Lecture 10 - Virtual Memory

Gidon Rosalki

2025-06-08

1 Introduction

In a modern operating system, even when not running many programs personally, it is not unusual to have hundreds of processes running. Each process uses memory which is reserved by the OS. Additionally, should one open a program such as Chrome, or Excel, it is not unusual to see a process use more memory than is physically installed in the PC. Also, how can so many processes share this memory? Often with an overall usage that is many times the total available physical memory? How come processes do not interfere with each other? These questions are all resolved by **virtual memory**.

1.1 Memory management

Before this week, the whole machine ran a single program. Resources were managed manually by the developer, and there was no external software with unknown requirements. Now, we use multi tasking. This is an effective way to share resources between processes, and allows the dynamic coexistence of many processes. When asking how many processes share memory, we need to consider the following:

- What if a processes tries to access the memory of another process?
- What if a process needs more memory, and none is available?
- What about processes that are not known to each other?
- What happens when a new process is spawned?

1.1.1 Requirements

We have the following requirements:

- Scalability: It should be possible to change the amount of memory available for a process, maybe even to more than the memory hpysically installed in the machine.
- Isolation: Different processes should be separated, due to concerns of security, interference, protection, and so on.
- Orthogonality: Each process should see the same address space (0x0000 - 0xffff), and thus be completely unaware of each other. In other words, each process runs inside its own sandbox, in which it thinks that there are no other processes on the machine.

1.1.2 Solution - virtual and physical memory

Instead of using the physical memory directly, the programmer accesses the **virtual address space**. This is a perfect abstraction, in which memory is contiguous, and completely belongs to a process. This can be the complete memory space (0x0 - 0xffffffff). This is in contrast to the **physical address space**, which is the real computer memory (DRAM), used by several processes. It is hidden from the programmer by the OS and by hardware.
This leaves us with the question, how to map the virtual address space to the physical address space?

1.1.3 Virtual memory

One process has access to one virual address space. We split both the physical and virtual address spaces into blocks of fixed size (usually 4KB) called pages. Only pages used by the process are stored ("mapped") in computer memory, since if a page is not being used, there is no need to map it from the virtual space. Pages can be in physical memory, or on the disk. Memory acts as a cache for the secondary storage, the disk. Processes may therefore use more pages than physical memory can store.

2 Virtual memory

2.1 Memory hierarchy

2.1.1 Scalability

We use the principles of caching to get the best of both worlds. We get the speed of fast and expensive memory, but with the size of cheap, slow memory. As we learnt previously, caches often offer speeds that approach that of an SRAM cache, with the size of a DRAM cache. We can use the same principles the next level up, the speed of DRAM memory, with the size of disk memory. We need to check if we can apply the same techniques:

- Where do we store the tag and valid bits?
- How do we handle associativity?
- What should we use as the replacement policy?

2.1.2 Isolation

This is security and protection for processes: It allows multiple processes to simultaneously occupy memory, and provides protection between them, provides the separation of memory that belongs to each process, and denies the ability for a process to read/write to the memory of another process. It also protects the OS space, by providing a separation of OS memory from application memory.

2.1.3 Orthogonality

This provides each program the illusion as if it has its own private memory. Each program has a different view of the memory, for example, in two programs, they can have the code begin at the same address, such as 0x40000000, but each program has its own code in memory.

2.2 Mapping

2.2.1 Terminology

- Physical memory: The computer's main memory, indexed using physical addresses.
- Virtual memory: The program's memory, accessed by a virtual address
- Page: A unit of memory allocated and mapped by the virtual memory translation (also called frame). This is equivalent in some ways to a block in cache.
- Page table: A table in physical memory, that holds the translation from virtual memory to physical memory
- TLB: Translation Look-Aside Buffer: A table (cache) of virtual to physical address translations.

2.2.2 Virtual to physical address translation

Each program operates in its own virtual address space, and one program runs on one processor at a time. Each program is protected from other programs, and the OS decides which physical memory each program can use, and how it is mapped. This control is provided by page tables, maintained by the OS. The hardware does the virtual to physical translation, using page tables provided by the OS.

