1 Virtual Memory

Virtual memory provides 3 important abilities:

1. It uses the main memory efficiently by treating a cache for a namespace stored on the disk.
2. It simplifies memory management by providing each process with a uniform address space.
3. It protects the address space of each process from being corrupted by other processes.

1.1 Introduction

1.1.1 Virtual address space

The main memory is organized as a M-byte consecutive array. Each byte has a unique **physical address** starting from 0. CPU can access memory with its physical address, namely **physical addressing**. Modern processors use **virtual addressing** to access memory. CPU generates a **virtual address** to access memory. The VA is translated into a physical address by the **memory management unit(MMU)** on the CPU chip before being sent to the main memory. CPU generates VA inside a **virtual address space** $\{0, 1, 2, \ldots, N-1\}$ with $N = 2^n$. Similarly there is a **physical address space** $\{0, 1, 2, \ldots, M-1\}$. We assume $M = 2^m$.

1.1.2 VM as cache

Conceptionally, VM is organized as a N-byte consecutive array on the disk. Each byte has a unique VA. Data on the disk is cached inside the main memory. Like other caches in the memory hierarchy, data on the disk is divided into blocks to serve as basic units for transportation between the disk and the main memory. The virtual memory is divided into **virtual pages(VP)** of size $P = 2^p$, and the physical memory is divided into **physical pages(PP)**, or **page frames** of the same size. At any time, the set of VPs is divided into 3 subsets:

- Unallocated: pages not yet created by the VM system. Not related to any data, thus occupies no disk space.
- Cached: Allocated and cached in the physical memory.
- Uncached: Allocated but not cached in the memory.

Cost of DRAM cache miss is enormous:

- DRAM 10 times slower than SRAM; hard disk 100000 times slower than DRAM.
- Cost of reading the first byte from a disk sector is 100000 times higher than that of reading further bytes consecutively.

Properties of DRAM cache: large pages (4KB-2MB); full associative (any VP can be put in any PP); write-back instead of write-through.

Page table is used to tell if a VP is cached inside a PP; if so, which PP; if not, where is it on disk. It's an array of PTE(page table entry)s stored in memory and managed by the OS. Each VP is related to a PTE in the page table. A PTE has a valid bit and an n-bit address segment, which is used to indicate the starting address of the PP in DRAM if the valid bit is set (1), or the starting address of the PP on disk if the valid bit is unset (0). If the VP is unallocated, the valid bit is 0 and the address is set to null.

A DRAM cache miss is called a page fault. An exception handler in the kernel will be called. The needed VP will be copied into a PP at the cost of a sacrificed page. The instruction that caused the page fault will be executed again.

The efficiency of VM is guaranteed by locality. Programs tend to work mainly on a small set of pages called the working set. After some initial cost, page fault is supposed to be rare.

1.1.3 VM as tool for memory management

- VM simplifies linking. A separate virtual address space allows each process to use the same basic format for its memory image, e.g. `.data` always starts at 0x400000. The implementation of the linker is greatly simplified.
- VM simplifies loading. Loader does not have to copy any data from disk to memory. All it does is to allocate VPs for `.text` and `.data` segments of the executable and mark them as uncached. They will be paged in by the VM automatically on demand.
- VM simplifies data & code sharing. Different processes can share data & code, e.g. C standard libraries and kernel codes, by mapping VPs in their virtual address spaces to the same PPs.
- VM simplifies memory allocation. Consecutive VPs do not have to be mapped to consecutive PPs.

1.1.4 VM as tool for memory protection

- Separate virtual address space makes it easy to isolate private memories of different processes.
- Additional control bits can be added to PTEs for memory protection. For example, a SUP bit can be used to restrict access of the VP to super user; READ/WRITE bits can be used to allow and forbid processes from reading and writing the VP. If such restrictions are violated, the CPU triggers a protection fault, which is reported as "segmentation fault" in Linux.

1.2 Memory Translation

Memory translation is a map from the virtual address space to the physical address space:

$$MAP : VAS \to PAS \cup \emptyset$$

$$MAP(A) = \begin{cases} A' & \text{If data at VA A is at PA A'} \\ \emptyset & \text{If data at VA A is not in physical memory} \end{cases}$$

PTBR Page table base register: a control register in the CPU. Points to the current page table.

