LIST[k]` being the sentinel for the list of free blocks of size 2^k . Each list is circular and doubly-linked. Initially, `FreeBlock[m]` contains the entire block of allocatable storage (of size 2^m) and all other free lists contain only their sentinel nodes (are empty, in other words).

IS_FREE is an array of true/false values, with `IS_FREE[X]` being true iff X is the index of a free block. Since each element is either true or false, this array may be represented compactly—perhaps as a bit vector. Initially, `IS_FREE[0]` is true and all others are false.

SIZE is an array of integers in the range 0 to m . If there is a block (free or allocated) of size 2^k that begins at location X , then `SIZE[X]` contains k .

Because these values tend to be small, and because X will always be divisible by 2^{k_0} , it is possible to represent `SIZE` compactly. Initially, `SIZE[0]` is m .

Allocation. To allocate under the buddy system, we first round the size request up to a power of 2. If no block of the desired size is free, we allocate a block of double the size (recursively) and then break it into its constituent buddies, putting one of them back on the free list and returning the other as the desired allocation.

```
unsigned int buddyAlloc(unsigned int N)
/* Return the index in MEMORY of a new block of storage at least */
/* N storage units large. */
{
    Choose the minimum  $k \geq k_0$  with  $2^k \geq N$  and set  $N$  to  $2^k$ .

    if (k > m)
        ERROR: insufficient storage.

    if (isEmpty(FREE_LIST[k])) {
        unsigned int R = buddyAlloc(2*N);
        IS_FREE[R] = TRUE;
        SIZE[R] = k;
        Add the block at R to FREE_LIST[k].
        return R+N; /* i.e., the second half of the size 2N block at R */
    }
    else {
        Remove an item, R, from FREE_LIST[k].
        IS_FREE[R] = FALSE;
        return R;
    }
}

Address malloc(unsigned int N)
{
    return & MEMORY[buddyAlloc(N)];
}
```

Freeing. To see if a newly-freed block may be coalesced with its buddy, we first see if the block at the buddy's location is free, and then see if that block has the right size (the buddy may have been broken down to satisfy a request for something smaller).

```

/** Free the storage at index L in MEMORY. */
void buddyFree(unsigned int L)
{
    int k = SIZE[L];
    int N = 1 << k;
    unsigned int Lbuddy = L \string^ N;

    if (k < m && IS_FREE[Lbuddy] && SIZE[Lbuddy] == k) {
        Remove Lbuddy from FREE_LIST[k]
        IS_FREE[Lbuddy] = FALSE;
        if (L > Lbuddy)
            L = Lbuddy;
        SIZE[L] = k+1;
        buddyFree(L); /* recursively free the coalesced block */
    }
    else {
        IS_FREE[L] = TRUE;
        Add L to FREE_LIST[k];
    }
}

void free(Address X)
{
    unsigned int L = (StorageUnit*) X - (StorageUnit*) MEMORY;

    if (X == NULL || IS_FREE[L])
        return;
    buddyFree(L);
}

```

10.3.3 “Quick fit”

The use of an array of free lists in the buddy system suggests a simple way to speed up allocation and deallocation. When there are certain sizes of object that you often request, maintain a separate free list for each of these sizes. Requests for other sizes may be satisfied with a heterogeneous list, as described in the sections above. Free items on the one-size lists need not be coalesced (except perhaps in an emergency, when there is insufficient storage to meet a larger request), and no searching is needed to find an item of one of those sizes on a non-empty list. This means, of course, that allocation and freeing go very fast for those sizes. The term *quick-fit* has been used to describe this scheme.

10.4 Automatic Freeing

There are two problems with having the programmer free dynamic storage explicitly. First, it complicates and obscures programs to do so. Second, it is prone to error.

