

determine the PPN. As with a single-level hierarchy, the PPO is identical to the VPO.

Accessing k PTEs may seem expensive and impractical at first glance. However, the TLB comes to the rescue here by caching PTEs from the page tables at the different levels. In practice, address translation with multi-level page tables is not significantly slower than with single-level page tables.

9.6.4 Putting It Together: End-to-End Address Translation

In this section, we put it all together with a concrete example of end-to-end address translation on a small system with a TLB and L1 d-cache. To keep things manageable, we make the following assumptions:

- The memory is byte addressable.
- Memory accesses are to *1-byte words* (not 4-byte words).
- Virtual addresses are 14 bits wide ($n = 14$).
- Physical addresses are 12 bits wide ($m = 12$).
- The page size is 64 bytes ($P = 64$).
- The TLB is 4-way set associative with 16 total entries.
- The L1 d-cache is physically addressed and direct mapped, with a 4-byte line size and 16 total sets.

Figure 9.19 shows the formats of the virtual and physical addresses. Since each page is $2^6 = 64$ bytes, the low-order 6 bits of the virtual and physical addresses serve as the VPO and PPO, respectively. The high-order 8 bits of the virtual address serve as the VPN. The high-order 6 bits of the physical address serve as the PPN.

Figure 9.20 shows a snapshot of our little memory system, including the TLB (Figure 9.20(a)), a portion of the page table (Figure 9.20(b)), and the L1 cache (Figure 9.20(c)). Above the figures of the TLB and cache, we have also shown how the bits of the virtual and physical addresses are partitioned by the hardware as it accesses these devices.

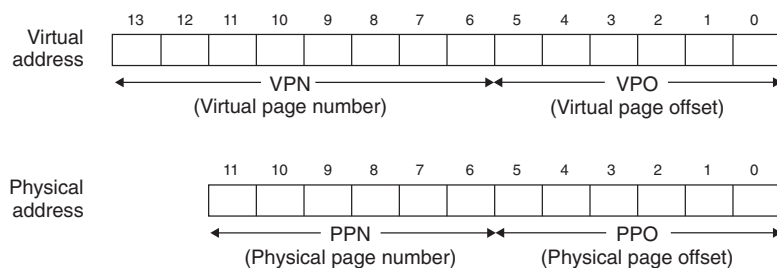
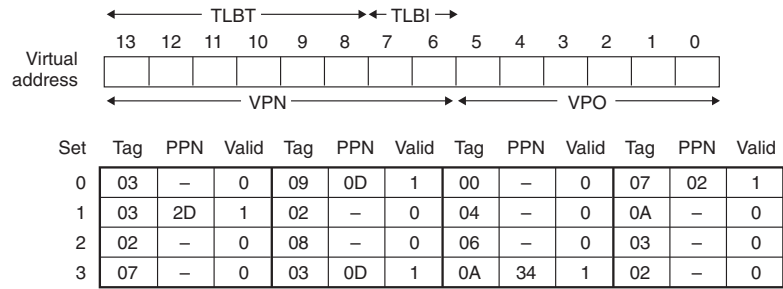


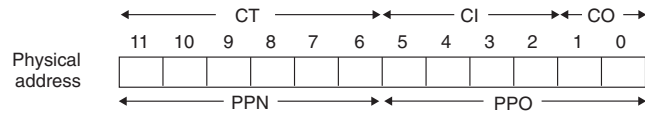
Figure 9.19 Addressing for small memory system. Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).



(a) TLB: 4 sets, 16 entries, 4-way set associative

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	—	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	—	0
04	—	0	0C	—	0
05	16	1	0D	2D	1
06	—	0	0E	11	1
07	—	0	0F	0D	1

(b) Page table: Only the first 16 PTEs are shown



Idx	Tag	Valid	Blk 0	Blk 1	Blk 2	Blk 3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	—	—	—	—
A	2D	1	93	15	DA	3B
B	0B	0	—	—	—	—
C	12	0	—	—	—	—
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	—	—	—	—

(c) Cache: 16 sets, 4-byte blocks, direct mapped

Figure 9.20 TLB, page table, and cache for small memory system. All values in the TLB, page table, and cache are in hexadecimal notation.

TLB. The TLB is virtually addressed using the bits of the VPN. Since the TLB has four sets, the 2 low-order bits of the VPN serve as the set index (TLBI). The remaining 6 high-order bits serve as the tag (TLBT) that distinguishes the different VPNs that might map to the same TLB set.

Page table. The page table is a single-level design with a total of $2^8 = 256$ page table entries (PTEs). However, we are only interested in the first 16 of these. For convenience, we have labeled each PTE with the VPN that indexes it; but keep in mind that these VPNs are not part of the page table and not stored in memory. Also, notice that the PPN of each invalid PTE is denoted with a dash to reinforce the idea that whatever bit values might happen to be stored there are not meaningful.

Cache. The direct-mapped cache is addressed by the fields in the physical address. Since each block is 4 bytes, the low-order 2 bits of the physical address serve as the block offset (CO). Since there are 16 sets, the next 4 bits serve as the set index (CI). The remaining 6 bits serve as the tag (CT).

Given this initial setup, let's see what happens when the CPU executes a load instruction that reads the byte at address 0x03d4. (Recall that our hypothetical CPU reads 1-byte words rather than 4-byte words.) To begin this kind of manual simulation, we find it helpful to write down the bits in the virtual address, identify the various fields we will need, and determine their hex values. The hardware performs a similar task when it decodes the address.

Bit position	TLBT						TLBI							
	0x03						0x03							
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								VPO						
0x0f								0x14						

To begin, the MMU extracts the VPN (0x0F) from the virtual address and checks with the TLB to see if it has cached a copy of PTE 0x0F from some previous memory reference. The TLB extracts the TLB index (0x03) and the TLB tag (0x3) from the VPN, hits on a valid match in the second entry of set 0x3, and returns the cached PPN (0x0D) to the MMU.

If the TLB had missed, then the MMU would need to fetch the PTE from main memory. However, in this case, we got lucky and had a TLB hit. The MMU now has everything it needs to form the physical address. It does this by concatenating the PPN (0x0D) from the PTE with the VPO (0x14) from the virtual address, which forms the physical address (0x354).

Next, the MMU sends the physical address to the cache, which extracts the cache offset CO (0x0), the cache set index CI (0x5), and the cache tag CT (0x0D) from the physical address.

	CT						CI				CO	
	0x0d						0x05				0x0	
Bit position	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0x354	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0d						0x14					

Since the tag in set 0x5 matches CT, the cache detects a hit, reads out the data byte (0x36) at offset C0, and returns it to the MMU, which then passes it back to the CPU.

Other paths through the translation process are also possible. For example, if the TLB misses, then the MMU must fetch the PPN from a PTE in the page table. If the resulting PTE is invalid, then there is a page fault and the kernel must page in the appropriate page and rerun the load instruction. Another possibility is that the PTE is valid, but the necessary memory block misses in the cache.

