asks the kernel to create a new read-only, private, demand-zero area of virtual memory containing `size` bytes. If the call is successful, then `bufp` contains the address of the new area.

The `munmap` function deletes regions of virtual memory:

```
#include <unistd.h>
#include <sys/mman.h>

int munmap(void *start, size_t length);
```

Returns: 0 if OK, −1 on error

The `munmap` function deletes the area starting at virtual address `start` and consisting of the next `length` bytes. Subsequent references to the deleted region result in segmentation faults.

Practice Problem 9.5 **(solution page 918)**

Write a C program `mmapcopy.c` that uses `mmap` to copy an arbitrary-size disk file to `stdout`. The name of the input file should be passed as a command-line argument.

9.9 Dynamic Memory Allocation

While it is certainly possible to use the low-level `mmap` and `munmap` functions to create and delete areas of virtual memory, C programmers typically find it more convenient and more portable to use a *dynamic memory allocator* when they need to acquire additional virtual memory at run time.

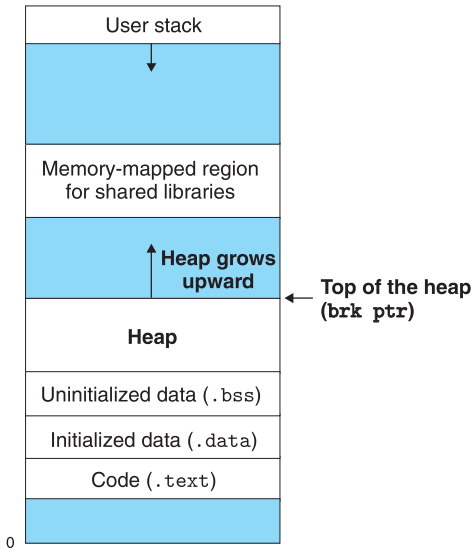
A dynamic memory allocator maintains an area of a process's virtual memory known as the *heap* (Figure 9.33). Details vary from system to system, but without loss of generality, we will assume that the heap is an area of demand-zero memory that begins immediately after the uninitialized data area and grows upward (toward higher addresses). For each process, the kernel maintains a variable `brk` (pronounced "break") that points to the top of the heap.

An allocator maintains the heap as a collection of various-size *blocks*. Each block is a contiguous chunk of virtual memory that is either *allocated* or *free*. An allocated block has been explicitly reserved for use by the application. A free block is available to be allocated. A free block remains free until it is explicitly allocated by the application. An allocated block remains allocated until it is freed, either explicitly by the application or implicitly by the memory allocator itself.

Allocators come in two basic styles. Both styles require the application to explicitly allocate blocks. They differ about which entity is responsible for freeing allocated blocks.

- *Explicit allocators* require the application to explicitly free any allocated blocks. For example, the C standard library provides an explicit allocator called the `malloc` package. C programs allocate a block by calling the `malloc`

Figure 9.33
The heap.



function, and free a block by calling the `free` function. The `new` and `delete` calls in `C++` are comparable.

- *Implicit allocators*, on the other hand, require the allocator to detect when an allocated block is no longer being used by the program and then free the block. Implicit allocators are also known as *garbage collectors*, and the process of automatically freeing unused allocated blocks is known as *garbage collection*. For example, higher-level languages such as Lisp, ML, and Java rely on garbage collection to free allocated blocks.

The remainder of this section discusses the design and implementation of explicit allocators. We will discuss implicit allocators in Section 9.10. For concreteness, our discussion focuses on allocators that manage heap memory. However, you should be aware that memory allocation is a general idea that arises in a variety of contexts. For example, applications that do intensive manipulation of graphs will often use the standard allocator to acquire a large block of virtual memory and then use an application-specific allocator to manage the memory within that block as the nodes of the graph are created and destroyed.

9.9.1 The `malloc` and `free` Functions

The C standard library provides an explicit allocator known as the `malloc` package. Programs allocate blocks from the heap by calling the `malloc` function.

```
#include <stdlib.h>

void *malloc(size_t size);
```

Returns: pointer to allocated block if OK, NULL on error

Aside How big is a word?

Recall from our discussion of machine code in Chapter 3 that Intel refers to 4-byte objects as *double words*. However, throughout this section, we will assume that *words* are 4-byte objects and that *double words* are 8-byte objects, which is consistent with conventional terminology.

The `malloc` function returns a pointer to a block of memory of at least `size` bytes that is suitably aligned for any kind of data object that might be contained in the block. In practice, the alignment depends on whether the code is compiled to run in 32-bit mode (`gcc -m32`) or 64-bit mode (the default). In 32-bit mode, `malloc` returns a block whose address is always a multiple of 8. In 64-bit mode, the address is always a multiple of 16.

If `malloc` encounters a problem (e.g., the program requests a block of memory that is larger than the available virtual memory), then it returns NULL and sets `errno`. `Malloc` does not initialize the memory it returns. Applications that want initialized dynamic memory can use `calloc`, a thin wrapper around the `malloc` function that initializes the allocated memory to zero. Applications that want to change the size of a previously allocated block can use the `realloc` function.

Dynamic memory allocators such as `malloc` can allocate or deallocate heap memory explicitly by using the `mmap` and `munmap` functions, or they can use the `sbrk` function:

```
#include <unistd.h>

void *sbrk(intptr_t incr);
                                    Returns: old brk pointer on success, −1 on error
```

The `sbrk` function grows or shrinks the heap by adding `incr` to the kernel's `brk` pointer. If successful, it returns the old value of `brk`, otherwise it returns −1 and sets `errno` to ENOMEM. If `incr` is zero, then `sbrk` returns the current value of `brk`. Calling `sbrk` with a negative `incr` is legal but tricky because the return value (the old value of `brk`) points to `abs(incr)` bytes past the new top of the heap.

Programs free allocated heap blocks by calling the `free` function.

```
#include <stdlib.h>

void free(void *ptr);
                                                    Returns: nothing
```

The `ptr` argument must point to the beginning of an allocated block that was obtained from `malloc`, `calloc`, or `realloc`. If not, then the behavior of `free` is undefined. Even worse, since it returns nothing, `free` gives no indication to the application that something is wrong. As we shall see in Section 9.11, this can produce some baffling run-time errors.

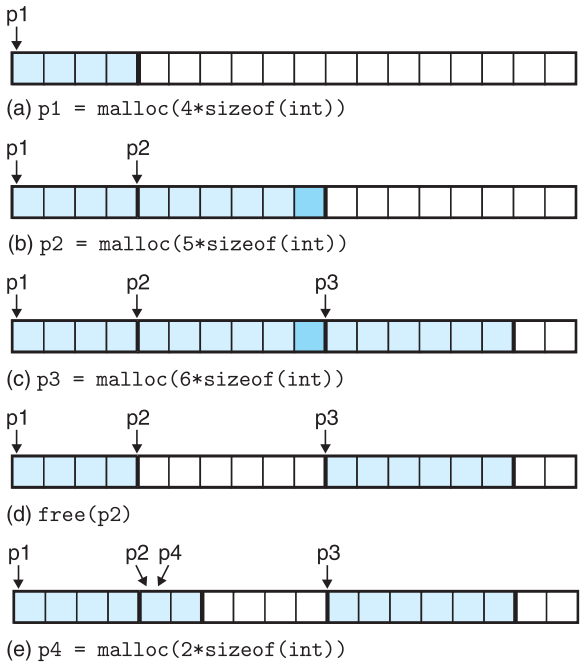


Figure 9.34
Allocating and freeing blocks with `malloc` and `free`. Each square corresponds to a word. Each heavy rectangle corresponds to a block. Allocated blocks are shaded. Padded regions of allocated blocks are shaded with a darker blue. Free blocks are unshaded. Heap addresses increase from left to right.

Figure 9.34 shows how an implementation of `malloc` and `free` might manage a (very) small heap of 16 words for a C program. Each box represents a 4-byte word. The heavy-lined rectangles correspond to allocated blocks (shaded) and free blocks (unshaded). Initially, the heap consists of a single 16-word double-word-aligned free block.[1]

Figure 9.34(a). The program asks for a four-word block. `Malloc` responds by carving out a four-word block from the front of the free block and returning a pointer to the first word of the block.

Figure 9.34(b). The program requests a five-word block. `Malloc` responds by allocating a six-word block from the front of the free block. In this example, `malloc` pads the block with an extra word in order to keep the free block aligned on a double-word boundary.

