Linux Memory Management

Ahmed Ali-Eldin

This lecture

- Kernel memory allocations
- User-space memory management
- Caching in memory

Numbers every programmer should know...

Yearly updated data: https://colin-scott.github.io/personal_website/research/interactive_latency.html

Approximate timing for various operations on a typical PC:

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

Kernel memory allocation

- Kernel processes require memory allocation
  - kernel cannot easily deal with memory allocation errors
  - kernel often cannot sleep
- The kernel memory allocation mechanisms differ from user-space allocation
- The kernel treats physical pages as the basic unit of memory management
- x-86 processors include a hardware Memory Management Unit (MMU)
- Memory management in Linux is a complex system
  - supports a variety of systems from MMU-less microcontrollers to supercomputers.

Pages in Linux

- Old days: Pages with default size 4 KB
- Now: Huge pages + 4KB pages (since kernel 2.6.3)
- Rationale
  - on a machine with 8KB pages and 32GB of memory, physical memory is divided into 4,194,304 distinct pages
  - 64-bit Linux allows up to 128 TB of virtual address space for individual processes, and can address approximately 64 TB of physical memory
  - While assigning memory to a process is relatively cheap, memory management is not!
    - Every memory access requires a page walk
    - Typically take 10 to 100 cpu cycles
    - A TLB can hold typically only 1024 entries (out of the 4 Million distinct pages above) and has to be flushed every context switch

Large Memory Applications

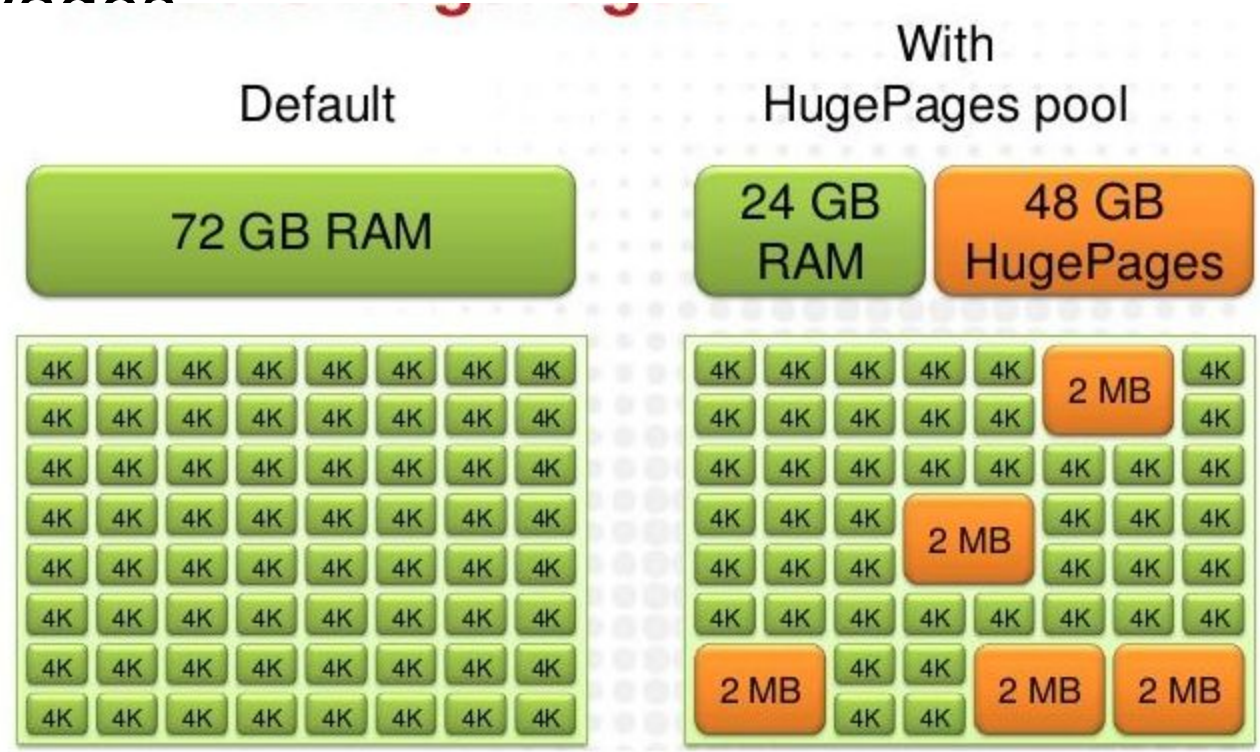
- Applications touching greater than 10 GB such as a large in-memory database with 17 GiB with 1000 connections
  - Uses approximately 4.4 million pages
  - With a page table size of 16 GiB
- Idea: Having larger page sizes enables a significant reduction in memory walks for such an application