Practice Problem 9.4 (solution page 917)

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, physical address, and cache byte value returned. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs. If there is a cache miss, enter “—” for “Cache byte returned.” If there is a page fault, enter “—” for “PPN” and leave parts C and D blank.

Virtual address: 0x03d7

A. Virtual address format

[illegible]

B. Address translation

Parameter	Value
VPN	
TLB index	
TLB tag	
TLB hit? (Y/N)	
Page fault? (Y/N)	
PPN	

C. Physical address format

11	10	9	8	7	6	5	4	3	2	1	0

D. Physical memory reference

Parameter	Value
Byte offset	_____
Cache index	_____
Cache tag	_____
Cache hit? (Y/N)	_____
Cache byte returned	_____

9.7 Case Study: The Intel Core i7/Linux Memory System

We conclude our discussion of virtual memory mechanisms with a case study of a real system: an Intel Core i7 running Linux. Although the underlying Haswell microarchitecture allows for full 64-bit virtual and physical address spaces, the current Core i7 implementations (and those for the foreseeable future) support a 48-bit (256 TB) virtual address space and a 52-bit (4 PB) physical address space, along with a compatibility mode that supports 32-bit (4 GB) virtual and physical address spaces.

Figure 9.21 gives the highlights of the Core i7 memory system. The *processor package* (chip) includes four cores, a large L3 cache shared by all of the cores, and

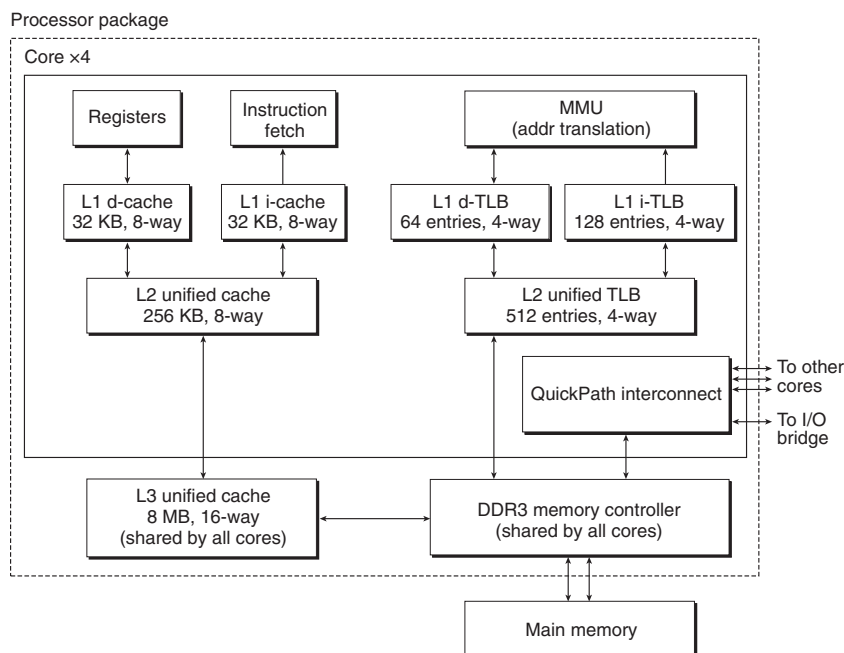


Figure 9.21 The Core i7 memory system.

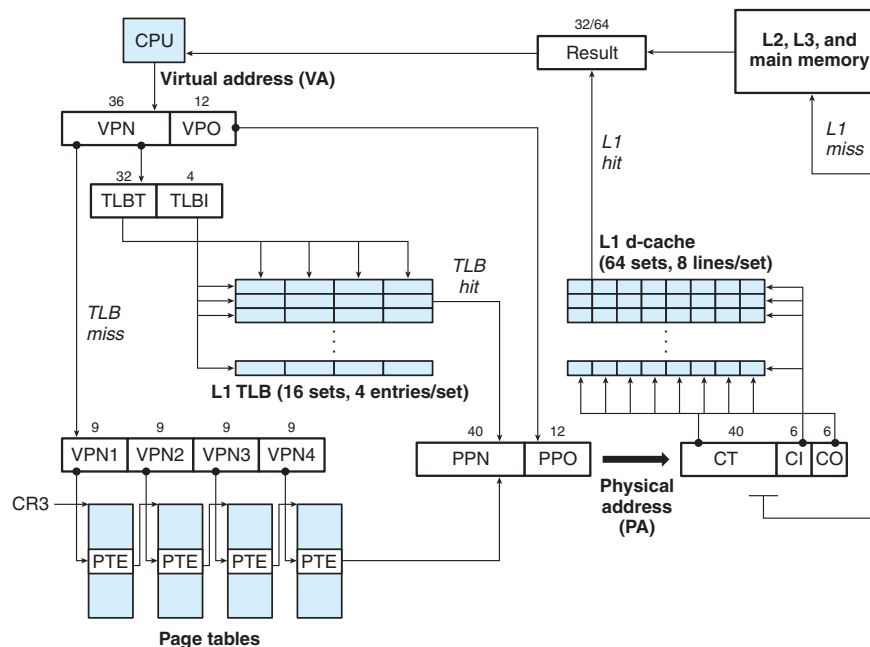


Figure 9.22 Summary of Core i7 address translation. For simplicity, the i-caches, i-TLB, and L2 unified TLB are not shown.

a DDR3 memory controller. Each core contains a hierarchy of TLBs, a hierarchy of data and instruction caches, and a set of fast point-to-point links, based on the QuickPath technology, for communicating directly with the other cores and the external I/O bridge. The TLBs are virtually addressed, and 4-way set associative. The L1, L2, and L3 caches are physically addressed, with a block size of 64 bytes. L1 and L2 are 8-way set associative, and L3 is 16-way set associative. The page size can be configured at start-up time as either 4 KB or 4 MB. Linux uses 4 KB pages.

9.7.1 Core i7 Address Translation

Figure 9.22 summarizes the entire Core i7 address translation process, from the time the CPU generates a virtual address until a data word arrives from memory. The Core i7 uses a four-level page table hierarchy. Each process has its own private page table hierarchy. When a Linux process is running, the page tables associated with allocated pages are all memory-resident, although the Core i7 architecture allows these page tables to be swapped in and out. The CR3 control register contains the physical address of the beginning of the level 1 (L1) page table. The value of CR3 is part of each process context, and is restored during each context switch.