Figure 9.34(c). The program requests a six-word block and `malloc` responds by carving out a six-word block from the free block.

Figure 9.34(d). The program frees the six-word block that was allocated in Figure 9.34(b). Notice that after the call to `free` returns, the pointer `p2`

1. Throughout this section, we will assume that the allocator returns blocks aligned to 8-byte double-word boundaries.

still points to the freed block. It is the responsibility of the application not to use p2 again until it is reinitialized by a new call to malloc.

Figure 9.34(e). The program requests a two-word block. In this case, malloc allocates a portion of the block that was freed in the previous step and returns a pointer to this new block.

9.9.2 Why Dynamic Memory Allocation?

The most important reason that programs use dynamic memory allocation is that often they do not know the sizes of certain data structures until the program actually runs. For example, suppose we are asked to write a C program that reads a list of n ASCII integers, one integer per line, from stdin into a C array. The input consists of the integer n, followed by the n integers to be read and stored into the array. The simplest approach is to define the array statically with some hard-coded maximum array size:

```
1   #include "csapp.h"
2   #define MAXN 15213
3
4   int array[MAXN];
5
6   int main()
7   {
8       int i, n;
9
10      scanf("%d", &n);
11      if (n > MAXN)
12          app_error("Input file too big");
13      for (i = 0; i < n; i++)
14          scanf("%d", &array[i]);
15      exit(0);
16  }
```

Allocating arrays with hard-coded sizes like this is often a bad idea. The value of MAXN is arbitrary and has no relation to the actual amount of available virtual memory on the machine. Further, if the user of this program wanted to read a file that was larger than MAXN, the only recourse would be to recompile the program with a larger value of MAXN. While not a problem for this simple example, the presence of hard-coded array bounds can become a maintenance nightmare for large software products with millions of lines of code and numerous users.

A better approach is to allocate the array dynamically, at run time, after the value of n becomes known. With this approach, the maximum size of the array is limited only by the amount of available virtual memory.

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int *array, i, n;
6
7       scanf("%d", &n);
8       array = (int *)Malloc(n * sizeof(int));
9       for (i = 0; i < n; i++)
10          scanf("%d", &array[i]);
11      free(array);
12      exit(0);
13  }
```

Dynamic memory allocation is a useful and important programming technique. However, in order to use allocators correctly and efficiently, programmers need to have an understanding of how they work. We will discuss some of the gruesome errors that can result from the improper use of allocators in Section 9.11.

9.9.3 Allocator Requirements and Goals

Explicit allocators must operate within some rather stringent constraints:

Handling arbitrary request sequences. An application can make an arbitrary sequence of allocate and free requests, subject to the constraint that each free request must correspond to a currently allocated block obtained from a previous allocate request. Thus, the allocator cannot make any assumptions about the ordering of allocate and free requests. For example, the allocator cannot assume that all allocate requests are accompanied by a matching free request, or that matching allocate and free requests are nested.

Making immediate responses to requests. The allocator must respond immediately to allocate requests. Thus, the allocator is not allowed to reorder or buffer requests in order to improve performance.

Using only the heap. In order for the allocator to be scalable, any nonscalar data structures used by the allocator must be stored in the heap itself.

Aligning blocks (alignment requirement). The allocator must align blocks in such a way that they can hold any type of data object.

Not modifying allocated blocks. Allocators can only manipulate or change free blocks. In particular, they are not allowed to modify or move blocks once they are allocated. Thus, techniques such as compaction of allocated blocks are not permitted.

Working within these constraints, the author of an allocator attempts to meet the often conflicting performance goals of maximizing throughput and memory utilization.

Goal 1: Maximizing throughput. Given some sequence of n allocate and free requests

$$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

we would like to maximize an allocator's *throughput*, which is defined as the number of requests that it completes per unit time. For example, if an allocator completes 500 allocate requests and 500 free requests in 1 second, then its throughput is 1,000 operations per second. In general, we can maximize throughput by minimizing the average time to satisfy allocate and free requests. As we'll see, it is not too difficult to develop allocators with reasonably good performance where the worst-case running time of an allocate request is linear in the number of free blocks and the running time of a free request is constant.

Goal 2: Maximizing memory utilization. Naive programmers often incorrectly assume that virtual memory is an unlimited resource. In fact, the total amount of virtual memory allocated by all of the processes in a system is limited by the amount of swap space on disk. Good programmers know that virtual memory is a finite resource that must be used efficiently. This is especially true for a dynamic memory allocator that might be asked to allocate and free large blocks of memory.

There are a number of ways to characterize how efficiently an allocator uses the heap. In our experience, the most useful metric is *peak utilization*. As before, we are given some sequence of n allocate and free requests

$$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

If an application requests a block of p bytes, then the resulting allocated block has a *payload* of p bytes. After request R_k has completed, let the *aggregate payload*, denoted P_k, be the sum of the payloads of the currently allocated blocks, and let H_k denote the current (monotonically nondecreasing) size of the heap.

Then the peak utilization over the first $k+1$ requests, denoted by U_k, is given by

$$U_k = \frac{\max_{i \leq k} P_i}{H_k}$$

The objective of the allocator, then, is to maximize the peak utilization U_{n-1} over the entire sequence. As we will see, there is a tension between maximizing throughput and utilization. In particular, it is easy to write an allocator that maximizes throughput at the expense of heap utilization. One of the interesting challenges in any allocator design is finding an appropriate balance between the two goals.

Aside Relaxing the monotonicity assumption

We could relax the monotonically nondecreasing assumption in our definition of U_k and allow the heap to grow up and down by letting H_k be the high-water mark over the first $k + 1$ requests.

9.9.4 Fragmentation

The primary cause of poor heap utilization is a phenomenon known as *fragmentation*, which occurs when otherwise unused memory is not available to satisfy allocate requests. There are two forms of fragmentation: *internal fragmentation* and *external fragmentation*.

Internal fragmentation occurs when an allocated block is larger than the payload. This might happen for a number of reasons. For example, the implementation of an allocator might impose a minimum size on allocated blocks that is greater than some requested payload. Or, as we saw in Figure 9.34(b), the allocator might increase the block size in order to satisfy alignment constraints.

Internal fragmentation is straightforward to quantify. It is simply the sum of the differences between the sizes of the allocated blocks and their payloads. Thus, at any point in time, the amount of internal fragmentation depends only on the pattern of previous requests and the allocator implementation.

External fragmentation occurs when there *is* enough aggregate free memory to satisfy an allocate request, but no single free block is large enough to handle the request. For example, if the request in Figure 9.34(e) were for eight words rather than two words, then the request could not be satisfied without requesting additional virtual memory from the kernel, even though there are eight free words remaining in the heap. The problem arises because these eight words are spread over two free blocks.

External fragmentation is much more difficult to quantify than internal fragmentation because it depends not only on the pattern of previous requests and the allocator implementation but also on the pattern of *future* requests. For example, suppose that after k requests all of the free blocks are exactly four words in size. Does this heap suffer from external fragmentation? The answer depends on the pattern of future requests. If all of the future allocate requests are for blocks that are smaller than or equal to four words, then there is no external fragmentation. On the other hand, if one or more requests ask for blocks larger than four words, then the heap does suffer from external fragmentation.

Since external fragmentation is difficult to quantify and impossible to predict, allocators typically employ heuristics that attempt to maintain small numbers of larger free blocks rather than large numbers of smaller free blocks.

9.9.5 Implementation Issues

The simplest imaginable allocator would organize the heap as a large array of bytes and a pointer p that initially points to the first byte of the array. To allocate

`size` bytes, `malloc` would save the current value of `p` on the stack, increment `p` by `size`, and return the old value of `p` to the caller. `Free` would simply return to the caller without doing anything.

This naive allocator is an extreme point in the design space. Since each `malloc` and `free` execute only a handful of instructions, throughput would be extremely good. However, since the allocator never reuses any blocks, memory utilization would be extremely bad. A practical allocator that strikes a better balance between throughput and utilization must consider the following issues:

Free block organization. How do we keep track of free blocks?

Placement. How do we choose an appropriate free block in which to place a newly allocated block?

Splitting. After we place a newly allocated block in some free block, what do we do with the remainder of the free block?

Coalescing. What do we do with a block that has just been freed?

The rest of this section looks at these issues in more detail. Since the basic techniques of placement, splitting, and coalescing cut across many different free block organizations, we will introduce them in the context of a simple free block organization known as an implicit free list.

9.9.6 Implicit Free Lists

Any practical allocator needs some data structure that allows it to distinguish block boundaries and to distinguish between allocated and free blocks. Most allocators embed this information in the blocks themselves. One simple approach is shown in Figure 9.35.

In this case, a block consists of a one-word *header*, the payload, and possibly some additional *padding*. The header encodes the block size (including the header and any padding) as well as whether the block is allocated or free. If we impose a double-word alignment constraint, then the block size is always a multiple of 8 and the 3 low-order bits of the block size are always zero. Thus, we need to store only the 29 high-order bits of the block size, freeing the remaining 3 bits to encode other information. In this case, we are using the least significant of these bits

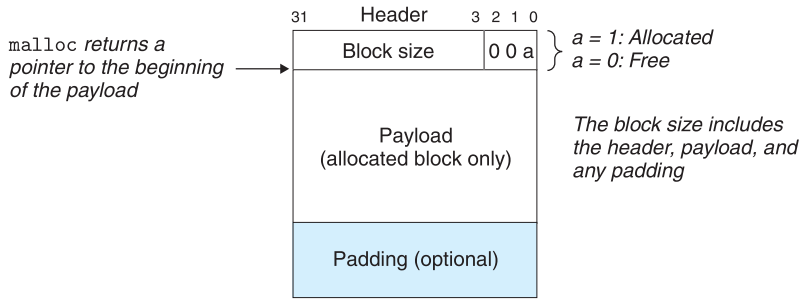


Figure 9.35
Format of a simple heap block.

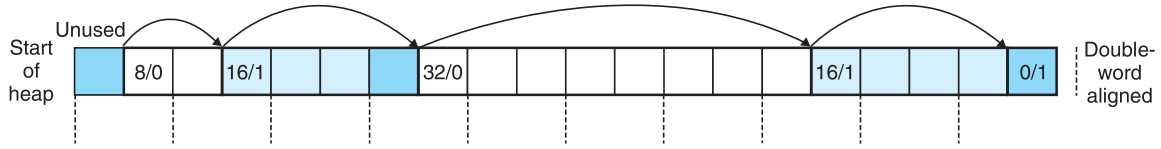


Figure 9.36 Organizing the heap with an implicit free list. Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

(the *allocated bit*) to indicate whether the block is allocated or free. For example, suppose we have an allocated block with a block size of 24 (`0x18`) bytes. Then its header would be