Huge Pages

- on x86, it is possible to map 2M and even 1G pages using entries in the second and the third level page tables.
- In Linux such pages are called huge
- Usage of huge pages significantly reduces pressure on TLB, improves TLB hit-rate and thus improves overall system performance.
- Allows power users to choose the correct Page size for their application
- However, it causes fragmentation

Huge P

- on x86 … the secon
- In Linu
- Usag… ves TLB hit-rat…



Transparent Huge Pages (THB)

- Builds on the above concept
- Default for all applications today
- Transparent to the application
- Idea: If an application asks for a large memory chunk, give it a huge page instead of a small one

So is it any good?

- It depends on the application:
  - https://alexandrnikitin.github.io/blog/transparent-hugepages-measuring-the-performance-impact/
  - https://docs.oracle.com/cd/E11882_01/install.112/e47689/pre_install.htm#LADBI1519
  - https://lwn.net/Articles/374424/
- In many cases, THB actually causes severe performance issues in applications.
  - It is still debatable how to fix this

Memory in the kernel

- The kernel represents every **physical** page on the system with a `struct page` structure
  - Define in https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h
  - no way to track which tasks are using a page
  - A much more complex definition than the one in the book, however let us look at the basics
  - The page structure is associated with physical pages, not virtual pages.
- The total size of all the structures can well exceed a 100 MB

`struct page` : Page flags

- The flags field stores the status of the page.
  - Such flags include whether the page is dirty or whether it is locked in memory.
  - Bit flags represent the various values, so at least 32 different flags are simultaneously available
  - Defined in https://github.com/torvalds/linux/blob/master/include/linux/page-flags.h

`struct page:` Tracking pages

- The `_count` field stores the usage count of the page
  - that is, how many references there are to this page.
  - When this count reaches negative one, no one is using the page, and it becomes available for use in a new allocation
- The `virtual` field is the page's virtual address.
- The kernel needs to know who owns the page. Possible owners include user-space processes, dynamically allocated kernel data, static kernel code, the page cache, and so on

Memory Zones

- Because of hardware limitations, the kernel cannot treat all pages as identical.
  - Some pages, because of their physical address in memory, cannot be used for certain tasks.
- The kernel divides pages into different zones.
  - The kernel uses the zones to group pages of similar properties
  - Linux has to deal with two shortcomings of hardware with respect to memory addressing:
    - Some hardware devices can perform DMA (direct memory access) to only certain memory addresses.
    - Some architectures can physically addressing larger amounts of memory than they can virtually address. Consequently, some memory is not permanently mapped into the kernel address space.
- Zones defined in: https://github.com/torvalds/linux/blob/cc12071ff39060fc2e47c58b43e249fe0d0061ee/include/linux/mmzone.h

The main zones

- Note: The actual use and layout of the memory zones is architecture-dependent, and not all zones need to exist!
- Each zone is represented by `struct zone` in the `mmzone.h` file
  - ZONE_DMA—This zone contains pages that can undergo DMA.
  - ZONE_DMA32—This zone contains pages that can undergo DMA. and are accessible only by 32-bit devices.
  - ZONE_NORMAL—This zone contains normal, regularly mapped, pages.
  - ZONE_HIGHMEM—This zone contains “high memory,” which are pages not permanently mapped into the kernel’s address space.
    - On 32-bit x86 systems, ZONE_HIGHMEM is all memory above the physical 896MB mark. On other architectures, ZONE_HIGHMEM is empty because all memory is directly mapped.