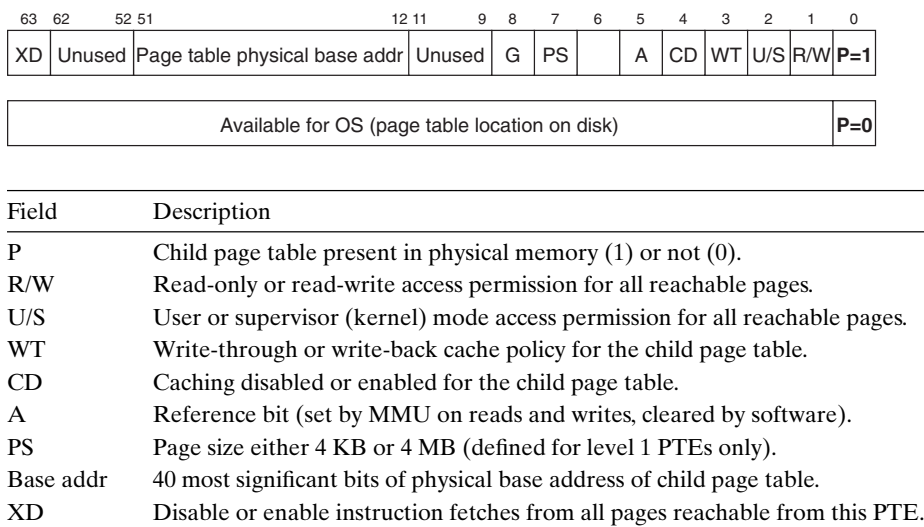


Figure 9.23 Format of level 1, level 2, and level 3 page table entries. Each entry references a 4 KB child page table.

Figure 9.23 shows the format of an entry in a level 1, level 2, or level 3 page table. When $P = 1$ (which is always the case with Linux), the address field contains a 40-bit physical page number (PPN) that points to the beginning of the appropriate page table. Notice that this imposes a 4 KB alignment requirement on page tables.

Figure 9.24 shows the format of an entry in a level 4 page table. When $P = 1$, the address field contains a 40-bit PPN that points to the base of some page in physical memory. Again, this imposes a 4 KB alignment requirement on physical pages.

The PTE has three permission bits that control access to the page. The R/W bit determines whether the contents of a page are read/write or read-only. The U/S bit, which determines whether the page can be accessed in user mode, protects code and data in the operating system kernel from user programs. The XD (execute disable) bit, which was introduced in 64-bit systems, can be used to disable instruction fetches from individual memory pages. This is an important new feature that allows the operating system kernel to reduce the risk of buffer overflow attacks by restricting execution to the read-only code segment.

As the MMU translates each virtual address, it also updates two other bits that can be used by the kernel's page fault handler. The MMU sets the A bit, which is known as a *reference bit*, each time a page is accessed. The kernel can use the reference bit to implement its page replacement algorithm. The MMU sets the D bit, or *dirty bit*, each time the page is written to. A page that has been modified is sometimes called a *dirty page*. The dirty bit tells the kernel whether or not it must

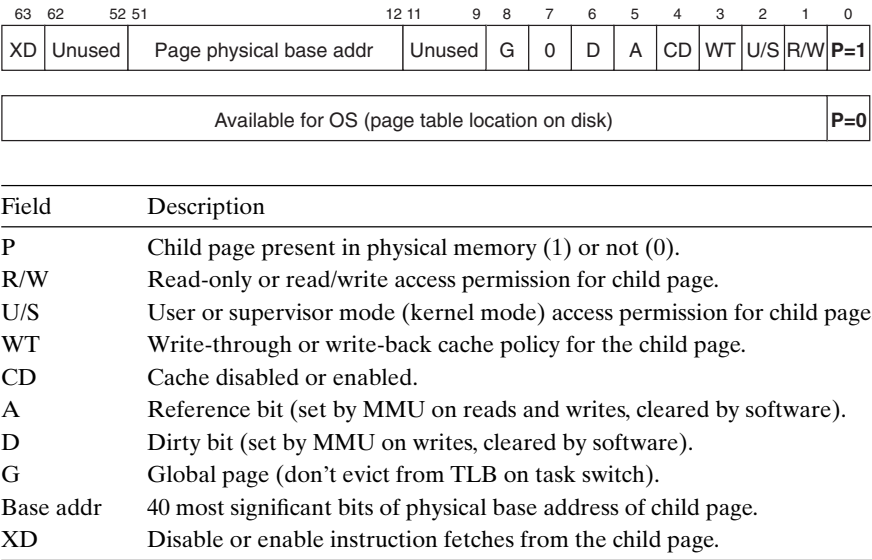


Figure 9.24 Format of level 4 page table entries. Each entry references a 4 KB child page.

write back a victim page before it copies in a replacement page. The kernel can call a special kernel-mode instruction to clear the reference or dirty bits.

Figure 9.25 shows how the Core i7 MMU uses the four levels of page tables to translate a virtual address to a physical address. The 36-bit VPN is partitioned into four 9-bit chunks, each of which is used as an offset into a page table. The CR3 register contains the physical address of the L1 page table. VPN 1 provides an offset to an L1 PTE, which contains the base address of the L2 page table. VPN 2 provides an offset to an L2 PTE, and so on.

9.7.2 Linux Virtual Memory System

A virtual memory system requires close cooperation between the hardware and the kernel. Details vary from version to version, and a complete description is beyond our scope. Nonetheless, our aim in this section is to describe enough of the Linux virtual memory system to give you a sense of how a real operating system organizes virtual memory and how it handles page faults.

Linux maintains a separate virtual address space for each process of the form shown in Figure 9.26. We have seen this picture a number of times already, with its familiar code, data, heap, shared library, and stack segments. Now that we understand address translation, we can fill in some more details about the kernel virtual memory that lies above the user stack.

The kernel virtual memory contains the code and data structures in the kernel. Some regions of the kernel virtual memory are mapped to physical pages that