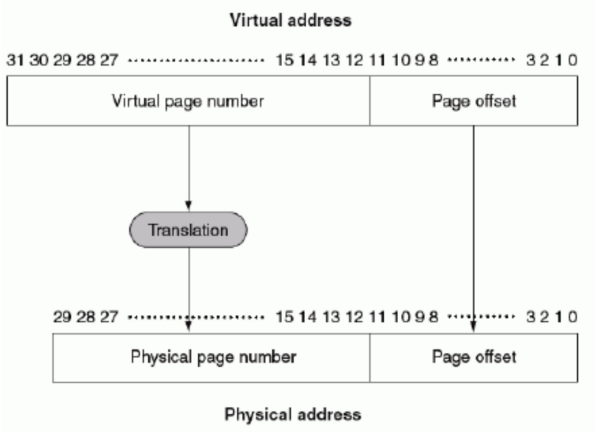

Figure 1:

A simple function cannot predict arbitrary mapping, so instead we use a table for the mappings, (virtual address to physical address translation). This is called the page table, and the page number is the index in the table. In the virtual memory mapping function, the virtual offset is the physical offset, and we can acquire the physical page number by indexing the table, with the virtual page number.

The page table has an entry for each possible virtual page. Let us consider an example page table memory usage, with 4K pages. We have 32 bits of virtual memory (4GB), and 30 bits of physical addresses (1GB of physical memory). The page size if 4KB (12 bits page offset). Each table netry requires 18 bits (PPN), and also 1 balid bit. We will round this up to 32 bits, since the entry size is always rounded to bytes, or words (4 bytes in our case). This is for ease of page table entries handling in memory (for software). The tab;e size for a full 32 bit virtual address space is

$$4 \cdot 2^{20} = 4\text{MB}$$

(4 bytes $\cdot 2^{20}$ virtual pages). If we have 16 processes, we will need 64MB of memory to hold their pages in memory. Now, this is quite a lot. Can we reduce this?

Let us first consider an example address mapping, where we have 32 bits of virtual memory, 30 bits of physical address, and a page size of 4KB (12 bits page offset). Given a virtual address $0\text{x}00045678$, this means a virtual page number $0\text{x}56$, and a page offset of $0\text{x}6787$. So, page table entry number $0\text{x}45$ will contain the PPN.

2.2.3 Page table

A page table is a structure, owned by the OS, which contains the mapping of virtual addresses to physical addresses. There are several different ways, all up to the OS, to manage this table. Each process has its own page table, and the page table contains Page Table Entries (PTE) which indicate the following:

- Whether the page is in memory, or on the disk
- The physical address of the page, if it is in memory, and the disk locator, if it is on the disk.
- Whether the data was modified (for evictions, if it was modified, then the modified data needs to first be written to the disk, whereas if it was not, it may be removed without concern)
- Access rights (read-only, read-write, etc. for security).

As said above, this is all managed by the OS, and the OS does all the complex tasks, such as initialisation, miss handling, and evictions. Teh hardware uses the page table to translate virtual addresses into physical addresses.

2.2.4 Translation failures

There are a few different possible translation failures. If we try and access an invalid virtual address, tehn this will cause a "segmentation fault", since no page table entry was allocated by the OS for this virtual address.
An invalid access will cause an exception, called "protection fault". Fore example, attempting to execute data from a data page, or by attempting to write to a read only page.
If a page is not in physical memory, then this will cause a "page fault". The hardware saves the current state, and gives control to the OS. The OS detects the page causing the fault, and evicts some page if no free pages are left in physical memory. The OS then loads the page from the disk to the physical memory, and updates the PTE (with the physical address, and the on disk bit). Finally, the OS gives control back to the hardware.

When there is not enough memory, the OS can move pages to the disk. When valid = 1, then the page is in memory, and when valid = 0, then the page is on the disk. Mapping of pages to the disk is done by the OS, using specific structures. Once a page is moved to disk, the physical memory can be re-assigned to another virtual page.

2.2.5 Performance

Performing a load, or a store, now requires an additional memory access. 1 access to the page table to translate the virtual address into a physical address, and 1 access to the actual memory. This is 2 physical memory accesses, which is **slow**. Let us observe, that since there is locality in data accesses, there is therefore also locality in virtual address translation. We can use a small cache of virtual to physical address translations to make translation faster, in which we cache the relevant information from the page table entry. For historical reasons, this cache is called the Translation Lookaside Buffer, or TLB

2.3 TLB

The page table is too large to be stored in hardware, and so it is stored in memory. Each translation requires one or more memory accesses. Translation is needed for every load, store and instruction fetch, which is too many. Caches help, but insufficiently. TLB caches recently used translation (PTEs: Page Table Entries). This speeds up translation, and normally contains 64 - 256 entries. The TLB access time is typically faster than cache access time.

In many cases, the TLB is fully associative. The translated address (physical address) is then used for accessing data. The look up is in caches, and on a miss, goes to memory. The physical address used for cache search, and memory access, is the same.