```
0x00000018 | 0x1 = 0x00000019
```

Similarly, a free block with a block size of 40 (`0x28`) bytes would have a header of

```
0x00000028 | 0x0 = 0x00000028
```

The header is followed by the payload that the application requested when it called `malloc`. The payload is followed by a chunk of unused padding that can be any size. There are a number of reasons for the padding. For example, the padding might be part of an allocator's strategy for combating external fragmentation. Or it might be needed to satisfy the alignment requirement.

Given the block format in Figure 9.35, we can organize the heap as a sequence of contiguous allocated and free blocks, as shown in Figure 9.36.

We call this organization an *implicit free list* because the free blocks are linked implicitly by the size fields in the headers. The allocator can indirectly traverse the entire set of free blocks by traversing *all* of the blocks in the heap. Notice that we need some kind of specially marked end block—in this example, a terminating header with the allocated bit set and a size of zero. (As we will see in Section 9.9.12, setting the allocated bit simplifies the coalescing of free blocks.)

The advantage of an implicit free list is simplicity. A significant disadvantage is that the cost of any operation that requires a search of the free list, such as placing allocated blocks, will be linear in the *total* number of allocated and free blocks in the heap.

It is important to realize that the system's alignment requirement and the allocator's choice of block format impose a *minimum block size* on the allocator. No allocated or free block may be smaller than this minimum. For example, if we assume a double-word alignment requirement, then the size of each block must be a multiple of two words (8 bytes). Thus, the block format in Figure 9.35 induces a minimum block size of two words: one word for the header and another to maintain the alignment requirement. Even if the application were to request a single byte, the allocator would still create a two-word block.

Practice Problem 9.6 (solution page 919)

Determine the block sizes and header values that would result from the following sequence of `malloc` requests. Assumptions: (1) The allocator maintains double-word alignment and uses an implicit free list with the block format from Figure 9.35. (2) Block sizes are rounded up to the nearest multiple of 8 bytes.

Request	Block size (decimal bytes)	Block header (hex)
`malloc(2)`	_____	_____
`malloc(9)`	_____	_____
`malloc(15)`	_____	_____
`malloc(20)`	_____	_____

9.9.7 Placing Allocated Blocks

When an application requests a block of k bytes, the allocator searches the free list for a free block that is large enough to hold the requested block. The manner in which the allocator performs this search is determined by the *placement policy*. Some common policies are first fit, next fit, and best fit.

First fit searches the free list from the beginning and chooses the first free block that fits. *Next fit* is similar to first fit, but instead of starting each search at the beginning of the list, it starts each search where the previous search left off. *Best fit* examines every free block and chooses the free block with the smallest size that fits.

An advantage of first fit is that it tends to retain large free blocks at the end of the list. A disadvantage is that it tends to leave "splinters" of small free blocks toward the beginning of the list, which will increase the search time for larger blocks. Next fit was first proposed by Donald Knuth as an alternative to first fit, motivated by the idea that if we found a fit in some free block the last time, there is a good chance that we will find a fit the next time in the remainder of the block. Next fit can run significantly faster than first fit, especially if the front of the list becomes littered with many small splinters. However, some studies suggest that next fit suffers from worse memory utilization than first fit. Studies have found that best fit generally enjoys better memory utilization than either first fit or next fit. However, the disadvantage of using best fit with simple free list organizations such as the implicit free list is that it requires an exhaustive search of the heap. Later, we will look at more sophisticated segregated free list organizations that approximate a best-fit policy without an exhaustive search of the heap.

9.9.8 Splitting Free Blocks

Once the allocator has located a free block that fits, it must make another policy decision about how much of the free block to allocate. One option is to use the entire free block. Although simple and fast, the main disadvantage is that it

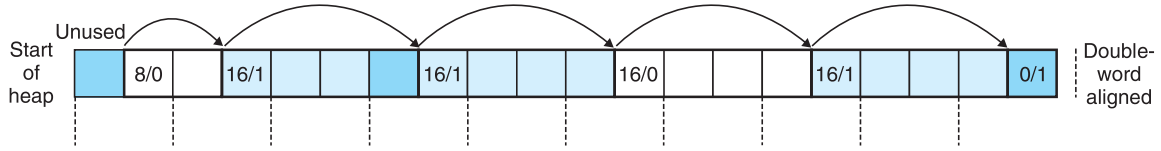


Figure 9.37 Splitting a free block to satisfy a three-word allocation request. Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

introduces internal fragmentation. If the placement policy tends to produce good fits, then some additional internal fragmentation might be acceptable.

However, if the fit is not good, then the allocator will usually opt to *split* the free block into two parts. The first part becomes the allocated block, and the remainder becomes a new free block. Figure 9.37 shows how the allocator might split the eight-word free block in Figure 9.36 to satisfy an application's request for three words of heap memory.

9.9.9 Getting Additional Heap Memory

What happens if the allocator is unable to find a fit for the requested block? One option is to try to create some larger free blocks by merging (coalescing) free blocks that are physically adjacent in memory (next section). However, if this does not yield a sufficiently large block, or if the free blocks are already maximally coalesced, then the allocator asks the kernel for additional heap memory by calling the `sbrk` function. The allocator transforms the additional memory into one large free block, inserts the block into the free list, and then places the requested block in this new free block.

9.9.10 Coalescing Free Blocks

When the allocator frees an allocated block, there might be other free blocks that are adjacent to the newly freed block. Such adjacent free blocks can cause a phenomenon known as *false fragmentation*, where there is a lot of available free memory chopped up into small, unusable free blocks. For example, Figure 9.38 shows the result of freeing the block that was allocated in Figure 9.37. The result is two adjacent free blocks with payloads of three words each. As a result, a subsequent request for a payload of four words would fail, even though the aggregate size of the two free blocks is large enough to satisfy the request.

To combat false fragmentation, any practical allocator must merge adjacent free blocks in a process known as *coalescing*. This raises an important policy decision about when to perform coalescing. The allocator can opt for *immediate coalescing* by merging any adjacent blocks each time a block is freed. Or it can opt for *deferred coalescing* by waiting to coalesce free blocks at some later time. For example, the allocator might defer coalescing until some allocation request fails, and then scan the entire heap, coalescing all free blocks.

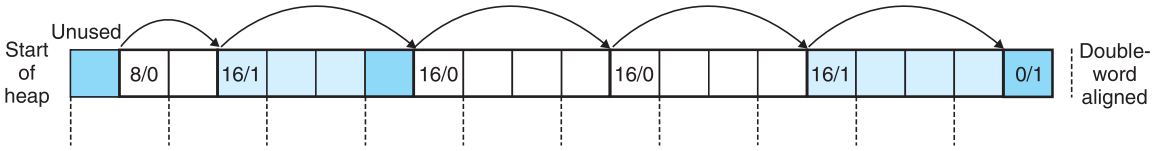