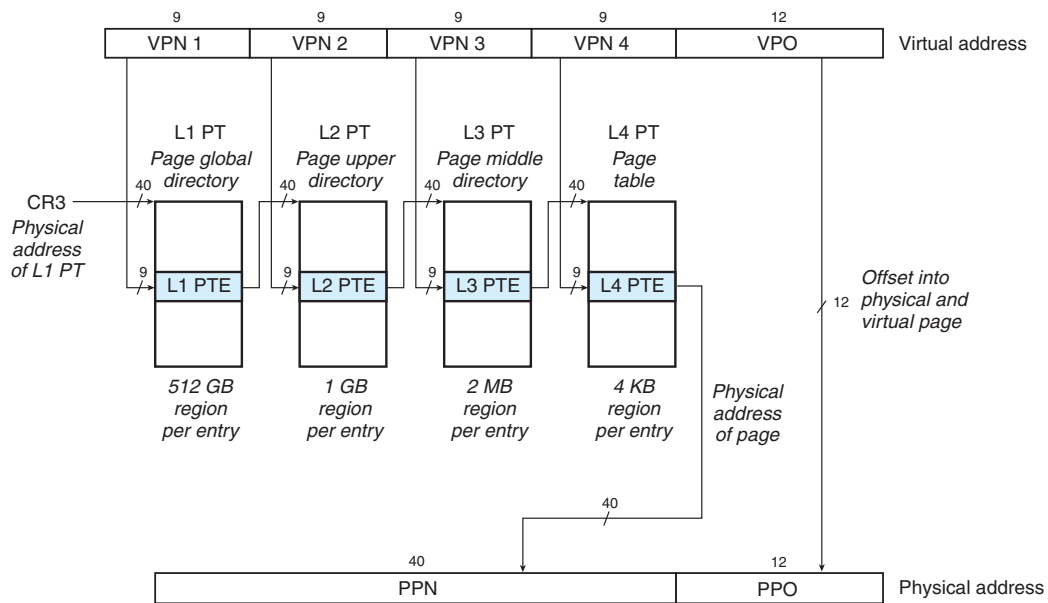
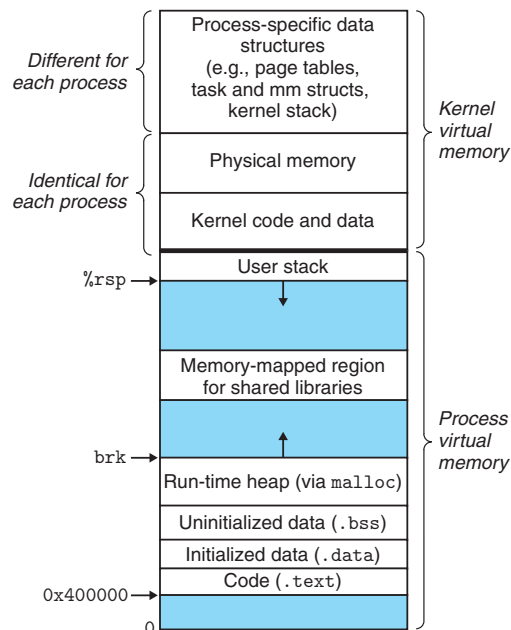


Figure 9.25 Core i7 page table translation. PT: page table; PTE: page table entry; VPN: virtual page number; VPO: virtual page offset; PPN: physical page number; PPO: physical page offset. The Linux names for the four levels of page tables are also shown.

Figure 9.26

The virtual memory of a Linux process.



Aside Optimizing address translation

In our discussion of address translation, we have described a sequential two-step process where the MMU (1) translates the virtual address to a physical address and then (2) passes the physical address to the L1 cache. However, real hardware implementations use a neat trick that allows these steps to be partially overlapped, thus speeding up accesses to the L1 cache. For example, a virtual address on a Core i7 with 4 KB pages has 12 bits of VPO, and these bits are identical to the 12 bits of PPO in the corresponding physical address. Since the 8-way set associative physically addressed L1 caches have 64 sets and 64-byte cache blocks, each physical address has 6 ($\log_2 64$) cache offset bits and 6 ($\log_2 64$) index bits. These 12 bits fit exactly in the 12-bit VPO of a virtual address, which is no accident! When the CPU needs a virtual address translated, it sends the VPN to the MMU and the VPO to the L1 cache. While the MMU is requesting a page table entry from the TLB, the L1 cache is busy using the VPO bits to find the appropriate set and read out the eight tags and corresponding data words in that set. When the MMU gets the PPN back from the TLB, the cache is ready to try to match the PPN to one of these eight tags.

are shared by all processes. For example, each process shares the kernel's code and global data structures. Interestingly, Linux also maps a set of contiguous virtual pages (equal in size to the total amount of DRAM in the system) to the corresponding set of contiguous physical pages. This provides the kernel with a convenient way to access any specific location in physical memory—for example, when it needs to access page tables or to perform memory-mapped I/O operations on devices that are mapped to particular physical memory locations.

Other regions of kernel virtual memory contain data that differ for each process. Examples include page tables, the stack that the kernel uses when it is executing code in the context of the process, and various data structures that keep track of the current organization of the virtual address space.

Linux Virtual Memory Areas

Linux organizes the virtual memory as a collection of *areas* (also called *segments*). An area is a contiguous chunk of existing (allocated) virtual memory whose pages are related in some way. For example, the code segment, data segment, heap, shared library segment, and user stack are all distinct areas. Each existing virtual page is contained in some area, and any virtual page that is not part of some area does not exist and cannot be referenced by the process. The notion of an area is important because it allows the virtual address space to have gaps. The kernel does not keep track of virtual pages that do not exist, and such pages do not consume any additional resources in memory, on disk, or in the kernel itself.

Figure 9.27 highlights the kernel data structures that keep track of the virtual memory areas in a process. The kernel maintains a distinct task structure (`task_struct` in the source code) for each process in the system. The elements of the task structure either contain or point to all of the information that the kernel needs to

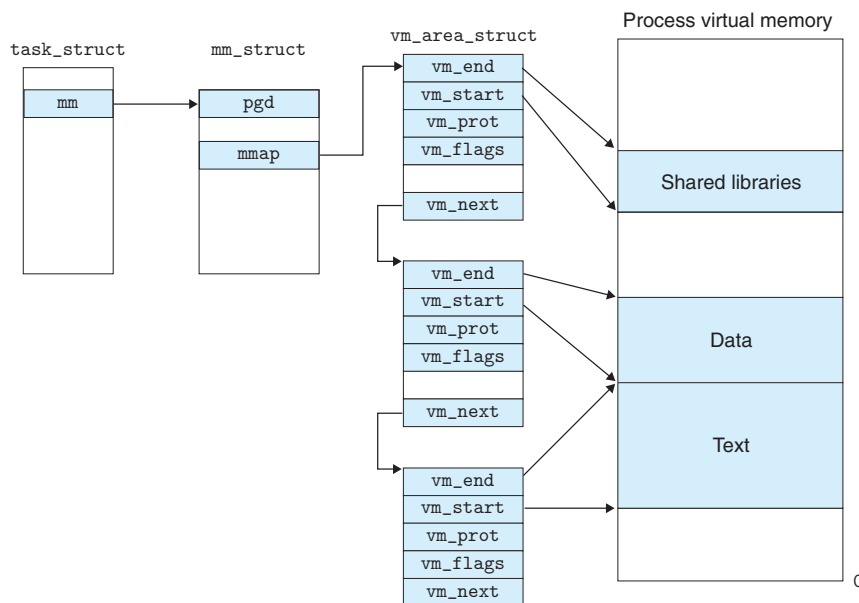


Figure 9.27 How Linux organizes virtual memory.

run the process (e.g., the PID, pointer to the user stack, name of the executable object file, and program counter).

One of the entries in the task structure points to an `mm_struct` that characterizes the current state of the virtual memory. The two fields of interest to us are `pgd`, which points to the base of the level 1 table (the page global directory), and `mmap`, which points to a list of `vm_area_structs` (area structs), each of which characterizes an area of the current virtual address space. When the kernel runs this process, it stores `pgd` in the CR3 control register.

For our purposes, the area struct for a particular area contains the following fields:

`vm_start`. Points to the beginning of the area.

