A memory allocator with only 0,006% fixed overhead written from scratch

2025-02-01

https://fosstodon.org/@slink

Behind the “clickbait“ 1/2

- Overhead is roughly 2bit/page
- for 4KB page size ~ 0.006%:

```
freemap_size(4KB, 1MB) = 576.000B (0.054932%)
freemap_size(4KB, 1GB) = 66.469KB (0.006339%)
freemap_size(4KB, 1TB) = 65.019MB (0.006201%)
freemap_size(4KB, 1PB) = 65.016GB (0.006200%)
freemap_size(4KB, 1EB) = 65.016TB (0.006200%)
```

- https://gitlab.com/uplex/varnish/slash/-/commit/7623cec0daff82c9cc75bb376b784cde5ddd6e2c

CC0 1.0 Universal / public domain

Behind the „blickbait“ 2/2

- For 256 byte pages ~ 0.1%

```
freemap_size(256B, 1MB) = 1872.000B (0.178528%)
freemap_size(256B, 1GB) = 1042.250KB (0.099397%)
freemap_size(256B, 1TB) = 1040.258MB (0.099207%)
freemap_size(256B, 1PB) = 1040.254GB (0.099206%)
freemap_size(256B, 1EB) = 1040.254TB (0.099206%)
```


CC0 1.0 Universal / public domain

A memory allocator with a fixed overhead of ~ $2^{(s - p - 2)}$ for an arena of size 2^s and a minimal page size of 2^p written from scratch

2025-02-01
https://fosstodon.org/@slink

CC0 1.0 Universal / public domain

/me

- Born when unix time fit into 28 bits
- Linux since ~1992 (kernel ~0.9.8 IIRC)
- MSc in Artificial Intelligence
- Set up two ISPs, ran a third
- Learned “everything” from FOSS
- Since 2009, Independent Developer and Consultant
- Varnish-Cache Maintainer
- Runs a small company

CC0 1.0 Universal / public domain

The Project

- SLASH/ https://gitlab.com/uplex/varnish/slash/ (LGPL)
- Two Storage Engines for https://varnish-cache.org/
- Buddy: in-memory / Fellow: tiered memory / disk
  - Eventually persistent
  - Always consistent (log stuctured)
  - Async I/O (io_uring)
- Sponsor #1 runs a Video CDN

CC0 1.0 Universal / public domain

Requirements / Design

- For „disk“ (ssd) storage (fellow)
  - No freemaps on disk (avoid I/O) == freemaps in RAM
    - Free space is implicit in log
  - Support PBs with reasonable amounts of RAM
- For both storages
  - LRU fairness: First come, first served for allocation requests while room is made by LRU
  - Fixed size
  - Efficiency

CC0 1.0 Universal / public domain

Buddy allocator to the rescue

- Cool idea „invented in 1963 by Harry Markowitz“ (Wikipedia)
- Widespread use: Linux, FreeBSD, …
- Main idea: Adjacent free pages merge into one larger page
- „Hierarchy“ of power of two page sizes

CC0 1.0 Universal / public domain

Buddy example

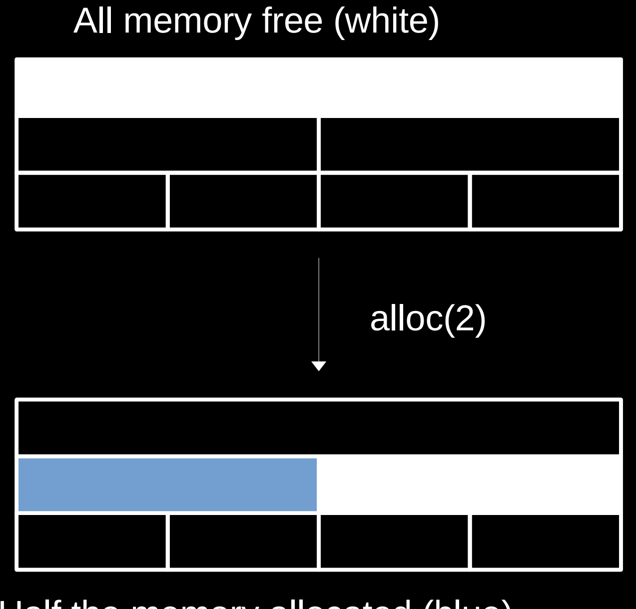
- https://en.wikipedia.org/wiki/Buddy_memory_allocation

