12

Kernel Memory Allocation

12.1 Introduction

The operating system must manage all the physical memory and allocate it both to other kernel subsystems and to user processes. When the system boots, the kernel reserves part of physical memory for its own text and static data structures. This portion is never released and hence is unavailable for any other purpose.[1] The rest of the memory is managed dynamically—the kernel allocates portions of it to various clients (processes and kernel subsystems), which release it when it is no longer needed.

UNIX divides memory into fixed-size frames or *pages*. The page size is a power of two, with 4 kilobytes being a fairly typical value.[2] Because UNIX is a *virtual memory* system, pages that are logically contiguous in a process address space need not be physically adjacent in memory. The next three chapters describe virtual memory. The memory management subsystem maintains mappings between the logical *(virtual)* pages of a process and the actual location of the data in physical memory. As a result, it can satisfy a request for a block of logically contiguous memory by allocating several physically non-contiguous pages.

This simplifies the task of page allocation. The kernel maintains a linked list of free pages. When a process needs some pages, the kernel removes them from the free list; when the pages are released, the kernel returns them to the free list. The physical location of the pages is unimportant.

[1] Many modern UNIX systems (AIX, for instance) allow part of the kernel to be pageable.

[2] This is a software-defined page size and need not equal the hardware page size, which is the granularity for protection and address translation imposed by the memory management unit.

The `memall()` and `memfree()` routines in 4.3BSD and the `get_page()` and `freepage()` routines in SVR4 implement this *page-level allocator*.

The page-level allocator has two principal clients (Figure 12-1). One is the *paging system*, which is part of the virtual memory system. It allocates pages to user processes to hold portions of their address space. In many UNIX systems, the paging system also provides pages for disk block buffers. The other client is the *kernel memory allocator*, which provides odd-sized buffers of memory to various kernel subsystems. The kernel frequently needs chunks of memory of various sizes, usually for short periods of time.

The following are some common users of the kernel memory allocator:

- The pathname translation routine may allocate a buffer (usually 1024 bytes) to copy a pathname from user space.
- The `allocb()` routine allocates STREAMS buffers of arbitrary size.
- Many UNIX implementations allocate `zombie` structures to retain exit status and resource usage information about deceased processes.
- In SVR4, the kernel allocates many objects (such as `proc` structures, vnodes, and file descriptor blocks) dynamically when needed.

Most of these requests are much smaller than a page, and hence the page-level allocator is inappropriate for this task. A separate mechanism is required to allocate memory at a finer granularity. One simple solution is to avoid dynamic memory allocation altogether. Early UNIX implementations [Bach 86] used fixed-size tables for vnodes, `proc` structures, and so forth. When memory was required for holding temporary pathnames or network messages, they borrowed buffers from the block buffer cache. Additionally, a few *ad hoc* allocation schemes were devised for special situations, such as the *clists* used by the terminal drivers.

This approach has several problems. It is highly inflexible, because the sizes of all tables and caches are fixed at boot time (often at compile time) and can not adjust to the changing demands on

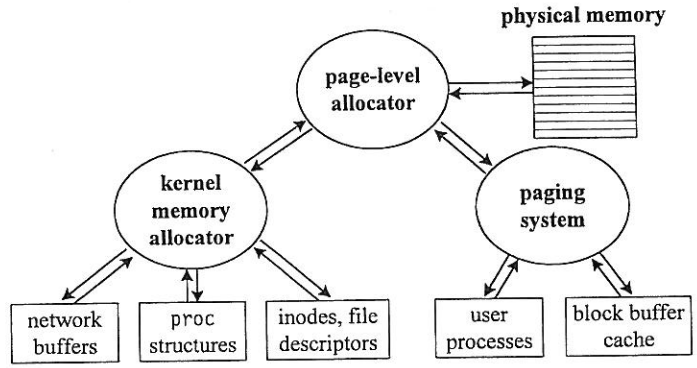

Figure 12-1. Memory allocators in the kernel.

the system. The default sizes of these tables are selected by the system developers based on the usage patterns expected with typical workloads. Although system administrators can usually tune these sizes, they have little guidance for doing so. If any table size is set too low, the table could overflow and perhaps crash the system without warning. If the system is configured conservatively, with large sizes of all tables, it wastes too much memory, leaving little for the applications. This causes the overall performance to suffer.

Clearly, the kernel needs a general-purpose memory allocator that can handle requests for large and small chunks of data efficiently. In the following section, we describe the requirements for this allocator and the criteria by which we can judge different implementations. We then describe and analyze various memory allocators used by modern UNIX systems.

12.2 Functional Requirements

The kernel memory allocator (KMA) services requests for dynamic memory allocation from several clients such as the pathname parser, STREAMS, and the interprocess communication facility. It does not handle requests for user process pages, which are the responsibility of the paging system.

When the system boots, the kernel first reserves space for its own text and static data structures, as well as some pre-defined pools such as the block buffer cache. The page-level allocator manages the remaining physical memory, which is contended for both by the kernel's own dynamic allocation requests and by user processes.

The page-level allocator pre-allocates part of this space to the KMA, which must use this memory *pool* efficiently. Some implementations allow no change in the total memory given to the KMA. Others allow the KMA to steal more memory from the paging system. Some even permit a two-way exchange, so the paging system can steal back excess free memory held by the KMA.

If the KMA runs out of memory, it blocks the caller until more memory is free. The caller may send a flag in the request, asking the KMA to return a failure status (usually a NULL pointer) instead of blocking. This option is most used by interrupt handlers, which must take some corrective action if the request fails. For example, if a network interrupt cannot allocate memory to hold an incoming packet, it may simply drop the packet, hoping the sender will retransmit it later.

The KMA must monitor which parts of its pool are allocated and which are free. Once a piece of memory is freed, it should be available to other requests. Ideally, a request for memory should fail only when memory is really full, that is, when the total free memory available to the allocator is less than the amount requested. In reality, the allocator fails sooner than that because of *fragmentation*—even if there is enough memory available to satisfy the request, it may not be available as one contiguous chunk.

12.2.1 Evaluation Criteria

An important criterion for evaluating a memory allocator is its ability to minimize wastage. Physical memory is limited, so the allocator must be space-efficient. One measure of efficiency is the *utilization factor,* which is the ratio of the total memory requested to that required to satisfy the requests. An ideal allocator would have 100% utilization; in practice, 50% is acceptable [Korn 85]. The major cause of wasted memory is fragmentation—the free memory is broken into chunks that are too

small to be useful. The allocator reduces fragmentation by coalescing adjacent chunks of free memory into a single large chunk.

A KMA must be fast, because it is used extensively by various kernel subsystems, including interrupt handlers, whose performance is usually critical. Both the average and the worst-case latency are important. Because kernel stacks are small, the kernel uses dynamic allocation in many situations where a user process would simply allocate the object on its stack. This makes allocation speed all the more important. A slow allocator degrades the performance of the entire system.

The allocator must have a simple programming interface that is suitable for a wide variety of clients. One possibility is to have an interface similar to the `malloc()` and `free()` functions of the user-level memory allocator provided by the standard library:

```
void* malloc (size_t nbytes);
void free (void* ptr);
```

An important advantage of this interface is that the `free()` routine does not need to know the size of the region being freed. Often, one kernel function allocates a chunk of memory and passes it to another subsystem, which eventually frees it. For example, a network driver may allocate a buffer for an incoming message and send it to a higher-level module to process the data and free the buffer. The module releasing the memory may not know the size of the allocated object. If the KMA can monitor this information, it will simplify the work of its clients.

Another desirable interface feature is that the client not be forced to release the entire allocated area all at once. If a client wants to release only part of the memory, the allocator should handle it correctly. The `malloc()/free()` interface does not permit this. The `free()` routine will release the entire region and will fail if called with a different address from that returned by `malloc()`. Allowing clients to grow a buffer (for instance by a `realloc()` function) would also be useful.

Allocated memory should be properly aligned for faster access. On many RISC architectures, this is a requirement. For most systems, longword alignment is sufficient, but 64-bit machines such as DEC's Alpha AXP [DEC 92] may require alignment on an eight-byte boundary. A related issue is the minimum allocation size, which is usually eight or sixteen bytes.