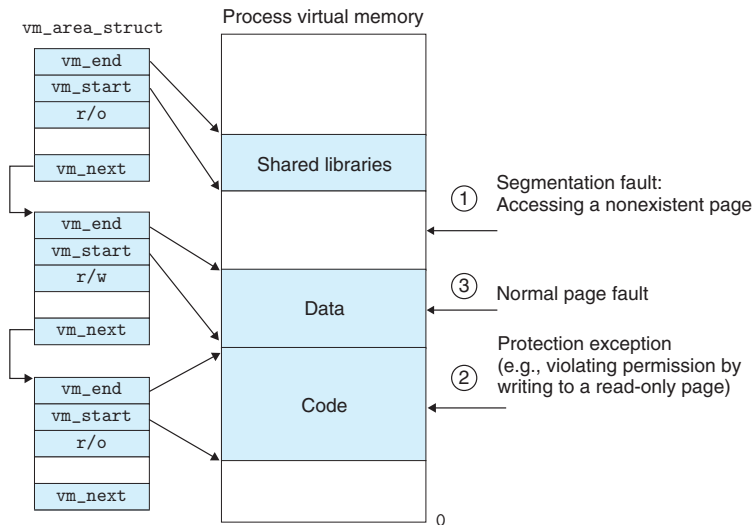
`vm_end`. Points to the end of the area.

`vm_prot`. Describes the read/write permissions for all of the pages contained in the area.

`vm_flags`. Describes (among other things) whether the pages in the area are shared with other processes or private to this process.

`vm_next`. Points to the next area struct in the list.

Figure 9.28
Linux page fault handling.



Linux Page Fault Exception Handling

Suppose the MMU triggers a page fault while trying to translate some virtual address *A*. The exception results in a transfer of control to the kernel's page fault handler, which then performs the following steps:

1. Is virtual address *A* legal? In other words, does *A* lie within an area defined by some area struct? To answer this question, the fault handler searches the list of area structs, comparing *A* with the `vm_start` and `vm_end` in each area struct. If the instruction is not legal, then the fault handler triggers a segmentation fault, which terminates the process. This situation is labeled "1" in Figure 9.28.
Because a process can create an arbitrary number of new virtual memory areas (using the `mmap` function described in the next section), a sequential search of the list of area structs might be very costly. So in practice, Linux superimposes a tree on the list, using some fields that we have not shown, and performs the search on this tree.
2. Is the attempted memory access legal? In other words, does the process have permission to read, write, or execute the pages in this area? For example, was the page fault the result of a store instruction trying to write to a read-only page in the code segment? Is the page fault the result of a process running in user mode that is attempting to read a word from kernel virtual memory? If the attempted access is not legal, then the fault handler triggers a protection exception, which terminates the process. This situation is labeled "2" in Figure 9.28.
3. At this point, the kernel knows that the page fault resulted from a legal operation on a legal virtual address. It handles the fault by selecting a victim page, swapping out the victim page if it is dirty, swapping in the new page,

and updating the page table. When the page fault handler returns, the CPU restarts the faulting instruction, which sends *A* to the MMU again. This time, the MMU translates *A* normally, without generating a page fault.

9.8 Memory Mapping

Linux initializes the contents of a virtual memory area by associating it with an *object* on disk, a process known as *memory mapping*. Areas can be mapped to one of two types of objects:

1. *Regular file in the Linux file system*: An area can be mapped to a contiguous section of a regular disk file, such as an executable object file. The file section is divided into page-size pieces, with each piece containing the initial contents of a virtual page. Because of demand paging, none of these virtual pages is actually swapped into physical memory until the CPU first *touches* the page (i.e., issues a virtual address that falls within that page's region of the address space). If the area is larger than the file section, then the area is padded with zeros.
2. *Anonymous file*: An area can also be mapped to an anonymous file, created by the kernel, that contains all binary zeros. The first time the CPU touches a virtual page in such an area, the kernel finds an appropriate victim page in physical memory, swaps out the victim page if it is dirty, overwrites the victim page with binary zeros, and updates the page table to mark the page as resident. Notice that no data are actually transferred between disk and memory. For this reason, pages in areas that are mapped to anonymous files are sometimes called *demand-zero pages*.

In either case, once a virtual page is initialized, it is swapped back and forth between a special *swap file* maintained by the kernel. The swap file is also known as the *swap space* or the *swap area*. An important point to realize is that at any point in time, the swap space bounds the total amount of virtual pages that can be allocated by the currently running processes.

9.8.1 Shared Objects Revisited

The idea of memory mapping resulted from a clever insight that if the virtual memory system could be integrated into the conventional file system, then it could provide a simple and efficient way to load programs and data into memory.

As we have seen, the process abstraction promises to provide each process with its own private virtual address space that is protected from errant writes or reads by other processes. However, many processes have identical read-only code areas. For example, each process that runs the Linux shell program `bash` has the same code area. Further, many programs need to access identical copies of read-only run-time library code. For example, every C program requires functions from the standard C library such as `printf`. It would be extremely wasteful for each process to keep duplicate copies of these commonly used codes in physical

memory. Fortunately, memory mapping provides us with a clean mechanism for controlling how objects are shared by multiple processes.

An object can be mapped into an area of virtual memory as either a *shared object* or a *private object*. If a process maps a shared object into an area of its virtual address space, then any writes that the process makes to that area are visible to any other processes that have also mapped the shared object into their virtual memory. Further, the changes are also reflected in the original object on disk.

Changes made to an area mapped to a private object, on the other hand, are not visible to other processes, and any writes that the process makes to the area are *not* reflected back to the object on disk. A virtual memory area into which a shared object is mapped is often called a *shared area*. Similarly for a *private area*.

Suppose that process 1 maps a shared object into an area of its virtual memory, as shown in Figure 9.29(a). Now suppose that process 2 maps the same shared ob-

Figure 9.29

A shared object. (a) After process 1 maps the shared object. (b) After process 2 maps the same shared object. (Note that the physical pages are not necessarily contiguous.)

