Physical memory anti-fragmentation mechanisms in the FreeBSD kernel

Bojan Novković

`$ whoami`

- `bnovkov@FreeBSD.org`
- Software developer, Ph.D. candidate based in Zagreb, Croatia
- GSoC '22 and '23 contributor, `src` commiter since 2024.

Introduction

- Memory fragmentation - a recurring issue
  - Practically eliminated by virtual memory
  - Reintroduced in modern systems
- Overview of several anti-fragmentation mechanisms
  - Talk will focus on `amd64`
- Parts of this work were sponsored by *GSoC '23*

Background - physical memory allocation

- FreeBSD manages memory using the *buddy allocator* algorithm
  - Manages power-of-two page blocks
  - Each block size has its own freelist
    - Page *order* - $\log_2(block\_size)$

- Blocks are broken up and coalesced during runtime

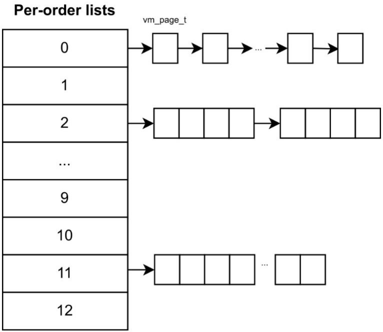
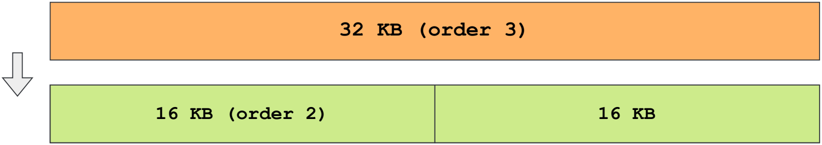


Figure 1: Buddy allocator freelists

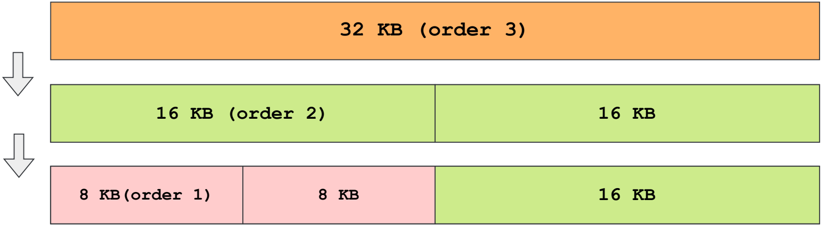
Background - physical memory allocation

32 KB (order 3)

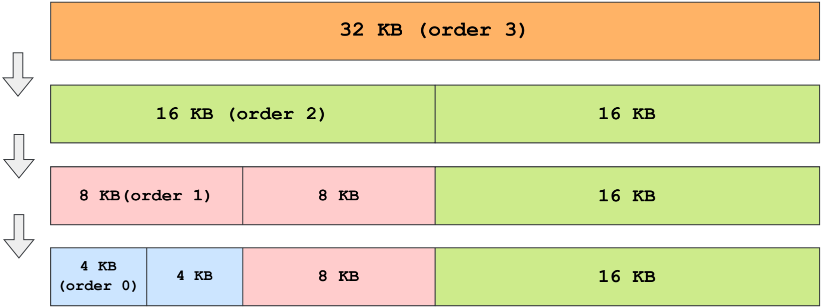
Background - physical memory allocation



Background - physical memory allocation



Background - physical memory allocation



Background - superpages

- Virtual address translation is costly
  - Can take up to 10%-30% of process runtime [1]
  - The *TLB* cache helps reduce performance cost
- Modern workloads are increasingly memory-hungry
  - Lower *TLB* efficiency
- Solution - ***superpages***
  - Pages of larger size than a standard page
  - Range from 2MB to 1G on `amd64`