Suppose, for example, that I introduce a string module into C. It provides a type, **String**, whose variables may contain arbitrary strings, of any length, and whose operations allow the programmer to form catenations, substrings, and so forth. I'd like to use **String** variables as conveniently as if they were integers. To make good use of space, it is convenient to use dynamic storage. This presents a problem, however. In contrast to the situation with **int** variables, my **String** variables don't entirely vanish when I exit the procedure that declares them. I must explicitly deallocate them—my string module will have provided a deallocation procedure, of course, but I (the programmer) must still write *something*. Worse yet, consider a procedure such as this.

```
/** Return the concatenation of the strings in X. */
String concatList(String X[], int N)
{
    int i;
    String R = nullString();
    for (i = 0; i < N; i += 1)
        R = concat(R, X[i]);
    return R;
}
```

This seems innocuous, but it is unlikely to work well. The problem is that the function `concat` does not know that the storage used by its first operand can be deallocated immediately after use (since the result of `concat` is going back into `R`). The programmer must explicitly deallocate each intermediate value of `R` instead, which will complicate this function considerably.

Perhaps the most common error found in programs that do explicit freeing is the *memory leak*: storage that is never deallocated, even after it is no longer needed. Other errors are possible, as well; attempts to access storage after it has been freed can lead to extremely obscure errors (I suspect, however, that these bugs are less common than memory leaks).

These considerations lead us to consider methods for automatically freeing dynamic storage that is no longer needed. This generally translates to dynamic storage that is no longer *reachable*—that the program can no longer reference since no pointers lead to it (directly or indirectly) from any named variables the program can access. Such storage is called *garbage*, and the process of reclaiming it *garbage collection*⁴.

Some assumptions. Automatic storage reclamation generally requires some cooperation from the compiler and the programming language being used. All of the methods

⁴Some authors reserve the term “garbage collection” for methods that use marking (see below), excluding reference counting. Here, I will use the term for all forms of automatic reclamation.

discussed below follow pointers that they find embedded in dynamically-allocated objects. In order to do this, they must first be able to find all such pointers. This requires a certain amount of what we generically call *type information*; the run-time routines must be able to find out at least enough about an object's type to deduce where its pointer fields are. There are various ways to arrange this.

- The language may have only one kind of dynamically-allocated object, whose pointers are all in the same places. For example, early Lisp systems had only **cons** cells (objects containing only a pair of pointers).
- The language may be *strongly typed* so that the type of all quantities is known by the compiler and conveyed somehow (by tables perhaps) to the run-time storage management routines.
- The system may store type information (indicating the positions of all pointers) *with every object* at some standard location, so that a storage-freeing routine can acquire this information without knowing anything beforehand about the program being run.
- The system may store type information *in the pointers*. Sometimes the possible addresses in a certain system leave certain bits of each pointer value 0, so that the runtime system may store useful information in these bits (masking them out when it really needs the pointer). Another approach is to put all objects of a particular type at particular ranges of addresses, so that by looking at a pointer's value, a storage deallocator may deduce its type.

In what follows, I'll just assume we have some way of finding this information, without going into particulars.

Automatic storage reclamation also requires that the values stored in pointers be under fairly strict control. A language or language implementation that allows arbitrary integers in pointer variables can seriously confuse a procedure that is trying to follow a trail of pointers through a data structure. In Lisp, for example, all pointer variables (that is to say, all variables) are initialized to values that the run-time system understands (when a variable is "undefined" or "unbound," it contains a special recognizable "undefined" value, even if the programmer can't mention such a thing directly). One can store numbers into variables in Lisp, but the representations of these numbers is such that they are always distinguishable from pointers⁵.

Finally, certain storage management schemes require that we be able to find all *roots* of dynamic data structures. A root, in this context, is a named variable (either static or local) that a program can possibly mention, and therefore might get used by the program. There various ways of insuring that a de-allocation routine (the usual customer) can find all roots. The compiler can leave around the necessary

⁵For example, in one common technique, small integers (say in the range 0–1023) are represented as in most C implementations, but any other integers are actually pointers to structures (called "bignums"). Arithmetic operations always check to see if they have created a big enough number to require allocation of a new dynamic structure.

information. In Lisp systems, the execution-time stacks contain only pointers, and therefore the roots simply comprise the entire stack, the registers, and a few fixed static variables⁶.