Many commercial environments have a cyclical usage pattern. For example, a machine may be used for database queries and transaction processing during the day and for backups and database reorganization at night. These activities may have different memory requirements. Transaction processing might consume several small chunks of kernel memory to implement database locking, while backups may require that most of the memory be dedicated to user processes.

Many allocators partition the pool into separate regions, or *buckets,* for requests of different sizes. For instance, one bucket may contain all 16-byte chunks, while another may contain all 64-byte chunks. Such allocators must guard against a bursty or cyclical usage pattern as described above. In some allocators, once memory has been assigned to a particular bucket, it cannot be reused for requests of another size. This may result in a large amount of unused memory in some buckets, and hence not enough in others. A good allocator provides a way to dynamically recover excess memory from one bucket for use by another.

Finally, the interaction with the paging system is an important criterion. The KMA must be able to borrow memory from the paging system when it uses up its initial quota. The paging system

must be able to recover unused memory from the KMA. This exchange should be properly controlled to ensure fairness and avoid starvation of either system.

We now look at several allocation methods, and analyze them using the above criteria.

12.3 Resource Map Allocator

The *resource map* is a set of <*base, size*> pairs that monitor areas of free memory (see Figure 12-2). Initially, the pool is described by a single map entry, whose *base* equals the starting address of the pool and *size* equals the total memory in the pool (Figure 12-2(a)). As clients allocate and free chunks of memory, the pool becomes fragmented, and the kernel creates one map entry for each contiguous free region. The entries are sorted in order of increasing base address, making it easy to coalesce adjacent free regions.

Using a resource map, the kernel can satisfy new allocation requests using one of three policies:

- **First fit** — Allocates memory from the first free region that has enough space. This is the fastest algorithm, but may not be optimal for reducing fragmentation.
- **Best fit** — Allocates memory from the smallest region that is large enough to satisfy the request. This has the drawback that it might leave several free regions that are too small to be useful.
- **Worst fit** — Allocates memory from the largest available region, unless a perfect fit is found. This may seem counter-intuitive, but its usefulness is based on the expectation that the region left behind after the allocation will be large enough to be used for a future request.

No one algorithm is ideal for all usage patterns. [Knut 73] provides a detailed analysis of these and other approaches. UNIX chooses the first-fit method.

Figure 12-2 describes a simple resource map that manages a 1024-byte region of memory. It supports two operations:

```
offset_t rmalloc (size);      /* returns offset of allocated region */
void rmfree (base, size);
```

Initially (Figure 12-2(a)), the entire region is free and is described by a single map entry. We then have two allocation requests, for 256 and 320 bytes respectively. This is followed by the release of 128 bytes starting at offset 256. Figure 12-2(b) shows the state of the map after these operations. We now have two free regions, and hence two map entries to describe them.

Next, another 128 bytes are released starting at offset 128. The allocator discovers that this region is contiguous with the free region that starts at offset 256. It combines them into a single, 256-byte, free region, resulting in the map shown in Figure 12-2(c). Finally, Figure 12-2(d) shows the map at a later time, after many more operations have occurred. Note that while the total free space is 256 bytes, the allocator cannot satisfy any request greater than 128 bytes.

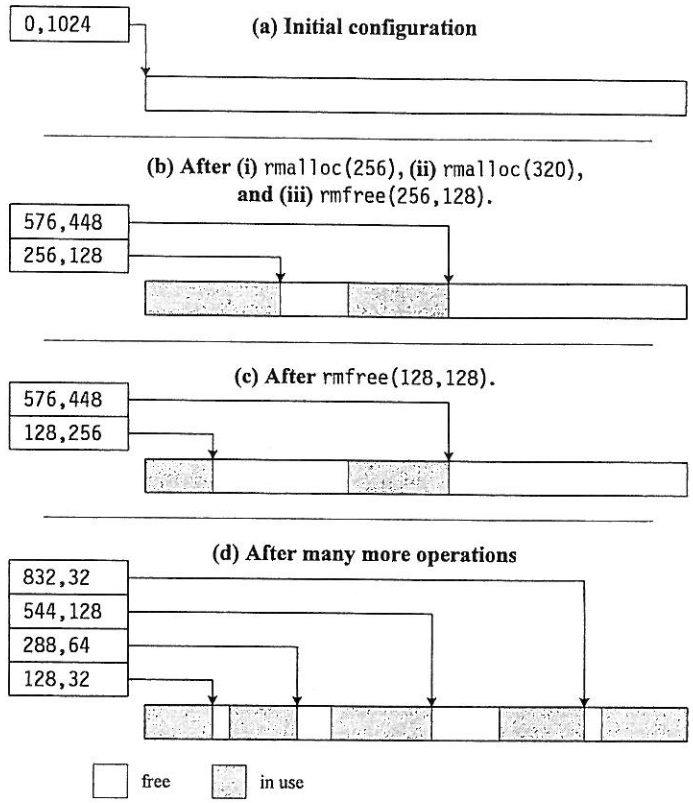

Figure 12-2. Using a resource map allocator.

12.3.1 Analysis

The resource map provides a simple allocator. The following are its main advantages:

- The algorithm is easy to implement.
- The resource map is not restricted to memory allocation. It can manage collections of arbitrary objects that are sequentially ordered and require allocation and freeing in contiguous chunks (such as page table entries and semaphores, as described below).

- It can allocate the exact number of bytes requested without wasting space. In practice, it will usually round up requests to four- or eight-byte multiples for simplicity and alignment.
- A client is not constrained to release the exact region it has allocated. As the previous example shows, the client can release any part of the region, and the allocator will handle it correctly. This is because the arguments to `rmfree()` provide the size of the region being freed, and the bookkeeping information (the map) is maintained separately from the allocated memory.
- The allocator coalesces adjacent free regions, allowing memory to be reused for different sized requests.

However, the resource map allocator also has some major drawbacks:

- After the allocator has been running for a while, the map becomes highly fragmented, creating many small free regions. This results in low utilization. In particular, the resource map allocator does poorly in servicing large requests.
- As the fragmentation increases, so does the size of the resource map, since it needs one entry for each free region. If the map is preconfigured with a fixed number of entries, it might overflow, and the allocator may lose track of some free regions.
- If the map grows dynamically, it needs an allocator for its own entries. This is a recursive problem, to which we offer one solution below.
- To coalesce adjacent free regions, the allocator must keep the map sorted in order of increasing base offsets. Sorting is expensive, even more so if it must be performed in-place, such as when the map is implemented as a fixed array. The sorting overhead is significant, even if the map is dynamically allocated and organized as a linked list.
- The allocator must perform a linear search of the map to find a free region that is large enough. This is extremely time consuming and becomes slower as fragmentation increases.
- Although it is possible to return free memory at the tail of the pool to the paging system, the algorithm is really not designed for this. In practice, the allocator never shrinks its pool.

The poor performance of the resource map is the main reason why it is unsuitable as a general-purpose kernel memory allocator. It is, however, used by some kernel subsystems. The System V interprocess communication facility uses resource maps to allocate semaphore sets and data areas for messages. The virtual memory subsystem in 4.3BSD uses this algorithm to manage system page table entries that map user page tables (see Section 13.4.2).

The map management can be improved in some circumstances. It is often possible to store the map entry in the first few bytes of the free region. This requires no extra memory for the map and no dynamic allocation for map entries. A single global variable can point to the first free region, and each free region stores its size and a pointer to the next free entry. This requires free regions to be at least two words long (one for the size, one for the pointer), which can be enforced by requiring allocation and freeing words in multiples of two. The Berkeley Fast File System (FFS), described in Section 9.5, uses a variation of this approach to manage free space within directory blocks.

While this optimization is suitable for the general memory allocator, it cannot be applied to other uses of the resource map, such as for semaphore sets or page table entries, where the managed objects have no room for map entry information.

12.4 Simple Power-of-Two Free Lists

The power-of-two free lists method is used frequently to implement `malloc()` and `free()` in the user-level C library. This approach uses a set of free lists. Each list stores buffers of a particular size, and all the sizes are powers of two. For example, in Figure 12-3, there are six free lists, storing buffers of sizes 32, 64, 128, 256, 512, and 1024 bytes.