Background - superpages

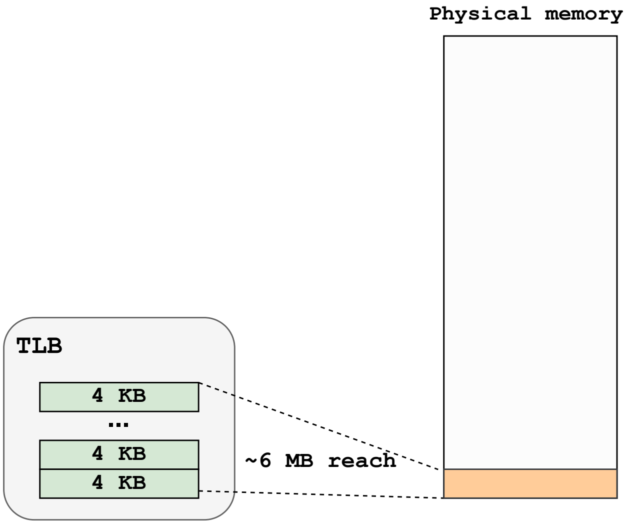


Figure 2: TLB reach on amd64 with regular pages.

Background - superpages

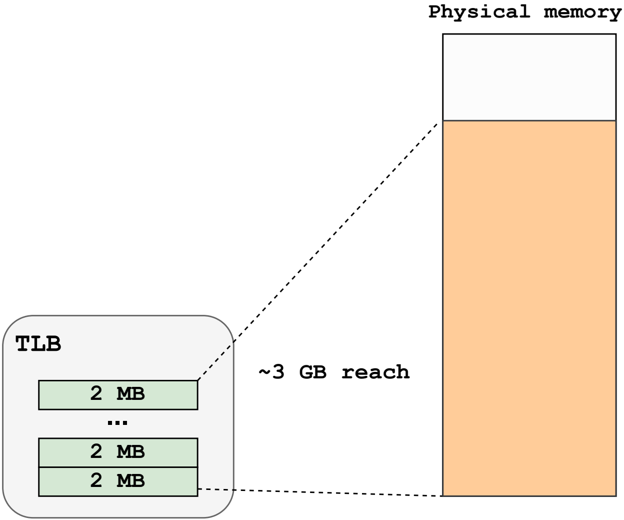


Figure 3: TLB reach on amd64 with superpages.

Background - superpages

- Superpages require a **contiguous** physical memory region
- OS needs a steady supply to maintain performance benefits
- Mixing 4K and 2M pages leads to **external fragmentation**
    - Superpage allocation often fail in fragmented environments

Background - external fragmentation

Page order	No. pages before	No. pages after
12 (16384K)	337	11
11 (8192K)	1	3
10 (4096K)	2	23
9 (2048K)	1	68
2 (16K)	9	1139
1 (8K)	1	1712
0 (4K)	1	2156

Table 1: State of a buddy allocator freelist before and after a `buildkernel` workload.

Background - external fragmentation

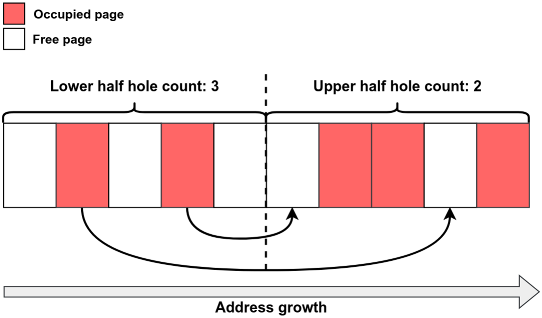


Figure 4: A fragmented memory region.

Background - external fragmentation

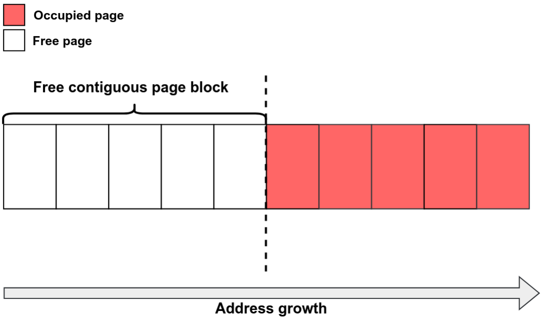


Figure 5: A rearranged memory region.

