Lecture 8 - Memory management

Gidon Rosalki

2025-05-18

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Memory hierarchy and caching

Until now we have considered memory in 3 parts, the CPU registers at the top of the pyramid, small but very fast, the main memory underneath this, slower but much larger, and beneath this the storage, even slower, but even larger. In reality, the main memory section is split into the L1, L2, and L3 caches, which are only a bit slower than the registers, and then the main memory, which is much slower than those. We can also add remote storage, (in the network) beneath the storage layer. The speed differences vary in order of 10^5, from 0.3ns in the registers, up through 1ns in L1, 3-10ns in L2, 10-20ns in L3, 50-100ns in RAM, and 20-40us in the storage. Recall what was found in exercise 1. The difference in space also varies by even more, where the registers clock in at around 1Kb, L1, L2, and L3 are 64Kb, 256Kb, and 2-16Mb accordingly, RAM tends to clock in order of 10s of GB, and storage in TB.

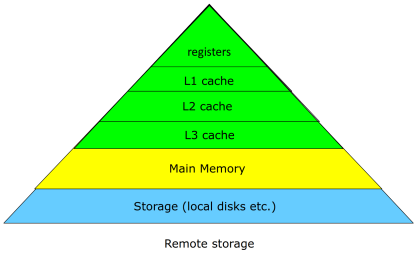

Figure 1: Storage pyramid

To analyse the memory, we review the speed, caching, implementation technology, volatility, core level ownership (shared/private), and management responsibility. We use caches to handle the CPU-DRAM gap, where the processors increased in speed in accordance with Moore's law, 1.5 times per year or so, but RAM only increased in speed by 7% per year. The first caches to compensate for this speed difference were implemented in 1989.

1.1 Caching

The caching concept is that small fast memory serves as a cache for large, slow memory. This leaves us with problems, such as how do we find things in the cache, which items should be stored in the cache, and which items should be evicted from the cache? We will not discuss these with regards to the memory cache (L1-L3 for RAM), but rather for RAM vs Disk.

Caching works because memory access is **not random**. It instead enables the following:

1. Temporal locality: If we accessed a certain address, the chances are high that we want to access it again shortly (for example, when updating data, or in loops)
2. Spatial locality: If we accessed a certain address, then the chances are high that we want to access its neighbours shortly (for example, in arrays, or sequential execution of instructions)

So the general concept here is to keep data and instructions that the CPU is most likely to need next in fast memory close to the CPU.

We can consider the RAM as a cache for the disk. The top few layers of RAM (Registers, L1-L3) are all made of SRAM (static RAM), and the main memory is made of DRAM (dynamic RAM). SRAM retains its value indefinitely, as long as it is kept powered, and is relatively insensitive to disturbances, such as electrical noise. It is also faster, but more expensive than DRAM, and is hence mostly used for registers and caches. DRAM has to be refreshed every 10-100ms, and is more sensitive to electrical noise. It is slower, but cheaper, than SRAM, and is used for main memory.

When reading from the disk, we have platters spinning at a fixed rotational rate, with a read/write head attached to the arm, that moves across the platter. By moving radially, the arm can access any part of the disk.

1.1.1 Cache details

The disk is non volatile, and everything above it in the pyramid is volatile. This means that if the power is turned off, then the non volatile data is lost. If we consider a multi core system, then everything L2 and above is private per core, and everything else is shared between the cores. If we consider what is the responsibility of whom, the compiler is responsible for the registers, the hardware for L1-L3, and the OS for the main memory and the disk.

2 Process address space

All memory is addressable by the program, but how, and how much, depends on the architecture. On a 32 bit architecture, it cannot address more than 4GB of memory. Some has to be set aside for the OS. Windows left about 2GB for user processes, and Linux left about 3GB (linux ftw). 64-bit architecture uses 48-bit addresses, and so can access 256TB of memory. Generally, half is set aside for the user processes, and half for the OS. Note, this is **for each and every process**. The address space is divided into 4 segments, with different uses. The stack, heap, data, and text. The data and text are both of fixed size, placed at the bottom of the memory (address 0, remember, as we store more, it tends to grow **upwards**). The heap is placed above them, and grows downwards towards the stack, and the stack is placed at the end of the memory, and grows up towards the heap.