Each buffer has a one-word header, which reduces its usable area by this amount. When the buffer is free, its header stores a pointer to the next free buffer. When the buffer is allocated, its header points to the free list to which it should be returned. In some implementations, it contains the size of the allocated area instead. This helps detect certain bugs, but requires the `free()` routine to compute the free list location from the size.

To allocate memory, the client calls `malloc()`, passing the required size as an argument. The allocator computes the size of the smallest buffer that is large enough to satisfy the request. This involves adding space for the header to the requested size and rounding the resulting value to the next power of two. The 32-byte buffers satisfy requests for 0–28 bytes, the 64-byte buffers satisfy requests for 29–60 bytes, and so on. The allocator then removes a buffer from the appropriate free list and writes a pointer to the free list in the header. It returns to the caller a pointer to the byte immediately following the header in the buffer.

When the client releases the buffer, it calls the `free()` routine, passing the pointer returned by `malloc()` as an argument. The user does not have to specify the size of the buffer being freed. It is essential, however, to free the entire buffer obtained from `malloc()`; there is no provision for freeing only part of the allocated buffer. The `free()` routine moves the pointer back four bytes to access the header. It obtains the free list pointer from the header and puts the buffer on that list.

The allocator can be initialized either by preallocating a number of buffers to each list or by leaving the lists empty at first and calling the page-level allocator to populate them as required. Sub-

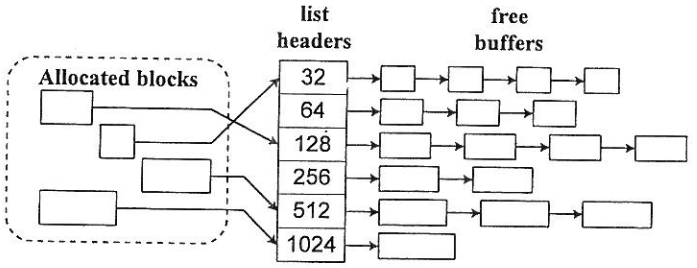

Figure 12-3. The power-of-two free list allocator.

sequently, if a list becomes empty, the allocator may handle a new `malloc()` request for that size in one of three ways:

- Block the request until a buffer of the appropriate size is released.
- Satisfy the request with a larger buffer, beginning with the next list and continuing the search until it finds a nonempty list.
- Obtain additional memory from the page-level allocator to create more buffers of that size.

Each method has its benefits and drawbacks, and the proper choice depends on the situation. For example, a kernel implementation of this algorithm may use an additional priority argument for allocation requests. In this case, the allocator may block low-priority requests that cannot be satisfied from the correct free list, but complete high-priority requests by one of the other two methods.

12.4.1 Analysis

The above algorithm is simple and reasonably fast. Its main appeal is that it avoids the lengthy linear searches of the resource map method and eliminates the fragmentation problem entirely. In situations where a buffer is available, its worst-case performance is well bounded. The allocator also presents a familiar programming interface, with the important advantage that the `free()` routine need not be given the buffer size as an argument. As a result, an allocated buffer can be passed to other functions and subsystems and eventually freed using only the pointer to the buffer. On the other hand, the interface does not allow a client to release only part of the allocated buffer.

There are many important drawbacks of this algorithm. The rounding of requests to the next power of two often leaves a lot of unused space in the buffer, resulting in poor memory utilization. The problem becomes worse due to the need to store the header in the allocated buffers. Many memory requests are for an exact power-of-two bytes. For such requests, the wastage is almost 100%, since the request must be rounded to the next power of two to allow for the header. For example, a 512-byte request would consume a 1024-byte buffer.

There is no provision for coalescing adjacent free buffers to satisfy larger requests. Generally, the size of each buffer remains fixed for its lifetime. The only flexibility is that large buffers may sometimes be used for small requests. Although some implementations allow the allocator to steal memory from the paging system, there is no provision to return surplus free buffers to the page-level allocator.

While the algorithm is much faster than the resource map method, it can be further improved. In particular, the round-up loop, shown in Example 12-1, is slow and inefficient:

```
void*malloc (size)
{
    int ndx = 0;                    /* free list index */
    int bufsize = 1 << MINPOWER;    /* size of smallest buffer */
    size += 4;                      /* account for header */
    assert (size <= MAXBUFSIZE);
    while (bufsize < size)  {
        ndx++;
        bufsize <<= 1;
    }
... /* at this point, ndx is the index of the appropriate free list */
}
```

Example 12-1. Crude implementation of `malloc()`.

The next section describes an improved algorithm that addresses many of these problems.

12.5 The McKusick-Karels Allocator

Kirk McKusick and Michael Karels introduced an improved power-of-two allocator [McKu 88], which is now used in several UNIX variants including 4.4BSD and Digital UNIX. In particular, it eliminates space wastage in the common case where the size of the requested memory was exactly a power of two. It also optimizes the round-up computation and eliminates it if the allocation size is known at the time of compilation.

The McKusick-Karels algorithm requires the memory managed by the allocator to comprise a set of contiguous pages and all buffers belonging to the same page to be the same size (a power of two). It uses an additional page usage array (`kmemsizes[]`) to manage its pages. Each page may be in one of three states:

- Free—the corresponding element of `kmemsizes[]` contains a pointer to the element for the next free page.
- Divided into buffers of a particular size—the `kmemsizes[]` element contains the size.
- Part of a buffer that spanned multiple pages—the `kmemsizes[]` element corresponding to the first page of the buffer contains the buffer size.

Figure 12-4 shows a simple example for a 1024-byte page size. `freelistarr[]` is the usual array of free list headers for all buffer sizes smaller than one page.

Since all buffers on the same page are of the same size, allocated buffers do not need a header to store a free list pointer. The `free()` routine locates the page by masking off the low-order bits of the buffer address and finding the size of the buffer in the corresponding element of the `kmemsizes[]` array. Eliminating the header in allocated buffers yields the greatest savings for memory requests whose sizes are an exact power of two.

The call to `malloc()` is replaced by a macro that rounds the request to the next power of two (allocated buffers have no header information) and removes a buffer from the appropriate free list. The macro calls the `malloc()` function for requests of one or more pages or if the appropriate free list is empty. In the latter case, `malloc()` calls a routine that consumes a free page and divides it into buffers of the required size. In the macro, the round-up loop is replaced by a set of conditional expressions. Example 12-2 provides an implementation for the pool shown in Figure 12-4:

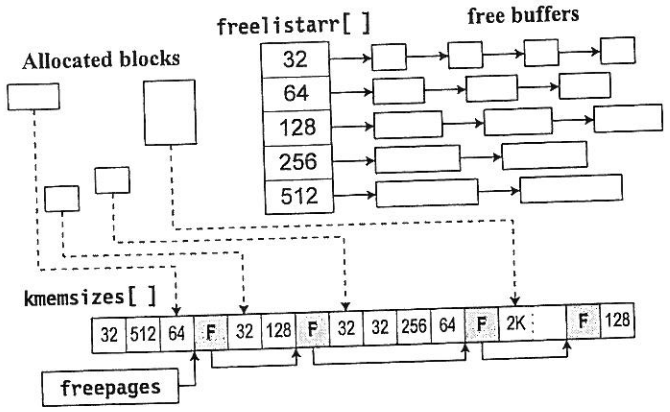

Figure 12-4. The McKusick - Karels allocator.

```
#define NDX(size)   \
    (size) > 128    \
        ? (size) > 256 ? 4 : 3  \
        : (size) > 64   \
            ? 2 \
            : (size) > 32 ? 1 : 0


#define MALLOC(space, cast, size, flags)    \
    {   \
        register struct freelisthdr* flh;   \
        if (size <= 512 &&  \
        (flh = freelistarr [NDX(size)]) != NULL)    {   \
            space = (cast)flh->next;    \
            flh->next = *(caddr_t *)space;  \
        } else  \
            space = (cast)malloc (size, flags); \
    }
```

Example 12-2. Using macros to speed up malloc.

The main advantage of using a macro is that when the allocation size is known at the time of compilation, the `NDX()` macro reduces to a compile-time constant, saving a substantial number of instructions. Another macro handles the simple cases of buffer release, calling the `free()` function in only a few cases, such as when freeing large buffers.