All memory free (white)

CC0 1.0 Universal / public domain

Buddy example

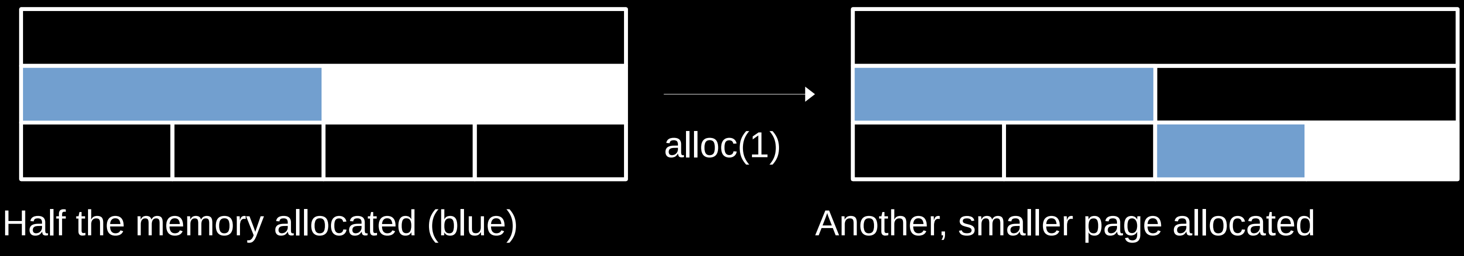
- https://en.wikipedia.org/wiki/Buddy_memory_allocation



CC0 1.0 Universal / public domain

Buddy example

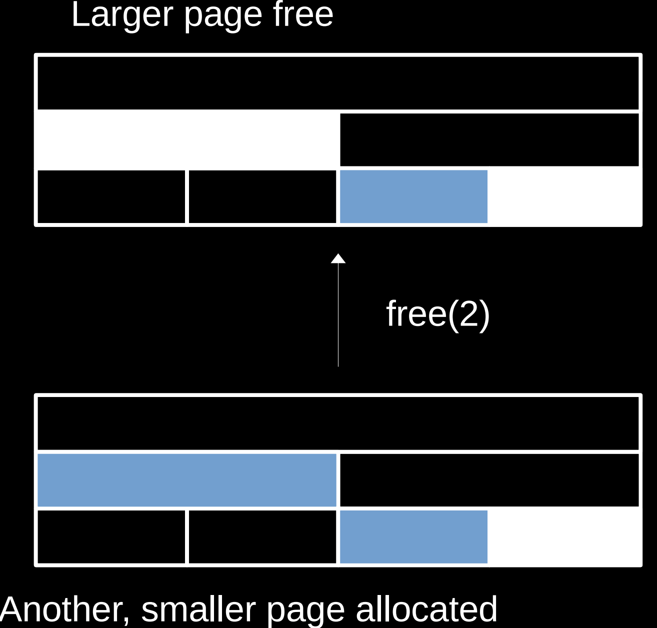
- https://en.wikipedia.org/wiki/Buddy_memory_allocation



CC0 1.0 Universal / public domain

Buddy example

- https://en.wikipedia.org/wiki/Buddy_memory_allocation



CC0 1.0 Universal / public domain

Buddy example

- https://en.wikipedia.org/wiki/Buddy_memory_allocation

CC0 1.0 Universal / public domain

Buddy example

- https://en.wikipedia.org/wiki/Buddy_memory_allocation

All memory free (white)

Larger page free

free(1)

alloc(2)

free(2)

alloc(1)

Half the memory allocated (blue)

Another, smaller page allocated

CC0 1.0 Universal / public domain

How do we organize free space?

