Page Frame Allocation

From OSDev Wiki

Contents

Physical Memory Allocators

These are the algorithms that will provide you a new page frame when you'll need it. The client of this algorithm is usually indifferent to which frame is returned, and especially, a request for n frames needn't to return contiguous frames (unless you are allocating memory for DMA operations like network packet buffers).

N will be the size of the memory in pages in the following text.

Bitmap

A large array of N/8 bytes is used as a large bit map of the memory usage (that is, bit #i in byte #n define the status of page #n*8+i). Setting the state of a given page is fast (O(1)), allocating a page may take time (O(N)).

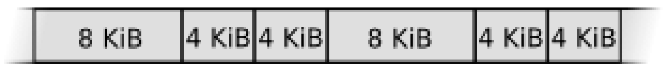
- an uint32_t comparison can test up to 32 bits at once and thus speed up allocations
- keeping a pointer to the last allocated bit may improve the performance of the next search (keeping information about the fact all the previous bytes were searched unsuccessfully)

Stack/List of pages

The address of each available physical frame is stored in a stack-like dynamic structure. Allocating a page is fast (O(1)), freeing a page too but checking the state of a page is not practical, unless additional metadata is stored sorted by physical address.

Sized Portion Scheme

You split each area of, say 16kb into (for example) chunks of 1 8kb, and 2 4kb's. Then you hand out each chunk. By doing this you can find closer fits to exact sizes. That means less waste. So say that you have an area of 32kb



You can even have 1, 2, even 3 or 4 (or more!) types of layouts for each portion. This way you have even more sizes to choose from.

Buddy Allocation System

This is the physical memory allocator of Linux kernel. Note that linux has several buddies depending on whether the memory is suitable for ISA DMA, or is coming from 'high physical memory' or just 'normal'. Each buddy contains k bitmaps, each indicating the availability of 2^i-sized and 2^i aligned blocks of free pages. Usually, linux uses from 4K to 512K blocks.

```
                0   4   8   12  16  20  24  28  32  36
                ###.#....#........#...###...########.... real memory pattern

  buddy[0]--->  ###.#.xx.#xxxxxxxx#.xx###.xx########xxxx 5  free 4K , 5-byte bitmap
  buddy[1]--->  # # # . # . x x . # . # # . # # # # x x  5  free 8K , 20-bits map
  buddy[2]--->  #   #   #   .   #   #   #   #   #   .    2  free 16K, 10-bits map
  buddy[3]--->  #       #       #       #       #        0  free 32K, 5-bits map
```

A buddy for N pages is about twice the size of a bitmap for the same area, but it allows faster location of collections of pages. Figure above shows a 4-buddy with free pages/blocks denoted as . and used pages/blocks denoted as #. When a block contains at least one used sub-block, it is itself marked as used and sub-blocks that are part of a larger block are also marked as used (x on the figure). Say we want to allocate a 12-K region on this buddy, we'll lookup the bitmap of free 16K blocks (which says we have one such starting at page #12 and another starting at page #36). buddy[2]->bit[4] is then set to 'used'. Now we only want 3 pages out of the 4 we got, so the remaining page is returned to the appropriated buddy bitmap (e.g. the single pages map). The resulting buddy is

```
                0   4   8   12  16  20  24  28  32  36
                ###.#....#..###...#...###...########.... real memory pattern

  buddy[0]--->  ###.#.xx.#xx###.xx#.xx###.xx########xxxx 6  free 4K , 5-byte bitmap
  buddy[1]--->  # # # . # . # # . # . # # . # # # # x x  5  free 8K , 20-bits map
  buddy[2]--->  #   #   #   #   #   #   #   #   #   .    1  free 16K, 10-bits map
  buddy[3]--->  #       #       #       #       #        0  free 32K, 5-bits map
```

Note that initially, only the largest regions are available, so if buddy[0] is apparently empty, we need to check buddy[1], then buddy[2] etc. for a free block to be split.

Hybrid scheme

Allocators may be chained so that (for instance) a stack only covers the last operations and that the 'bottom' of the stack is committed to a bitmap (for compact storage). If the stack lacks pages, it can scan the bitmap to find some (possibly in a background job).

Hybrid scheme #2

Instead of keeping track of just bits representing pages, or just page numbers on a stack, use a big array of structs to represent the memory. In these page frame structs, store a single link to a next page (pointer-sized) and a 8-16 bit information block indicating its status. Also include the virtual page pointer and the TCB to which the page number belongs. Keep pointers to each type of page, to both the start and the end of their lists. This way, you can easily display information about their content, the amount of pages for each type available, mix types, allow dynamic cleaning threads, do copy-on-write fairly easily and keep clear & concise overviews of the pages. It functions as a reverse page mapping table that lists types of pages too.

