Slab Allocation

Researcher: Mobin Kheibary

Student No: [994421017]

Fall – 2022

Abstract:

Slab allocation is a memory management mechanism intended for the efficient memory allocation of objects. In comparison with earlier mechanisms, it reduces fragmentation caused by allocations and deallocations. This technique is used for retaining allocated memory containing a data object of a certain type for reuse upon subsequent allocations of objects of the same type. It is analogous to an object pool, but only applies to memory, not other resources.

Slab allocation was first introduced in the Solaris 2.4 kernel by Jeff Bonwick. It is now widely used by many Unix and Unix-like operating systems including FreeBSD and Linux.

Basis

Slab allocation renders infrequent the very costly practice (in CPU time) of initialization and destruction of kernel data-objects, which can outweigh the cost of allocating memory for them. When the kernel creates and deletes objects often, overhead costs of initialization can result in significant performance drops. Object caching leads to less frequent invocation of functions which initialize object state: when a slab-allocated object is released after use, the slab allocation system typically keeps it cached (rather than doing the work of destroying it) ready for re-use next time an object of that type is needed (thus avoiding the work of constructing and initializing a new object).

With slab allocation, a cache for a certain type or size of data object has a number of pre-allocated "slabs" of memory; within each slab there are memory chunks of fixed size suitable for the objects. The slab allocator keeps track of these chunks, so that when it receives a request to allocate memory for a data object of a certain type, usually it can satisfy the request with a free slot (chunk) from an existing slab. When the allocator is asked to free the object's memory, it just adds the slot to the containing slab's list of free (unused) slots. The next call to create an object of the same type (or allocate memory of the same size) will return that memory slot (or some other free slot) and remove it from the list of free slots. This process eliminates the need to search for suitable memory space and greatly alleviates memory fragmentation. In this context, a slab is one or more contiguous pages in the memory containing pre-allocated memory chunks.

Implementation

Understanding the slab allocation algorithm requires defining and explaining some terms:

1. **Cache**: cache represents a small amount of very fast memory. A cache is a storage for a specific type of object, such as semaphores, process descriptors, file objects, etc.
2. **Slab**: slab represents a contiguous piece of memory, usually made of several physically contiguous pages. The slab is the actual container of data associated with objects of the specific kind of the containing cache.

When a program sets up a cache, it allocates a number of objects to the slabs associated with that cache. This number depends on the size of the associated slabs.

Slabs may exist in one of the following states:

1. *empty* – all objects on a slab marked as free
2. *partial* – slab consists of both used and free objects
3. *full* – all objects on a slab marked as used

Initially, the system marks each slab as "empty". When the process calls for a new kernel object, the system tries to find a free location for that object on a partial slab in a cache for that type of object. If no such location exists, the system allocates a new slab from contiguous physical pages and assigns it to a cache. The new object gets allocated from this slab, and its location becomes marked as "partial".

The allocation takes place quickly, because the system builds the objects in advance and readily allocates them from a slab.

Slabs

A slab is the amount by which a cache can grow or shrink. It represents one memory allocation to the cache from the machine, and whose size is customarily a multiple of the page size. A slab must contain a list of free buffers (or bufctls), as well as a list of the bufctls that have been allocated (in the case of a large slab size).

Large slabs

These are for caches that store objects that are at least 1/8 of the page size for a given machine. The reason for the large slabs having a different layout from the small slabs is that it allows large slabs to pack better into page-size units, which helps with fragmentation. The slab contains a list of bufctls, which are simply controllers for each buffer that can be allocated (a buffer is the memory that the user of a slab allocator would use).

Small slabs

The small slabs contain objects that are less than 1/8 of the page size for a given machine. These small slabs need to be optimized further from the logical layout, by avoiding using bufctls (which would be just as large as the data itself and cause memory usage to be much greater). A small slab is exactly one page, and has a defined structure that allows bufctls to be avoided. The last part of the page contains the 'slab header', which is the information needed to retain the slab. Starting at the first address of that page, there are as many buffers as can be allocated without running into the slab header at the end of the page.

Instead of using bufctls, we use the buffers themselves to retain the free list links. This allows the small slab's bufctl to be bypassed.

