Memory Allocation

From OSDev Wiki

This page is about the allocation of memory from memory that is already available to the process (like malloc() and new()). For the allocation of page frames see Page Frame Allocation.

One of the most basic functions of a kernel is the memory management, i.e. the allocating and freeing of memory.

At square one, the kernel is the only process in the system. But it is not alone: BIOS data structures, memory-mapped hardware registers etc. populate the address space. Among the first things a kernel must do is to start bookkeeping about which areas of physical memory are available for use and which are to be considered "occupied".

The free space will subsequently be used for kernel data structures, application binaries, their heap and stack etc. - the kernel needs a function that marks a memory area as reserved, and makes that memory available to the process requiring it. In the C Standard Library, this is handled by malloc() and free(); in C++ by new() and delete().

Contents

A very very simple Memory Manager

The easiest you can do is the WaterMark allocator. Just keep track of how far you've allocated and forget about the notion of "freeing".

```
          before ...
                                                       <-N-->
  +----+-----+--+--------//--+             +----+-----+--+----+---//--+
  |##A#|##B##|C#|  free      |             |##A#|##B##|C#|##D#|free   |
  +----+-----+--+---------//-+             +----+-----+--+----+---//--+
                ^freebase    ^freetop                    ^d   ^fbase  ^ftop
```

When allocating N bytes for D, simply check that freetop-freebase>N and increment freebase by N. Period. A very simple Memory Manager

Now, if you need to free things, one of the easiest solution is to put at the start of the freed zone a descriptor that allows you to insert it in a list of free zones. Keeping that list sorted by address helps you identifying contiguous free zones and allows you to merge them in larger free zones.

```
   first free                        Structure of a   +--.--.--.--+    +---> +-- ...
     \/                              free zone        |F  R  E  E |    |     | FREE
+----+-----+--+----+---+-----//--+                    |nextfreeptr|----+
|##A#|free |C#|free|#E#|   free  |               <----|prevfreeptr|
+----+-|---+--+-|--+---+-----//--+                    | zone size |
       +-next>--+                                     +///////////+
```

Fixed size allocation

Allocating and deallocating fixed sized areas of memory is extremely simple. You can basically treat all free memory as a linked list of nodes. To allocate memory, you remove the front node from the linked list. To free memory, you return the front node to the linked list. This provides a constant allocation/releasing time and there is no fragmentation.

In the real world, programs like to allocate different sized chunks of memory so it's unlikely you can solely rely on this method.

However, it is theoretically possible for a microkernel to be designed so that all memory structures are exactly the same size (the Process struct, Thread struct, Message struct, etc). This would be very fast and efficient.

On this note, most Lisp implementations have a single 'box-and-pointer' base data type. A box-and-pointer type is a pair of values, usually pointer/pointer or atom/pointer (atom means a numeric value). On a 32-bit system, this structure is 8 bytes big. All other data structures (lists, trees, objects, etc) are built on top of this type. As a consequence memory allocation on a Lisp system is very fast.

Tips to go further

- Sometimes, and especially when you're working with objects, you have to allocate many objects that always have a certain size. It is wise to create pools of pre-divided large blocks for such objects.
- It's way easier to keep the size of allocated objects in a header hidden from the requester, so that a call to `free` doesn't require the object's size. Normally this hidden header is kept just before the block returned by `malloc`.
- It's way easier to design a memory allocator in a host OS than in your kernel. Also, if you implement the full malloc interface (`malloc`, `calloc`, `realloc` and `free` is enough on Linux) a good sanity test is to compile your `malloc` into a shared library, then compile something (like your whole host OS tree) with your malloc using LD_PRELOAD.
- Magic words like "F R E E" and "U S E D" will make your life easier when debugging. TimRobinson even allows 32 bits to store the address of the requester so that you can see "okay, this is a N-bytes block that was requested by `MySillyFunction()`, line 3405" ...

Memory & Microkernels

In a microkernel environment, there comes up a question: where the hell shall I put the memory management? In sense of heap management: give the kernel a dedicated allocator and a dedicated memory area to use - you might need two of them: one for the messages, and one for all the other stuff.

The other task of memory management: Process address space management, keeping track of used pages (yes, lets talk about paging, it is nice, it is neat, it causes a warm fuzzy feeling beneath the toes) and assigning memory to processes as needed, you can split it up. To be more precise, you have to split this task up - or keep every aspect of memory management in kernel land to make it easy. A recommended method for managing process address space: handle it in a per-process-manner. The actual process needs memory: allocate the memory and hand it out to the process. So you can keep the page allocating routines easy and straight forward. For this task of allocating/deallocating memory, you should take into consideration, that the task performing such actions should be able to slip into address spaces at needs (it loads the required page directory and does what it has to do - slimy little weasel thou' it is.) Take those things into good consideration and put quite an amount of brainwork into them. It's worth doing good engineering here.