Allocating pages in the kernel

Flag	Description
`alloc_page(gfp_mask)`	Allocates a single page and returns a pointer to its
`alloc_pages(gfp_mask, order)`	Allocates 2^{order} pages and returns a pointer to the first page's `page` structure
`__get_free_page(gfp_mask)`	Allocates a single page and returns a pointer to its logical address
`__get_free_pages(gfp_mask, order)`	Allocates 2^{order} pages and returns a pointer to the first page's logical address
`get_zeroed_page(gfp_mask)`	Allocates a single page, zero its contents and returns a pointer to its logical address

Gfp_mask

Flag	Description
__GFP_WAIT	The allocator can sleep.
__GFP_HIGH	The allocator can access emergency pools.
__GFP_IO	The allocator can start disk I/O.
__GFP_FS	The allocator can start filesystem I/O.
__GFP_COLD	The allocator should use cache cold pages.
__GFP_NOWARN	The allocator does not print failure warnings.
__GFP_REPEAT	The allocator repeats the allocation if it fails, but the allocation can potentially fail.
__GFP_NOFAIL	The allocator indefinitely repeats the allocation. The allocation cannot fail.
__GFP_NORETRY	The allocator never retries if the allocation fails.
__GFP_NOMEMALLOC	The allocator does not fall back on reserves.
__GFP_HARDWALL	The allocator enforces "hardwall" cpuset boundaries.
__GFP_RECLAIMABLE	The allocator marks the pages reclaimable.
__GFP_COMP	The allocator adds compound page metadata (used internally by the hugetlb code).

Flag	Description
__GFP_DMA	Allocates only from ZONE_DMA
__GFP_DMA32	Allocates only from ZONE_DMA32
__GFP_HIGHMEM	Allocates from ZONE_HIGHMEM or ZONE_NORMAL

Easier than masks, flags

Flag	Description
GFP_ATOMIC	The allocation is high priority and must not sleep. This is the flag to use in interrupt handlers, in bottom halves, while holding a spin-lock, and in other situations where you cannot sleep.
GFP_NOWAIT	Like GFP_ATOMIC, except that the call will not fallback on emergency memory pools. This increases the liklihood of the memory allocation failing.
GFP_NOIO	This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when you cannot cause more disk I/O, which might lead to some unpleasant recursion.
GFP_NOFS	This allocation can block and can initiate disk I/O, if it must, but it will not initiate a filesystem operation. This is the flag to use in filesystem code when you cannot start another filesystem operation.
GFP_KERNEL	This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to do to obtain the memory requested by the caller. This flag should be your default choice.
GFP_USER	This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.
GFP_HIGHUSER	This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes.
GFP_DMA	This is an allocation from ZONE_DMA. Device drivers that need DMA-able memory use this flag, usually in combination with one of the preceding flags.

Freeing pages in the kernel

- Three main functions
  - `void __free_pages(struct page *page, unsigned int order)`
  - `void free_pages(unsigned long addr, unsigned int order)`
  - `void free_page(unsigned long addr)`
- You must be careful to free only pages you allocate.
  - Passing the wrong struct page or address, or the incorrect order, can result in corruption.
  - kernel trusts itself. unlike with user-space

Example

```
unsigned long page;


page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
        /* insufficient memory: you must handle this error! */
        return -ENOMEM;
}

/* 'page' is now the address of the first of eight contiguous pages ... */
```

free_pages(page, 3);

Allocating byte sized chunks

- Use `kmalloc()` if you need physically contiguous memory (mostly needed for hardware devices), and `vmalloc()` if you only need virtually contiguous memory

void * kmalloc(size_t size, gfp_t flags)