VPN Virtual page number.

VPO Virtual page offset in bytes. Last p bits of VA.

PPN Physical page number.

PPO Physical page offset in bytes. Last p bits of PA. Equal to VPO.

Page hit:

1. CPU generates a VA and sends it to the MMU.
2. MMU generates the address of the appropriate PTE (PTBR + VPN), and asks SRAM/DRAM for the PTE.
3. SRAM/DRAM returns the PTE to the MMU.
4. MMU obtains the PPN from the PTE, and constructs the PP(PPN + VPN). The PP is sent to SRAM/DRAM.
5. SRAM/DRAM returns the asked word to the CPU.

Page fault:

1. 1-3: the same as page hit.
2. The valid bit of the PTE is 0. MMU triggers an exception. Control is transferred to the page fault exception handler in the kernel.
3. The handler identifies a victim page in the physical memory. If it has been modified, it's paged out to the disk.
4. The handler pages in the new page (whose address on disk is in the PTE) and updates the PTE.
5. The handler returns control to the original process and the instruction is executed again.

TLB Translation lookaside buffer. A small buffer in the MMU dedicated to PTEs. It's usually highly associative. If TLB has 2^t sets, then the least significant t bits of the VPN is used as the TLBI(TLB index) and the rest is used as TLBT(TLB tag).

Multi-level Page Tables One single page table calls for too much memory footprint. With a 32-bit address space, 4KB pages and 4B PTEs, a 4MB page table has to stay in memory. For a 64-bit address space it's even large. The page table can be compacted with a hierarchy of page tables.

As shown in the figure, each entry in the 1st level page table is responsible for a 4MB chunk in the VAP, which is 1024 VPs. If none of the pages in the chunk is allocated, the entry in the 1st level page table is empty. If at least one of them is allocated, the entry points to the address of a 2nd level page table.

1.3 Case study: Intel Core i7/Linux VM system

1.3.1 Intel Core i7 memory translation

- Core i7 uses 48-bit VAS (256TB) and 52-bit PAS (4PB).
- L1 d-cache: 32KB, 64B block, 8-way associative, thus 64 sets. PA: 40-bit CT, 6-bit CI, 6-bit CO.