Memory compaction - overview

- Core idea - rearrange pages to increase contiguity
- An *active* defragmentation mechanism
    - Focused on maintaining superpage pool
- Very invasive
    - Interferes with running processes
    - Moving pages is expensive
- Still a WIP

Memory compaction - moving pages

```
static size_t
vm_phys_compact_region(vm_paddr_t start, vm_paddr_t end, int domain)
{
  vm_page_t free, scan;
  ...
  free = PHYS_TO_VM_PAGE(start);
  scan = PHYS_TO_VM_PAGE(end - PAGE_SIZE);
  ...
  while (free < scan) {
      ...
      /* Find suitable destination page ("hole"). */
      while (free < scan && !vm_phys_compact_page_free(free)) {
          free++;
      }
      ...
      /* Find suitable relocation candidate. */
      while (free < scan && !vm_phys_compact_page_relocatable(scan)) {
          scan--;
      }
      ...
      /* Swap the two pages and move "fingers". */
      error = vm_page_relocate_page(scan, free, domain);
      if (error == 0) {
          nrelocated++;
          scan--;
          free++;
      }
      ...
  }
  ...
}
```

Listing 1: Two-finger compaction algorithm.

Memory compaction - metadata

- **Which regions do we compact?**
- Idea - maintain page stats for blocks of memory
  - Must hook into the buddy allocator
- Two important requirements:
  - Minimal performance overhead
  - Must work with sparse physical memory

Memory compaction - metadata

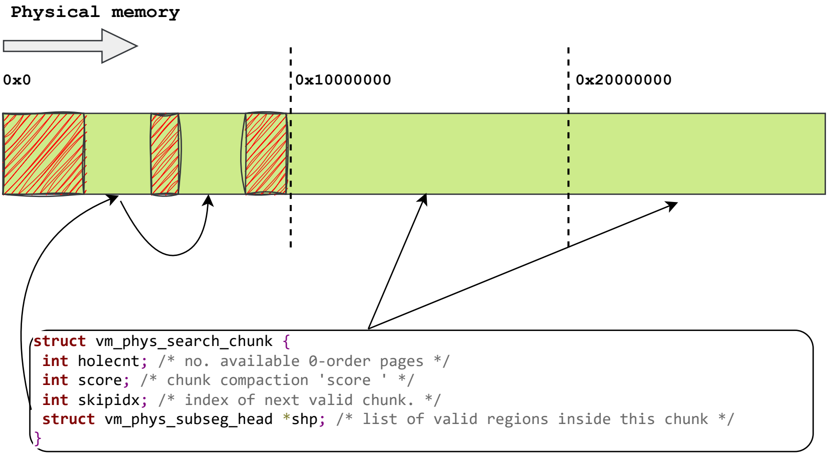


Figure 6: Tracking compaction metadata.

Memory compaction - quantifying fragmentation

- **When should we compact?**
- *Free Memory Fragmentation Index (FMFI)* [2]
  - Quantifies external fragmentation of a freelist
  - Values range from negative to 1

$$F_i(o) = 1 - \frac{NoPagesFree/2^o}{BlocksFree}$$

Memory compaction - background compaction

- Putting it all together - ***compactd***
  - Monitors fragmentation for superpage order
  - Compacts when *FMFI* drops below a threshold
    - Tunable - **`vm.phys_compact_thresh`**
  - Rudimentary back-off mechanism

- One compaction thread per NUMA domain

- Evaluation
  - Ryzen 5 5600 X, 48 GB DDR4 RAM
  - Benchmark - **`buildkernel`** x 10

