Shared Huge Pages with a Memory Allocator to Avoid Fragmentation

Samuel Grayson

ABSTRACT

Huge pages present certain advantages, but they have some problems with page fragmentation. The OS can allocate a huge page to a group of processes which trust each other as long as each of the processes agree to use non-overlapping ranges. This mitigates some of the problems with huge pages while offering some advantages of its own.

I want to formalize this idea, develop measurement tools to see if it is worth pursuing, and—time permitting—implement it.

1 INTRODUCTION

Memory-allocation has two regimes:

(1) Intrapage memory-allocation allocates variable size ranges to a specific process
(2) Interpage memory-allocation allocates fixed-sized pages to each process

Thus interpage memory-allocation has much greater problems with internal fragmentation (many partly filled pages) and external fragmentation (no contiguous unused space big enough for a whole page). These problems is only worsened if huge pages are utilized. However, huge pages have desirable characteristics: the same TLB maps more physical memory, resulting in fewer page walks.

I want to create a group-malloc combining these techniques, where multiple processes share a huge page and allocate memory from it using intrapage memory allocation. This has the benefits of both: TLB performance of huge pages with the fragmentation performance of intrapage memory allocation. This behavior can be selectively enabled at runtime without source-code change, and traditional share-nothing processes would have unchanged performance.

2 HOW SHARING PAGES COULD WORK

The user could tells the OS to execute a group of tasks with sharing with a syscall like exec. Then the OS would allocate a huge page for the group and initialize a memory-allocator (`grmalloc()`) on it. When the process calls `malloc()`, the memory could come out of this shared page. [1] The other program segments can be shared in the following way:

(1) Sharing the data segment (both initialized and uninitialized) is straightforward: call `grmalloc()` and load the data there. [2]
(2) Sharing the text segment is mostly straightforward with one complication: absolute branch addresses. At load-time, these would have to be rewritten to be the address plus the ranges offset. In a production system, this could be done easily in hardware without the need for software rewriting.
(3) Sharing the heap is somewhat complicated. I divide this into three cases:
    (a) If a program uses libc's `malloc()`, then is straightforward: proxy to `grmalloc()`.
    (b) If a program uses `mmap()` to create an 'anonymous mapping' significantly smaller than a page, then this is straightforward. Otherwise, it is best to allocate a set of entire pages. With huge pages, this case will be rare.
    (c) If a program uses `brk()`, `sbrk()`, or other memory primitives directly, the OS should not attempt this optimization.
(4) Sharing the stack is less reliable. If the OS could know that the stack would not grow beyond a certain size (this could be inferred from prior runs), then the OS could allocate a range for the stack.

All of these optimizations are independent, so they can be implemented progressively, maintaining correctness along the way.

In order to limit the scope to something feasible, I will assume a uniprocessor system. In a multithreaded case, `grmalloc` would need to either use a lock or atomics. This work is still relevant because it either proves the unfeasibility of the idea in the simplest-case or it gives us an expectation to strive for in a world.

3 COSTS AND BENEFITS

3.1 Pros

(1) Huge pages give a greater TLB reach which minimizes TLB misses.
(2) When context-switching to another process in the shared group, the kernel does not have to flush the TLB.

Pro 2 is especially important for processes that trade off control at a high-frequency, common in patterns such as the actor model, communicating sequential processes, and threads with locks.

3.2 Cons

(1) Shared huge pages could still suffer from some memory fragmentation (both internal and external), although this problem is less drastic than in traditional huge pages.
(2) Sharing gives less isolation against invalid memory accesses.
(3) Individual processes might use more pages because they get less of each page. This is mitigated by having larger pages in general. If there are n processes in a group, then the pages should be more than n times larger to see beneficial TLB performance.

Con 2 is a valid flaw, but there are situations where its impact is low:

(1) In containers and VMs, applications commonly run as root because they are sandboxed.

[1] POSIX does not require that successive calls to `malloc()` be contiguous,[4] and glibc already breaks this anyway.[2]

[2] The OS should not attempt this optimization for software that uses non-standard system-defined pointers such as etext, edata, and end[1].

(2) High-level languages that do not have manual memory management do not suffer from this class of bugs.
(3) High-performance computing where access is already strictly controlled.
(4) Applications or application-suites that use multiple processes written by the same vendor or individual might trust each other.

4 PRIOR WORK

This idea is similar to multithreading, where processes share a virtual-address space. However unlike threads sharing processes do not need to know that they are sharing a virtual-address space; they won't modify each others data because a correct program cannot to derive a pointer to memory it did not allocate.

This idea is similar to using `mmap()` to create shared pages, but again the programs don't have to be written with this optimization in mind. Shared huge pages could share more than just the range allocated by `mmap()`; they could share the data and text segments as well.

This idea is similar to single address-space OSes like Opal[3] and Nemesis, but shared huge pages can coexist with traditional processes that want their own memory space. It can easily be built into a POSIX OS, so many apps can run with the optimization without modification.

5 APPLICATIONS

I am not sure which OS I will target, but it should already supports mixed page-sizes to support shared huge pages.

I am not sure which applications to target, but the ideal conditions are:

(1) applications that have sporadic memory accesses (sporadic even at the granularity of a 4k page)
(2) applications that have processes that trade off control at a high frequency
(3) possibly inside a virtual machine

Other literature uses traditional benchmarks (SPEC CPU, PARSEC 3.0), machine learning workloads (MapReduce web search, Spark MovieRecmd), or database workloads (Redis, MongoDB). [5]

6 METHODOLOGY

First I would attempt to predict how big of a problem this is and how much room is there for improvement.

This will probably involve creating tools to get real-world measurements on existing systems (not modifying the OS yet).

Then if the numbers are promising, I will implement a part of this optimization. Which part (`malloc` vs memory-segments?, is not flushing TLB on context-switch important?) can be driven by the data from previously.

REFERENCES

[1] *end(3) Linux man page*, Mar. 2019. Version 5.02.
[2] *malloc(3) Linux man page*, Mar. 2019. Version 5.02.
[3] Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems 12*, 4 (Nov. 1994), 271–307.
[4] IEEE and The Open Group. *malloc - POSIX.1-2017*, 2018.
[5] Kwon, Y., Yu, H., Peter, S., and Rossbach, C. J. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation.* (Nov. 2016).