Figure 9.38 An example of false fragmentation. Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

Immediate coalescing is straightforward and can be performed in constant time, but with some request patterns it can introduce a form of thrashing where a block is repeatedly coalesced and then split soon thereafter. For example, in Figure 9.38, a repeated pattern of allocating and freeing a three-word block would introduce a lot of unnecessary splitting and coalescing. In our discussion of allocators, we will assume immediate coalescing, but you should be aware that fast allocators often opt for some form of deferred coalescing.

9.9.11 Coalescing with Boundary Tags

How does an allocator implement coalescing? Let us refer to the block we want to free as the *current block*. Then coalescing the next free block (in memory) is straightforward and efficient. The header of the current block points to the header of the next block, which can be checked to determine if the next block is free. If so, its size is simply added to the size of the current header and the blocks are coalesced in constant time.

But how would we coalesce the previous block? Given an implicit free list of blocks with headers, the only option would be to search the entire list, remembering the location of the previous block, until we reached the current block. With an implicit free list, this means that each call to `free` would require time linear in the size of the heap. Even with more sophisticated free list organizations, the search time would not be constant.

Knuth developed a clever and general technique, known as *boundary tags*, that allows for constant-time coalescing of the previous block. The idea, which is shown in Figure 9.39, is to add a *footer* (the boundary tag) at the end of each block, where the footer is a replica of the header. If each block includes such a footer, then the allocator can determine the starting location and status of the previous block by inspecting its footer, which is always one word away from the start of the current block.

Consider all the cases that can exist when the allocator frees the current block:

1. The previous and next blocks are both allocated.
2. The previous block is allocated and the next block is free.
3. The previous block is free and the next block is allocated.
4. The previous and next blocks are both free.

Figure 9.39
Format of heap block that uses a boundary tag.

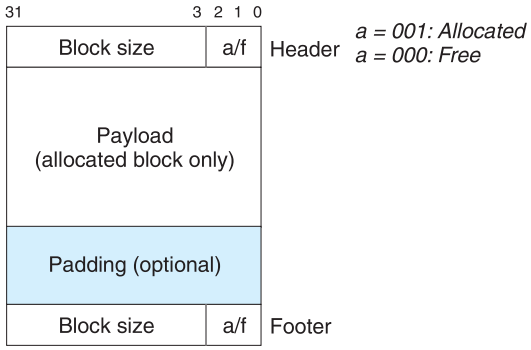


Figure 9.40 shows how we would coalesce each of the four cases.

In case 1, both adjacent blocks are allocated and thus no coalescing is possible. So the status of the current block is simply changed from allocated to free. In case 2, the current block is merged with the next block. The header of the current block and the footer of the next block are updated with the combined sizes of the current and next blocks. In case 3, the previous block is merged with the current block. The header of the previous block and the footer of the current block are updated with the combined sizes of the two blocks. In case 4, all three blocks are merged to form a single free block, with the header of the previous block and the footer of the next block updated with the combined sizes of the three blocks. In each case, the coalescing is performed in constant time.

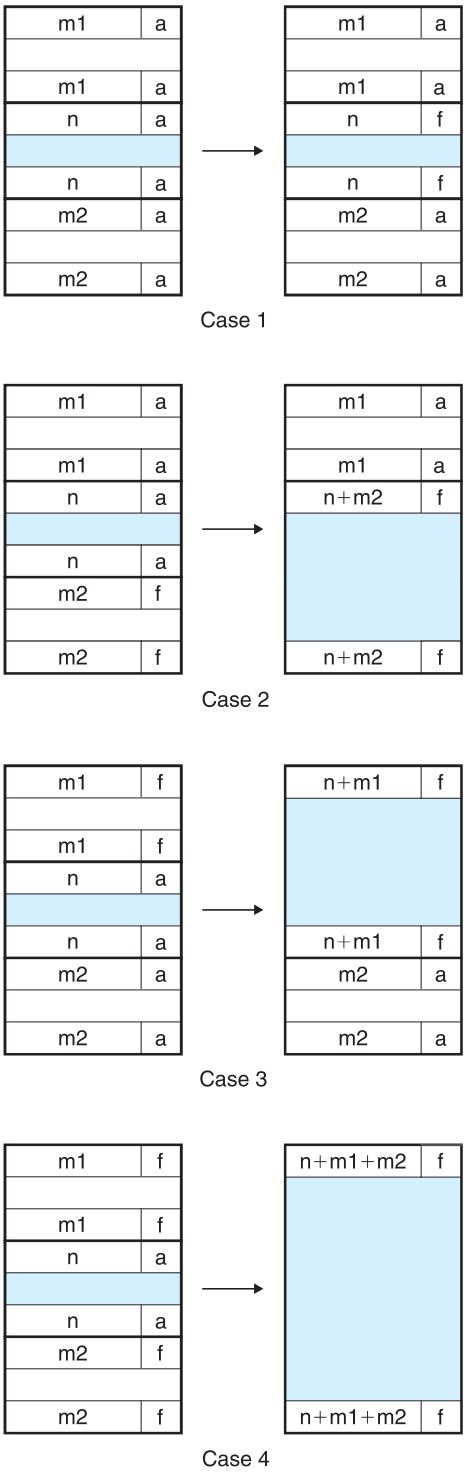
The idea of boundary tags is a simple and elegant one that generalizes to many different types of allocators and free list organizations. However, there is a potential disadvantage. Requiring each block to contain both a header and a footer can introduce significant memory overhead if an application manipulates many small blocks. For example, if a graph application dynamically creates and destroys graph nodes by making repeated calls to `malloc` and `free`, and each graph node requires only a couple of words of memory, then the header and the footer will consume half of each allocated block.

Fortunately, there is a clever optimization of boundary tags that eliminates the need for a footer in allocated blocks. Recall that when we attempt to coalesce the current block with the previous and next blocks in memory, the size field in the footer of the previous block is only needed if the previous block is *free*. If we were to store the allocated/free bit of the previous block in one of the excess low-order bits of the current block, then allocated blocks would not need footers, and we could use that extra space for payload. Note, however, that free blocks would still need footers.

Practice Problem 9.7 (solution page 919)

Determine the minimum block size for each of the following combinations of alignment requirements and block formats. Assumptions: Implicit free list, zero-size payloads are not allowed, and headers and footers are stored in 4-byte words.

Figure 9.40
Coalescing with boundary tags. Case 1: prev and next allocated. Case 2: prev allocated, next free. Case 3: prev free, next allocated. Case 4: next and prev free.



Alignment	Allocated block	Free block	Minimum block size (bytes)
Single word	Header and footer	Header and footer	_____
Single word	Header, but no footer	Header and footer	_____
Double word	Header and footer	Header and footer	_____
Double word	Header, but no footer	Header and footer	_____

9.9.12 Putting It Together: Implementing a Simple Allocator

Building an allocator is a challenging task. The design space is large, with numerous alternatives for block format and free list format, as well as placement, splitting, and coalescing policies. Another challenge is that you are often forced to program outside the safe, familiar confines of the type system, relying on the error-prone pointer casting and pointer arithmetic that is typical of low-level systems programming.

While allocators do not require enormous amounts of code, they are subtle and unforgiving. Students familiar with higher-level languages such as C++ or Java often hit a conceptual wall when they first encounter this style of programming. To help you clear this hurdle, we will work through the implementation of a simple allocator based on an implicit free list with immediate boundary-tag coalescing. The maximum block size is $2^{32} = 4$ GB. The code is 64-bit clean, running without modification in 32-bit (`gcc -m32`) or 64-bit (`gcc -m64`) processes.

General Allocator Design

Our allocator uses a model of the memory system provided by the `memlib.c` package shown in Figure 9.41. The purpose of the model is to allow us to run our allocator without interfering with the existing system-level `malloc` package.

The `mem_init` function models the virtual memory available to the heap as a large double-word aligned array of bytes. The bytes between `mem_heap` and `mem_brk` represent allocated virtual memory. The bytes following `mem_brk` represent unallocated virtual memory. The allocator requests additional heap memory by calling the `mem_sbrk` function, which has the same interface as the system's `sbrk` function, as well as the same semantics, except that it rejects requests to shrink the heap.

The allocator itself is contained in a source file (`mm.c`) that users can compile and link into their applications. The allocator exports three functions to application programs:

```
1   extern int mm_init(void);
2   extern void *mm_malloc (size_t size);
3   extern void mm_free (void *ptr);
```

The `mm_init` function initializes the allocator, returning 0 if successful and −1 otherwise. The `mm_malloc` and `mm_free` functions have the same interfaces and semantics as their system counterparts. The allocator uses the block format

code/vm/malloc/memlib.c

```
1   /* Private global variables */
2   static char *mem_heap;     /* Points to first byte of heap */
3   static char *mem_brk;      /* Points to last byte of heap plus 1 */
4   static char *mem_max_addr; /* Max legal heap addr plus 1*/
5
6   /*
7    * mem_init - Initialize the memory system model
8    */
9   void mem_init(void)
10  {
11      mem_heap = (char *)Malloc(MAX_HEAP);
12      mem_brk = (char *)mem_heap;
13      mem_max_addr = (char *)(mem_heap + MAX_HEAP);
14  }
15
16  /*
17   * mem_sbrk - Simple model of the sbrk function. Extends the heap
18   *    by incr bytes and returns the start address of the new area. In
19   *    this model, the heap cannot be shrunk.
20   */
21  void *mem_sbrk(int incr)
22  {
23      char *old_brk = mem_brk;
24
25      if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
26          errno = ENOMEM;
27          fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
28          return (void *)-1;
29      }
30      mem_brk += incr;
31      return (void *)old_brk;
32  }
```

code/vm/malloc/memlib.c

Figure 9.41 `memlib.c`: **Memory system model.**

shown in Figure 9.39. The minimum block size is 16 bytes. The free list is organized as an implicit free list, with the invariant form shown in Figure 9.42.

The first word is an unused padding word aligned to a double-word boundary. The padding is followed by a special *prologue block*, which is an 8-byte allocated block consisting of only a header and a footer. The prologue block is created during initialization and is never freed. Following the prologue block are zero or more regular blocks that are created by calls to `malloc` or `free`. The heap always ends with a special *epilogue block*, which is a zero-size allocated block

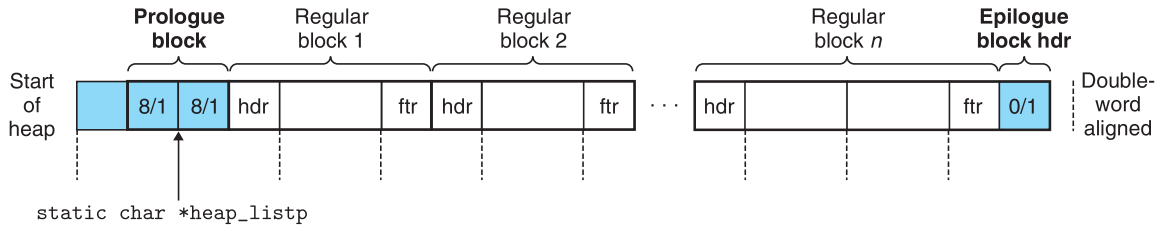


Figure 9.42 **Invariant form of the implicit free list.**

that consists of only a header. The prologue and epilogue blocks are tricks that eliminate the edge conditions during coalescing. The allocator uses a single private (`static`) global variable (`heap_listp`) that always points to the prologue block. (As a minor optimization, we could make it point to the next block instead of the prologue block.)

Basic Constants and Macros for Manipulating the Free List

Figure 9.43 shows some basic constants and macros that we will use throughout the allocator code. Lines 2–4 define some basic size constants: the sizes of words (WSIZE) and double words (DSIZE), and the size of the initial free block and the default size for expanding the heap (CHUNKSIZE).

Manipulating the headers and footers in the free list can be troublesome because it demands extensive use of casting and pointer arithmetic. Thus, we find it helpful to define a small set of macros for accessing and traversing the free list (lines 9–25). The PACK macro (line 9) combines a size and an allocate bit and returns a value that can be stored in a header or footer.

The GET macro (line 12) reads and returns the word referenced by argument `p`. The casting here is crucial. The argument `p` is typically a (`void *`) pointer, which cannot be dereferenced directly. Similarly, the PUT macro (line 13) stores `val` in the word pointed at by argument `p`.

The GET_SIZE and GET_ALLOC macros (lines 16–17) return the size and allocated bit, respectively, from a header or footer at address `p`. The remaining macros operate on *block pointers* (denoted `bp`) that point to the first payload byte. Given a block pointer `bp`, the HDRP and FTRP macros (lines 20–21) return pointers to the block header and footer, respectively. The NEXT_BLKP and PREV_BLKP macros (lines 24–25) return the block pointers of the next and previous blocks, respectively.

The macros can be composed in various ways to manipulate the free list. For example, given a pointer `bp` to the current block, we could use the following line of code to determine the size of the next block in memory:

```
size_t size = GET_SIZE(HDRP(NEXT_BLKP(bp)));
```

code/vm/malloc/mm.c

```
1   /* Basic constants and macros */
2   #define WSIZE       4       /* Word and header/footer size (bytes) */
3   #define DSIZE       8       /* Double word size (bytes) */
4   #define CHUNKSIZE  (1<<12)  /* Extend heap by this amount (bytes) */
5
6   #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8   /* Pack a size and allocated bit into a word */
9   #define PACK(size, alloc)  ((size) | (alloc))
10
11  /* Read and write a word at address p */
12  #define GET(p)       (*(unsigned int *)(p))
13  #define PUT(p, val)  (*(unsigned int *)(p) = (val))
14
15  /* Read the size and allocated fields from address p */
16  #define GET_SIZE(p)  (GET(p) & ~0x7)
17  #define GET_ALLOC(p) (GET(p) & 0x1)
18
19  /* Given block ptr bp, compute address of its header and footer */
20  #define HDRP(bp)       ((char *)(bp) - WSIZE)
21  #define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
22
23  /* Given block ptr bp, compute address of next and previous blocks */
24  #define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
25  #define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

code/vm/malloc/mm.c

Figure 9.43 **Basic constants and macros for manipulating the free list.**

Creating the Initial Free List

Before calling `mm_malloc` or `mm_free`, the application must initialize the heap by calling the `mm_init` function (Figure 9.44).

The `mm_init` function gets four words from the memory system and initializes them to create the empty free list (lines 4–10). It then calls the `extend_heap` function (Figure 9.45), which extends the heap by CHUNKSIZE bytes and creates the initial free block. At this point, the allocator is initialized and ready to accept allocate and free requests from the application.

The `extend_heap` function is invoked in two different circumstances: (1) when the heap is initialized and (2) when `mm_malloc` is unable to find a suitable fit. To maintain alignment, `extend_heap` rounds up the requested size to the nearest

code/vm/malloc/mm.c

```
1  int mm_init(void)
2  {
3      /* Create the initial empty heap */
4      if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
5          return -1;
6      PUT(heap_listp, 0);                          /* Alignment padding */
7      PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
8      PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
9      PUT(heap_listp + (3*WSIZE), PACK(0, 1));     /* Epilogue header */
10     heap_listp += (2*WSIZE);
11
12     /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13     if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14         return -1;
15     return 0;
16 }
```

code/vm/malloc/mm.c

Figure 9.44 `mm_init` **creates a heap with an initial free block.**

code/vm/malloc/mm.c

```
1  static void *extend_heap(size_t words)
2  {
3      char *bp;
4      size_t size;
5
6      /* Allocate an even number of words to maintain alignment */
7      size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8      if ((long)(bp = mem_sbrk(size)) == -1)
9          return NULL;
10
11     /* Initialize free block header/footer and the epilogue header */
12     PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
13     PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
14     PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
15
16     /* Coalesce if the previous block was free */
17     return coalesce(bp);
18 }
```

code/vm/malloc/mm.c

Figure 9.45 `extend_heap` **extends the heap with a new free block.**

multiple of 2 words (8 bytes) and then requests the additional heap space from the memory system (lines 7–9).