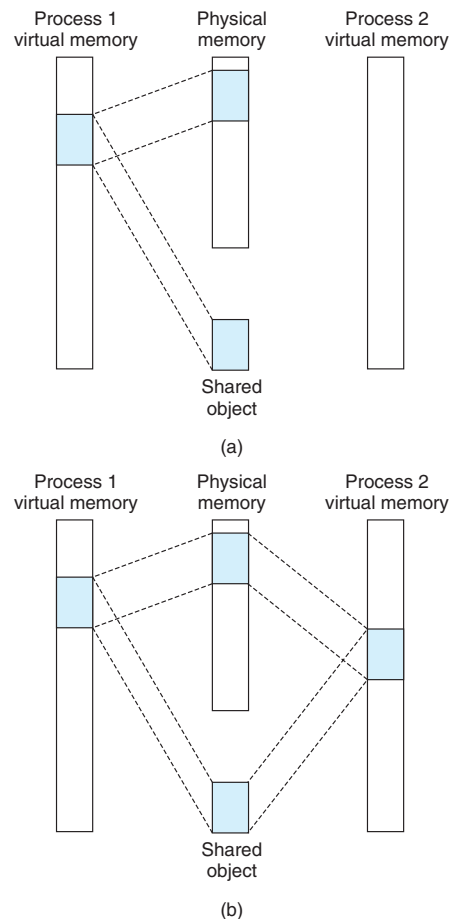
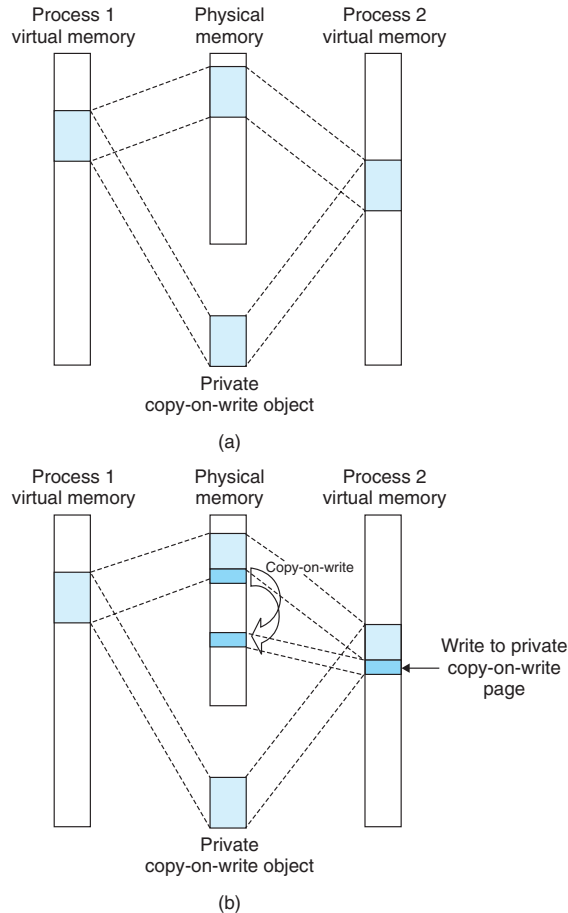


Figure 9.30
A private copy-on-write object. (a) After both processes have mapped the private copy-on-write object. (b) After process 2 writes to a page in the private area.



ject into its address space (not necessarily at the same virtual address as process 1), as shown in Figure 9.29(b).

Since each object has a unique filename, the kernel can quickly determine that process 1 has already mapped this object and can point the page table entries in process 2 to the appropriate physical pages. The key point is that only a single copy of the shared object needs to be stored in physical memory, even though the object is mapped into multiple shared areas. For convenience, we have shown the physical pages as being contiguous, but of course this is not true in general.

Private objects are mapped into virtual memory using a clever technique known as *copy-on-write*. A private object begins life in exactly the same way as a shared object, with only one copy of the private object stored in physical memory. For example, Figure 9.30(a) shows a case where two processes have mapped a private object into different areas of their virtual memories but share the same

physical copy of the object. For each process that maps the private object, the page table entries for the corresponding private area are flagged as read-only, and the area struct is flagged as *private copy-on-write*. So long as neither process attempts to write to its respective private area, they continue to share a single copy of the object in physical memory. However, as soon as a process attempts to write to some page in the private area, the write triggers a protection fault.

When the fault handler notices that the protection exception was caused by the process trying to write to a page in a private copy-on-write area, it creates a new copy of the page in physical memory, updates the page table entry to point to the new copy, and then restores write permissions to the page, as shown in Figure 9.30(b). When the fault handler returns, the CPU re-executes the write, which now proceeds normally on the newly created page.

By deferring the copying of the pages in private objects until the last possible moment, copy-on-write makes the most efficient use of scarce physical memory.

9.8.2 The fork Function Revisited

Now that we understand virtual memory and memory mapping, we can get a clear idea of how the `fork` function creates a new process with its own independent virtual address space.

When the `fork` function is called by the *current process*, the kernel creates various data structures for the *new process* and assigns it a unique PID. To create the virtual memory for the new process, it creates exact copies of the current process's `mm_struct`, area structs, and page tables. It flags each page in both processes as read-only, and flags each area struct in both processes as private copy-on-write.

When the `fork` returns in the new process, the new process now has an exact copy of the virtual memory as it existed when the `fork` was called. When either of the processes performs any subsequent writes, the copy-on-write mechanism creates new pages, thus preserving the abstraction of a private address space for each process.

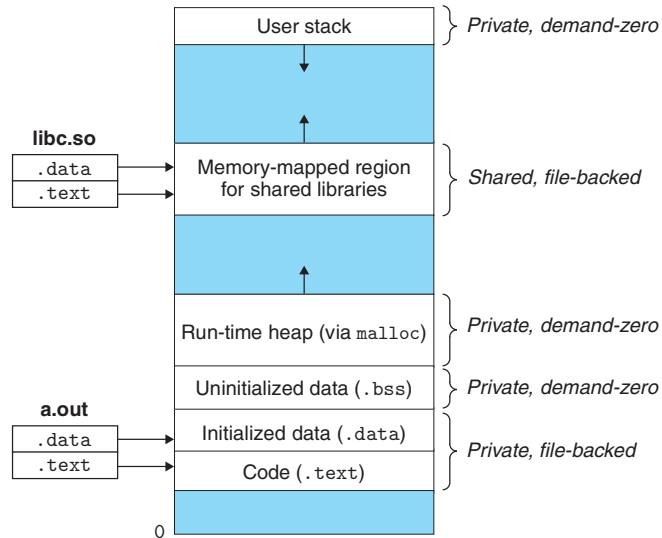
9.8.3 The execve Function Revisited

Virtual memory and memory mapping also play key roles in the process of loading programs into memory. Now that we understand these concepts, we can understand how the `execve` function really loads and executes programs. Suppose that the program running in the current process makes the following call:

```
execve("a.out", NULL, NULL);
```

As you learned in Chapter 8, the `execve` function loads and runs the program contained in the executable object file `a.out` within the current process, effectively replacing the current program with the `a.out` program. Loading and running `a.out` requires the following steps:

Figure 9.31
How the loader maps the
areas of the user address
space.



