repeat until we find a block that fits. If none of the free lists yields a block that fits, then we request additional heap memory from the operating system, allocate the block out of this new heap memory, and place the remainder in the appropriate size class. To free a block, we coalesce and place the result on the appropriate free list.

The segregated fits approach is a popular choice with production-quality allocators such as the GNU `malloc` package provided in the C standard library because it is both fast and memory efficient. Search times are reduced because searches are limited to particular parts of the heap instead of the entire heap. Memory utilization can improve because of the interesting fact that a simple first-fit search of a segregated free list approximates a best-fit search of the entire heap.

Buddy Systems

A *buddy system* is a special case of segregated fits where each size class is a power of 2. The basic idea is that, given a heap of 2^m words, we maintain a separate free list for each block size 2^k, where $0 \leq k \leq m$. Requested block sizes are rounded up to the nearest power of 2. Originally, there is one free block of size 2^m words.

To allocate a block of size 2^k, we find the first available block of size 2^j, such that $k \leq j \leq m$. If $j = k$, then we are done. Otherwise, we recursively split the block in half until $j = k$. As we perform this splitting, each remaining half (known as a *buddy*) is placed on the appropriate free list. To free a block of size 2^k, we continue coalescing with the free buddies. When we encounter an allocated buddy, we stop the coalescing.

A key fact about buddy systems is that, given the address and size of a block, it is easy to compute the address of its buddy. For example, a block of size 32 bytes with address

$$xxx \ldots x00000$$

has its buddy at address

$$xxx \ldots x10000$$

In other words, the addresses of a block and its buddy differ in exactly one bit position.

The major advantage of a buddy system allocator is its fast searching and coalescing. The major disadvantage is that the power-of-2 requirement on the block size can cause significant internal fragmentation. For this reason, buddy system allocators are not appropriate for general-purpose workloads. However, for certain application-specific workloads, where the block sizes are known in advance to be powers of 2, buddy system allocators have a certain appeal.

9.10 Garbage Collection

With an explicit allocator such as the C `malloc` package, an application allocates and frees heap blocks by making calls to `malloc` and `free`. It is the application's responsibility to free any allocated blocks that it no longer needs.

Failing to free allocated blocks is a common programming error. For example, consider the following C function that allocates a block of temporary storage as part of its processing:

```
1  void garbage()
2  {
3      int *p = (int *)Malloc(15213);
4
5      return; /* Array p is garbage at this point */
6  }
```

Since `p` is no longer needed by the program, it should have been freed before `garbage` returned. Unfortunately, the programmer has forgotten to free the block. It remains allocated for the lifetime of the program, needlessly occupying heap space that could be used to satisfy subsequent allocation requests.

A *garbage collector* is a dynamic storage allocator that automatically frees allocated blocks that are no longer needed by the program. Such blocks are known as *garbage* (hence the term "garbage collector"). The process of automatically reclaiming heap storage is known as *garbage collection*. In a system that supports garbage collection, applications explicitly allocate heap blocks but never explicitly free them. In the context of a C program, the application calls `malloc` but never calls `free`. Instead, the garbage collector periodically identifies the garbage blocks and makes the appropriate calls to `free` to place those blocks back on the free list.

Garbage collection dates back to Lisp systems developed by John McCarthy at MIT in the early 1960s. It is an important part of modern language systems such as Java, ML, Perl, and Mathematica, and it remains an active and important area of research. The literature describes an amazing number of approaches for garbage collection. We will limit our discussion to McCarthy's original *Mark&Sweep* algorithm, which is interesting because it can be built on top of an existing `malloc` package to provide garbage collection for C and C++ programs.

9.10.1 Garbage Collector Basics

A garbage collector views memory as a directed *reachability graph* of the form shown in Figure 9.49. The nodes of the graph are partitioned into a set of *root nodes* and a set of *heap nodes*. Each heap node corresponds to an allocated block in the heap. A directed edge $p \rightarrow q$ means that some location in block p points to some location in block q. Root nodes correspond to locations not in the heap that contain pointers into the heap. These locations can be registers, variables on the stack, or global variables in the read/write data area of virtual memory.

We say that a node p is *reachable* if there exists a directed path from any root node to p. At any point in time, the unreachable nodes correspond to garbage that can never be used again by the application. The role of a garbage collector is to maintain some representation of the reachability graph and periodically reclaim the unreachable nodes by freeing them and returning them to the free list.