12.5.1 Analysis

The McKusick-Karels algorithm is a significant improvement over the simple power-of-two allocator described in Section 12.4. It is faster, wastes less memory, and can handle large and small requests efficiently. However, the algorithm suffers from some of the drawbacks inherent in the power-of-two approach. There is no provision for moving memory from one list to another. This makes the allocator vulnerable to a bursty usage pattern that consumes a large number of buffers of one particular size for a short period. Also, there is no way to return memory to the paging system.

12.6 The Buddy System

The *buddy system* [Pete 77] is an allocation scheme that combines free buffer coalescing with a power-of-two allocator.[3] Its basic approach is to create small buffers by repeatedly halving a large buffer and coalescing adjacent free buffers whenever possible. When a buffer is split, each half is called the *buddy* of the other.

To explain this method, let us consider a simple example (Figure 12-5), where a buddy algorithm is used to manage a 1024-byte block with a minimum allocation size of 32 bytes. The allocator uses a bitmap to monitor each 32-byte chunk of the block; if a bit is set, the corresponding chunk is in use. It also maintains free lists for each possible buffer size (powers of two between 32 and 512). Initially, the entire block is a single buffer. Let us consider the effect of the following sequence of requests and allocator actions:

1. *allocate (256):* Splits the block into two buddies—**A** and **A′**—and puts **A′** on the 512-byte free list. It then splits **A** into **B** and **B′**, puts **B′** on the 256-byte free list, and returns **B** to the client.

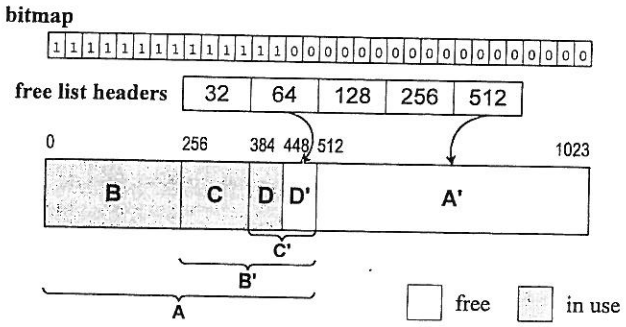

Figure 12-5. The buddy system.

[3] This is the binary buddy system, which is the simplest and most popular buddy system. We can implement other buddy algorithms by splitting buffers into four, eight, or more pieces.

2. *allocate (128):* Finds the 128-byte free list empty. It checks the 256-byte list, removes **B′** from it, and splits it into **C** and **C′**. Then, it puts **C′** on the 128-byte free list and returns **C** to the client.
3. *allocate (64):* Finds the 64-byte list empty, and hence removes **C′** from 128-byte free list. It splits **C′** into **D** and **D′**, puts **D′** on the 64-byte list, and returns **D** to the client. Figure 12-5 shows the situation at this point.
4. *allocate (128):* Finds the 128-byte and 256-byte lists empty. It then checks the 512-byte free list and removes **A′** from it. Next, it splits **A′** into **E** and **E′**, and further splits **E** into **F** and **F′**. Finally, it puts **E′** onto the 256-byte list, puts **F′** on the 128-byte list, and returns **F** to the client.
5. *release (C, 128):* Returns **C** to the 128-byte free list. This leads to the situation shown in Figure 12-6.

So far, there has been no coalescing. Suppose the next operation is

6. *release (D, 64):* The allocator will note that **D′** is also free and will coalesce **D** with **D′** to obtain **C′**. It will further note that **C** is also free and will coalesce it with **C′** to get back **B′**. Finally, it will return **B′** to the 256-byte list, resulting in the situation in Figure 12-7.

A few points of interest need to be clarified:

- There is the usual rounding of the request to the next power of two.
- For each request in this example, the corresponding free list is empty. Often, this is not the case. If there is a buffer available on the appropriate free list, the allocator uses it, and no splitting is required.

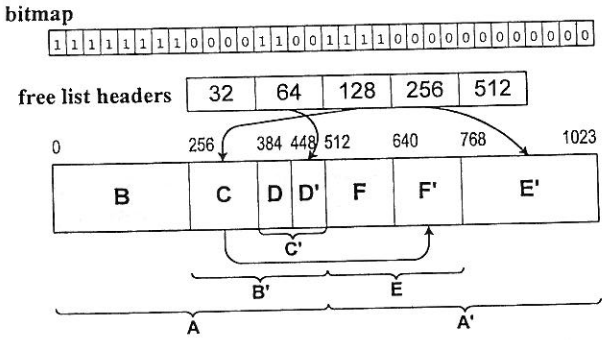

Figure 12-6. The buddy system, stage 2.

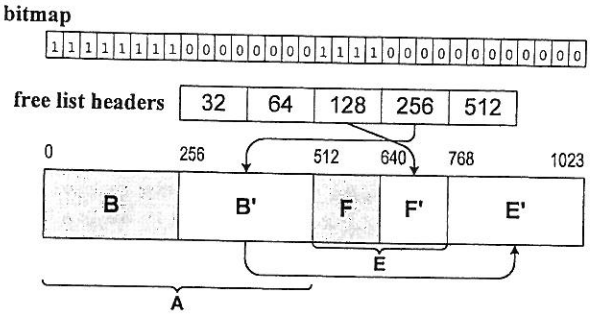

Figure 12-7. The buddy system, stage 3.

- The address and size of a buffer provide all the information required to locate its buddy. This is because the algorithm automatically gives each buffer an alignment factor equal to its size. Thus, for example, a 128-byte buffer at offset 256 has its buddy at offset 384, while a 256-byte buffer at the same offset has its buddy at offset 0.
- Each request also updates the bitmap to reflect the new state of the buffer. While coalescing, the allocator examines the bitmap to determine whether a buffer's buddy is free.
- While the above example uses a single, 1024-byte page, the allocator can manage several disjoint pages simultaneously. The single set of free list headers can hold free buffers from all pages. The coalescing will work as before, since the buddy is determined from the buffer's offset in the page. The allocator will, however, maintain a separate bitmap for each page.

12.6.1 Analysis

The buddy system does a good job of coalescing adjacent free buffers. That provides flexibility, allowing memory to be reused for buffers of a different size. It also allows easy exchange of memory between the allocator and the paging system. Whenever the allocator needs more memory it can obtain a new page from the paging system and split it as necessary. Whenever the release routine coalesces an entire page, the page can be returned to the paging system.

The main disadvantage of this algorithm is its performance. Every time a buffer is released, the allocator tries to coalesce as much as possible. When allocate and release requests alternate, the algorithm may coalesce buffers, only to split them again immediately. The coalescing is recursive, resulting in extremely poor worst-case behavior. In the next section, we examine how SVR4 modifies this algorithm to overcome this performance bottleneck.

Another drawback is the programming interface. The release routine needs both the address and size of the buffer. Moreover, the allocator requires that an entire buffer be released. Partial release is insufficient, since a partial buffer has no buddy.

12.7 The SVR4 Lazy Buddy Algorithm

The major problem with the simple buddy algorithm is the poor performance due to repetitive coalescing and splitting of buffers. Normally, memory allocators are in a steady state, where the number of in-use buffers of each size remains in a fairly narrow range. Under such conditions, coalescing offers no advantage and is only a waste of time. Coalescing is necessary only to deal with bursty conditions, where there are large, temporary, variations in the buffer usage pattern.

We define a *coalescing delay* as the time taken to either coalesce a single buffer with its buddy or determine that its buddy is not free. The coalescing results in a buffer of the next larger size, and the process is recursive until we find a buffer whose buddy is not free. In the buddy algorithm, each release operation incurs at least one coalescing delay, and often more than one.

A straightforward solution is to defer coalescing until it becomes necessary, and then to coalesce as many buffers as possible. Although this reduces the average time for allocation and release, the few requests that invoke the coalescing routine are slow. Because the allocator may be called from time-critical functions such as interrupt handlers, it is essential to control this worst-case behavior. We need an intermediate approach that defers coalescing, but does not wait until the situation is critical, and amortizes the cost of coalescing over several requests. [Lee 89] suggests a solution based on low- and high-watermarks on each buffer class. The SVR4 approach [Bark 89] described below is based on the same idea, but is more efficient.

12.7.1 Lazy Coalescing

Buffer release involves two steps. First, the buffer is put on the free list, making it available for other allocation requests. Second, the buffer is marked as free in the bitmap and coalesced with adjacent buffers if possible; this is the coalescing operation. The normal buddy system performs both steps on each release operation.