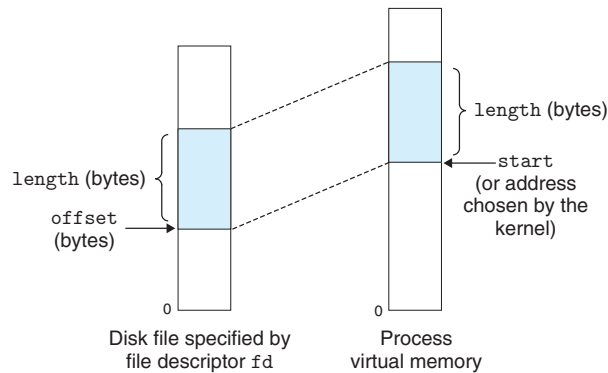
1. *Delete existing user areas.* Delete the existing area structs in the user portion of the current process's virtual address.
2. *Map private areas.* Create new area structs for the code, data, bss, and stack areas of the new program. All of these new areas are private copy-on-write. The code and data areas are mapped to the `.text` and `.data` sections of the `a.out` file. The bss area is demand-zero, mapped to an anonymous file whose size is contained in `a.out`. The stack and heap area are also demand-zero, initially of zero length. Figure 9.31 summarizes the different mappings of the private areas.
3. *Map shared areas.* If the `a.out` program was linked with shared objects, such as the standard C library `libc.so`, then these objects are dynamically linked into the program, and then mapped into the shared region of the user's virtual address space.
4. *Set the program counter (PC).* The last thing that `execve` does is to set the program counter in the current process's context to point to the entry point in the code area.

The next time this process is scheduled, it will begin execution from the entry point. Linux will swap in code and data pages as needed.

9.8.4 User-Level Memory Mapping with the `mmap` Function

Linux processes can use the `mmap` function to create new areas of virtual memory and to map objects into these areas.

Figure 9.32
Visual interpretation of
mmap arguments.



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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
           Returns: pointer to mapped area if OK, MAP_FAILED (-1) on error
```

The `mmap` function asks the kernel to create a new virtual memory area, preferably one that starts at address `start`, and to map a contiguous chunk of the object specified by file descriptor `fd` to the new area. The contiguous object chunk has a size of `length` bytes and starts at an offset of `offset` bytes from the beginning of the file. The `start` address is merely a hint, and is usually specified as `NULL`. For our purposes, we will always assume a `NULL` start address. Figure 9.32 depicts the meaning of these arguments.

The `prot` argument contains bits that describe the access permissions of the newly mapped virtual memory area (i.e., the `vm_prot` bits in the corresponding area struct).

PROT_EXEC. Pages in the area consist of instructions that may be executed by the CPU.

PROT_READ. Pages in the area may be read.

PROT_WRITE. Pages in the area may be written.

PROT_NONE. Pages in the area cannot be accessed.

The `flags` argument consists of bits that describe the type of the mapped object. If the `MAP_ANON` flag bit is set, then the backing store is an anonymous object and the corresponding virtual pages are demand-zero. `MAP_PRIVATE` indicates a private copy-on-write object, and `MAP_SHARED` indicates a shared object. For example,

```
bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

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.

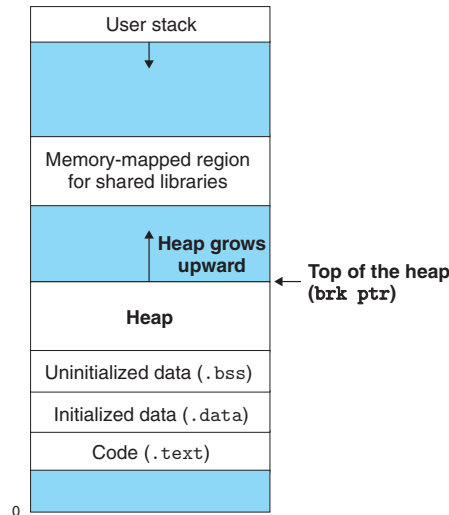
A dynamic memory allocator maintains an area of a process's virtual memory known as the *heap* (Figure 9.33). Details vary from system to system, but without loss of generality, we will assume that the heap is an area of demand-zero memory that begins immediately after the uninitialized data area and 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.

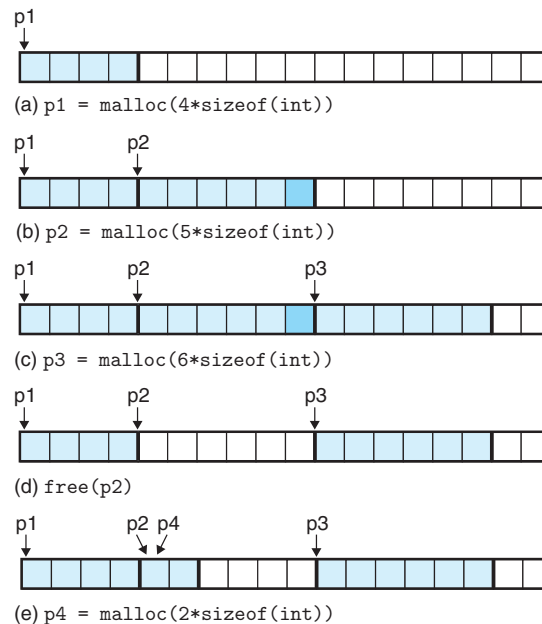


Figure 9.34 shows how an implementation of `malloc` and `free` might manage a (very) small heap of 16 words for a C program. Each box represents a 4-byte word. The heavy-lined rectangles correspond to allocated blocks (shaded) and free blocks (unshaded). Initially, the heap consists of a single 16-word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

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

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

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

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.

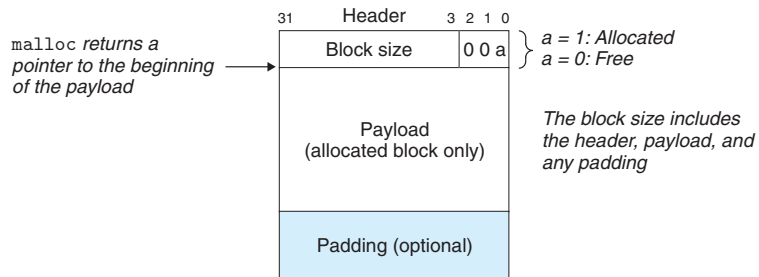
9.9.6 Implicit Free Lists

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.



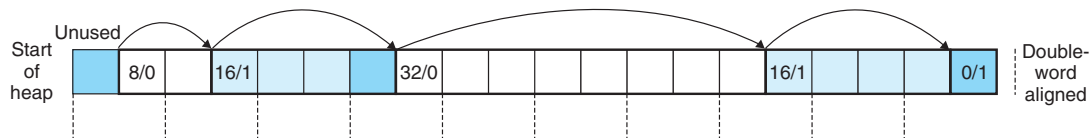


Figure 9.36 Organizing the heap with an implicit free list. Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

(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)
<code>malloc(2)</code>	_____	_____
<code>malloc(9)</code>	_____	_____
<code>malloc(15)</code>	_____	_____
<code>malloc(20)</code>	_____	_____

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

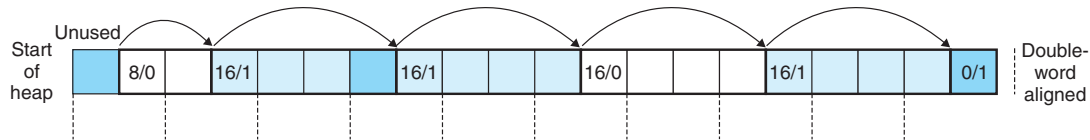


Figure 9.37 Splitting a free block to satisfy a three-word allocation request. Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

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.