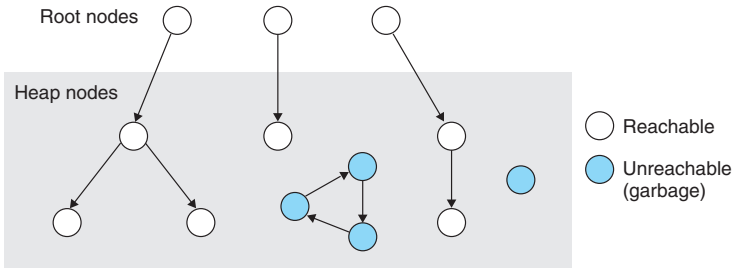

Figure 9.49
A garbage collector's view of memory as a directed graph.

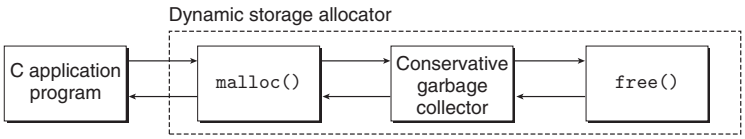

Figure 9.50
Integrating a conservative garbage collector and a C `malloc` package.

Garbage collectors for languages like ML and Java, which exert tight control over how applications create and use pointers, can maintain an exact representation of the reachability graph and thus can reclaim all garbage. However, collectors for languages like C and C++ cannot in general maintain exact representations of the reachability graph. Such collectors are known as *conservative garbage collectors*. They are conservative in the sense that each reachable block is correctly identified as reachable, while some unreachable nodes might be incorrectly identified as reachable.

Collectors can provide their service on demand, or they can run as separate threads in parallel with the application, continuously updating the reachability graph and reclaiming garbage. For example, consider how we might incorporate a conservative collector for C programs into an existing `malloc` package, as shown in Figure 9.50.

The application calls `malloc` in the usual manner whenever it needs heap space. If `malloc` is unable to find a free block that fits, then it calls the garbage collector in hopes of reclaiming some garbage to the free list. The collector identifies the garbage blocks and returns them to the heap by calling the `free` function. The key idea is that the collector calls `free` instead of the application. When the call to the collector returns, `malloc` tries again to find a free block that fits. If that fails, then it can ask the operating system for additional memory. Eventually, `malloc` returns a pointer to the requested block (if successful) or the NULL pointer (if unsuccessful).

9.10.2 Mark&Sweep Garbage Collectors

A Mark&Sweep garbage collector consists of a *mark phase*, which marks all reachable and allocated descendants of the root nodes, followed by a *sweep phase*, which frees each unmarked allocated block. Typically, one of the spare low-order bits in the block header is used to indicate whether a block is marked or not.

(a) `mark` function

```
void mark(ptr p) {
   if ((b = isPtr(p)) == NULL)
     return;
   if (blockMarked(b))
     return;
   markBlock(b);
   len = length(b);
   for (i=0; i < len; i++)
     mark(b[i]);
   return;
}
```

(b) `sweep` function

```
void sweep(ptr b, ptr end) {
   while (b < end) {
      if (blockMarked(b))
         unmarkBlock(b);
      else if (blockAllocated(b))
         free(b);
      b = nextBlock(b);
   }
   return;
}
```

Figure 9.51 Pseudocode for the `mark` and `sweep` functions.

Our description of Mark&Sweep will assume the following functions, where `ptr` is defined as `typedef void *ptr`:

`ptr isPtr(ptr p)`. If `p` points to some word in an allocated block, it returns a pointer `b` to the beginning of that block. Returns NULL otherwise.

`int blockMarked(ptr b)`. Returns `true` if block `b` is already marked.

`int blockAllocated(ptr b)`. Returns `true` if block `b` is allocated.

`void markBlock(ptr b)`. Marks block `b`.

`int length(ptr b)`. Returns the length in words (excluding the header) of block `b`.

`void unmarkBlock(ptr b)`. Changes the status of block `b` from marked to unmarked.

`ptr nextBlock(ptr b)`. Returns the successor of block *b* in the heap.

The mark phase calls the `mark` function shown in Figure 9.51(a) once for each root node. The `mark` function returns immediately if `p` does not point to an allocated and unmarked heap block. Otherwise, it marks the block and calls itself recursively on each word in block. Each call to the `mark` function marks any unmarked and reachable descendants of some root node. At the end of the mark phase, any allocated block that is not marked is guaranteed to be unreachable and, hence, garbage that can be reclaimed in the sweep phase.

The sweep phase is a single call to the `sweep` function shown in Figure 9.51(b). The `sweep` function iterates over each block in the heap, freeing any unmarked allocated blocks (i.e., garbage) that it encounters.

Figure 9.52 shows a graphical interpretation of Mark&Sweep for a small heap. Block boundaries are indicated by heavy lines. Each square corresponds to a word of memory. Each block has a one-word header, which is either marked or unmarked.