The lazy buddy system always performs the first step, which makes the buffer *locally free* (available for allocation within the class, but not for coalescing). Whether it performs the second step depends on the state of the buffer class. At any time, a class has N buffers, of which A buffers are active, L are locally free, and G are *globally free* (marked free in the bitmap, available for coalescing). Hence,

$$N = A + L + G$$

Depending on the values of these parameters, a buffer class is said to be in one of three states:

- **lazy** — buffer consumption is in a steady state (allocation and release requests are about equal) and coalescing is not necessary.
- **reclaiming** — consumption is borderline; coalescing is needed.
- **accelerated** — consumption is not in a steady state, and the allocator must coalesce faster.

The critical parameter that determines the state is called the *slack*, defined as

$$slack = N - 2L - G$$

The system is in the lazy state when *slack* is 2 or more, in the reclaiming state when *slack* equals 1, and in the accelerated state when *slack* is zero. The algorithm ensures that *slack* is never negative. [Bark 89] provides comprehensive proof of why the *slack* is an effective measure of the buffer class state.

When a buffer is released, the SVR4 allocator puts it on the free list and examines the resulting state of the class. If the list is in the lazy state, the allocator does no more. The buffer is not marked as free in the bitmap. Such a buffer is called a *delayed* buffer and is identified as such by a flag in the buffer header (the header is present only on buffers on the free list). Although it is available for other same-size requests, it cannot be coalesced with adjacent buffers.

If the list is in the reclaiming state, the allocator marks the buffer as free in the bitmap and coalesces it if possible. If the list is in accelerated state, the allocator coalesces two buffers—the one just released and an additional delayed buffer, if there is one. When it releases the coalesced buffer to the next higher-sized list, the allocator checks the state of that class to decide whether to coalesce further. Each of these operations changes the *slack* value, which must be recomputed.

To implement this algorithm efficiently, the buffers are doubly linked on the free lists. Delayed buffers are released to the head of the list, and non-delayed buffers to the tail. This way, delayed buffers are reallocated first; this is desirable because they are the least expensive to allocate (no bitmap update is required). Moreover, in the accelerated stage, the additional delayed buffer can be quickly checked for and retrieved from the head of the list. If the first buffer is non-delayed, there are no delayed buffers on the list.

This is a substantial improvement over the basic buddy system. In steady state, all lists are in the lazy state, and no time is wasted in coalescing and splitting. Even when a list is in the accelerated state, the allocator coalesces at most two buffers on each request. Hence, in the worst-case situation, there are at most two coalescing delays per class, which is at most twice as bad as the simple buddy system.

[Bark 89] analyzes the performance of the buddy and lazy buddy algorithms under various simulated workloads. It shows that the average latency of the lazy buddy method is 10% to 32% better than that of the simple buddy system. As expected, however, the lazy buddy system has greater variance and poorer worst-case behavior for the release routine.

12.7.2 SVR4 Implementation Details

SVR4 uses two types of memory pools—large and small. Each small pool begins with a 4096-byte block, divided into 256-byte buffers. The first two buffers are used to maintain the data structures (such as the bitmap) for this pool, while the rest are available for allocation and splitting. This pool allocates buffers whose sizes are powers of two ranging from 8 to 256 bytes. A large pool begins with a 16-kilobyte block and allocates buffers of size 512 to 16K bytes. In steady state, there are numerous active pools of both types.

The allocator exchanges memory with the paging system in pool-sized units. When it needs more memory, it acquires a large or small pool from the page-level allocator. When all memory in a pool is coalesced, the allocator returns the pool to the paging system.

A large pool is coalesced by the lazy buddy algorithm alone, because the pool size equals that of the largest buffer class. For a small pool, the buddy algorithm only coalesces up to 256 bytes,

and we need a separate function to gather the 256-byte buffers of a pool. This is time-consuming, and should be performed in the background. A system process called the *kmdaemon* runs periodically to coalesce the pools and return free pools to the page-level allocator.

12.8 The Mach-OSF/1 Zone Allocator

The *zone* allocator used in Mach and OSF/1 [Sciv 90] provides fast memory allocation and performs garbage collection in the background. Each class of dynamically allocated objects (such as `proc` structures, credentials, or message headers) is assigned its own *zone*, which is simply a pool of free objects of that class. Even if objects of two classes have the same size, they have their own zone. For example, both *port translations* and *port sets* (see Chapter 6) are 104 bytes in size [DEC 93], but each has its own zone. There is also a set of power-of-two-sized zones used by miscellaneous clients that do not require a private pool of objects.

Zones are initially populated by allocating memory from the page-level allocator, which also provides additional memory when required. Any single page is only used for one zone; hence, all objects on the same physical page belong to the same class. The free objects of each zone are maintained on a linked list, headed by a `struct zone`. These themselves are dynamically allocated from a *zone of zones*, each element of which is a `struct zone`.

Each kernel subsystem initializes the zones it will need, using the function

```
zinit (size, max, alloc, name);
```

where `size` is the size of each object, `max` is the maximum size in bytes the zone may reach, `alloc` is the amount of memory to add to the zone each time the free list becomes empty (the kernel rounds it to a whole number of pages), and `name` is a string that describes the objects in the zone. `zinit()` allocates a `zone` structure from the *zone of zones* and records the `size`, `max`, and `alloc` values in it. `zinit()` then allocates an initial `alloc`-byte region of memory from the page-level allocator and divides it into `size`-byte objects, which it puts on the free list. All active `zone` structures are maintained on a linked list, described by the global variables `first_zone` and `last_zone` (Figure 12-8). The first element on this list is the *zone of zones*, from which all other elements are allocated.

Thereafter, allocation and release are extremely fast, and involve nothing more than removing objects from and returning objects to the free list. If an allocation request finds the free list empty, it asks the page-level allocator for `alloc` more bytes. If the size of the pool reaches `max` bytes, further allocations will fail.

12.8.1 Garbage Collection

Obviously, a scheme like this requires garbage collection, otherwise a bursty usage pattern will leave a lot of memory unusable. This happens in the background, so that it does not lead to deviant worst-case behavior of a few operations. The allocator maintains an array called the *zone page map*, with one element for each page that is assigned to a zone. Each map entry contains two counts:

- `in_free_list` is the number of objects from that page on the free list.

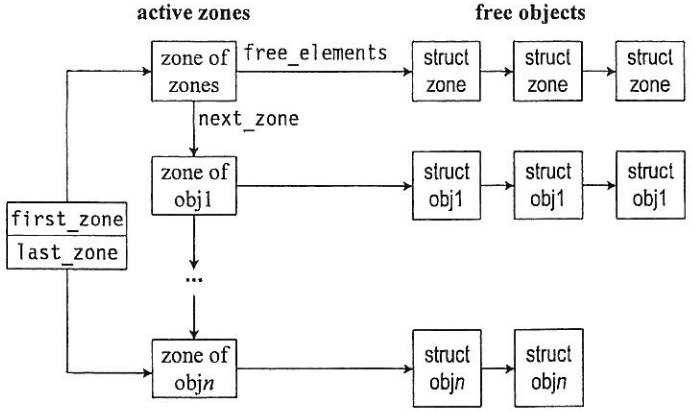

Figure 12-8. The zone allocator.

- `alloc_count` is the total number of objects from that page assigned to the zone.

The `alloc_count` is set whenever the page is acquired by the zone from the page-level allocator. Since the page size may not be an exact multiple of the object size, an object may occasionally span two pages. In this case, it is included in the `alloc_count` of both pages. The `in_free_list` count is not updated with each allocation and release operation, but is recomputed each time the garbage collector runs. This minimizes the latency of individual allocator requests.

The garbage collector routine, `zone_gc()`, is invoked by the swapper task each time it runs. It walks through the list of zones and, for each zone, makes two passes through its free list. In the first pass, it scans each free element and increments the `in_free_count` of the page to which it belongs. At the end of this scan, if the `in_free_list` and `alloc_count` of any page are equal, all objects on that page are free, and the page can be recaptured.[4] Hence, in the second pass, `zone_gc()` removes all such objects from the free list. Finally, it calls `kmem_free()` to return each free page to the page-level allocator.