- The function returns a pointer to a region of memory that is at least `size` bytes in length
  - Kernel allocations always succeed, unless an insufficient amount of memory is available.
  - You must check for NULL after all calls to kmalloc() and handle the error appropriately
- The counterpart to `kmalloc()` and `vmalloc()` is `kfree()` and `vfree()`

```
void kfree(const void *ptr)
```

- `kmalloc` used more as it has better performance

Linux Slab layer

- The slab layer acts as a generic data structure-caching layer.
  - Most kernel programmers introduce free lists in their code. This is memory that was allocated for a data structure that no longer exists.
    - Rather deallocating the memory, it is added to a free list that can later be used for new data structures instead of trying to allocate new memory
    - Acts like an internal caching layer in the kernel
- Can be problematic when the memory becomes scarce
  - To enable the kernel more control, the kernel provides the slab layer
  - Implemented in https://github.com/torvalds/linux/blob/master/mm/slab.c and definitions in slab.h

Slab- layer basic tenets (1)

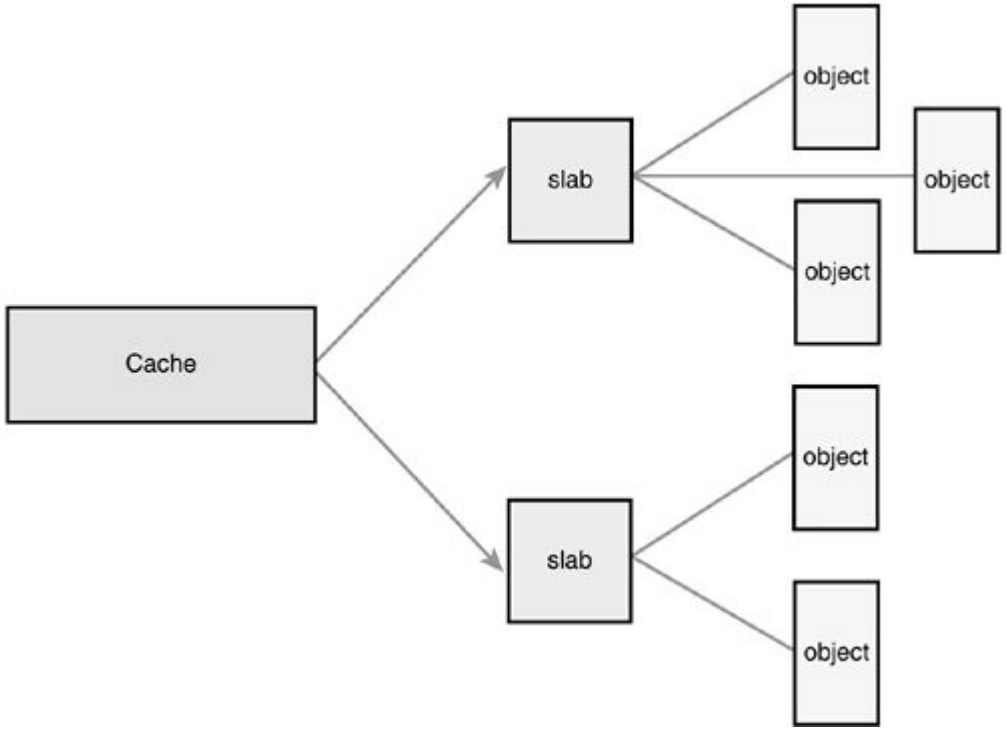
- Frequently used data structures tend to be allocated and freed often, so cache them.
- Frequent allocation and deallocation can result in memory fragmentation
    - To prevent this, the cached free lists are arranged contiguously. Because freed data structures return to the free list, there is no resulting fragmentation.
- The free list provides improved performance during frequent allocation and deallocation because a freed object can be immediately returned to the next allocation
- If the allocator is aware of concepts such as object size, page size, and total cache size, it can make more intelligent decisions.