We can consider all the dynamic data in a program as a giant graph structure, where objects are the vertices and pointers are the edges. Any dynamically-allocated object that is not reachable from some root will never again be used by the program, and is therefore garbage. The problem is to find this garbage and free it.

10.4.1 Reference counting garbage collection

One way to determine when storage can be deallocated is to keep track of how many copies there are of a pointer to a particular object. When this number reaches 0, the object can no longer be reached from any root, and may therefore be deleted. The most convenient technique is to put a *reference count* in each object (initially 0). Whenever the compiler encounters an assignment of one variable to another,

```
X ← Y;
```

it generates code with the following effect.

```
if (Y is a non-null pointer)
    increment the reference count of Y;
if (X contains a non-null pointer) {
    decrement the reference count of X;
    if (reference count of X == 0)
        freeStructure(X);
}
X = Y;

/** Free the object pointed to by X, decrementing the reference
 * counts of any of its fields. */
void freeStructure(Address X)
{
    for each field, F, in the object pointed to by X {
        if (F contains a non-null pointer) {
            decrement the reference count of F;
            if (reference count of F == 0)
                freeStructure(F);
        }
    }
}
```

The assignment procedure must be used not only for explicit assignments, but also when a function exits (all local variables of the function, including by-value parameters, are in effect assigned NULL), when a value from a variable is passed

⁶For example, there is typically have one static variable that points to a hash table containing all symbols.

as a parameter to a function (in effect, this is an assignment of value to a new variable), when a variable's value is returned from a function (if it is a pointer, this creates a temporary copy of it), and when a function's value is ignored (if it is a pointer, this destroys a copy of it).

Reference counting is used, for example, in the UNIX file system. The objects that contain pointers are directories. They contain pointers ("hard links") to the actual files ("inodes"). Removing a file (the `rm` command) merely removes a certain directory entry and the pointer it contains. Only if this is the last pointer does the file really get deleted.

There is a problem with reference counting: circular structures (like doubly-linked lists) will always have pointers to themselves, even when they cannot be reached from a root⁷. In programs or languages that do not allow circular structures, this poses no particular problem. Otherwise, the system must make some other provision for circular structures (such as periodic marking garbage collection, described below, or 'planned' periodic crashing).

In addition, reference counting (at least in the naive form described here) requires a great deal of work. Each assignment performs incrementing and decrementing, which considerably increases the cost of so otherwise simple an operation. The remaining automatic schemes perform their collection all at once, generally avoiding much of the work of done by reference-counting techniques.

10.4.2 Mark-and-sweep garbage collection

Providing garbage collection of circular structures is a nice practical application of graph traversal. To find the currently-reachable objects, we can perform depth-first traversals starting from each of the roots, where visiting a node does nothing but mark it (the marks may either be on the objects themselves or in a separate bit vector, indexed by object addresses). Doing this clearly requires both that the storage de-allocator be able to find all the pointers in any given object, but also (unlike reference counting) that it be able to find all roots. The objects marked by this traversal (known as the *marking phase*) are precisely the reachable objects; all others are garbage and may be freed. The procedure clearly is not confused by unreachable circular structures—it simply never gets to them.

Assume that all dynamically-allocated objects are laid out consecutively in memory (as is usually the case) and that (as before) we can obtain the size of each object once we have a pointer to it. Then we can collect garbage by means of a *sweep* through memory:

⁷The UNIX directory structure is doubly-linked, which is why one must delete directory structures starting from the leaves and working up, breaking the double-links on the way up.


```

/** Return a list of all unmarked objects between addresses L and
 * U, inclusive. All objects are unmarked at the conclusion. */
ListOfObjects sweepGarbage(Address L, Address U)
{
    ListOfObjects freeList;
    Address m;

    freeList = nullList();

    for (m = L; m <= U; m += objectSize(m)) {
        if (! MARK(m)) {
            (Optional) coalesce the object at m with any following
              free object.
            Place the object at location m of size objectSize(m)
              on freeList.
            SET_MARK(m, FALSE);
        }
    }

    return freeList;
}

