ALLOCATING KERNEL MEMORY

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel. This list is typically populated using a page-replacement algorithm and most likely contains free pages scattered throughout physical memory. If a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes.

There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.

2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory —without the benefit of a virtual memory interface —and consequently may require memory residing in physically contiguous pages.

In the following sections, we examine two strategies for managing free memory that is assigned to kernel processes: the “buddy system” and slab allocation.

1. Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

Let’s consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into

two **buddies**—which we will call *AL* and *AR* —each 128 KB in size. One of these buddies is further divided into two 64-KB buddies —
BL and *BR*. However, the next-highest power of 2 from 21 KB is 32 KB so either
BL or *BR* is again divided into two 32-KB buddies, *CL* and *CR*. One of these buddies is used to satisfy the 21-KB request. This scheme is illustrated in Figure 9.26, where *CL* is the segment allocated to the 21-KB request.

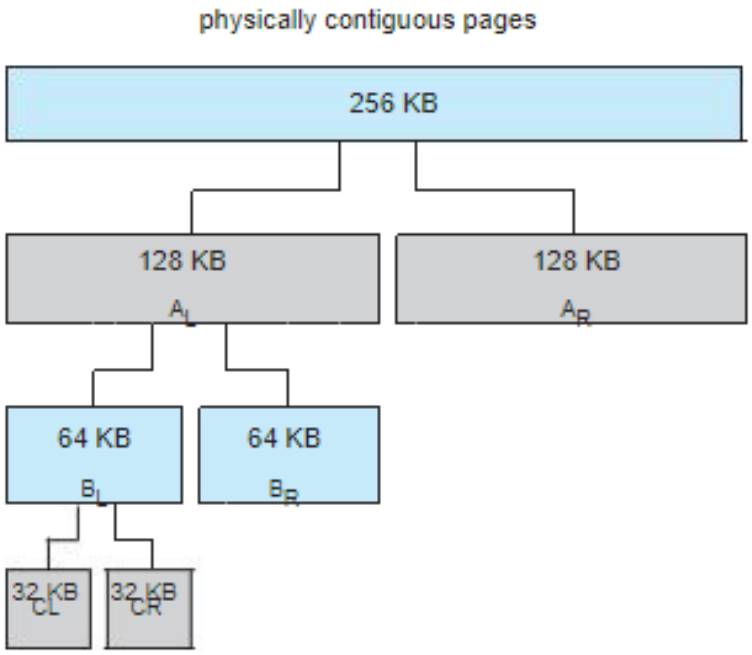

Fig : Buddy System Allocation

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as coalescing. In the above Figure for example, when the kernel releases the *CL* unit it was allocated, the system can coalesce *CL* and *CR* into a 64-KB segment.

This segment, *BL* , can in turn be coalesced with its buddy *BR* to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64- KB segment.

In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation. In the following section, we explore a memory allocation scheme where no space is lost due to fragmentation.

2. Slab Allocation

A second strategy for allocating kernel memory is known as slab allocation. A slab is made up of one or more physically contiguous pages. A cache consists of one or more slabs. There is a single cache for each unique kernel data structure —for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth.

Each cache is populated with objects that are instantiations of the kernel data structure the cache represents.

For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth. The relationship among slabs, caches, and objects is shown in Figure

The figure shows two kernel objects 3 KB in size and three objects 7 KB in size, each stored in a separate cache.

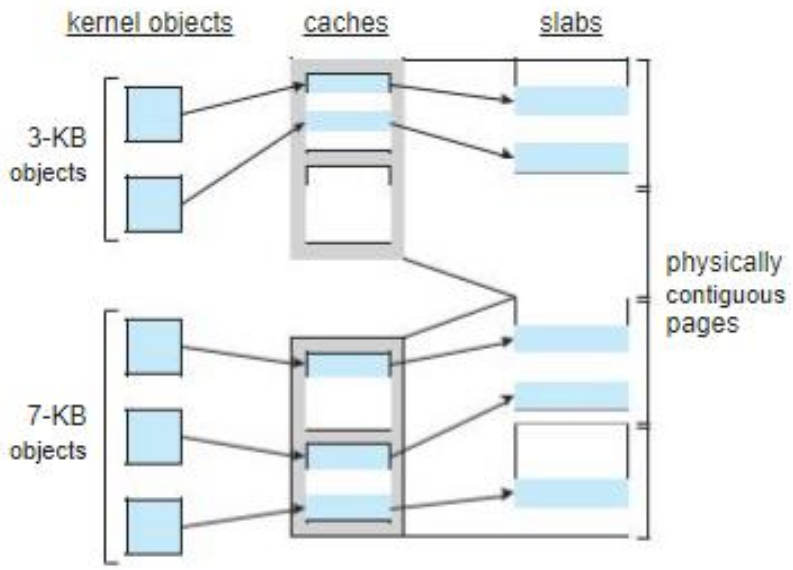

Fig : Slab Alocation

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects which are initially marked as free are allocated to the cache. The number of objects in the cache depends on the size of the associated slab.

For example, a 12-KB slab (made up of three continguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object

for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type struct task struct, which requires approximately 1.7 KB of memory.

When the Linux kernel creates a new task, it requests the necessary memory for the struct task struct object from its cache. The cache will fulfill the request using a struct task struct object that has already been allocated in a slab and is marked as free.
In Linux, a slab may be in one of three possible states:

1. Full. All objects in the slab are marked as used.
2. Empty. All objects in the slab are marked as free.
3. Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented.

Thus, when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.

2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel. The act of allocating —and releasing — memory can be a time-consuming process.

However, objects are created in advance and thus can be quickly allocated from the cache. Furthermore, when the kernel has finished with an object and releases it, it is

marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

The slab allocator first appeared in the Solaris 2.4 kernel. Because of its general-purpose nature, this allocator is now also used for certain user-mode memory requests in Solaris. Linux originally used the buddy system; however, beginning with Version 2.2, the Linux kernel adopted the slab allocator.

Recent distributions of Linux now include two other kernel memory allocators — the SLOB and SLUB allocators. (Linux refers to its slab implementation as SLAB.)

The SLOB allocator is designed for systems with a limited amount of memory, such as embedded systems. SLOB (which stands for Simple List of Blocks) works by maintaining three lists of objects: ***small*** (for objects less than 256 bytes), ***medium*** (for objects less than 1,024 bytes), and ***large*** (for objects less than 1,024 bytes). Memory requests are allocated from an object on an appropriately sized list using a first-fit policy.

Beginning with Version 2.6.24, the SLUB allocator replaced SLAB as the default allocator for the Linux kernel. SLUB addresses performance issues with slab allocation by reducing much of the overhead required by the SLAB allocator.

One change is to move the metadata that is stored with each slab under SLAB allocation to the page structure the Linux kernel uses for each page. Additionally, SLUB removes the per-CPU queues that the SLAB allocator maintains for objects in each cache.

For systems with a large number of processors, the amount of memory allocated to these queues was not insignificant. Thus, SLUB provides better performance as the number of processors on a system increases.