- A memory allocator is not concerned with allocations, it manages free space
- „Easy“ if the free space is memory: The free space can hold metadata
- But for managing free space on disk, we have no memory, so all metadata needs RAM which we then can not use for caching (or we would need to do I/O, which we don’t)…
- For traditional approaches, metadata scales with fragmentation (the more free regions, the more metadata) → bad for predictable behavior wrt RAM usage

CC0 1.0 Universal / public domain

Use single bits to mark free pages

All memory free (white)

free(1)

Larger page free

alloc(2)

free(2)

alloc(1)

Half the memory allocated (blue)

Another, smaller page allocated

CC0 1.0 Universal / public domain

Bitfields

- In this implementation, the boxes from the example are literally single bits
- One bitfield per level
- Set bits mark free pages

- We can map the bitfields into another process and watch the allocator live

CC0 1.0 Universal / public domain

Demo

```
                                                               1544408186880 bytes free           keys:
                                                              12749242974208 bytes alloced        q : (q)uit
                                                              14293651161088 bytes total          f : show (f)ree sizes
                                             16KB: 23958               89.20 % used               + : zoom in
                                              8KB: 168789439                                      - : zoom out
                                              4KB: 29890386                * allocated
                                                                           _ free                 [pg](up/down): scroll map
                                                                           / partially free       left/right   : select storage

    0 - 14293651161087     - 1 char = 4GB                                                                    2/2 fellow.fellow.dsk
************************************************************************************************************************************
************************************************************************************************************************////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////<
```


CC0 1.0 Universal / public domain

Buddy operations

- Allocation and return map to
  - Freeing a page
  - Taking a page
  - Splitting a page
  - Merging pages
- Finding space (FFS)

The following contains slightly pseudoficated C-code

CC0 1.0 Universal / public domain

Free & Take

- Free: `bitf_set(bitfield, page);`
- Take: `bitf_clr(bitfield, page);`

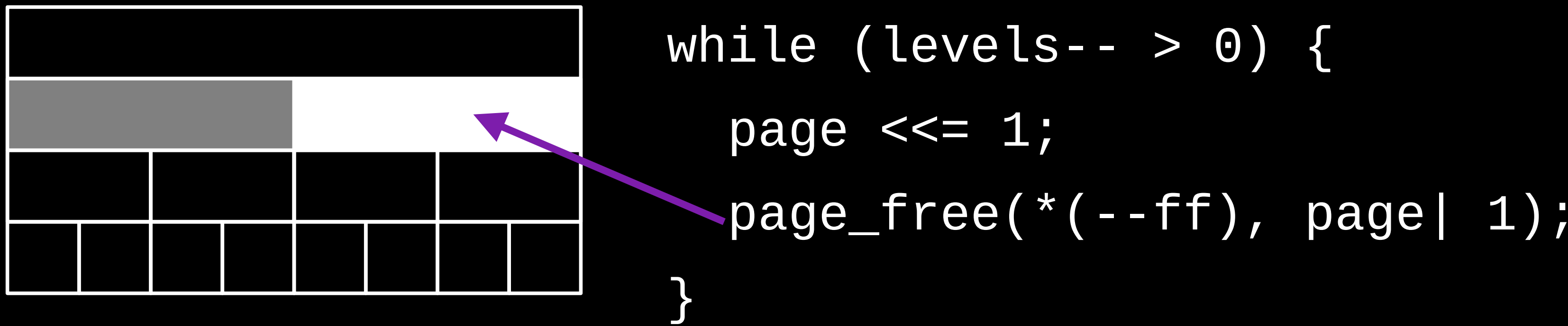
CC0 1.0 Universal / public domain

Split

```
AN(page_take(*ff, page));
```


CC0 1.0 Universal / public domain

Split



CC0 1.0 Universal / public domain

Split

```
while (levels-- > 0) {
  page <<= 1;
  page_free(*(--ff), page| 1);
}
```