```

As you might expect, `objectSize` applied to an `Address` gives the size of the object at that address, while `MARK` and `SET_MARK` manipulate the mark bit associated with it. Figure 10.3 gives a sample configuration of objects just before a garbage collection. Figure 10.4 shows how this might be laid out in storage before sweeping and after marking, and Figure 10.5 shows the configuration after sweeping.

The optional coalesce operation is inappropriate for applications in which there are only a few sizes of objects. In Lisp, for example, most allocations tend to be `cons` cells, and coalescing is not a good idea. When sizes are many and varied, coalescing has the same advantages as in the previous sections on explicit freeing.

The system will typically perform a garbage collection whenever an attempt to allocate storage fails (no sufficiently-large block on the free list). The time required to sweep memory is proportional to the number of objects in it, while the time required for marking is proportional to the total number of roots and of pointers in reachable objects (the latter correspond to the number of edges in a graph).

10.4.3 Copying garbage collection

In mark-and-sweep garbage collection, as for explicit storage allocation, storage can become increasingly fragmented. When there are few distinct object sizes, this is not a problem, of course. Otherwise, one way to overcome this difficulty is to use a type of garbage collection that *copies* reachable objects rather than collecting unreachable ones, in the process *compressing* out unallocated (garbage) space between reachable objects. To do this, we divide storage into two areas, called *to-space* and *from-space*. Before a garbage collection, all dynamic storage

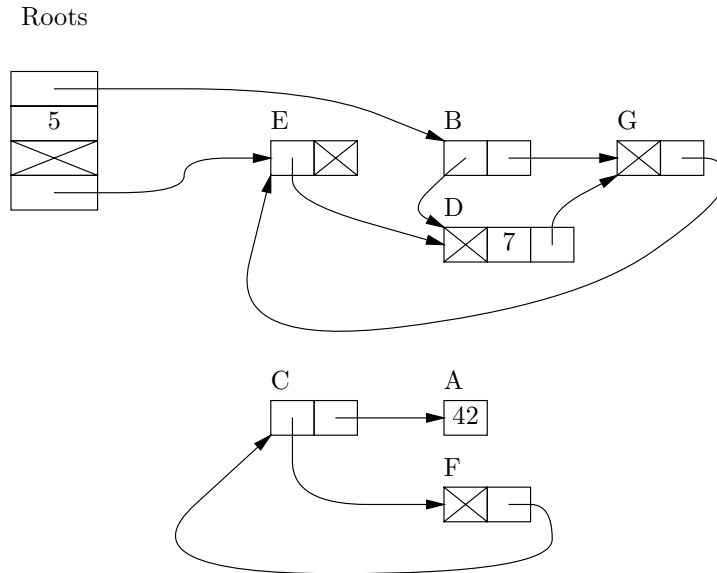


Figure 10.3: An example of dynamically-allocated storage. Labels above the upper left corners of objects are for reference only; they are not variable names. The Roots include all the named variables (their names are not shown). Objects contain either pointers, nulls (crossed out), or other things (represented by numbers here). The objects labeled C, A, and F are unreachable garbage; other objects are reachable and must be preserved during garbage collection.

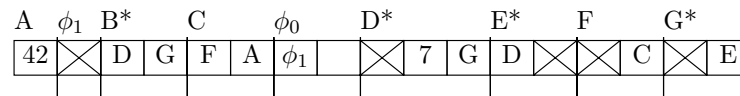


Figure 10.4: A possible layout of the objects depicted in Figure 10.3 just after the marking phase and before the sweep of a garbage collection. Marked nodes are indicated by asterisks. The reference labels from the preceding figure appear at the upper left of each object. To avoid a nest of arrows, pointers are represented by the reference labels of the objects they point to. Objects labeled ϕ_i are on the free list, which starts at ϕ_0 . Presumably there is about to be a garbage collection because the program has made a request for an object larger than two words.