Porting an existing Memory Allocator

It's not always desirable or practical to write your own memory allocator. Writing an efficient memory allocator can be an entire project in itself and fortunately it's extremely easy to port an existing memory allocator to your OS (to run in either kernel or user space). The advantages of using an existing memory allocator are; porting one is much faster than writing your own especially when you want to focus on other areas of your OS, it is likely to be well-tested so you do not have to debug the memory allocator, it takes a minimal amount of work to port it, and finally, someone else has down the hard work to make it fast, scalable, stable, etc.

Porting a memory allocator is fairly simple to do. Most of them are no more than one source and/or header file. The functionality you must hook is for allocating and freeing pages to your program, so the memory allocator has memory to work with.

There are a pair of hooks that some memory allocators use which allow the allocator to request memory from the kernel in terms of 'pages'. These memory allocators will have a constant stored somewhere in the source on how many large a page is (e.g. '#define PAGE_SIZE 4096' for 4KB pages) so the allocator knows how many pages to request at a time. With these allocators you must hook/implement:

- void *alloc_page(size_t pages) - Allocates 'pages' consecutive pages in virtual memory and returns a pointer to the beginning of the group.
- void free_page(void *start, size_t pages) - Frees 'pages' consecutive pages in virtual memory from 'start' back to the kernel.

In addition, some memory allocators will require you to hook locking functionality to ensure critical sections of the memory allocator aren't executed simultaneously by multiple threads. Typically the functions will appear like;

- void wait_and_lock_mutex() - Locks a mutex before entering a critical section. The simplest 'lock' solution is to disable interrupts which may be suitable as a starting point. For best performance, it is recommended to implement a spin lock.
- void unlock_mutex() - Unlocks the mutex after leaving a critical section. You either enable interrupts again or reset the spin lock. This allows any waiting threads to enter the critical section.

Choosing a Memory Allocator

There are many memory allocators to choose from. There is no perfect memory allocator because there are many different goals different allocators try to achieve. Usually these are conflicting goals, so different allocators have different trade-offs.

Some allocators are;

- ..fast. They can perform the most allocations and releases per second. 'Fast' is context sensitive, and an allocator that is fast when in some circumstances (allocating lots of large chunks) may not be in other circumstances (allocating and releasing lots of small chunks).
- ..space efficient. While other allocators may be aligning memory to page boundaries or have huge internal structures taking up megabytes, these allocators make sure all memory is packed into it's tightest fitting area and no byte is left to waste. This is especially important if you do not have much RAM.
- ..stable. The other allocators may be fast, but these allocators are designed to run for a very long time. They focus on minimising memory fragmentation, which is important for a server that runs for months on end.
- ..scalable. There are other allocators that are faster when allocating from a single thread but other threads must lock and wait for their turn. Scalable allocators can handle allocations from hundreds of threads on the latest quad-core CPUs concurrently with no significant performance penalty and minimal locking.
- ..real-time. There may be a fast allocator that on 'average' takes 75 cycles to allocate a chunk, but has the occasional worse case situation where it'll take 350 cycles. A real-time allocator might guarantee to return a memory pointer in less than 200 cycles. This is desirable in media decoding and real-time systems.

There are a lot of allocators to choose from, so this is far from being a comprehensive list;

- liballoc (https://github.com/blanham/liballoc/) - Excellent allocator that was originally a part of the Spoon Operating System and designed to be plugged into hobby OS's.
- dlmalloc (http://g.oswego.edu/dl/html/malloc.html) - Doug Lea's Memory Allocator. A good all purpose memory allocator that is widely used and ported.
- TCMalloc (http://goog-perftools.sourceforge.net/doc/tcmalloc.html) Thread-Caching Malloc. An experimental scalable allocator.
- nedmalloc (http://www.nedprod.com/programs/portable/nedmalloc/) A very fast and very scalable allocator. These two properties have made it somewhat popular in multi-threaded video games as an alternative to the default provided allocator.
- ptmalloc (http://www.malloc.de/en/) A widely used memory allocator included with glibc that scales reasonably while being space efficient.

See Also

Tutorials

- Writing a memory manager

External Links

- Memory Management 1 (http://www.osdever.net/tutorials/view/memory-management-1) - Part one of a two part series on memory management by Tim Robinson

- Memory Management 2 (http://www.osdever.net/tutorials/view/memory-management-2) - Part two of a two part series on memory management by Tim Robinson
- Publications about 'Memory Management' (http://www.cs.ucsb.edu/~grze/papers/Keyword/MEMORY-MANAGEMENT.html) - A list of some nice articles
- TLSF: Memory Allocator for Real-Time (http://rtportal.upv.es/rtmalloc/) General purpose dynamic memory allocator specifically designed to meet real-time requirements

Retrieved from "http://wiki.osdev.org/index.php?title=Memory_Allocation&oldid=17398"
Category: Memory management