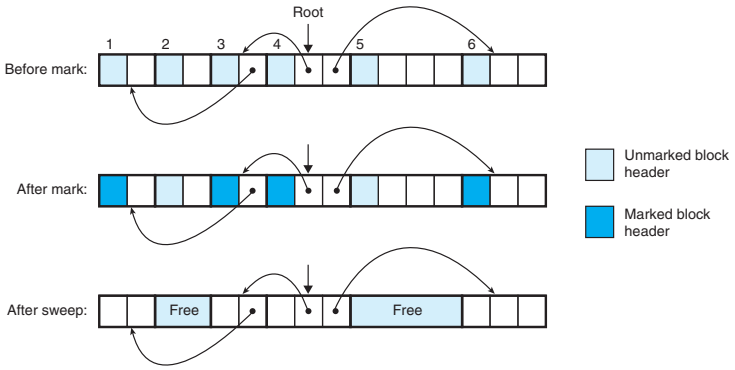

Figure 9.52
Mark&Sweep example. Note that the arrows in this example denote memory references, not free list pointers.

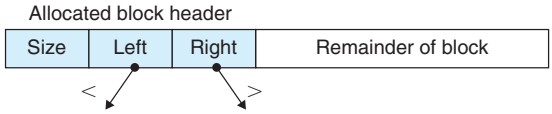

Figure 9.53
Left and right pointers in a balanced tree of allocated blocks.

Initially, the heap in Figure 9.52 consists of six allocated blocks, each of which is unmarked. Block 3 contains a pointer to block 1. Block 4 contains pointers to blocks 3 and 6. The root points to block 4. After the mark phase, blocks 1, 3, 4, and 6 are marked because they are reachable from the root. Blocks 2 and 5 are unmarked because they are unreachable. After the sweep phase, the two unreachable blocks are reclaimed to the free list.

9.10.3 Conservative Mark&Sweep for C Programs

Mark&Sweep is an appropriate approach for garbage collecting C programs because it works in place without moving any blocks. However, the C language poses some interesting challenges for the implementation of the `isPtr` function.

First, C does not tag memory locations with any type information. Thus, there is no obvious way for `isPtr` to determine if its input parameter `p` is a pointer or not. Second, even if we were to know that `p` was a pointer, there would be no obvious way for `isPtr` to determine whether `p` points to some location in the payload of an allocated block.

One solution to the latter problem is to maintain the set of allocated blocks as a balanced binary tree that maintains the invariant that all blocks in the left subtree are located at smaller addresses and all blocks in the right subtree are located in larger addresses. As shown in Figure 9.53, this requires two additional fields (`left` and `right`) in the header of each allocated block. Each field points to the header of some allocated block. The `isPtr(ptr p)` function uses the tree to perform a binary search of the allocated blocks. At each step, it relies on the size field in the block header to determine if `p` falls within the extent of the block.

The balanced tree approach is correct in the sense that it is guaranteed to mark all of the nodes that are reachable from the roots. This is a necessary guarantee, as application users would certainly not appreciate having their allocated blocks prematurely returned to the free list. However, it is conservative in the sense that it may incorrectly mark blocks that are actually unreachable, and thus it may fail to free some garbage. While this does not affect the correctness of application programs, it can result in unnecessary external fragmentation.

The fundamental reason that Mark&Sweep collectors for C programs must be conservative is that the C language does not tag memory locations with type information. Thus, scalars like `int`s or `float`s can masquerade as pointers. For example, suppose that some reachable allocated block contains an `int` in its payload whose value happens to correspond to an address in the payload of some other allocated block *b*. There is no way for the collector to infer that the data is really an `int` and not a pointer. Therefore, the allocator must conservatively mark block *b* as reachable, when in fact it might not be.

9.11 Common Memory-Related Bugs in C Programs

Managing and using virtual memory can be a difficult and error-prone task for C programmers. Memory-related bugs are among the most frightening because they often manifest themselves at a distance, in both time and space, from the source of the bug. Write the wrong data to the wrong location, and your program can run for hours before it finally fails in some distant part of the program. We conclude our discussion of virtual memory with a look at of some of the common memory-related bugs.

9.11.1 Dereferencing Bad Pointers

As we learned in Section 9.7.2, there are large holes in the virtual address space of a process that are not mapped to any meaningful data. If we attempt to dereference a pointer into one of these holes, the operating system will terminate our program with a segmentation exception. Also, some areas of virtual memory are read-only. Attempting to write to one of these areas terminates the program with a protection exception.

A common example of dereferencing a bad pointer is the classic `scanf` bug. Suppose we want to use `scanf` to read an integer from `stdin` into a variable. The correct way to do this is to pass `scanf` a format string and the *address* of the variable:

```
scanf("%d", &val)
```

However, it is easy for new C programmers (and experienced ones too!) to pass the *contents* of `val` instead of its address:

```
scanf("%d", val)
```