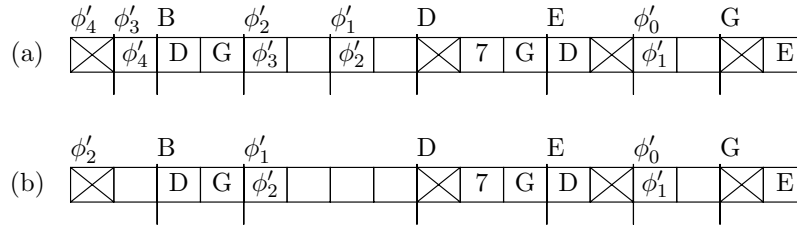


Figure 10.5: State of storage after a sweep that starts with the situation in Figure 10.4 (a) without coalescing and (b) with coalescing of adjacent free areas. The new free list starts at ϕ'_0 .

is in from-space and to-space is empty. The effect of collection is to move the reachable objects in from-space to to-space, changing all pointers in the roots and in the objects themselves to point to the new copies of the objects. At the next garbage collection, to-space and from-space change places, and the contents of what was from-space are simply ignored.

Updating the pointers correctly requires one new trick. If, as we traverse the objects copied from from-space, we encounter a pointer to an object we have previously copied, it is necessary to find the new location of that object. The usual method is to leave behind a *forwarding pointer* in the old object pointing to the new copy (since the old object's contents have been copied to its new location, the system is free to use its storage for such purposes). When we encounter a marked object, we know that it has been copied and that it contains a forwarding pointer, which may use to update the value of the pointer we are processing. The resulting program is given below. Assume that `FETCH(X)` fetches the pointer value at the Address `X` in memory, and `SET(X,V)` sets the contents of Address `X` in memory to `V` (it doesn't change `X` itself). Figure 10.6 illustrates copying garbage collection for the objects shown in Figure 10.3.

```

static Address to_space, from_space;
/** The first free location in to_space */
static Address nextFree;

void copyReachables(void)
{
    Address toDo;

    Swap from_space and to_space.
    nextFree = to_space;

    for each root, R {
        if (R is a pointer into from_space)
            R = copyObject(R);
    }
    /* All roots contain their correct new values */

    for (toDo = to_space; toDo < nextFree; toDo += sizeof(Address)) {
        /* The copied objects between to_space and toDo contain
         * correct pointers to new objects in to_space. Objects between
         * toDo and nextFree contain only pointers into from_space. */
        if (toDo is the address of a pointer field in to_space) {
            if (FETCH(toDo) is a pointer into from_space) {
                if (MARK(FETCH(toDo)))
                    /* FETCH(FETCH(toDo)) is a forwarding pointer */
                    SET(toDo, FETCH(FETCH(toDo)));
                else
                    SET(toDo, copyObject(FETCH(toDo)));
            }
        }
    }
}

/** Copy the from_space object X into to_space, mark X, leave a
 * forwarding pointer, and return the Address of the copy */
static Address copyObject(Address X)
{
    Address newObject = nextFree;
    nextFree += objectSize(X);
    copy objectSize(X) bytes from location X to location newObject
    SET(X, newObject); SET_MARK(X);
    return newObject;
}

```

As you can see from Figure 10.6, the free storage in to-space is contiguous after

garbage collection. The “good” storage has all been collected into one contiguous area, leaving the rest free (this sort of garbage collecting is therefore sometimes called *compacting*). This fact makes subsequent storage allocation extremely easy and fast. There is no free list to search; to allocate n bytes of storage, we simply increment the pointer `nextFree` by n .