Slab- layer basic tenets (2)

- If part of the cache is made per-processor (separate and unique to each processor on the system), allocations and frees can be performed without an SMP lock.
- If the allocator is NUMA-aware, it can fulfill allocations from the same memory node as the requestor.
- Stored objects can be colored to prevent multiple objects from mapping to the same cache lines.

Slab design

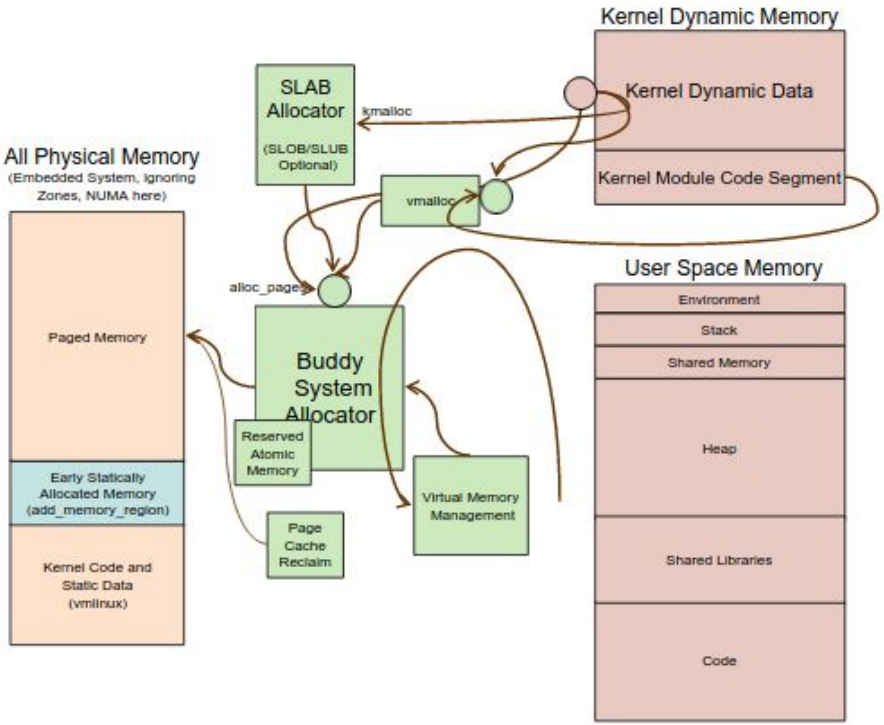
- The slab layer divides different objects into groups called caches
  - The caches are then divided into slabs
  - The slabs are composed of one or more physically contiguous pages.
- There is one cache per object type
  - one cache is for process descriptors (a free list of task_struct structures, etc)
  - kmalloc() is actually implemented on top of slab
    - Try to assign memory from the cache rather than new memory
- Each slab contains some number of objects, which are the data structures being cached.
- Each slab is in one of three states: full, partial, or empty.



The kernel stack

- Historically, unlike user-space programs, kernel processes have non-dynamic stacks
  - Usually the stack per process is one (normal sized) memory page
  - However, since 2016, new approaches have been suggested, including Virtually mapped kernel stacks, see: https://lwn.net/Articles/692208/

Overall memory architecture in Linux



Source: https://slideplayer.com/slide/11404795/

The Process Address Space

Address Space Layout

- Determined (mostly) by the application
- Determined at compile time
  - Link directives can influence this
- OS usually reserves part of the address space to map itself
  - Upper GB on x86 Linux
- Application can dynamically request new mappings from the OS, or delete mappings
  - Dynamically asks kernel for “anonymous” pages for its heap and stack