Systems using slab allocation

- AmigaOS (introduced in AmigaOS 4)
- DragonFly BSD (introduced in release 1.0)
- FreeBSD (introduced in 5.0)

- GNU Mach
- Haiku (introduced in alpha 2)
- Horizon (Nintendo Switch microkernel)
- HP-UX (introduced in 11i)
- Linux (introduced in kernel 2.2, it's now one of three memory allocator implementations together with SLOB and SLUB. The three allocators provides a kind of front-end to the zoned buddy allocator for those sections of the kernel that require more flexible memory allocation than the standard 4KB page size).
- NetBSD (introduced in 4.0)
- Solaris (introduced in 2.4)
- The Perl 5 compiler uses a slab allocator for internal memory management
- Memcached uses slab allocation for memory management
- illumos

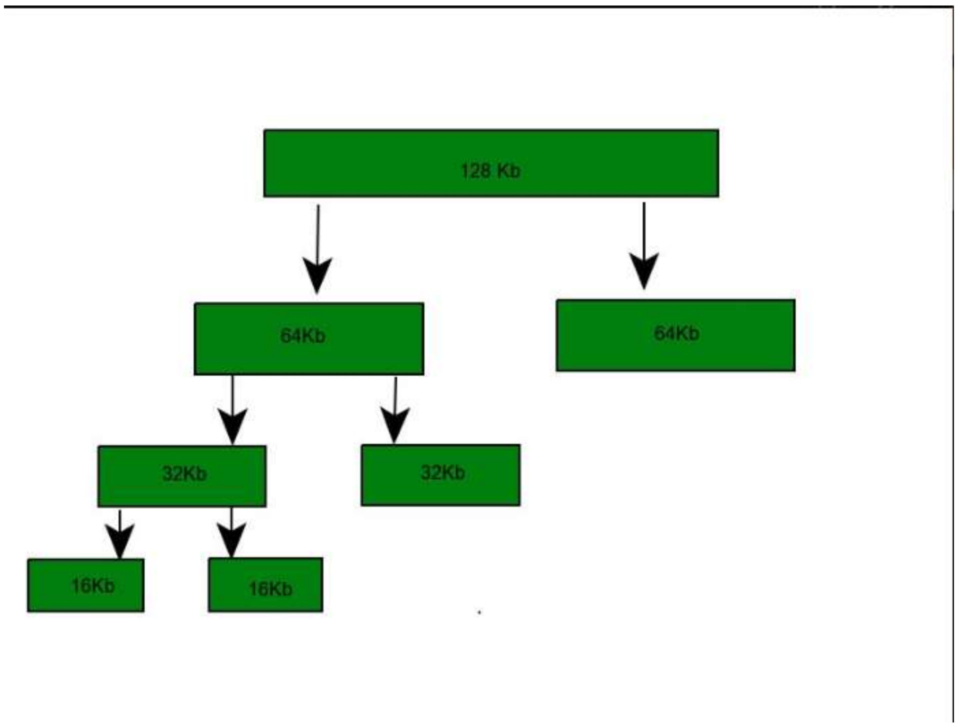
Allocating kernel memory (Buddy system and Slab system)

Two strategies for managing free memory that is assigned to kernel processes:

1. Buddy system –

Buddy allocation system is an algorithm in which a larger memory block is divided into small parts to satisfy the request. This algorithm is used to give best fit. The two smaller parts of block are of equal size and called as buddies. In the same manner one of the two buddies will further divide into smaller parts until the request is fulfilled. Benefit of this technique is that the two buddies can combine to form the block of larger size according to the memory request.

Example – If the request of 25Kb is made then block of size 32Kb is allocated.



Four Types of Buddy System –

1. Binary buddy system
2. Fibonacci buddy system
3. Weighted buddy system
4. Tertiary buddy system

Why buddy system?
If the partition size and process size are different then poor match occurs and may use space inefficiently.
It is easy to implement and efficient then dynamic allocation.

Binary buddy system –
The buddy system maintains a list of the free blocks of each size (called a free list), so that it is easy to find a block of the desired size, if one is available. If no block of the requested size is available, allocate searches for the first non-empty list for blocks of at least the size requested. In either case, a block is removed from the free list.

Example – Assume the size of memory segment is initially 256kb and the kernel requests 25kb of memory. The segment is initially divided into two buddies. Let we call A1 and A2 each 128kb in size. One of these buddies is further divided into two 64kb buddies let say B1 and B2. But the next highest power of 25kb is 32kb so, either B1 or B2 is further divided into two 32kb buddies(C1 and C2) and finally one of these buddies is used to satisfy the 25kb request. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from.

Fibonacci buddy system –
This is the system in which blocks are divided into sizes which are Fibonacci numbers. It satisfies the following relation:

$Z_i = Z_{(i-1)}+Z_{(i-2)}$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 144, 233, 377, 610. The address calculation for the binary and weighted buddy systems is straight forward, but the original procedure for the Fibonacci buddy system was either limited to a small, fixed number of block sizes or a time-consuming computation.