Of course, you may well object to the fact that from-space (one half of allocatable storage) is unused between garbage collections. While this is a disadvantage, it is not as bad as it seems. In particular, because of virtual memory, it is not necessary to waste half of the computer’s physical memory. Further improvement is possible through the technique of *generational garbage collection*.

10.4.4 Generational garbage collection

Copying garbage collection shares one problem with mark-and-sweep: long-lived objects are repeatedly traversed, even though they tend not to change very quickly after they are allocated and initialized. Also, in typical programs written for languages like Lisp, objects that become garbage often tend to do so early in their lifetimes. This suggests that it would be nice to restrict garbage collection to young (recently-allocated) objects, ignoring ones that have remain reachable for a certain period of time. With some care, this can be done; the result is known as *generational* garbage collection.

The idea is to divide objects into generations, each in a separate area of memory. Objects are initially “born” into the youngest generation. When the to-space for this generation fills up, it is garbage collected using copying, but pointers into older generations (whose objects were allocated before those in the youngest generation) are mostly ignored (i.e., not traversed). Objects that survive one or more of these collections of the youngest generation (details vary from system to system) are *tenured*—that is, copied into the to-space for the next-older generation. Because objects tend to die young, this older generation fills up much more slowly than the youngest. It is also made to be much larger than the youngest generation, so that the need to perform garbage collection for the older generations is relatively uncommon.

I said that pointers to older objects are “mostly” ignored because older objects generally do not point at younger ones, and so need not be traversed in order to mark and copy younger objects. The reason should be clear—after one allocates an object, one initializes its fields to point to objects that *already exist* and are therefore older than it is. The only time an object contains a pointer to something younger is when one of its fields is assigned to after its initial allocation and initialization. Statistics show these events to be relatively rare in Lisp programs, for example (`cons` is a common operation; `set-car!` is not). Therefore, systems that use generational garbage collection simply keep an array (called a “remembered list”) of pointers to old objects that have had young pointers assigned to their fields. The young-generation pointers in these objects are then counted as roots during garbage collection of the youngest generation.