The TLB is a cache of page table mappings. Teh access time is comparable to that of the cache, i.e. significantly less than main memory access. The dirty bit indicates that the page was modified in memory (Stored to). This is updated by the CPU, and used by the OS to determine wheter to write the page back to disk when replaced (write back). Access rights / permissions, and other bits are typically a copy of the PTE bits, such as write protection, and user/OS data. There is also optionally the LRU bits, for TLB replacement. Here is a table example:

Valid	Tag (VPN)	PPN	Dirty	Access writes and misc.

Table 1:

2.3.1 TLB miss

What if a page translation is not in the TLB? Then we have two options:

1. Software managed TLB (used by MIPS): Here the hardware traps to the OS (page fault), and it is up to the OS to decide what to do. The CPU does not know the page table format, and the OS must know the TLB entry structure to set it up. The TLB entry requires a Dirty bit to support OS page swapping.
2. Hardware managed TLB: The hardware loads information from the PTE to the TLB. If PTE.valid = 0, i.e. the page is not present, then the CPU will trap to the OS to handle a page fault, and the OS will load the page from the disk into memory (potentially replacing another page). The CPU **must** know the page table format, and the OS need not know the TLB entry structure.

Most modern CPUs use the second option, since it is much faster.

2.4 Hierarchical page tables

These reduce the size of the Page Table, especially in the case where the program is small (most cases). The virtual address is 64 bits in size, 12 for the page offset, and 52 for the page number. How can we store 2^{52} PTEs in memory? This is a huge amount. Let us suppose, for the purposes of example, we use a 32 bits virtual address space, or 2^{32} bytes. Assuming 4KB pages, and each PTE is 4B, then we need $\frac{2^{32}}{2^{12}} = 2^{20}$ b page table entries to cover the virtual address space, and $2^{20} \cdot 4 = 2^{22}$ bytes, is 4MB. So what happens if we have 256 processes in memory, each consuming 16MB of memory? Then we need 4MB $\cdot$256 =1GB of memory. Since each process uses 16MB of memory, then on average each processes only uses 16KB of the page table. A total of 4MB. So, less than 1% of the page table is actually used. With a 64 bit address space, this is impractical to implement, since we'll need 16 million GB of memory, just for the page table. The solution to this is to use Hierarchical page tables:

2.5 Explanation

The page table structures is defined by the ISA. Accessing a given page involves first walking the tables, which each point to each other. This is particularly effective when pages are sparse in the virtual address space. In this system, we split the linear address into an offset, table, and directory.

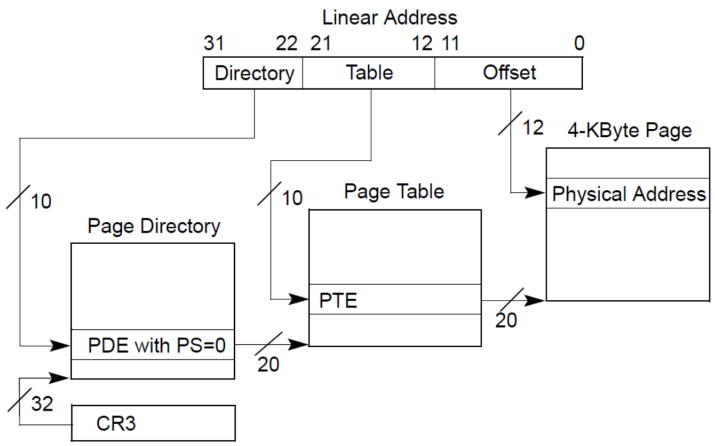

Figure 2: Linear address translation for a 4KB page, using 32 bit paging

2.5.1 Example - page table

Flat page table (no hierarchy):
Our example system will have:

- 4KB pages
- PTE size = 4 bytes
- 32 bit birtual address space
- 30 bit physical address space

Our flat page table will havbe

$$\frac{2^{32}}{2^{12}} = 2^{20}$$

pages, and so our page table memory is $2^{20} \cdot 4 = 2^2 2$ =4MB. So the total memory used for the flat table is 4MB. However, let us consider the system, but with hierarchy.
Usage: 10 pages, address 0x01004000 − 0x01013FFF, 16 pages of 4KB. The page directory will be 2^{10} entries $\cdot 4B = 2^{12}B$ = 4KB. A single page direcotyr is required, so we are using a single entry, entry number 4. The page table has 2^{10} entries $\cdot 4B = 2^{12}B$ =4KB. So we are using 16 entries, or 16 PTEs, used to point to 16 pages, entries 4 to 19. So, the total memory used by the Hierarchical page table: 8KB.