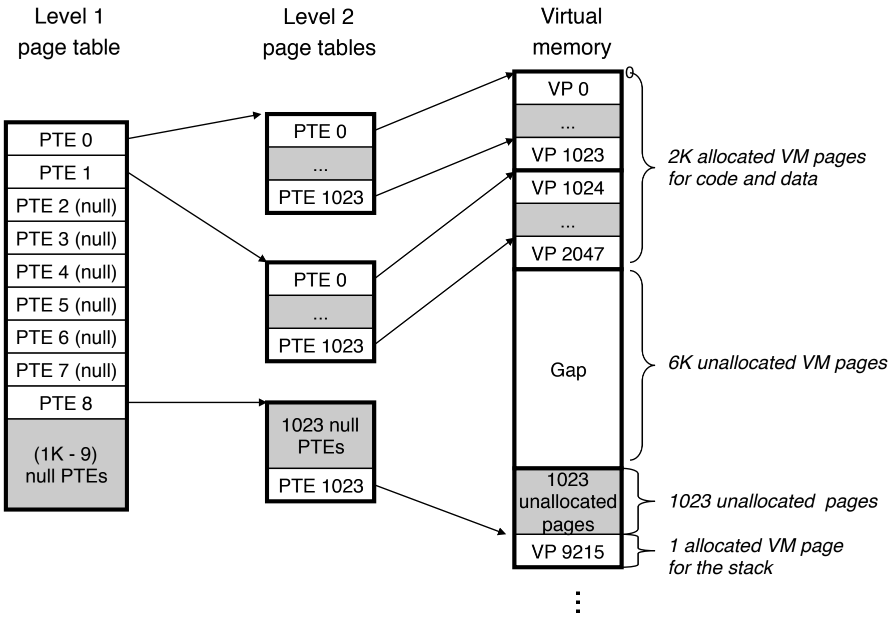


Figure 1: A 2-level page table hierarchy

- 4-level page table. VA: 4× 9-bit VPNs + 12-bit VPO. Level 1: 512GB each entry. Level 2: 1GB. Level 3: 2MB. Level 4: 4KB (page size).
- Parallel address translation: 12-bit VPO = 12-bit PPO = 6-bit CI + 6-bit CO, thus PPN = CT. During address translation, VPN is transferred to MMU while VPO is transferred to L1 d-cache. When MMU asks TLB for a PTE, L1-cache finds the appropriate set according to CI and reads the 8 tags and data words (according to CO). After the MMU obtains PPN from TLB, it simply compares it against the 8 tags and choose the word whose tag matches (in case of a cache hit).

1.3.2 Linux VM system

Linux maintains a separate VM space for each process. VM is organized as a collection of areas(segments). An area is a contiguous chunk of existing (allocated) VM whose pages are related in some way. Each existing VP is contained in an area. Any VP that is not part of some area does not exist, cannot be referenced by the process, and does not consume any additional resources.

Kernel data structures used to keep track of VM areas of a process:

- A separate task struct for each process (`task_struct`). Its elements either contain of point to information needed by the kernel to run the process: PID, pointer to the user stack, name of the executable file, PC, etc.
- One of the entries points to an `mm_struct` that characterizes the current state of the VM. It contains:

  - `pgd`: base of the level 1 page table.
  - `mmap`: a list of `vm_area_structs`, each of which characterizes an area.

- `vm_area_structs` contains:
  - `vm_start`: points to the beginning of the area.
  - `vm_end`: points to the end of the area.
  - `vm_prot`: read/write permissions for all pages in the area.
  - `vm_flags`: whether pages in the area are shared with other processes, etc.
  - `vm_next`: points to the next area struct in the list.

Linux page fault exception handling:

1. Check if the VA is legal, i.e. if the VA is contained in an area, by comparing it against `vm_start` and `vm_end` in each area struct. If illegal: segmentation fault.
2. Check if the access is legal, i.e. if the process has enough permissions to read-/write/execute the pages in the area. If illegal: protection exception.
3. Page out victim page and page in the new page. Re-execute the instruction.

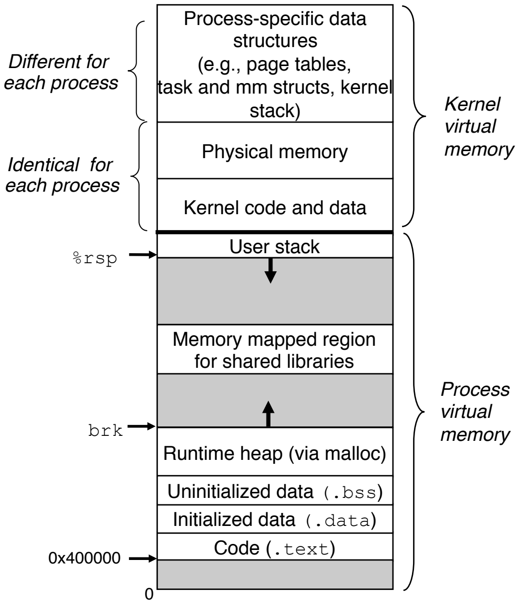

Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Different for each process
Identical for each process
Kernel virtual memory
User stack
%rsp
Memory mapped region for shared libraries
brk
Runtime heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Code (.text)
0x400000
0
Process virtual memory

Figure 2: VM of a Linux process

1.4 Memory mapping

Linux **initializes**[1] the contents of a VM area by associating it with an object on disk. The process is called memory mapping. Areas can be mapped to two kinds of objects:

- Regular. A contiguous section of a regular disk file. The section is divided into page-size pieces, each containing the initial contents of a VP. Because of demand paging, no VP is actually swapped into physical memory until the page is touched by the CPU.
- Anonymous. An anonymous file created by the kernel containing only binary zeros. When the page is first touched, a victim page in the physical memory is overwritten with 0. No data is transferred between disk and memory.