Example memory layout

- **ldd** prints the shared objects (shared libraries) required by each program or shared object specified on the command line.
- An example of its use and output is the following:

```
$ ldd /bin/ls
        linux-vdso.so.1 (0x00007ffcc3563000)
        libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f87e5459000)
        libcap.so.2 => /lib64/libcap.so.2 (0x00007f87e5254000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f87e4e92000)
        libpcre.so.1 => /lib64/libpcre.so.1 (0x00007f87e4c22000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f87e4a1e000)
        /lib64/ld-linux-x86-64.so.2 (0x00005574bf12e000)
        libattr.so.1 => /lib64/libattr.so.1 (0x00007f87e4817000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f87e45fa000)
```

The Process memory areas

- Memory areas contain, for example
  - A memory map of the executable file's code, called the text section.
  - A memory map of the executable file's initialized global variables, called the data section.
  - A memory map of the zero page containing uninitialized global variables (called bss section)
  - A memory map of the zero page used for the process's user-space stack
  - An additional text, data, and bss section for each shared library, such as the C library and dynamic linker, loaded into the process's address space
  - Any memory mapped files.
  - Any shared memory segments.
  - Any anonymous memory mappings, such as those associated with malloc()

Virtual memory areas

- Linux represents portions of a process with a vm_area_struct, or vma
  - Includes:
    - Start address (virtual)
    - End address (first address after vma)
    - Protection (read, write, execute, etc)
- Defined in https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h

The Memory Descriptor

- The kernel represents a process's address space with a data structure called the memory descriptor.
  - contains all the information related to the process address space.
  - represented by `struct mm_struct` and defined in https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h
  - Very well documented in the code above
  - The memory descriptor associated with a given task is stored in the `mm` field of the task's process descriptor.
  -

The Page Cache and Page Writeback

The page cache

- RAM can be orders of magnitude faster than disk
- The page cache consists of physical pages in RAM, the contents of which correspond to physical blocks on a disk.
  - The size of the page cache is dynamic; it can grow to consume any free memory and shrink to relieve memory pressure.
  - the storage device being cached is called the backing store because the disk stands behind the cache as the source of the canonical version of any cached data.
- Entire files need not be cached; the page cache can hold some files in their entirety while storing only a page or two of other files. What is cached depends on what has been accessed.

Write Caching

- One of three strategies
  - No-write: Cache gets invalidated and needs to be repopulated from disk
  - Write-through-cache: Update both memory and disk together keeping the cache coherent
  - Write-back: default policy in Linux
- The write-back caching policy requires that a write operation occurs at the cache only
  - The backing store is not immediately or directly updated. Instead, the written-to pages in the page cache are marked as dirty and are added to a dirty list.
  - Periodically, pages in the dirty list are written back to disk in a process called writeback, bringing the on-disk copy in line with the in-memory cache.
  - The pages are then marked as no longer dirty
- Write therefore can be performed in bulk, optimizing access to the slow disk
- Application can force immediate write back with sync system calls (and some open/mmap options)

The Linux Page Cache

- A page in the page cache can consist of multiple non-contiguous physical disk blocks
- the kernel must check for the existence of a page in the page cache before initiating any page I/O,
  - the overhead of searching and checking the page cache could nullify any benefits from the cache
  - Thus some parts of the search is implemented as an efficient Radix-Tree

Cache reclamation

- Kernel caches and processes can continue assigning memory until memory becomes scarce
  - Low memory, hibernation, free memory below a “goal”
- Memory pages can be divided into one of four categories
  - Unreclaimable – free pages (obviously), pages pinned in memory by a process, temporarily locked pages, pages used for certain purposes by the kernel
  - Swappable – anonymous pages, tmpfs, shared IPC memory
  - Syncable – cached disk data
  - Discardable – unused pages in cache allocators

Cache eviction policies

- Least Recently Used
- The two list strategy
  - Linux keeps two lists: the active list and the inactive list.
  - Pages on the active list are considered “hot” and are not available for eviction.
  - Pages on the inactive list are available for cache eviction
  - Pages are placed on the active list only when they are accessed while already residing on the inactive list.
  - The lists are kept in balance
  - Approach is also known as LRU/2; it can be generalized to n-lists, called LRU/n