OS Lab 1 Report: Physical Memory Management
ZHL
4/7/2025

Overview:

In this lab, we need to create a physical memory allocator for the Abstract Machine, a hardware-abstraction-layer framework provided by the instructor. Key constraints from the description are: 1) the allocator must work under the SMP configuration, 2) the largest request would be 16 MB, 3) most requests would not exceed 128 bytes, and 4) all data structures used to manage the memory should be dynamically allocated.

Design and Implementation:

To reduce internal fragmentation and take the advantage of multi-core systems, the single memory allocator actually requires three “mini-allocators”. All of them were used in the Linux-2.4.22 kernel.

The first one is the binary buddy allocator [1]. Proposed by Knowlton in 1965, it is designed to manage large memory chunks in orders. Each order contains a number of blocks, and each block has a power of two pages. For example, each order-12 block consists of 4096 pages. One doubly-linked list (free_area) is used to track the free blocks at every order; a bitmap is used to track the state of a pair of blocks (buddies) at each order. The bitmap is the workhorse of the entire system, responsible for splitting and merging the blocks correctly.

Due to constraint (1), I designed my buddy system to have 13 orders (starting from order 0 to order 12). Because 16 MB is exactly 4096 pages, an order-12 block is sufficient to handle the largest request.

In addition, it is known to us where the physical memory we manage starts (0x300000) and ends (0x8000000). Therefore, I split the region into two smaller ones: the first is from 0x300000 to 0x1000000 and the second is from 0x1000000 to 0x8000000. The reason behind is to initialize the management metadata. Yes, all allocations have to be dynamic, according to the lab description. But we still need to create the core data structures at boot time before mpe_init().

The second component is the slab allocator [2]. It was proposed by Bonwick in 1994. The command line “sudo cat /proc/slabinfo” helped to show how it works. It works by allocating small memory chunks from dedicated caches named size-cache. For instance, a “cache-16” exists in today’s Linux memory management system to just handle requests between 9 to 16 bytes. This way reduces the chance of internal fragmentation. Each cache can have a number of slabs. Each slab takes care of exactly one physical page and fills with a list of pointers of the same size. There are 10 caches in my design, from 4 to 2048.

In this part, there are three main problems I solved. One is the allocation of slabs metadata. At first, like what I did for the buddy system, I allocated the slabs using bootmem_alloc(), placing them in the smaller region. Soon I realized the slabs grow together with the number of requests and the boot memory region is not big enough for this dynamic allocation.

The second problem is the timing to recycle the free slabs in a free slab list, returning the pages to the buddy allocator by calling pgfree(). To achieve this, I set a limit for the number of free slabs each cache can have. Based on the constraint (3), I designed the limit for caches of size less than or equal to 128 bytes to be 8, and the rest to be 4. Once the last

object inside a slab is going to be freed and the slab will be moved from the partial list to the free list, the check will decide whether to trigger the “garbage collection” or not.

The third component is the SMP local caches [3]. Every core will have its own array of objects in each cache. The allocation and free calls must begin at the local level. The performance benefit of enabling local cache is increased not only by the usage of multi-threading, but also by reducing the cost to acquire and release the lock for accessing the global cache.

In this part, the main problem I solved is the double-free issue. To avoid this by halting the program accordingly, I set the pointer to NULL after allocation and check if the object going to be freed is in the cache by comparing the pointers.

Challenge:

I spent countless hours thinking about the “chicken and egg problem”. At first, I was struggling at how to get the memory I need for the metadata. Conceptually, my mind still lives in the user space as I was trying to use “malloc” from klib (which is designed for user programs running on the Abstract Machine) to allocate space for my data structures at the physical memory level.

The second challenge is the alignment and bitwise operations, the buddy allocators heavily rely on the bitmap operations to get the index of the block, the pair, and the buddy.

The third challenge is to just combine everything and make it work correctly. Currently, the allocator will crash when freeing objects through both slab and buddy allocator under multiple CPUs, for example, when two threads running the workload below:

```
void *ptr1 = pmm->alloc(4096);
void *ptr2 = pmm->alloc(SIZE);
pmm->free(ptr1);
```

[1] Knowlton, Kenneth C. “A Fast Storage Allocator.” *Communications of the ACM*, vol. 8, no. 10, Oct. 1965, pp. 623–624, https://doi.org/10.1145/365628.365655. Accessed 7 Apr. 2025.

[2] Bonwick, Jeff. “The Slab Allocator: An Object-Caching Kernel Memory Allocator.” *USENIX Summer Technical Conference*, 6 June 1994, pp. 6–6. Accessed 7 Apr. 2025.

[3] Dowd, Kevin, and Charles R Severance. *High Performance Computing*. O’Reilly Media, 1998.