[1] Further changes made to VM areas are not necessarily reflected back to the original objects on disk. See below (private areas).

Once a VP gets initialized, it is swapped back and forth between the physical memory and the swap file on disk[2]. The size of the swap file bounds the number of VPs that can be allocated by the running processes.

Let's revisit some concepts introduced before.

Shared Object An object can be mapped to an area of VM as either a shared object or a private object. Any writes made by an process to the area in its VM mapped to a shared object are visible to other processes that also map the shared object into their VMs, and are reflected in the original object on disk. On the contrary, changes made to an area mapped to an private object are not visible to other processes. Neither are they reflected back to the object on disk.

Private objects are mapped into VM using copy-on-write technique. Multiple processes can map the same private object on disk into their VMs. Related PTEs are flagged as read-only, and related area structs are flagged as **private copy-on-write**. Only a single copy of the private object is kept in the physical memory until either process attempts to write to some page in the private area, when a protection fault is triggered. A new copy of the page will be created in the memory, with the related PTE updated. The process will have write permission to the new page.

fork When the current process calls the `fork` function, the kernel creates various data structures for the new process and assigns it a unique PID. In order to create VM for the new process, it create exact copies of the current process's `mm_struct`, area structs and page table. All pages in both processes are flagged as read-only, and all area structs are flagged as private copy-on-write. When either process performs writes, new pages will be created by the copy-on-write mechanism. The abstraction of a private address space for each process is preserved.

execve `execve("a.out", NULL, NULL);` loads and runs the program contained in the executable object file `a.out` within the current process, effectively replacing the current program with the `a.out` program. This is accomplished in the following steps:

- Delete existing area structs in the user portion of the current process's VM.[3]
- Map private areas. New area structs are created for the code, data, bss, stack and heap areas of the new programs, all of whom are private copy-on-write. Code and data areas are mapped to `.text` and `.data` sections of `a.out`. Bss is mapped to an anonymous file whose length is contained in `"a.out"`. Stack and heap are mapped to an anonymous file of initial length 0.
- Map shared areas. Shared objects such as `libc.so` are mapped into the shared region of the VM.
- Set the PC of the current process's context to the entry point in the code area.

Linux processes can use the `mmap` function to create new VM areas and map objects to these areas.