Programs need to be loaded to main memory before being executed. Most addresses are known only after the load module (e.g., EXE file) is loaded into the main memory, since the load module contains relocatable addresses.

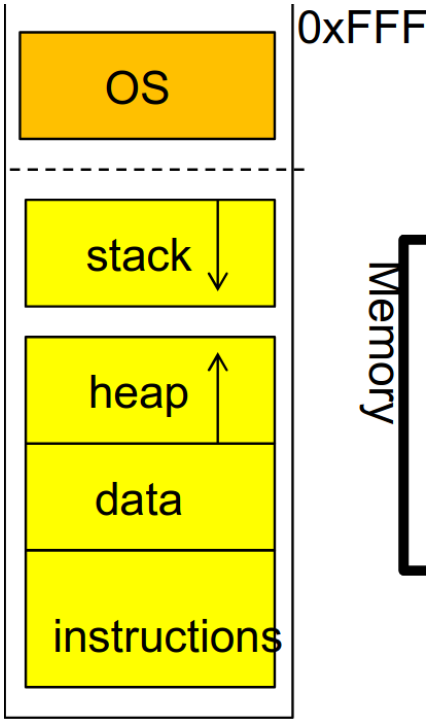

Figure 2: Memory segments

3 Memory management

We discussed that we need to translate logical positions to actual memory addresses. This is handled by the Memory Management Unit (MMU) .The CPU sends virtual addresses to the MMU, which then sends physical addresses to the memory. We need to consider how to map a logical process address to a physical RAM address, the separation of process address spaces, and how to handle the size gap between logical address space, and physical RAM.

3.1 Contiguous allocation: base + bound

This was used about 50-60 years ago. Each process address space as a continuous block in physical memory. The addresses are translated by each process getting a **base** address, and the logical address x in the process is mapped to the physical address **base** + x. This leads to a very simple MMU implementation. However, since we need to allocate space to grow for the stack and the heap, this can lead to fragmentation.

So what is the role of the OS here? Memory access is done by hardware, at cycle speed, the CPU issues the request, and the MMU translates the address. We have not called the OS here in the slightest. The OS decides **where** to map each process, and manages the physical RAM allocation, loading the base + bound into HW registers.

Each process needs a contiguous region of memory, have different sizes, when a process starts, memory is allocated for it, and when it terminates, the memory is freed. However, all of this results in **fragmentation**:

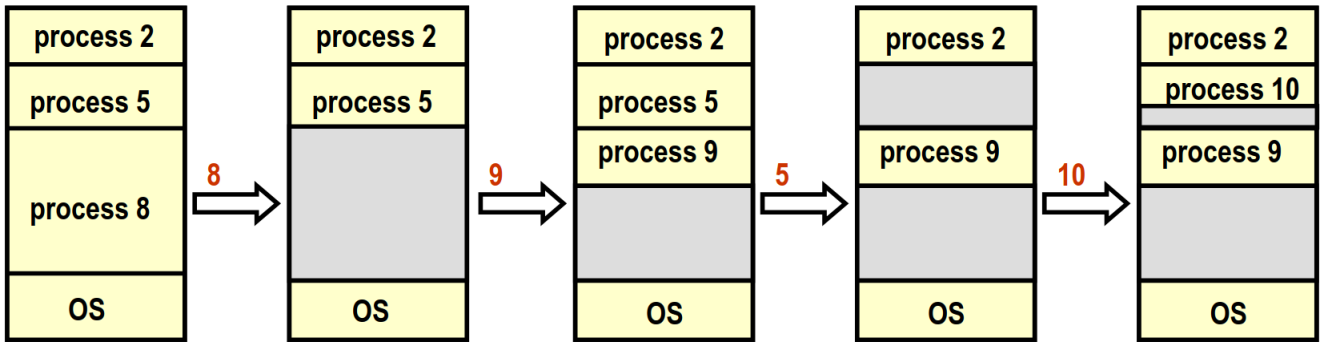

Figure 3: Fragmentation

Fragmentation is a situation in which we have enough memory to allocate to a process, but not contiguously, as the holes are scattered all over the memory. Internal Fragmentation is when we have free memory inside process' allocation, and External Fragmentation: free memory between processes' allocations. We will **only** try to solve External Fragmentation.