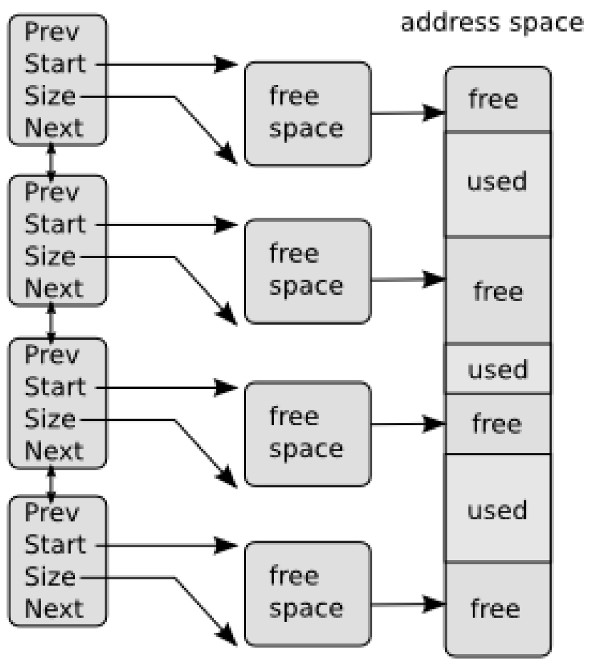
For an example implementation see AtlantisOS 0.0.2 or higher.

Virtual Addresses Allocator

Flat List

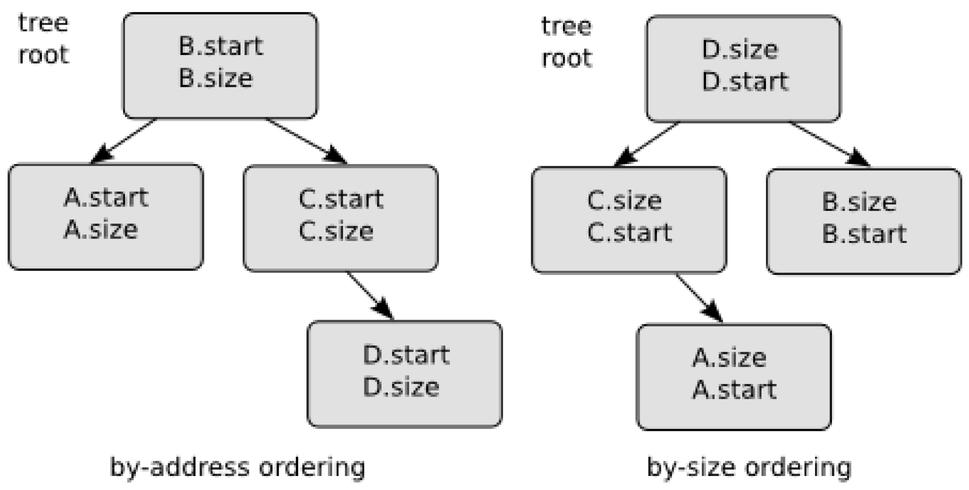
One straightforward way to manage big areas of addresses space is a linked-list (as depicted below). Each "free" region is associated with a descriptor giving its size and its base address. When some space needs to be allocated, the list is scanned for a region being large enough with a "first fit" or "best fit" (or whatever) algorithm. This was e.g. the way memory was managed by MS-DOS. When memory is allocated, the suitable entry is either removed (the whole region is allocated) or resized (only a portion of the region is allocated).

Note that with flat linked-lists, both "is memory at address XXX free" or "where can i get a block of size YYY" questions may require a complete traversal of the list to get answered. If virtual memory gets fragmented and the list gets longer, that may become an issue. "Is memory at address XXX free?" is mainly used to merge two free zone into a new (bigger) one when a block is released, and it is easier to deal with if the list is kept ordered by growing addresses.



Tree-based approach.

Since it is frequent that the list is searched for a given address or a given size, it may be interesting to use more efficient data structures. One of them that still keeps the ability of traversing the whole list is the AVL Tree. Each "node" in the AVL tree will describe a memory region and has pointer to the subtree of lower nodes and to the subtree of higher nodes.



While insertion/deletion in such a balanced tree requires more complex operations than linked list manipulation, searching the tree is usually achieved with O(log2(N)) rather than O(N) for linked lists -- that is, if you have 1000 entries, it requires 1000 iterations to scan the list against 10 iterations to find a given interval in the tree.

Linux has used AVL trees for virtual addresses management for quite a while. Note however that it uses it for regions (like what you find in /proc/xxxx/maps), not for a malloc-like interface.

See Also

Articles

- Memory Allocation
- Memory management
- Writing a memory manager - a tutorial

Threads

- Allocating memory for an allocator without an allocator
- A bitmap based allocation technique
- Ways to keep track of allocated RAM
- Questions about Memory Allocation
- Memory Management
- Memory Management to the X'th
- MM Questions
- (about)Tim Robinson Memory Management Tutorial #1
- Managing used/free pages
- Malloc, etc. (tute by curufir)
- Physical MM (by Freanan)
- Concepts and key points on alternative memory management schemes

External Links

- mystran's Basic VMM for Dummies (cached) (http://replay.web.archive.org/20081206102136/http://www.cs.hut.fi/~tvoipio/memtutor.html)
- Page replacement algorithm on Wikipedia