12.8.2 Analysis

The zone allocator is fast and efficient. It has a simple programming interface. Objects are allocated by

```
obj = void* zalloc (struct zone* z);
```

[4] If the page being recaptured has objects at its top or bottom that span two pages, such objects must be removed from the free list, and the `alloc_count` of the other page must be decremented to reflect this.

where z points to the zone for that class of objects, set up by an earlier call to `zinit()`. The objects are released by

```
void zfree(struct zone* z, void* obj);
```

This requires that clients release allocated objects in their entirety and that they know to which zone the objects must be released. There is no provision for releasing only part of the allocated object.

One interesting property is that the zone allocator uses itself to allocate zone structures for newly created zones (using the *zone of zones*). This leads to a "chicken-and-egg problem" when the system is bootstrapped. The memory management system needs zones to allocate its own data structures, while the zones subsystem needs the page-level allocator, which is part of the memory management system. This problem is addressed by using a small, statically configured, region of memory to create and populate the *zone of zones*.

Zone objects are exactly the required size and do not incur the space wastage inherent in power-of-two methods. The garbage collector provides a mechanism for memory reuse—free pages can be returned to the paging system and later recovered for other zones.

The efficiency of the garbage collector is a major concern. Because it runs as a background task, it does not directly impact the performance of individual allocation or release requests. The garbage collection algorithm is slow, as it involves a linear traversal, first of all free objects and then of all pages in the zone. This affects the system responsiveness, for the garbage collector ties up the CPU until it completes.

[Sciv 90] claims that the addition of garbage collection did not significantly change the performance of a parallel compilation benchmark. There are, however, no definitive published measurements of garbage collection overhead, and it is difficult to estimate its impact on the overall system performance. The garbage collection algorithm, however, is complex and inefficient. Compare this with the slab allocator (Section 12.10), which has a simple and fast garbage collection mechanism and also exhibits better worst-case behavior.

12.9 A Hierarchical Allocator for Multiprocessors

Memory allocation for a shared-memory multiprocessor raises some additional concerns. Data structures such as free lists and allocation bitmaps used by traditional systems are not multiprocessor-safe and must be protected by locks. In large, parallel systems, this results in heavy contention for these locks, and CPUs frequently stall while waiting for the locks to be released.

One solution to this problem is implemented in Dynix, a multiprocessor UNIX variant for the Sequent S2000 machines [McKe 93]. It uses a hierarchical allocation scheme that supports the System V programming interface. The Sequent multiprocessors are used in large on-line transaction-processing environments, and the allocator performs well under that load.

Figure 12-9 describes the design of the allocator. The lowest *(per-CPU)* layer allows the fastest operations, while the highest *(coalesce-to-page)* layer is for the time-consuming coalescing process. There is also (not shown) a *coalesce-to-vmblock* layer, which manages page allocation within large (4MB-sized) chunks of memory.

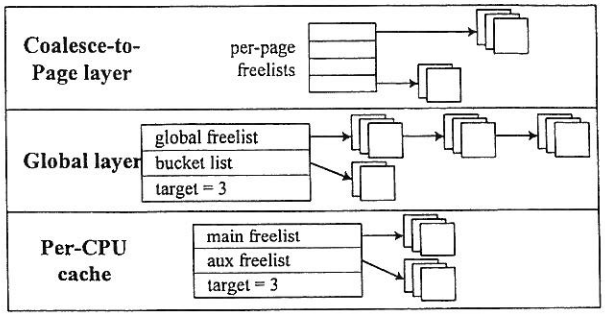

Figure 12-9. A hierarchical allocator for a multiprocessor.

The *per-CPU* layer manages one set of power-of-two pools for each processor. These pools are insulated from the other processors, and hence can be accessed without acquiring global locks. Allocation and release are fast in most cases, as only the local free list is involved.

Whenever the per-CPU free list becomes empty, it can be replenished from the *global layer*, which maintains its own power-of-two pools. Likewise, excess buffers in the per-CPU cache can be returned to the global free list. As an optimization, buffers are moved between these two layers in *target*-sized groups (three buffers per move in the case shown in Figure 12-9), preventing unnecessary linked-list operations.

To accomplish this, the per-CPU layer maintains two free lists—*main* and *aux*. Allocation and release primarily use the main free list. When this becomes empty, the buffers on *aux* are moved to *main*, and the *aux* list is replenished from the global layer. Likewise, when the *main* list overflows (size exceeds *target*), it is moved to *aux*, and the buffers on *aux* are returned to the global layer. This way, the global layer is accessed at most once per *target*-number of accesses. The value of target is a tunable parameter. Increasing *target* reduces the number of global accesses, but ties up more buffers in per-CPU caches.

The global layer maintains global power-of-two free lists, and each list is subdivided into groups of *target* buffers. Occasionally, it is necessary to transfer odd-sized groups of blocks to the global layer, due to low-memory operations or per-CPU cache flushes. Such blocks are added to a separate bucket list, which serves as a staging area for the global free list.

When a global list exceeds a global target value, excess buffers are returned to the *coalesce-to-page layer*. This layer maintains per-page free lists (all buffers from the page are the same size). This layer places the buffers on the free list to which they belong, and increases the free count for that page. When all buffers on a page are returned to this list, the page can be given back to the paging system. Conversely, the coalesce-to-page layer can borrow memory from the paging system to create new buffers.

The coalesce-to-page layer sorts its lists based on the number of free blocks on each page. This way, it allocates buffers from pages having the fewest free blocks. Pages with many free blocks

get more time to recover other free blocks, increasing the probability of returning them to the paging system. This results in a high coalescing efficiency.

12.9.1 Analysis

The Dynix algorithm provides efficient memory allocation for shared memory multiprocessors. It supports the standard System V interface, and allows memory to be exchanged between the allocator and the paging system. The per-CPU caches reduce the contention on the global lock, and the dual free lists provide a fast exchange of buffers between the per-CPU and global layers.

It is interesting to contrast the Dynix coalescing approach with that of the Mach zone-based allocator. The Mach algorithm employs a mark-and-sweep method, linearly scanning the entire pool each time. This is computationally expensive, and hence is relegated to a separate background task. In Dynix, each time blocks are released to the coalesce-to-page layer, the per-page data structures are updated to account for them. When all the buffers in a page are freed, the page can be returned to the paging system. This happens in the foreground, as part of the processing of release operations. The incremental cost for each release operation is small; hence it does not lead to unbounded worst-case performance.

Benchmark results [McKe 93] show that for a single CPU, the Dynix algorithm is faster than the McKusick-Karels algorithm by a factor of three to five. The improvement is even greater for multiprocessors (a hundred to a thousand-fold for 25 processors). These comparisons, however, are for the best-case scenario, where allocations occur from the per-CPU cache. This study does not describe more general measurements.

12.10 The Solaris 2.4 *Slab* Allocator

The slab allocator [Bonw 94], introduced in Solaris 2.4, addresses many performance problems that are ignored by the other allocators described in this chapter. As a result, the slab allocator delivers better performance and memory utilization than other implementations. Its design focuses on three main issues—object reuse, hardware cache utilization, and allocator footprint.

12.10.1 Object Reuse

The kernel uses the allocator to create various kinds of temporary objects, such as inodes, `proc` structures, and network buffers. The allocator must execute the following sequence of operations on an object:

1. Allocate memory.
2. Construct (initialize) the object.
3. Use the object.
4. Deconstruct it.
5. Free the memory.

Kernel objects are usually complex, and contain sub-objects such as reference counts, linked list headers, mutexes, and condition variables. Object construction involves setting these fields to a fixed, *initial* state. The deconstruction phase deals with the same fields, and in many cases, leaves them in their initial state before deallocating the memory.

For instance, a vnode contains the header of a linked list of its resident pages. When the vnode is initialized, this list is empty. In many UNIX implementations [Bark 90], the kernel deallocates the vnode only after all its pages have been flushed from memory. Hence, just before freeing the vnode (the deconstruction stage), its linked list is empty again.

If the kernel reuses the same object for another vnode, it does not need to reinitialize the linked list header, for the deconstruction took care of that. The same principle applies to other initialized fields. For instance, the kernel allocates objects with an initial reference count of one, and deallocates them when the last reference is released (hence, the reference count is one, and is about to become zero). Mutexes are initialized to an *unlocked* state, and must be unlocked before releasing the object.