Memory compaction - results

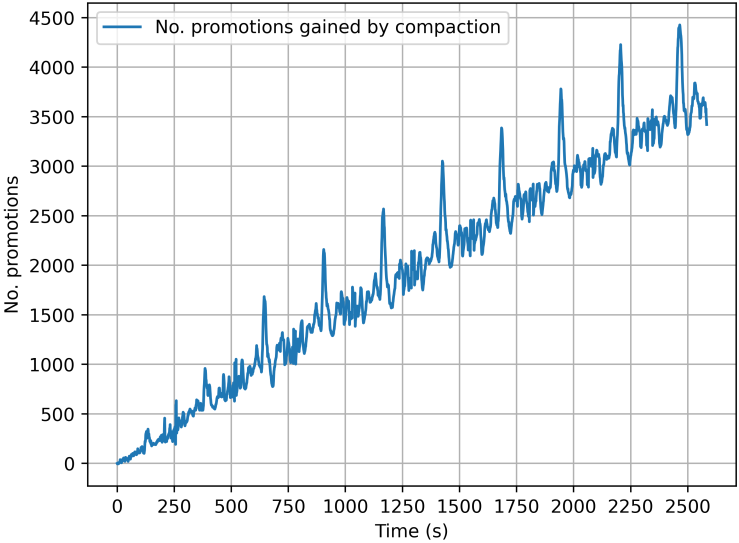


Figure 7: Compaction benchmark results.

Reworking kernel stack allocations

- Fragmentation issues in kernel stack allocation
  - “Guard” pages
  - Each kernel stack leaves an unused 0-order page
- Issue - `vm_object_t` page offset calculation
  - **`KVA >> PAGE_SHIFT`**

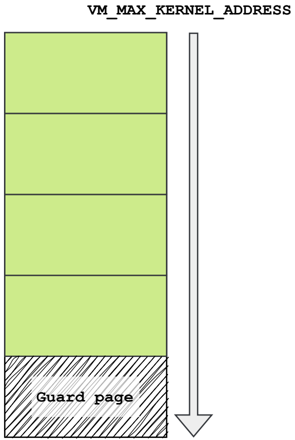
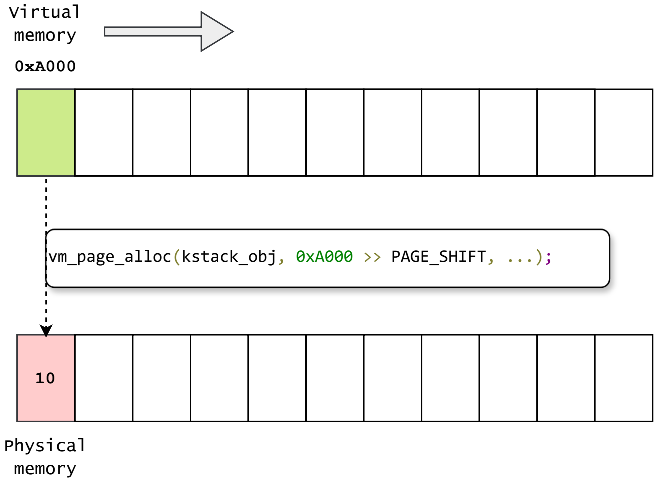
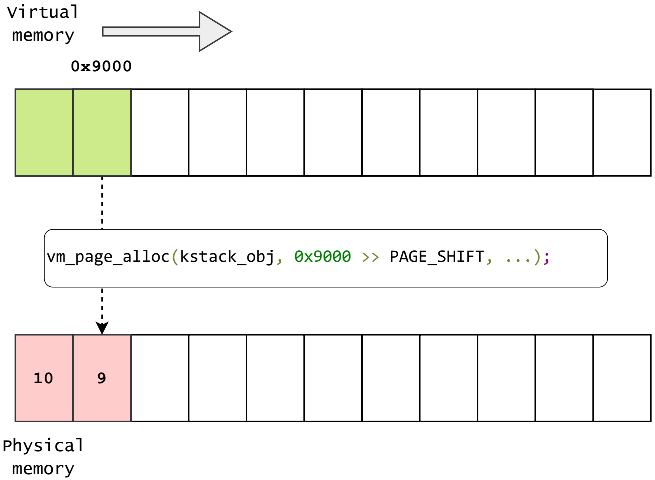


Figure 8: `amd64` kstack layout.

