Cottontail Memory Management: A System for Allocation, Deallocation, and Accounting

By: Frank Millea

The Cottontail memory management system tracks memory address space usage for a flat-mode address space. It is somewhat platform-dependent, as it is designed for an Intel(r)-style system with a 32-bit total address space of 4GB, taking advantage of the x86 processor's paging feature. This means there are approximately 4.3 billion individual addressable byte locations within this address space (4GB), broken down into approximately 1.05 million (1M) logical page locations of 4096 (4KB) bytes each. The only information one needs to know when allocating address space is whether or not a page of address space is allocated. This shall be represented by a 1 for those in use, and a 0 for those free for allocation. The size of the bitmap needed to store this information is calculated as follows:

1 bit per page x 2^20 pages x 1 byte per 8 bits = 131,072 bytes, or 128KB

From a programming standpoint, a linear search of this bitmap would be ungodfully slow, and this would occur each time memory would need to be allocated. The following C snippet represents a search for a free page:

```
unsigned char page_usage_bitmap[131072];
int i=0;
while(page_usage_bitmap[i / 8] & (1 << (i % 8)))
{
        i++;
}
```

For the sake of argument, let's say that the first 4MB in the address space are allocated. This means the first 1,024 pages are allocated, and in order to find a free page the program must search through the preceding 1,024 until #1,025 reveals that it is free. 4MB isn't a lot for an operating system. Let's say Half-Life is running and has allocated 200MB of address space. I say address space because not all of the data may be in physical memory, or RAM, but may have been swapped out to disk. Nevertheless, it still takes up address space. Remember that that is what we are allocating here, not actual RAM. But back to Half-Life. With 200MB allocated, the program would need to search through 50,000 pages before finding a free page! This may be a little more or a little less, depending on where the memory was allocated in the address space, but in general it should be fairly contiguous. This is 50,000 logical operations of the loop, but how long would it take in real time? After compilation to assembly with GCC, analysis reveals about 20 machine language instructions are being performed per iteration. 20 x 50,000 = 1,000,000 machine language instructions per memory allocation. With an average of 2 clock cycles per instruction, that's 2 million clock cycles per memory allocation. On a 500MHz computer with 500 million clock cycles per second, only 250 memory allocations would be possible per second, and that's if all the computer did was allocate memory! Memory allocation is a very low-level function used extensively by higher-level procedures such as those that read data in from the hard drive. Suffice to say that memory allocation must be as close to lightning-fast as possible. The above implementation is certainly not good performance for a decent operating system. A few simple modifications and tweaks will make the algorithm much faster. First, take into account the fact that the x86 is a 32-bit processor and thus takes the same speed to read a byte (8 bits) from memory as a double word (32 bits). Let us then change our bitmap from a 128K bytes to 128/4 = 32K double words. Let us then observe that a double word describing 32 pages which are all used would be equal to 0xFFFFFFFF in hex, or -1. Rather than checking each single bit, now only each double word needs to be checked, reducing our number of iterations

by a factor of 32. Once one is found with a free page, determining the free page's offset within the double word is a trivial matter. Here is the modified code, which will perform only 1,563 iterations to find a free page:

```
unsigned long page_usage_bitmap[32768];
int i=0;
while(page_usage_bitmap[i] == 0xFFFFFFFF)
{
        i++;
}
```

Due to the more simplistic algorithm, the number of machine instructions needed to describe it are reduced further, to a mere 8. 8 x 1,563 = 12,504 instructions x 2 clock cycles per instruction = about 25,000 clock cycles per memory allocation, a far cry from the 2 million in the first algorithm. But this is still too slow. What if there was even more memory allocated already? What if the operating system needed to allocate memory thousands of times per second over and over? These are some very realistic considerations for the designer of a successful high-demand high-performance OS, and therefore more optimization is needed. Divide the address space into 1,024 4MB "superpages" and create a second bitmap which tracks only 1 simple fact: if the superpage has at least one free page (4KB) in it or not. The size of this bitmap will be 1,024 superpages x 1 bit per superpage x 1 byte per 8 bits = 128 bytes total, a nominal sum. With the 32-bit optimization applied, this results in an array of a mere 32 doublewords. By searching this array first, one may skip over a massive amount of pages at a time. Let's go back to our example of 200MB allocated by Half-Life. Since each of the 32 superpage doublewords describes 128MB of address space, this entire 200MB would fall into the first two superpage doublewords. The first 50 superpages would then be automatically skipped over for linear searching, and the 51st would have at least some free pages. The number of iterations to determine a free page then would be 32, since there are 1,024 pages per superpage, and this is a worst-case scenario! This yields 32 iterations x 8 instructions x 2 clock cycles = 512 clock cycles per memory allocation! But what if one needs to allocate more than just one page at a time, say 64? All you would then need to do is use the above algorithm to find two consecutive doublewords of 0, or a doubleword of 0 with a certain number of free pages before and after it. This would be slightly more complicated and may take longer, say 2,000 clock cycles, but still is incredibly fast considering even an ancient 80486 processor has 50,000 clock cycles in a millisecond. Once a memory allocation is complete, it needs to A.) Update the page bitmap with the pages it allocated and B.) Check within each of the superpage areas it allocated address space in to make sure at least one page is free. This is very quick and its overhead is immaterial to this discussion. Deallocation is slightly simpler; merely update the page bitmap with the pages being deallocated and then set the bits of all affected superpages in the superpage bitmap to 0, because we know intuitively from the deallocation that there is at least one free page in that superpage now.

In conclusion, the Cottontail memory management system is the best I have been able to think up at the moment. Constructive criticism and/or comments on this paper would be appreciated from members of this newsgroup. This is only a first draft. Once discussion of it has ceased, I will do an edit and then add it to the material for the first issue of _The OS Development Journal_.

e-mail: frankm29a (at) no canned meat dot hotmail dot com

http://cottontail-os.tripod.com/

Download(ZIP 5.5kb) the source to Frank Millea's memory managment system.

This tutorial is here with Frank Millea's permission.