3.1.1 Allocation algorithms

Allocation algorithms take
input: List of current free ranges and Request for new allocation (size)
output: Decision which free range to use. Can use only part.

Best fit: Scan the and select the range that provides the snuggest fit. This runs in linear time (in the process list size), maintains large ranges, and causes small fragments. This has the pro of good fragment optimisation, but the con of being slow.

First fit: Scan the list and select the first range that fits. This runs in linear time, with a constant < 1. It may, however, cause excessive fragmentation at low addresses. It has the pros of being fast, but the cons of being likely to fragment low addresses.

Next fit: Scan the list *starting from where you left off last time*, and select the first range that fits. This runs in linear time, with a constant < 1, and spreads the fragmentation out more evenly. It comes with the pros of being fast, but the cons of being worse than best fit.

3.1.2 Fragmentation solution

Compaction: We move around data to unify holes, and create large enough holes to accommodate future processes. This has the problem of being a very costly operation, and we have to be careful with data copying (up vs down). However, we are still left with the problem of there not being enough physical memory for a full address space of all processes.

3.2 Summary

To summarise, we access the base address, plus the logical address, and place memory using a next fit policy. Two processes cannot access each other's memory, which is good, however, the size gap is unsolved. Additionally, there is an inherent address space dilemma. We need to announce the size before running, and if it too big, then we will run out of physical memory very quickly. However, if we try for too small, then our processes will run out of logical memory.
Therefore, we see from this, this method is not the best for the modern world.

3.3 Paging

3.3.1 Concept

The process address space is divided into pages. All pages have a fixed size (typically 4KB). The physical memory is (conceptually) divided into frames, with the same size as pages. Any page can be mapped to any frame (this mapping is performed by the OS), and is used by the MMU to access memory. We thus create a page table per process, where each page is mapped to a specific frame in the physical memory. We store each address as `page|offset`, so if we consider the address `6 = 00110`, then this is the same as `001|10`, meaning page 1, with offset of 2. This is then translated by the MMU into `frame|offset`, where the frame stores the corresponding frame in physical memory to which the page refers, and the offset remains the same.

3.3.2 Page size tradeoff

Small pages have less internal fragmentation, but require larger page tables. Let us consider p to be the page size, s the size of the process, and e the size of the entry in the page table, then the overhead can be defined as follows:

$$\text{Overhead} = \left(\frac{s \cdot e}{p}\right) + \frac{p}{2}$$

$$\text{Optimal page size} = \sqrt{2 \cdot s \cdot e}$$

For $s = 1MB, e = 8B$ (64 bit), then $p = 4KB$.

3.3.3 Address translation

Let us consider on a 32 bit example, to make life simple. Divide virtual address into two parts: Page number (top 20 bits), and offset into page (bottom 12 bits). Use page number as index into page table. Get frame number from page table (20 bit). Combine frame number and offset to create the physical address.

We are however left with a problem. Consider a 1MB process, which then requires 2KB per process, and this needs to be stored in memory! This means we need an additional memory access, for each memory access! The solution is to create a Transition Lookaside Buffer (TLB), which is a special cache for the MMU. It is usually fully associative, and small with about 64 entries, made of SRAM. In almost all cases the CPU will request memory locations stored in the TLB, and on the odd occasion there is a miss, then the new locations will be loaded into the TLB (and these are also cached in L1, L2, L3 and so on).

3.3.4 Allocating new pages

The OS keeps track of free frames. When a process requires n pages, it will allocated any n free frames, by

- Removing the frames from the list of free frames
- Copy data to the physical memory
- Create a page table for the process

3.4 Virtual memory

3.4.1 Logical vs Physical memory

Each process thinks that it runs alone, and may access the **entire** physical memory. The number of processes is in general, not limited. Therefore, the required logical memory (sum of all the logical memory of all processes) is **much** larger than the actual physical memory of the computer.

We resolve this given only part of the program and data needs to be in memory for execution. Therefore, the logical address pace can be much larger than the physical address space. This allows for more processes to coexist in the main memory, and allows for more efficient process creation. Additionally, not currently used pages can be stored on the disk, instead of in main memory. This concept will be continued next week.