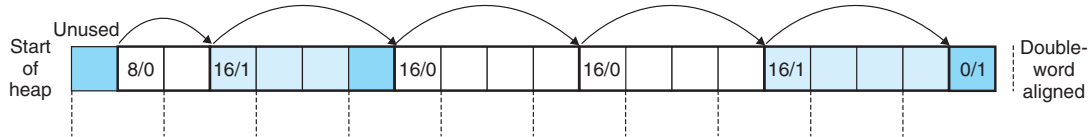


Figure 9.38 An example of false fragmentation. Allocated blocks are shaded. Free blocks are unshaded. Headers are labeled with (size (bytes)/allocated bit).

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.

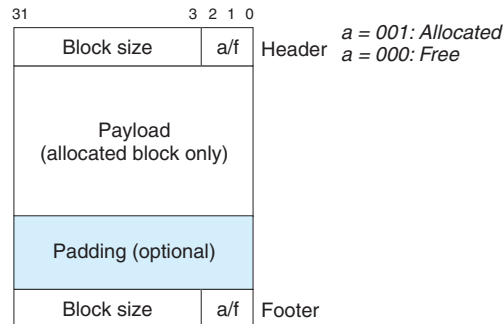


Figure 9.40 shows how we would coalesce each of the four cases.

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.

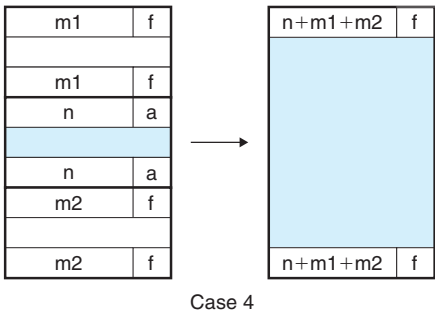
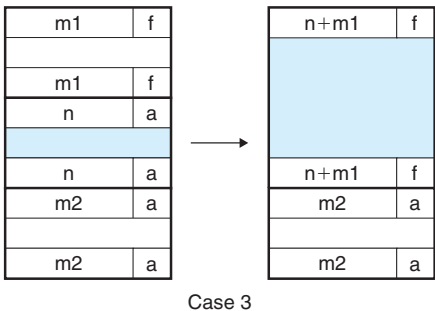
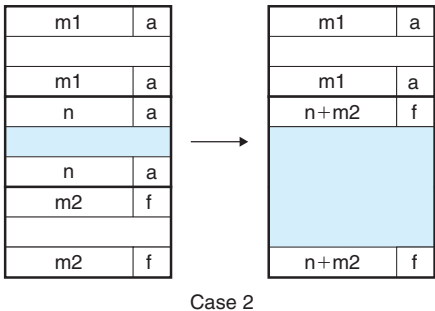
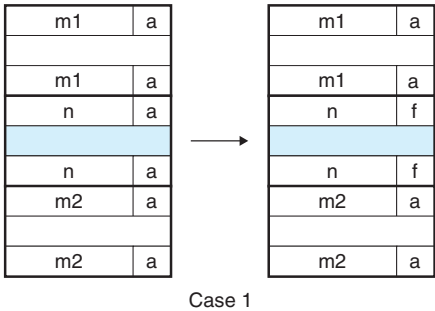
The idea of boundary tags is a simple and elegant one that generalizes to many different types of allocators and free list organizations. However, there is a potential disadvantage. Requiring each block to contain both a header and a footer can introduce significant memory overhead if an application 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

```

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 }

```

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

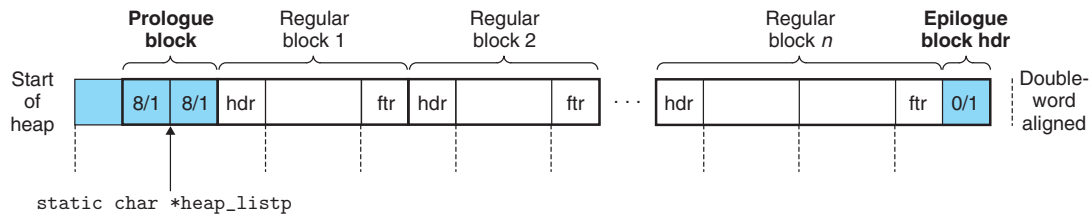


Figure 9.42 Invariant form of the implicit free list.

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)));
```

```

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)))

```

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

```

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.

```

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.

```

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_BLKPTR(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKPTR(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_BLKPTR(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_BLKPTR(bp)));
28         PUT(FTRP(bp), PACK(size, 0));
29         PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
30         bp = PREV_BLKPTR(bp);
31     }
32
33     else {                                  /* Case 4 */
34         size += GET_SIZE(HDRP(PREV_BLKPTR(bp))) +
35             GET_SIZE(FTRP(NEXT_BLKPTR(bp)));
36         PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
37         PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 0));
38         bp = PREV_BLKPTR(bp);
39     }
40     return bp;
41 }

```

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

```

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.

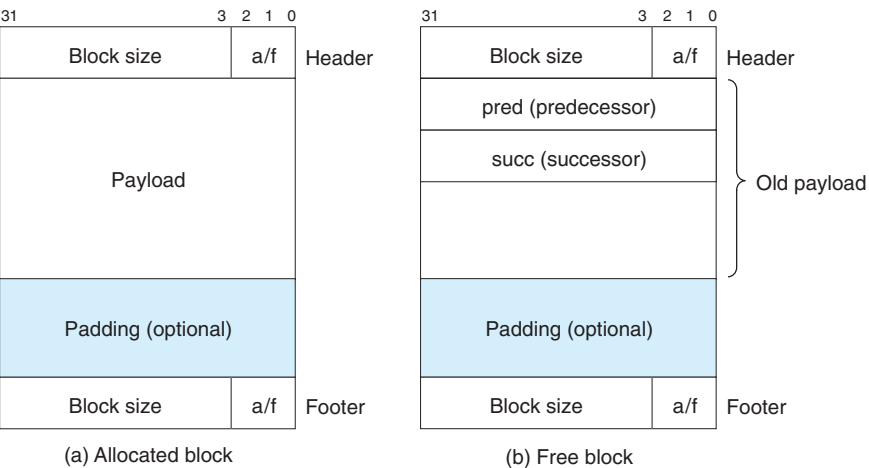


Figure 9.48 Format of heap blocks that use doubly linked free lists.

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:

$$\{1\}, \{2\}, \{3, 4\}, \{5-8\}, \dots, \{1,025-2,048\}, \{2,049-4,096\}, \{4,097-\infty\}$$

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

$$\{1\}, \{2\}, \{3\}, \dots, \{1,023\}, \{1,024\}, \{1,025-2,048\}, \{2,049-4,096\}, \{4,097-\infty\}$$

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 ... x00000`

has its buddy at address

`xxx ... 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.