The remainder of the `extend_heap` function (lines 12–17) is somewhat subtle. The heap begins on a double-word aligned boundary, and every call to `extend_heap` returns a block whose size is an integral number of double words. Thus, every call to `mem_sbrk` returns a double-word aligned chunk of memory immediately following the header of the epilogue block. This header becomes the header of the new free block (line 12), and the last word of the chunk becomes the new epilogue block header (line 14). Finally, in the likely case that the previous heap was terminated by a free block, we call the `coalesce` function to merge the two free blocks and return the block pointer of the merged blocks (line 17).

Freeing and Coalescing Blocks

An application frees a previously allocated block by calling the `mm_free` function (Figure 9.46), which frees the requested block (`bp`) and then merges adjacent free blocks using the boundary-tags coalescing technique described in Section 9.9.11.

The code in the `coalesce` helper function is a straightforward implementation of the four cases outlined in Figure 9.40. There is one somewhat subtle aspect. The free list format we have chosen—with its prologue and epilogue blocks that are always marked as allocated—allows us to ignore the potentially troublesome edge conditions where the requested block `bp` is at the beginning or end of the heap. Without these special blocks, the code would be messier, more error prone, and slower because we would have to check for these rare edge conditions on each and every free request.

Allocating Blocks

An application requests a block of `size` bytes of memory by calling the `mm_malloc` function (Figure 9.47). After checking for spurious requests, the allocator must adjust the requested block size to allow room for the header and the footer, and to satisfy the double-word alignment requirement. Lines 12–13 enforce the minimum block size of 16 bytes: 8 bytes to satisfy the alignment requirement and 8 more bytes for the overhead of the header and footer. For requests over 8 bytes (line 15), the general rule is to add in the overhead bytes and then round up to the nearest multiple of 8.

Once the allocator has adjusted the requested size, it searches the free list for a suitable free block (line 18). If there is a fit, then the allocator places the requested block and optionally splits the excess (line 19) and then returns the address of the newly allocated block.

If the allocator cannot find a fit, it extends the heap with a new free block (lines 24–26), places the requested block in the new free block, optionally splitting the block (line 27), and then returns a pointer to the newly allocated block.

code/vm/malloc/mm.c

```
1   void mm_free(void *bp)
2   {
3       size_t size = GET_SIZE(HDRP(bp));
4
5       PUT(HDRP(bp), PACK(size, 0));
6       PUT(FTRP(bp), PACK(size, 0));
7       coalesce(bp);
8   }
9
10  static void *coalesce(void *bp)
11  {
12      size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13      size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14      size_t size = GET_SIZE(HDRP(bp));
15
16      if (prev_alloc && next_alloc) {            /* Case 1 */
17          return bp;
18      }
19
20      else if (prev_alloc && !next_alloc) {      /* Case 2 */
21          size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22          PUT(HDRP(bp), PACK(size, 0));
23          PUT(FTRP(bp), PACK(size,0));
24      }
25
26      else if (!prev_alloc && next_alloc) {      /* Case 3 */
27          size += GET_SIZE(HDRP(PREV_BLKP(bp)));
28          PUT(FTRP(bp), PACK(size, 0));
29          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
30          bp = PREV_BLKP(bp);
31      }
32
33      else {                                     /* Case 4 */
34          size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
35              GET_SIZE(FTRP(NEXT_BLKP(bp)));
36          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
37          PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
38          bp = PREV_BLKP(bp);
39      }
40      return bp;
41  }
```

code/vm/malloc/mm.c

Figure 9.46 `mm_free` **frees a block and uses boundary-tag coalescing to merge it with any adjacent free blocks in constant time.**

code/vm/malloc/mm.c

```
1  void *mm_malloc(size_t size)
2  {
3      size_t asize;      /* Adjusted block size */
4      size_t extendsize; /* Amount to extend heap if no fit */
5      char *bp;
6
7      /* Ignore spurious requests */
8      if (size == 0)
9          return NULL;
10
11     /* Adjust block size to include overhead and alignment reqs. */
12     if (size <= DSIZE)
13         asize = 2*DSIZE;
14     else
15         asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
16
17     /* Search the free list for a fit */
18     if ((bp = find_fit(asize)) != NULL) {
19         place(bp, asize);
20         return bp;
21     }
22
23     /* No fit found. Get more memory and place the block */
24     extendsize = MAX(asize,CHUNKSIZE);
25     if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26         return NULL;
27     place(bp, asize);
28     return bp;
29 }
```

code/vm/malloc/mm.c

Figure 9.47 `mm_malloc` **allocates a block from the free list.**

Practice Problem 9.8 (solution page 920)

Implement a `find_fit` function for the simple allocator described in Section 9.9.12.

```
static void *find_fit(size_t asize)
```

Your solution should perform a first-fit search of the implicit free list.

Practice Problem 9.9 (solution page 920)

Implement a `place` function for the example allocator.

```
static void place(void *bp, size_t asize)
```

Your solution should place the requested block at the beginning of the free block, splitting only if the size of the remainder would equal or exceed the minimum block size.

9.9.13 Explicit Free Lists

The implicit free list provides us with a simple way to introduce some basic allocator concepts. However, because block allocation time is linear in the total number of heap blocks, the implicit free list is not appropriate for a general-purpose allocator (although it might be fine for a special-purpose allocator where the number of heap blocks is known beforehand to be small).

A better approach is to organize the free blocks into some form of explicit data structure. Since by definition the body of a free block is not needed by the program, the pointers that implement the data structure can be stored within the bodies of the free blocks. For example, the heap can be organized as a doubly linked free list by including a `pred` (predecessor) and `succ` (successor) pointer in each free block, as shown in Figure 9.48.

Using a doubly linked list instead of an implicit free list reduces the first-fit allocation time from linear in the total number of blocks to linear in the number of *free* blocks. However, the time to free a block can be either linear or constant, depending on the policy we choose for ordering the blocks in the free list.

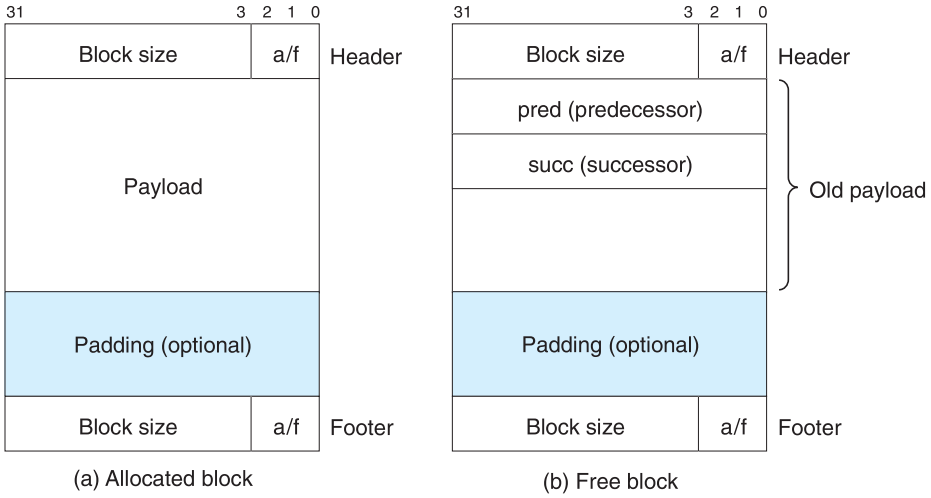


Figure 9.48 **Format of heap blocks that use doubly linked free lists.**

One approach is to maintain the list in *last-in first-out* (LIFO) order by inserting newly freed blocks at the beginning of the list. With a LIFO ordering and a first-fit placement policy, the allocator inspects the most recently used blocks first. In this case, freeing a block can be performed in constant time. If boundary tags are used, then coalescing can also be performed in constant time.

Another approach is to maintain the list in *address order*, where the address of each block in the list is less than the address of its successor. In this case, freeing a block requires a linear-time search to locate the appropriate predecessor. The trade-off is that address-ordered first fit enjoys better memory utilization than LIFO-ordered first fit, approaching the utilization of best fit.

A disadvantage of explicit lists in general is that free blocks must be large enough to contain all of the necessary pointers, as well as the header and possibly a footer. This results in a larger minimum block size and increases the potential for internal fragmentation.