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void* start, size_t length);
```

[2] Looks like an error in the book. "Swapped between a swap file"?

[3] Maybe also the page table?

`mmap` asks the kernel to create a new VM area mapped to a contiguous chunk of the object specified by file descriptor `fd`. The size of the chunk is `length` and it starts from `offset` bytes from the beginning of the file. `start` is the suggested beginning address of the new area, usually specified as NULL. `prot` contains access permissions of the new area (i.e. `vm_prot` in its area struct). It can be

- PROT_EXEC: pages within the area are composed of instructions executable by the CPU.
- PROT_READ: pages within the area are readable.
- PROT_WRITE: pages within the area are writeable.
- PROT_NONE: pages within the area cannot be accessed.

`flags` describes type of the object. They can be ORed with |.

- MAP_ANON: anonymous object.
- MAP_PRIVATE: private copy-on-write.
- MAP_SHARED: shared object.

`munmap` deletes an area created by `mmap`.

1.5 Dynamic memory allocation

Dynamic memory allocation is necessary because we often do not know the size of some data structures until runtime.

1.5.1 Requirements

- Handle arbitrary sequence of requests. No assumption should be made about the sequence of allocate / free requests except that each free request should correspond to a previous successful allocate request.
- Reply to requests immediately. The allocator is not allowed to reorder or buffer requests to improve efficiency.
- Use only the heap.
- Align the blocks so that any data object can be stored.
- Leave allocated blocks unmodified. The allocator can only manipulate or change free blocks.

1.5.2 Performance goals

- Maximize throughput, i.e. requests handled per minute. Reasonable performance: the worst running time of an allocate request is linear in the number of free blocks, while the free request can be finished in constant time.
- Maximize memory utilization. Peak utilization:

  $$U_k = \frac{\max_{i \leq k} P_i}{\max_{i \leq k} H_k}.$$

  P_k is the aggregate payload after request R_k and H_k is the size of the heap (the heap can grow or shrink).

1.5.3 Fragmentation

There is enough memory left but it cannot be used to fulfil an allocate request. Internal fragmentation: an allocated block is larger than the payload, usually to fulfil aligning requirement. It can be quantified. External fragmentation: no single block is large enough for an allocate request, though there is enough aggregate free memory. It's hard to quantify because it depends on future allocate requests.

1.5.4 Implicit free lists

Any allocator needs some data structure to tell the boundaries of blocks and to tell allocated blocks from free ones. Most allocators embed this information in the block itself.

- Implicit free list uses a word (4B) at the beginning of each block as its header.
- The lowest bit is used to indicated if it's allocated (1) or free(0).
- The highest 29 bits are used to indicated its size (8B alignment is required so that the lowest 3 bits are not needed).
- A special ending block is needed, e.g. one with size 0 and flagged as allocated.
- Simple. Searching within free blocks takes time linear in the number of all blocks.

1.5.5 Placement policies

- First fit: Search from the beginning and use the first suitable free block. Leaves large free blocks at the end of the list, but tends to leave small free blocks near the beginning of the list, which increases the time to search for large free blocks.
- Next fit: Search from where the last query ends and use the first suitable free block. Faster than first fit but results in lower memory utilization.
- Best fit: Check each free block and choose the smallest appropriate one. Requires thorough search of the heap.

1.5.6 Splitting free blocks

If the chosen free block is larger than the requested size, the allocator can split it into 2 parts: an allocated block and a new free block.

1.5.7 Obtaining additional heap memory

If no appropriate free block is available, the allocator has to call the `sbrk` function to ask the kernel for more heap memory.

```
/* Increase pointer brk by incr bytes. Returns the old brk. When incr is negative,
    it shrinks the heap. */
#include <unistd.h>
void *sbrk(intptr_t incr);
```

The obtained memory is turned into a large free block at the end of the list.

1.5.8 Coalescing free blocks

Fault fragmentation: a series of contiguous small free blocks. They should be coalesced into a large free block.

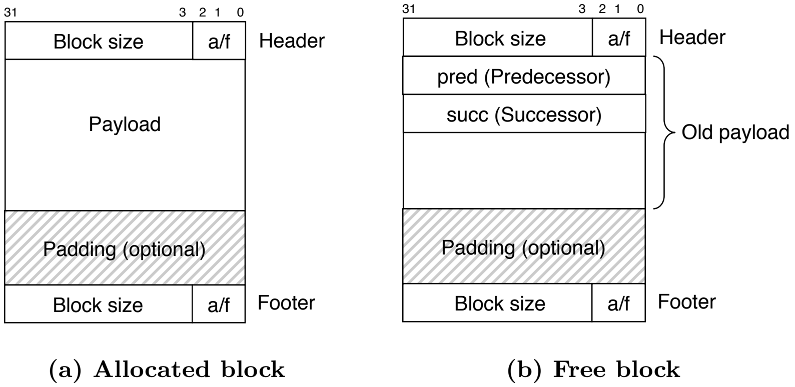
- Immediate coalescing: each time a block is freed, coalesce neighbouring free blocks. Simple but might cause thrashing.
- Deferred coalescing: coalesce sometime later, e.g. after an allocate request fails.

Coalescing the next free block is simple. To coalesce the previous one, **boundary tag** is needed. A copy of the block header is put at the end of the block as a footer, so that the allocator can decide to coalesce the previous block or not by checking its footer. Boundary is elegant but causes memory overhead, especially when blocks are small. The method can be optimized by eliminating footers of allocated blocks: store the allocated/free bit of the previous block inside an excess low bit of the current block, so that we no longer have to check its footer when it's allocated.

1.5.9 Explicit free lists

Implicit free list: allocation time linear to total number of blocks.

Explicit free list: allocation time linear to total number of free blocks.



(a) Allocated block

(b) Free block

Figure 3: Format of heap blocks using doubly linked free lists