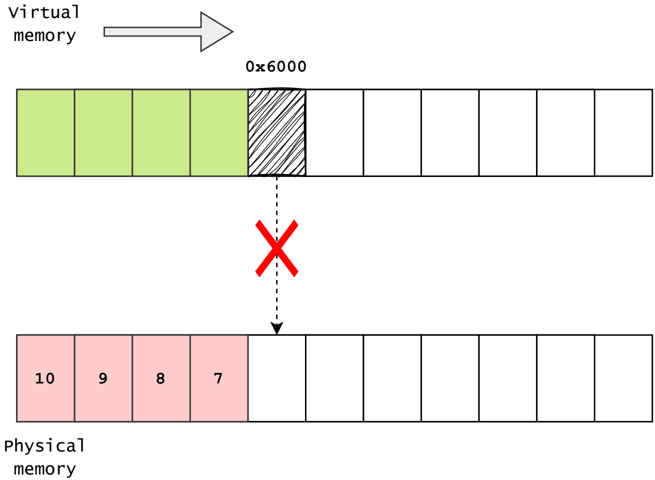
Reworking kernel stack allocations



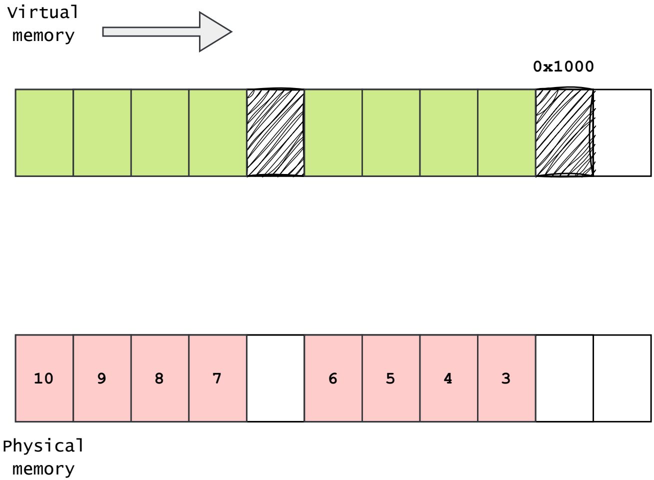
Reworking kernel stack allocations



Reworking kernel stack allocations



Reworking kernel stack allocations



Reworking kernel stack allocations

- Kernel stacks have two nice properties
  1. Fixed size
  2. Guard pages at fixed offsets

- These can be used to mathematically "pack" the pages together
  - Other backing mechanisms required

- Additional benefits
  - Guard pages at each end
  - More room for kernelspace superpages

Reworking kernel stack allocations

```
vm_pindex_t
vm_kstack_pindex(vm_offset_t ks, int kpages)
{
	vm_pindex_t pindex = atop(ks - VM_MIN_KERNEL_ADDRESS);

#ifdef __ILP32__
	return (pindex);
#else
	/*
	 * Return the linear pindex if guard pages aren't active or if we are
	 * allocating a non-standard kstack size.
	 */
	if (KSTACK_GUARD_PAGES == 0 || kpages != kstack_pages) {
		return (pindex);
	}
	KASSERT(pindex % (kpages + KSTACK_GUARD_PAGES) >= KSTACK_GUARD_PAGES,
	    ("%s: Attempting to calculate kstack guard page pindex", __func__));

	return (pindex -
	    (pindex / (kpages + KSTACK_GUARD_PAGES) + 1) * KSTACK_GUARD_PAGES);
#endif
}
```