Advantages –

- In comparison to other simpler techniques such as dynamic allocation, the buddy memory system has little external fragmentation.
- The buddy memory allocation system is implemented with the use of a binary tree to represent used or unused split memory blocks.
- The buddy system is very fast to allocate or deallocate memory.
- In buddy systems, the cost to allocate and free a block of memory is low compared to that of best-fit or first-fit algorithms.
- Other advantage is coalescing.
- Address calculation is easy.

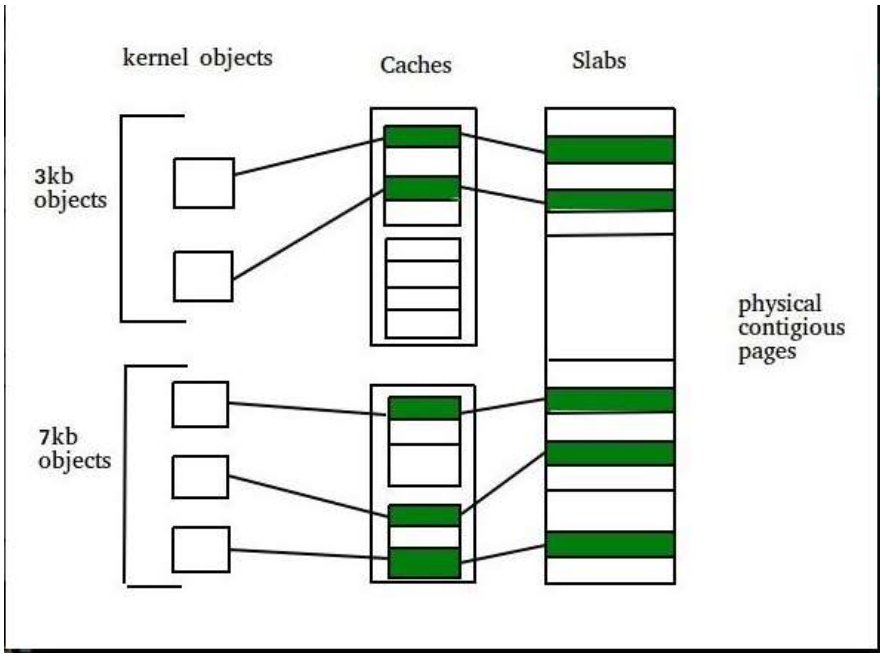
What is coalescing?
It is defined as how quickly adjacent buddies can be combined to form larger segments this is known as coalescing.
For example, when the kernel releases the C1 unit it was allocated, the system can coalesce C1 and C2 into a 64kb segment. This segment B1 can in turn be coalesced with its buddy B2 to form a 128kb segment. Ultimately, we can end up with the original 256kb segment.

Drawback –
The main drawback in buddy system is internal fragmentation as larger block of memory is acquired then required. For example, if a 36 kb request is made then it can only be satisfied by 64 kb segment and remaining memory is wasted.

2. Slab Allocation –

A second strategy for allocating kernel memory is known as slab allocation. It eliminates fragmentation caused by allocations and deallocations. This method is used to retain allocated memory that contains a data object of a certain type for reuse upon subsequent allocations of objects of the same type. In slab allocation memory chunks suitable to fit data objects of certain type or size are preallocated. Cache does not free the space immediately after use although it keeps track of data which are required frequently so that whenever request is made the data will reach very fast. Two terms required are:

- **Slab –** A slab is made up of one or more physically contiguous pages. The slab is the actual container of data associated with objects of the specific kind of the containing cache.
- **Cache –** Cache represents a small amount of very fast memory. A cache consists of one or more slabs. There is a single cache for each unique kernel data structure.



Example –

- A separate cache for a data structure representing processes descriptors
- Separate cache for file objects
- Separate cache for semaphores etc.

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.

Implementation –
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 size of the associated slab. *Example* – A 12 kb slab (made up of three contiguous 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.

In Linux, a slab may 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

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.

Benefits of slab allocator –

- No memory is wasted due to fragmentation because each unique kernel data structure has an associated cache.
- Memory request can be satisfied quickly.
- The slab allocating scheme is particularly effective for managing when objects are frequently allocated or deallocated. 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. When the kernel has finished with an object and releases it, it is marked as free and return to its cache, thus making it immediately available for subsequent request from the kernel.

Main Resources:

i. https://en.wikipedia.org/wiki/Slab_allocation
ii. https://www.geeksforgeeks.org/operating-system-allocating-kernel-memory-buddy-system-slab-system/
iii. https://www.kernel.org/doc/gorman/html/understand/understand011.html

Special Thanks to Dr. Sharifi for all her efforts.

The End.