9.9.14 Segregated Free Lists

As we have seen, an allocator that uses a single linked list of free blocks requires time linear in the number of free blocks to allocate a block. A popular approach for reducing the allocation time, known generally as *segregated storage*, is to maintain multiple free lists, where each list holds blocks that are roughly the same size. The general idea is to partition the set of all possible block sizes into equivalence classes called *size classes*. There are many ways to define the size classes. For example, we might partition the block sizes by powers of 2:

$$\{1\}, \{2\}, \{3, 4\}, \{5\text{–}8\}, \cdots, \{1{,}025\text{–}2{,}048\}, \{2{,}049\text{–}4{,}096\}, \{4{,}097\text{–}\infty\}$$

Or we might assign small blocks to their own size classes and partition large blocks by powers of 2:

$$\{1\}, \{2\}, \{3\}, \cdots, \{1{,}023\}, \{1{,}024\}, \{1{,}025\text{–}2{,}048\}, \{2{,}049\text{–}4{,}096\}, \{4{,}097\text{–}\infty\}$$

The allocator maintains an array of free lists, with one free list per size class, ordered by increasing size. When the allocator needs a block of size n, it searches the appropriate free list. If it cannot find a block that fits, it searches the next list, and so on.

The dynamic storage allocation literature describes dozens of variants of segregated storage that differ in how they define size classes, when they perform coalescing, when they request additional heap memory from the operating system, whether they allow splitting, and so forth. To give you a sense of what is possible, we will describe two of the basic approaches: *simple segregated storage* and *segregated fits*.

Simple Segregated Storage

With simple segregated storage, the free list for each size class contains same-size blocks, each the size of the largest element of the size class. For example, if some size class is defined as {17–32}, then the free list for that class consists entirely of blocks of size 32.

To allocate a block of some given size, we check the appropriate free list. If the list is not empty, we simply allocate the first block in its entirety. Free blocks are never split to satisfy allocation requests. If the list is empty, the allocator requests a fixed-size chunk of additional memory from the operating system (typically a multiple of the page size), divides the chunk into equal-size blocks, and links the blocks together to form the new free list. To free a block, the allocator simply inserts the block at the front of the appropriate free list.

There are a number of advantages to this simple scheme. Allocating and freeing blocks are both fast constant-time operations. Further, the combination of the same-size blocks in each chunk, no splitting, and no coalescing means that there is very little per-block memory overhead. Since each chunk has only same-size blocks, the size of an allocated block can be inferred from its address. Since there is no coalescing, allocated blocks do not need an allocated/free flag in the header. Thus, allocated blocks require no headers, and since there is no coalescing, they do not require any footers either. Since allocate and free operations insert and delete blocks at the beginning of the free list, the list need only be singly linked instead of doubly linked. The bottom line is that the only required field in any block is a one-word `succ` pointer in each free block, and thus the minimum block size is only one word.

A significant disadvantage is that simple segregated storage is susceptible to internal and external fragmentation. Internal fragmentation is possible because free blocks are never split. Worse, certain reference patterns can cause extreme external fragmentation because free blocks are never coalesced (Practice Problem 9.10).

Practice Problem 9.10 **(solution page 921)**

Describe a reference pattern that results in severe external fragmentation in an allocator based on simple segregated storage.

Segregated Fits

With this approach, the allocator maintains an array of free lists. Each free list is associated with a size class and is organized as some kind of explicit or implicit list. Each list contains potentially different-size blocks whose sizes are members of the size class. There are many variants of segregated fits allocators. Here we describe a simple version.

To allocate a block, we determine the size class of the request and do a first-fit search of the appropriate free list for a block that fits. If we find one, then we (optionally) split it and insert the fragment in the appropriate free list. If we cannot find a block that fits, then we search the free list for the next larger size class. We

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.

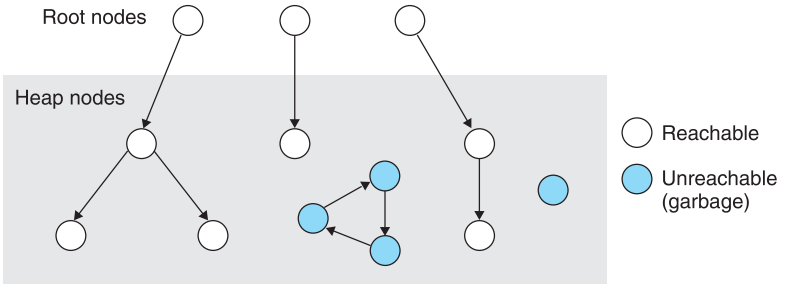


Figure 9.49
A garbage collector's view of memory as a directed graph.

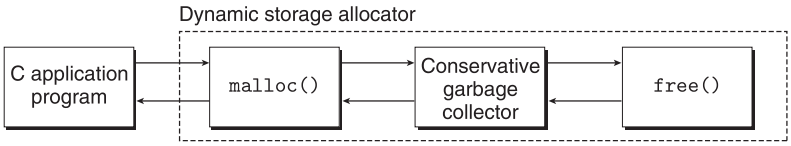


Figure 9.50
Integrating a conservative garbage collector and a C `malloc` package.

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.

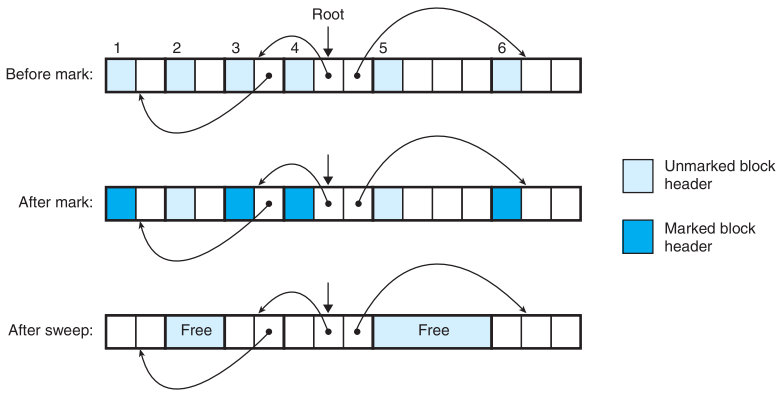


Figure 9.52
Mark&Sweep example. Note that the arrows in this example denote memory references, not free list pointers.

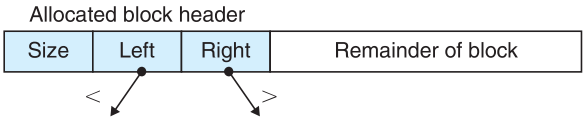


Figure 9.53
Left and right pointers in a balanced tree of allocated blocks.

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)
```

In this case, `scanf` will interpret the contents of `val` as an address and attempt to write a word to that location. In the best case, the program terminates immediately with an exception. In the worst case, the contents of `val` correspond to some valid read/write area of virtual memory, and we overwrite memory, usually with disastrous and baffling consequences much later.

9.11.2 Reading Uninitialized Memory

While bss memory locations (such as uninitialized global C variables) are always initialized to zeros by the loader, this is not true for heap memory. A common error is to assume that heap memory is initialized to zero:

```
1   /* Return y = Ax */
2   int *matvec(int **A, int *x, int n)
3   {
4       int i, j;
5
6       int *y = (int *)Malloc(n * sizeof(int));
7
8       for (i = 0; i < n; i++)
9           for (j = 0; j < n; j++)
10              y[i] += A[i][j] * x[j];
11      return y;
12  }
```

In this example, the programmer has incorrectly assumed that vector `y` has been initialized to zero. A correct implementation would explicitly zero `y[i]` or use `calloc`.

9.11.3 Allowing Stack Buffer Overflows

As we saw in Section 3.10.3, a program has a *buffer overflow bug* if it writes to a target buffer on the stack without examining the size of the input string. For example, the following function has a buffer overflow bug because the `gets` function copies an arbitrary-length string to the buffer. To fix this, we would need to use the `fgets` function, which limits the size of the input string.