CC0 1.0 Universal / public domain

Split

```
page_free(*(--ff), page| 1);
}
return(page);
```


CC0 1.0 Universal / public domain

Merge

Basically split in reverse: Check if in bounds of the bitmap and if the buddy page is free (can be taken), if so continue upwards (shift right, select the other page (xor))

```
while (bitf_nbits(f) > 1 && buddy < bitf_nbits(f) &&
  page_take(f, buddy)) {
    f = *++ff;
    page >>= 1;
    buddy = page ^ (size_t)1;
}
page_free(f, page);
```


CC0 1.0 Universal / public domain

Finding space

- For a 64bits: `__builtin_ffsll(word)`

- Problem: How do we find the first set bit in a multi-GB/TB bitfield?
- Solution: Index of bitfields:
  - Mark each word (64bit) != 0 as a single bit in the next index
  - Chase pointers (can probably be optimized)

CC0 1.0 Universal / public domain

Finding the right size

- If the requested page size is available, take it
- Otherwise look „upwards“ and split
- Additional: `cram` paramter
  - Also return a $2^{abs(cram)}$ smaller page

CC0 1.0 Universal / public domain

Trim

- Traditional wisdom: Buddy always rounds up to the next power of two.
- Yes, but: For anything which is not a power of two:
  - Allocate next larger page
  - Return „leftovers“

CC0 1.0 Universal / public domain

Fragmentation

- Allocation requests wait until they can be fulfiled by the cram rules
  - So LRU might need to evict a _lot_ of objects until free space happens to coalesce
- „Solution“
  - Set a „standard page size“ (chunk_size) and LRU until it can be allocated
  - Put in an array owned by LRU
  - LRU returns it when other requests are waiting

CC0 1.0 Universal / public domain

Usage examples

CC0 1.0 Universal / public domain

Basic (C)

- NO metadata, the allocation must be returned as received
- „what exactly is this thing?“
  - `struct buddy_ptr_extent { void *; size_t; }`
  - `struct buddy_off_extent { buddyoff_t; size_t; }`
  - `struct buddy_ptr_page { void *; uint8_t; }`
  - `struct buddy_off_page { buddyoff_t; uint8_t; }`
- Alloc: `buddy_alloc1_{ptr,off}_{extent,page}`
- Free: `buddy_return1_{ptr,off}_{extent,page}`

CC0 1.0 Universal / public domain

Batched alloc/return

- Entering the allocator is expensive (mutex)
- For page merges to happen, we want to return many pages at once
- Create structure on stack, enter allocator to execute

CC0 1.0 Universal / public domain

Batched alloc/return (C)

```
reqs = BUDDY_REQS_STK(buddy, howmany); // „alloca()-ish“
buddy_req_{extent,page}(reqs, …)
…
n = buddy_alloc(reqs);
while (n--)
  buddy_get_next_{ptr,off}_{extent,page}(reqs);
```

- Same idea for returns

CC0 1.0 Universal / public domain

Waiting allocations

- For a Web cache, we have two endpoints:
  - Cache misses & disk reads need memory
  - LRU eviction makes room
- Easiest if the memory allocator supports this scenario
- Prioritized queues
- Waiting requests are served by returns (frees)

- Usage:
  - `buddy_alloc1_{ptr,off}_{extent,page}_wait`
  - `n = buddy_alloc_wait(reqs);`

CC0 1.0 Universal / public domain

Async waiting

- Usage example (batched as above)

```
do {
  u = buddy_alloc_async_ready(reqs);
  if (u == 0) do_something_else();
} while (u == 0)
buddy_get_next_{ptr,off}_{extent,page}(reqs);
```

- Needs to be re-done with a callback to support my next project

CC0 1.0 Universal / public domain

Performance

- SLASH/fellow delivers bare metal performance (e.g. 60GB/s from RAM)
- allocator was never the bottleneck
- Can easily scale to multiple allocators

CC0 1.0 Universal / public domain

And now what?

- I wanted to share these ideas in case they might help...
- Is this really new?
    - I can hardly believe that no-one else had the bitfield idea before?
- Is this of general interest?
    - Fixed size case is special, I guess
- Anyone wants to (help) turn this into a standalone project?

CC0 1.0 Universal / public domain

Thank you!

CC0 1.0 Universal / public domain