This shows the advantage of caching and reusing the same object, rather than allocating and initializing arbitrary chunks of memory. Object caches are also space-efficient, as we avoid the typical rounding to the next power of two. The zone allocator (Section 12.8) is also based on object caching and gives efficient memory utilization. However, because it is not concerned with the object state, it does not eliminate the reinitialization overhead.

12.10.2 Hardware Cache Utilization

Traditional memory allocators create a subtle but significant problem with hardware cache utilization. Many processors have a small, simple, level-1 data cache whose size is a power of two. (Section 15.13 explains hardware caches in detail.) The MMU maps addresses to cache locations by

```
cache location = address % cache size;
```

When the hardware references an address, it first checks the cache location to see if the data is in the cache. If it is not, the hardware fetches the data from main memory into the cache, overwriting the previous contents of that cache location.

Typical memory allocators such as the McKusick-Karels and the buddy algorithms round memory requests to the next power of two and return objects aligned to that size. Moreover, most kernel objects have their important, frequently accessed fields at the beginning of the object.

The combined effect of these two factors is dramatic. For instance, consider an implementation where the in-core inode is about 300 bytes, the first 48 bytes of which are frequently accessed. The kernel allocates a 512-byte buffer aligned at a 512-byte boundary. Of these 512 bytes, only 48 bytes (9%) are frequently used.

As a result, parts of the hardware cache that are close to the 512-byte boundary suffer serious contention, whereas the rest of the cache is underutilized. In this case, inodes can utilize only 9% of the cache. Other objects exhibit similar behavior. This anomaly in buffer address distribution results in inefficient use of the hardware cache and, hence, poor memory performance.

The problem is worse for machines that interleave memory access across multiple main buses. For instance, the SPARCcenter 2000 [Cekl 92] interleaves data in 256-byte stripes across two buses. For the above example of inode use, most of the accesses involve bus 0, resulting in unbalanced bus use.

12.10.3 Allocator Footprint

The footprint of an allocator is the portion of the hardware cache and the *translation lookaside buffer (TLB)* that is overwritten by the allocation itself. (Section 13.3.1 explains TLBs in detail.) Memory allocators using resource maps or buddy algorithms must examine several objects to find a suitable buffer. These objects are distributed in many different parts of memory, often far from each other. This causes many cache and TLB misses, reducing the performance of the allocator. The impact is even greater, because the allocator's memory accesses overwrite *hot* (active) cache and TLB entries, requiring them to be fetched from main memory again.

Allocators such as McKusick-Karels and zone have a small footprint, since the allocator determines the correct pool by a simple computation and merely removes a buffer from the appropriate free list. The slab allocator uses the same principles to control its footprint.

12.10.4 Design and Interfaces

The slab allocator is a variant of the zone method and is organized as a collection of object caches. Each cache contains objects of a single type; hence, there is one cache of vnodes, one of `proc` structures, and so on. Normally, the kernel allocates objects from, and releases them to, their respective caches. The allocator also provides mechanisms to give more memory to a cache, or recover excess memory from it.

Conceptually, each cache is divided into two parts—a front end and a back end (Figure 12-10). The front end interacts with the memory client. The client obtains constructed objects from the cache and returns deconstructed objects to it. The back end interacts with the page-level allocator, exchanging slabs of unconstructed memory with it as the system usage patterns change.

A kernel subsystem initializes a cache to manage objects of a particular type, by calling

```
cachep = kmem_cache_create (name, size, align, ctor, dtor);
```

where `name` is a character string describing the object, `size` is the size of the object in bytes, `align` is the alignment required by the objects, and `ctor` and `dtor` are pointers to functions that construct and deconstruct the object respectively. The function returns a pointer to the cache for that object.

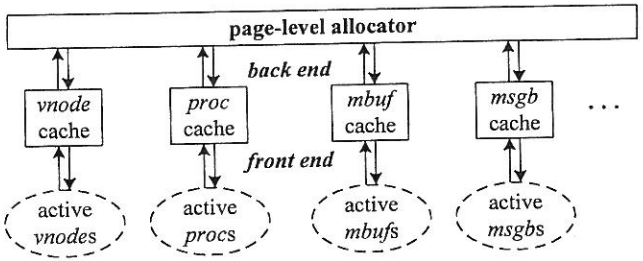

Figure 12-10. Slab allocator design.

Thereafter, the kernel allocates an object from the cache by calling

```
objp = kmem_cache_alloc (cachep, flags);
```

and releases it with

```
kmem_cache_free (cachep, objp);
```

This interface does not construct or deconstruct objects when reusing them. Hence the kernel must restore the object to its *initial* state before releasing it. As explained in Section 12.10.1, this usually happens automatically and does not require additional actions.

When the cache is empty, it calls `kmem_cache_grow()` to acquire a *slab* of memory from the page-level allocator and create objects from it. The slab is composed of several contiguous pages managed as a monolithic chunk by the cache. It contains enough memory for several instances of the object. The cache uses a small part of the slab to manage the memory in the slab and divides the rest of the slab into buffers that are the same size as the object. Finally, it initializes the objects by calling their constructor (specified in the `ctor` argument to `kmem_cache_create()`), and adds them to the cache.

When the page-level allocator needs to recover memory, it calls `kmem_cache_reap()` on a cache. This function finds a slab whose objects are all free, deconstructs these objects (calling the function specified in the `dtor` argument to `kmem_cache_create()`), and removes the slab from the cache.

12.10.5 Implementation

The slab allocator uses different techniques for large and small objects. We first discuss small objects, many of which can fit into a one-page slab. The allocator divides the slab into three parts—the `kmem_slab` structure, the set of objects, and some unused space (Figure 12-11). The `kmem_slab` structure occupies 32 bytes and resides at the end of the slab. Each object uses an extra four bytes to store a free list pointer. The unused space is the amount left over after creating the maximum possible number of objects from the slab. For instance, if the inode size is 300 bytes, a 4096-byte slab will hold 13 inodes, leaving 104 bytes unused (accounting for the `kmem_slab` structure and the free list pointers). This space is split into two parts, for reasons explained below.

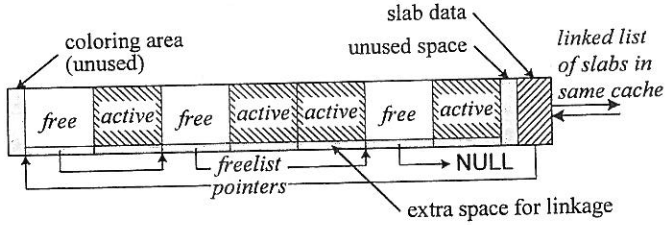

Figure 12-11. Slab organization for small objects.

The kmem_slab structure contains a count of its in-use objects. It also contains pointers to chain it in a doubly linked list of slabs of the same cache, as well as a pointer to the first free object in the slab. Each slab maintains its own, singly linked, free buffer list, storing the linkage information in a four-byte field immediately following the object. This field is needed only for free objects. It must be distinct from the object itself, since we do not want to overwrite the constructed state of the object.

The unused space is split into two parts: a *slab coloring area* at the head of the slab and the rest just before the kmem_slab structure. The cache tries to use a different-sized coloring area in each of its slabs, subject to alignment restrictions. In our inode cache example, if the inodes require an 8-byte alignment, the slabs can have 14 different coloring sizes (0 through 104 in 8-byte increments). This allows a better distribution of the starting offsets of the objects of this class, resulting in more balanced and efficient use of the hardware cache and memory buses.

The kernel allocates an object from a slab by removing the first element from the free list and incrementing the slab's in-use count. When freeing the object, it identifies the slab by a simple computation:

```
slab address = object address % slab size;
```

It then returns the object to the slab's free list, and decrements the in-use count.

When its in-use count becomes zero, the slab is free, or eligible for reclaiming. The cache chains all its slabs on a partly sorted, doubly linked list. It stores fully active slabs (all objects in use) at the beginning, partially active slabs in the middle, and free slabs at the tail. It also maintains a pointer to the first slab that has a free object and satisfies allocations from that slab. Hence the cache does not allocate objects from a completely free slab until all partly active slabs are exhausted. If the page-level allocator must reclaim memory, it checks the slab at the tail of the list and removes the slab if free.