Generational garbage collection has proven to be extremely effective. In one

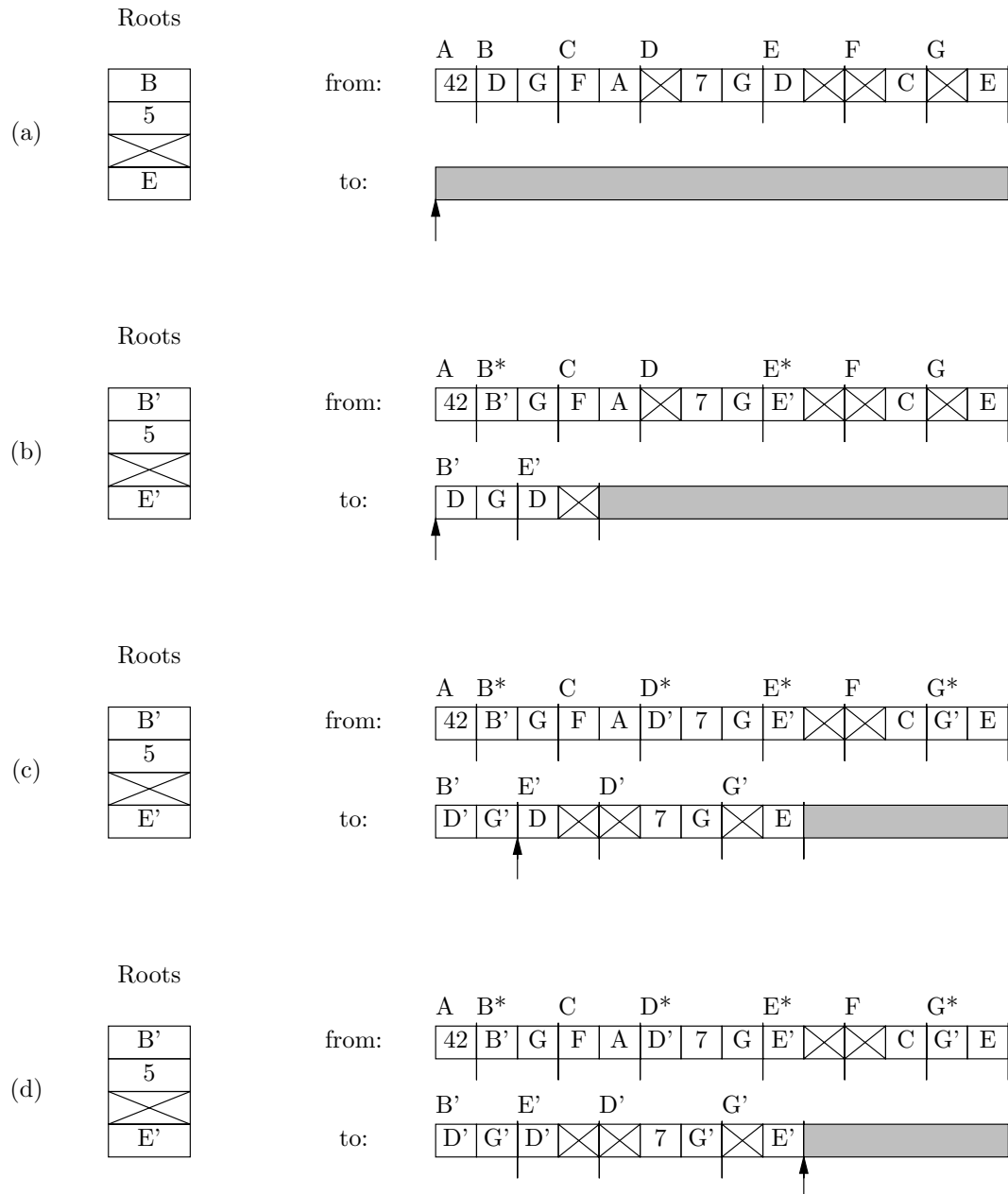


Figure 10.6: Example of copying garbage collection. (a) shows the configuration of from-space and to-space just before copying the roots. The arrow indicates the position of the `toDo` pointer and the area to the right of `nextFree` is shaded. (b) shows the configuration immediately after the roots are copied. Primes after a label distinguish a to-space copy from the from-space original. Marked nodes in from-space are marked with asterisks; their first words have been replaced with forwarding pointers. (c) shows the configuration immediately after `copyReachables` finishes processing the pointers in object B'. (d) shows the final configuration. Nothing further is copied after (c); the rest of the processing involves replacing from-space pointers with the appropriate forwarding pointers.

Smalltalk system developed at Berkeley, generational garbage collection accounted for only about 3% of the total execution time.

10.4.5 Parallel garbage collection

By a simple trick, the copying garbage collectors described above can be made to run simultaneously with the program that is generating garbage (which is called the *mutator*).

The trick does require some coöperation from the operating system: it must be possible to temporarily *read protect* blocks of memory under program control so that when the mutator attempts to read from one of these blocks, it will be interrupted and made to do something else. Our Unix implementation, for example, provides a function `mprotect` that allows a program to set the protection of blocks of memory in units called *pages*.

The technique for getting a parallel algorithm now follows from a few of observations.

- Immediately after the roots are moved, all pointers in the roots are into to-space.
- A program can only read from memory locations a root points to (to compute something like, e.g., `X->tail->tail`, the computer in effect first reads `X->tail` into a register, which is one of the roots).
- The area of to-space that can contain unprocessed pointers (i.e., pointers to old, uncollected objects in from-space) is between `toDo` and `nextFree` while copying garbage collection is in progress.

Therefore, if the garbage collector protects the storage between `toDo` and `newFree` so that whenever the mutator tries to read from there, it is interrupted, we can make sure that the mutator never sees an old, unprocessed pointer into from-space. Whenever it tries to read such a pointer from to-space, it is interrupted, and can be made to wait while the pointers area it is trying to read are processed.