Effiziente Programmierung in c

Dynamic Memory Allocation

A presentation from: Marcel Ellermann
Agenda

• Introduction to Memory Management
• Basics of dynamic memory allocation
• Introduction to First-Fit, Next-Fit and Best-Fit
• Runtime comparison
• Summary
Introduction to Memory Management

- Physical memory is described with pages
- Page size is architecture dependent
- X86 Architecture has 4KB Pages
- Often more than one page size is supported
- ARM supports 4KB, 8KB and 32KB Pages*

*http://www.makelinux.net/books/lkd2/ch19lev1sec6
Introduction to Memory Management

- Kernel has its own address space
- Each process has its own address space
- Processes run in a virtual address space
- Virtual addresses are mapped to physical addresses by pages
Introduction to Memory Management

Accessed on 09th January 2014
Introduction to Memory Management

- The full virtual address space is not mapped to physical memory right from the start.

- On a system with 4KB Pages and 2GB Memory and a page size of 40 Byte the paging table would consume about 20MB.

- The process has to request mapping from its virtual address space to physical memory.

- Linux offers the brk/sbrk and mmap interface.
Introduction to Memory Management

• Each thread receives a default mapping of 8 MB at its highest available memory address
• Space is the program stack
• Stack pointer to the "top" of the stack
• Grows from highest address to lowest
• Only elements at the "top" can be removed
Basics of dynamic memory allocation
Basics of dynamic memory allocation

- If we free an allocated block we create wholes of free memory
- We do not want to waste that free memory
- Need to track free memory
- Need to handle our own memory space – can not use the heap
Basics of dynamic memory allocation

Source: The C programming Language; W. Kerningham & M. Ritchie; Pentice-Hall 1988
Basics of dynamic memory allocation

Requesting memory

- Usually the requested size is given in bytes
- Allocator has to select a free memory block from his free list
- Various approaches how to select the free memory block
- Allocator guarantees to return at least the requested block size
- Usually NULL is returned if no free memory is available
Basics of dynamic memory allocation

Requesting memory from OS

- Processes have nearly no virtual memory mapped by default
- **Program break** = the end of the process’s data segment
- Increase that end with a call to the system's `sbrk` method
- System calls are slow
- Usually larger memory amounts are requested and then distributed to the process upon request.
Basics of dynamic memory allocation

Freeing memory

• The memory chunk has to be returned into the free list
• Free list is ordered by ascending memory address
• Memory block is inserted at its respective place within the order
Basics of dynamic memory allocation

- **External fragmentation**: A form of fragmentation that arises when memory is allocated in units of arbitrary size. When a large amount of memory is released, part of it may be used to meet a subsequent request, leaving an unused part that is too small to meet any further requests.*

Basics of dynamic memory allocation

Coalescing

- To reduce external fragmentation blocks returned to the free list are coalesced with each other
- If block start at the end of a free block they’re merged
- If block ends at the start of a free block they’re merged
Basics of dynamic memory allocation

Miscellaneous

• Though the allocator has freed its memory, it’s not guaranteed to be returned to the operating system
• OS can only reserve full memory pages for a process
Basics of dynamic memory allocation

Header structure

typedef long Align;

union Header {
    struct {
        union Header *ptr;
        unsigned size;
    } s;
    Align x; /*Never used but forces alignment*/
};

• Header also used to align data
• Only multiples of the headers’ size are allocated
Dynamic memory allocation methods
Dynamic memory allocation methods

First-Fit

- First fit searches the Free-List from the beginning
- Returns the first chunk big enough to hold at least the requested amount of bytes
- If the chunk is too big it’s split and the remaining space is left in the Free-List.
Dynamic memory allocation methods

Next-Fit

- Similar to First-Fit

- Start searching from the block that pointed to the last allocated one

- Should drastically increase the performance
  - Average lookups should decrease
  - First-Fit tends to accumulate small block at its beginning but Next-Fit distributes the blocks uniformly.
Dynamic memory allocation methods

Best-Fit

- Uses the free block that fits the best to the requested size.
- Has to search the whole list for a best match
- Tends to leave small unusable blocks
- Therefore produces external fragmentation
Dynamic memory allocation methods

Optimization

• To reduce external fragmentation usually a minimal block size is defined

• If \textit{requested} < \textit{available} and \textit{requested} + \textit{min} > \textit{available} the whole block is returned and not split.

• Store available chunks in bins grouped by size
Dynamic memory allocation methods

Bins

<table>
<thead>
<tr>
<th>index</th>
<th>2 exact bins</th>
<th>4</th>
<th>...</th>
<th>64</th>
<th>65</th>
<th>sorted bins</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>...</td>
<td>512</td>
<td>576</td>
<td>640</td>
</tr>
</tbody>
</table>

Dynamic memory allocation methods

Bins

• 128 fixed-width bins
• Bins up to 512 bytes are each just spaced 8 bytes apart
• Only hold the exact size they’re representing
• Bins over the 512 byte mark hold a range of chunks
Dynamic memory allocation methods

Bins

- Till 1995 the oldest-first rule has been applied
- Nowadays chunks are sorted within bins
- It is has been proven that “the minor time investment was worth it to avoid observed bad cases”*
- Best-Fit with the presented optimizations even reduces external fragmentation compared to other methods

Dynamic memory allocation methods

Caching

• Caching refers to the fact that small blocks are not coalesced immediately

• Each split and each coalesce requires time

• Small blocks are kept up to a given number

• Very effective for many small allocations and frees i.e. when working with trees
Dynamic memory allocation methods

glib - malloc

“This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs.”*

*http://code.woboq.org/userspace/glibc/malloc/malloc.c.html
Dynamic memory allocation methods

glib - malloc

• For Blocks with a size <= 64 bytes the algorithm uses a caching allocator

• Block with a size >= 512 bytes the algorithm uses a pure best-fit allocator

• In between “it does the best it can trying to meet both goals at once”*

*http://code.woboq.org/userspace/glibc/malloc/malloc.c.html
Runtime comparison
Runtime comparison

- We perform INS_MAX cycles
- Each cycles between 1 and 10 allocations are performed
- Each allocation allocates between 1 and 8 kb
- Each allocation lasts between 1 clock and 10 seconds
- All random values are distributed uniformly
- Comparison done for First-Fit, Next-Fit, glib’s malloc
Runtime comparison

- For 100,000 cycles
- First-Fit: 157 seconds
- Next-Fit: 168 seconds
- glib’s malloc: 78 seconds
Runtime comparison

Evaluation

• Malloc outperforms each of the custom implementations

• Despite the fact that next-fit should be a huge improvement it’s even slower

• Differences might be to flaws in own implementations and some missing or ineffective optimizations
Summary

- Virtual memory
- Sbrk, mmap
- External fragmentation
- Best-Fit, Next-Fit, First-Fit
- Free-List
- Binning
- Caching
Literature

• Donald E. Knuth, The art of computer programming - Fundamental algorithms, 2006, Addison Wesley
• Carter Bays, A Comparison of Next-fit, First-fit and Best-fit, 1977, Volume 20, Number 3, Communications of the ACM, pp. 191—192
• Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall, 1988