```
1   void bufoverflow()
2   {
3       char buf[64];
4
5       gets(buf); /* Here is the stack buffer overflow bug */
6       return;
7   }
```

9.11.4 Assuming That Pointers and the Objects They Point to Are the Same Size

One common mistake is to assume that pointers to objects are the same size as the objects they point to:

```
1   /* Create an nxm array */
2   int **makeArray1(int n, int m)
3   {
4       int i;
5       int **A = (int **)Malloc(n * sizeof(int));
6
7       for (i = 0; i < n; i++)
8           A[i] = (int *)Malloc(m * sizeof(int));
9       return A;
10  }
```

The intent here is to create an array of n pointers, each of which points to an array of m `int`s. However, because the programmer has written `sizeof(int)` instead of `sizeof(int *)` in line 5, the code actually creates an array of `int`s.

This code will run fine on machines where `int`s and pointers to `int`s are the same size. But if we run this code on a machine like the Core i7, where a pointer is larger than an `int`, then the loop in lines 7–8 will write past the end of the `A` array. Since one of these words will likely be the boundary-tag footer of the allocated block, we may not discover the error until we free the block much later in the program, at which point the coalescing code in the allocator will fail dramatically and for no apparent reason. This is an insidious example of the kind of "action at a distance" that is so typical of memory-related programming bugs.

9.11.5 Making Off-by-One Errors

Off-by-one errors are another common source of overwriting bugs:

```
1   /* Create an nxm array */
2   int **makeArray2(int n, int m)
3   {
4       int i;
5       int **A = (int **)Malloc(n * sizeof(int *));
6
7       for (i = 0; i <= n; i++)
8           A[i] = (int *)Malloc(m * sizeof(int));
9       return A;
10  }
```

This is another version of the program in the previous section. Here we have created an n-element array of pointers in line 5 but then tried to initialize $n + 1$ of its elements in lines 7 and 8, in the process overwriting some memory that follows the `A` array.

9.11.6 Referencing a Pointer Instead of the Object It Points To

If we are not careful about the precedence and associativity of C operators, then we incorrectly manipulate a pointer instead of the object it points to. For example, consider the following function, whose purpose is to remove the first item in a binary heap of `*size` items and then reheapify the remaining `*size - 1` items:

```
1   int *binheapDelete(int **binheap, int *size)
2   {
3       int *packet = binheap[0];
4
5       binheap[0] = binheap[*size - 1];
6       *size--; /* This should be (*size)-- */
7       heapify(binheap, *size, 0);
8       return(packet);
9   }
```

In line 6, the intent is to decrement the integer value pointed to by the `size` pointer. However, because the unary `--` and `*` operators have the same precedence and associate from right to left, the code in line 6 actually decrements the pointer itself instead of the integer value that it points to. If we are lucky, the program will crash immediately. But more likely we will be left scratching our heads when the program produces an incorrect answer much later in its execution. The moral here is to use parentheses whenever in doubt about precedence and associativity. For example, in line 6, we should have clearly stated our intent by using the expression `(*size)--`.

9.11.7 Misunderstanding Pointer Arithmetic

Another common mistake is to forget that arithmetic operations on pointers are performed in units that are the size of the objects they point to, which are not necessarily bytes. For example, the intent of the following function is to scan an array of `int`s and return a pointer to the first occurrence of `val`:

```
1   int *search(int *p, int val)
2   {
3       while (*p && *p != val)
4           p += sizeof(int); /* Should be p++ */
5       return p;
6   }
```

However, because line 4 increments the pointer by 4 (the number of bytes in an integer) each time through the loop, the function incorrectly scans every fourth integer in the array.

9.11.8 Referencing Nonexistent Variables

Naive C programmers who do not understand the stack discipline will sometimes reference local variables that are no longer valid, as in the following example:

```
1   int *stackref ()
2   {
3       int val;
4
5       return &val;
6   }
```

This function returns a pointer (say, `p`) to a local variable on the stack and then pops its stack frame. Although `p` still points to a valid memory address, it no longer points to a valid variable. When other functions are called later in the program, the memory will be reused for their stack frames. Later, if the program assigns some value to `*p`, then it might actually be modifying an entry in another function's stack frame, with potentially disastrous and baffling consequences.

9.11.9 Referencing Data in Free Heap Blocks

A similar error is to reference data in heap blocks that have already been freed. Consider the following example, which allocates an integer array `x` in line 6, prematurely frees block `x` in line 10, and then later references it in line 14:

```
1   int *heapref(int n, int m)
2   {
3       int i;
4       int *x, *y;
5
6       x = (int *)Malloc(n * sizeof(int));
7
8       . . . // Other calls to malloc and free go here
9
10      free(x);
11
12      y = (int *)Malloc(m * sizeof(int));
13      for (i = 0; i < m; i++)
14          y[i] = x[i]++; /* Oops! x[i] is a word in a free block */
15
16      return y;
17  }
```

Depending on the pattern of `malloc` and `free` calls that occur between lines 6 and 10, when the program references `x[i]` in line 14, the array `x` might be part of some other allocated heap block and may have been overwritten. As with many

memory-related bugs, the error will only become evident later in the program when we notice that the values in y are corrupted.

9.11.10 Introducing Memory Leaks

Memory leaks are slow, silent killers that occur when programmers inadvertently create garbage in the heap by forgetting to free allocated blocks. For example, the following function allocates a heap block x and then returns without freeing it:

```
1   void leak(int n)
2   {
3       int *x = (int *)Malloc(n * sizeof(int));
4
5       return;  /* x is garbage at this point */
6   }
```

If `leak` is called frequently, then the heap will gradually fill up with garbage, in the worst case consuming the entire virtual address space. Memory leaks are particularly serious for programs such as daemons and servers, which by definition never terminate.

9.12 Summary

Virtual memory is an abstraction of main memory. Processors that support virtual memory reference main memory using a form of indirection known as virtual addressing. The processor generates a virtual address, which is translated into a physical address before being sent to the main memory. The translation of addresses from a virtual address space to a physical address space requires close cooperation between hardware and software. Dedicated hardware translates virtual addresses using page tables whose contents are supplied by the operating system.

Virtual memory provides three important capabilities. First, it automatically caches recently used contents of the virtual address space stored on disk in main memory. The block in a virtual memory cache is known as a page. A reference to a page on disk triggers a page fault that transfers control to a fault handler in the operating system. The fault handler copies the page from disk to the main memory cache, writing back the evicted page if necessary. Second, virtual memory simplifies memory management, which in turn simplifies linking, sharing data between processes, the allocation of memory for processes, and program loading. Finally, virtual memory simplifies memory protection by incorporating protection bits into every page table entry.

The process of address translation must be integrated with the operation of any hardware caches in the system. Most page table entries are located in the L1 cache, but the cost of accessing page table entries from L1 is usually eliminated by an on-chip cache of page table entries called a TLB.

Modern systems initialize chunks of virtual memory by associating them with chunks of files on disk, a process known as memory mapping. Memory mapping provides an efficient mechanism for sharing data, creating new processes, and loading programs. Applications can manually create and delete areas of the virtual address space using the `mmap` function. However, most programs rely on a dynamic memory allocator such as `malloc`, which manages memory in an area of the virtual address space called the heap. Dynamic memory allocators are application-level programs with a system-level feel, directly manipulating memory without much help from the type system. Allocators come in two flavors. Explicit allocators require applications to explicitly free their memory blocks. Implicit allocators (garbage collectors) free any unused and unreachable blocks automatically.

Managing and using memory is a difficult and error-prone task for C programmers. Examples of common errors include dereferencing bad pointers, reading uninitialized memory, allowing stack buffer overflows, assuming that pointers and the objects they point to are the same size, referencing a pointer instead of the object it points to, misunderstanding pointer arithmetic, referencing nonexistent variables, and introducing memory leaks.

Bibliographic Notes

Kilburn and his colleagues published the first description of virtual memory [63]. Architecture texts contain additional details about the hardware's role in virtual memory [46]. Operating systems texts contain additional information about the operating system's role [102, 106, 113]. Bovet and Cesati [11] give a detailed description of the Linux virtual memory system. Intel Corporation provides detailed documentation on 32-bit and 64-bit address translation on IA processors [52].

Knuth wrote the classic work on storage allocation in 1968 [64]. Since that time, there has been a tremendous amount of work in the area. Wilson, Johnstone, Neely, and Boles have written a beautiful survey and performance evaluation of explicit allocators [118]. The general comments in this book about the throughput and utilization of different allocator strategies are paraphrased from their survey. Jones and Lins provide a comprehensive survey of garbage collection [56]. Kernighan and Ritchie [61] show the complete code for a simple allocator based on an explicit free list with a block size and successor pointer in each free block. The code is interesting in that it uses unions to eliminate a lot of the complicated pointer arithmetic, but at the expense of a linear-time (rather than constant-time) free operation. Doug Lea developed a widely used open-source malloc package called `dlmalloc` [67].

Homework Problems

9.11 ◆

In the following series of problems, you are to show how the example memory system in Section 9.6.4 translates a virtual address into a physical address and accesses the cache. For the given virtual address, indicate the TLB entry accessed,