Large Object Slabs

The above implementation is not space-efficient for large objects, which are usually multiples of a page in size. For such objects, the cache allocates slab management data structures from a separate pool of memory (another object cache, of course). In addition to the kmem_slab structure, the cache uses a kmem_bufctl structure for each object in the cache. This structure contains the free list linkage, a pointer to the kmem_slab, and a pointer to the object itself. The slab also maintains a hash table to provide a reverse translation from the object to the kmem_bufctl structure.

12.10.6 Analysis

The slab allocator is a well-designed, powerful facility. It is space-efficient, because its space overhead is limited to the kmem_slab structure, the per-object linkage field, and an unused area no larger than one object per slab. Most requests are serviced extremely quickly by removing an object from the free list and updating the in-use count. Its coloring scheme results in better hardware cache and memory bus utilization, thus improving overall system performance. It also has a small footprint, as it accesses only one slab for most requests.

The garbage collection algorithm is much simpler than that of the zone allocator, which is based on similar principles. The cost of garbage collection is spread over all requests, since each operation changes the in-use count. The actual reclaim operation involves some additional overhead, for it must scan the different caches to find a free slab. The worst-case performance is proportional to the total number of caches, not the number of slabs.

One drawback of the slab allocator is the management overhead inherent in having a separate cache for each type of object. For common classes of objects, where the cache is large and often used, the overhead is insignificant. For small, infrequently used caches, the overhead is often unacceptable. This problem is shared by the Mach zone allocator and is solved by having a set of power-of-two buffers for objects that do not merit a cache of their own.

The slab allocator would benefit from the addition of per-processor caches such as those of Dynix. [Bonw 94] acknowledges this and mentions it as a possible future enhancement.

12.11 Summary

The design of a general-purpose kernel memory allocator raises many important issues. It must be fast, easy to use, and use memory efficiently. We have examined several allocators and analyzed their advantages and drawbacks. The resource map allocator is the only one that permits release of part of the allocated object. Its linear search methods yield unacceptable performance for most applications. The McKusick-Karels allocator has the simplest interface, using the standard malloc() and free() syntax. It has no provision for coalescing buffers or returning excess memory to the page-level allocator. The buddy system constantly coalesces and breaks buffers to adjust to shifting memory demands. Its performance is usually poor, particularly when there is frequent coalescing. The zone allocator is normally fast, but has inefficient garbage collection mechanisms.

The Dynix and slab allocators offer significant improvements over these methods. Dynix uses a power-of-two method, but adds per-processor caches and fast garbage collection (the coalesce-to-page layer). The slab allocator is a modified zone algorithm. It improves performance through object reuse and balanced address distribution. It also uses a simple garbage collection algorithm that bounds the worst-case performance. As was previously noted, adding per-CPU caches to the slab algorithm would provide an excellent allocator.

Table 12-1 summarizes the results of a set of experiments [Bonw 94] comparing the slab allocator with the SVR4 and McKusick-Karels allocators. The experiments also show that object reuse reduces the time required for allocation plus initialization by a factor of 1.3 to 5.1, depending on the object. This benefit is in addition to the improved allocation time noted in the table.

Table 12-1. Performance measurements of popular allocators

	SVR4	McKusick-Karels	slab
Average time for alloc + free (microseconds)	9.4	4.1	3.8
Total fragmentation (waste)	46%	45%	14%
Kenbus benchmark performance (number of scripts executed per minute)	199	205	233

Many of these techniques can also be applied to user-level memory allocators. However, the requirements of user-level allocators are quite different; hence, a good kernel allocator may not work as well at the user level, and vice versa. User-level allocators deal with a very large amount of (virtual) memory, practically limitless for all but the most memory-intensive applications. Hence, coalescing and adjusting to shifting demands are less critical than rapid allocation and deallocation. A simple, standard interface is also extremely important, since they are used by many diverse, independently written applications. [Korn 85] describes several different user-level allocators.

12.12 Exercises

1. In what ways do the requirements for a kernel memory allocator differ from those for a user-level allocator?
2. What is the maximum number of resource map entries required to manage a resource with n items?
3. Write a program that evaluates the memory utilization and performance of a resource map allocator, using a simulated sequence of requests. Use this to compare the first-fit, best-fit, and worst-fit approaches.
4. Implement the free() function for the McKusick-Karels allocator.
5. Write a scavenge() routine that coalesces free pages in the McKusick-Karels allocator and releases them to the page-level allocator.
6. Implement a simple buddy algorithm that manages a 1024-byte area of memory with a minimum allocation size of 16 bytes.
7. Determine a sequence of requests that would cause the worst-case behavior for the simple buddy algorithm.
8. In the SVR4 lazy buddy algorithm described in Section 12.7, how would each of following events change the values of N, A, L, G, and *slack?*
   (a) A buffer is released when *slack* is greater than 2.
   (b) A delayed buffer is reallocated.
   (c) A non-delayed buffer is allocated (there are no delayed buffers).
   (d) A buffer is released when *slack* equals 1, but none of the free buffers can be coalesced because their buddies are not free.
   (e) A buffer is coalesced with its buddy.
9. Which of the other memory allocators can be modified to have a Dynix-style per-CPU free list in case of multiprocessors? Which algorithms cannot adopt this technique? Why?
10. Why does the slab allocator use different implementations for large and small objects?
11. Which of the allocators described in this chapter have simple programming interfaces?
12. Which of the allocators allow a client to release part of an allocated block?
13. Which of the allocators can reject an allocation request even if the kernel has a block of memory large enough to satisfy the request?

12.13 References

[Bach 86] Bach, M.J., *The Design of the UNIX Operating System,* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Bark 89] Barkley, R.E., and Lee, T.P., "A Lazy Buddy System Bound By Two Coalescing Delays per Class," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles,* Dec. 1989, pp. 167–176.

[Bark 90] Barkley, R.E., and Lee, T.P., "A Dynamic File System Inode Allocation and Reclaim Policy," *Proceedings of the Winter 1990 USENIX Technical Conference,* Jan. 1990, pp. 1–9.

[Bonw 94] Bonwick, J., "The Slab Allocator: An Object-Caching Kernel Memory Allocator," *Proceedings of the Summer 1994 USENIX Technical Conference,* Jun. 1994, pp. 87–98.

[Cekl 92] Cekleov, M., Frailong, J.-M., and Sindhu, P., *Sun–4D Architecture,* Revision 1.4, Sun Microsystems, 1992.

[DEC 92] Digital Equipment Corporation, *Alpha Architecture Handbook,* Digital Press, 1992.

[DEC 93] Digital Equipment Corporation, *DEC OSF/1 Internals Overview—Student Workbook,* 1993.

[Knut 73] Knuth, D., *The Art of Computer Programming, Vol. 1, Fundamental Algorithms,* Addison-Wesley, Reading, MA, 1973.

[Korn 85] Korn, D.G., and Vo, K.-P., "In Search of a Better Malloc," *Proceedings of the Summer 1985 USENIX Technical Conference,* Jun. 1985, pp. 489–505.

[Lee 89] Lee, T.P., and Barkley, R.E., "A Watermark-Based Lazy Buddy System for Kernel Memory Allocation," *Proceedings of the Summer 1989 USENIX Technical Conference,* Jun. 1989, pp. 1–13.

[McKe 93] McKenney, P.E., and Slingwine, J., "Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors," *Proceedings of the Winter 1993 USENIX Technical Conference,* Jan. 1993, pp. 295–305.

[McKu 88] McKusick, M.K., and Karels, M.J., "Design of a General-Purpose Memory Allocator for the 4.3BSD UNIX Kernel," *Proceedings of the Summer 1988 USENIX Technical Conference,* Jun. 1988, pp. 295–303.

[Pete 77] Peterson, J.L., and Norman, T.A., "Buddy Systems," *Communications of the ACM,* Vol. 20, No. 6, Jun. 1977, pp. 421–431.

[Sciv 90] Sciver, J.V., and Rashid, R.F., "Zone Garbage Collection," *Proceedings of the USENIX Mach Workshop,* Oct. 1990, pp. 1–15.

[Step 83] Stephenson, C.J., "Fast Fits: New Methods for Dynamic Storage Allocation," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles,* Vol. 17, no. 5, 1983, pp. 30–32.