Listing 2: Improved page offset calculation

Reworking kernel stack allocations

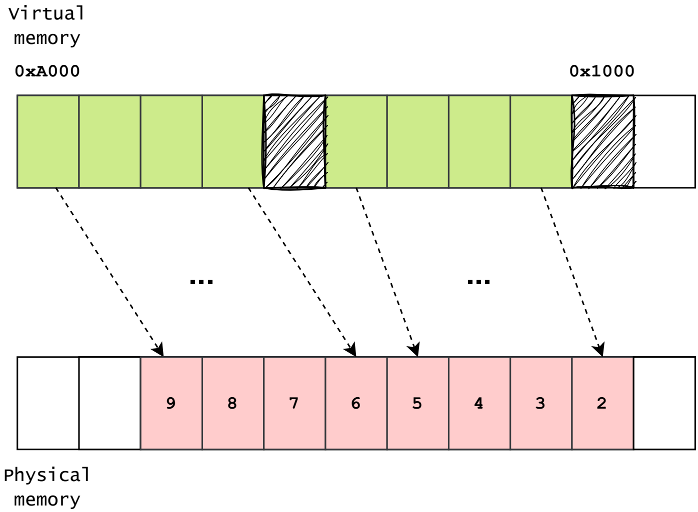


Figure 9: Adjusted kstack allocations.

Batched page allocations

- Common idiom - allocate 0-order pages in a tight loop
- Two issues:
  1. Allocated pages might not be contiguous
  2. Poor cache usage

```
...
for (i = 1; i <= *rbehind; i++) {
    p = vm_page_alloc(object,
                      ma[0]->pindex - i,
                      VM_ALLOC_NORMAL);
    if (p == NULL)
            break;
    p->oflags |= VPO_SWAPINPROG;
}
*rbehind = i - 1;
...
```

Listing 3: Swap pager - allocating multiple pages

Batched page allocations

- New page allocation routine - **`vm_page_alloc_pages`**
  - Promotes contiguity
  - Cache-friendly
- Microbenchmark evaluation
  - Measuring the time it takes to allocate N pages
  - $N \in \{1, 2, 4, ..., 65536\}$

Batched page allocations - results

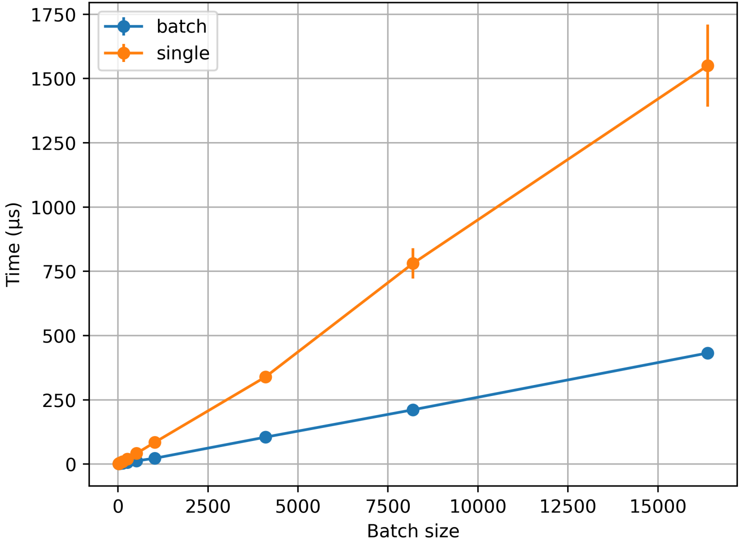


Figure 10: Batched allocation benchmark results. Smaller is better.

Speeding up `mlock(2)`

- Motivation - wiring large amounts of memory is slow
  - Especially problematic for hypervisors

- `mlock(2)` allocates and maps one 0-order page at a time

- Idea - preallocate and insert higher order pages

- Evaluated by booting **`bhyve`** VMs

Speeding up `mlock(2)` - results

	baseline	patched
Avg (ms)	875.02	**92.49**
Median (ms)	883.77	**79.98**
Stddev	80.79	18.56
Min	761.68	76.76
Max	992.12	115.02

Table 2: `mlock` benchmark results. Smaller is better.

Future work

- Issues with “permanent” fragmentation
  - Improving placement of long-lived wired (unmovable) pages

- Compaction efficiency
  - Smarter heuristics

Conclusion

- Reviews:
  - D44450, D43622, D40772, D38852
- Thanks to `markj@` for his mentorship

References

- [1] *Gupta, Siddharth, et al. "Rebooting virtual memory with midgard." 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021.*

- [2] *Gorman, Mel, and Andy Whitcroft. "The what, the why and the where to of anti-fragmentation." Ottawa Linux Symposium. Vol. 1. Citeseer, 2006.*