

Lecture 21 — Memory: Segmentation and Paging

Jeff Zarnett

Memory Segmentation

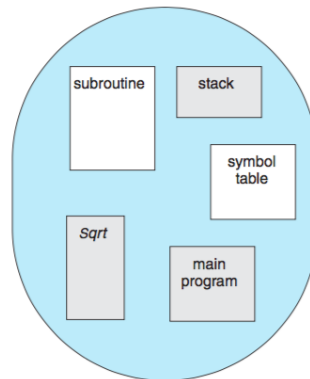
Though you've been repeatedly told that memory is a linear array of bytes, you have also likely been told that there's the stack and the heap, libraries and instructions. Both of these are true; they are simply views of memory at two different levels of abstraction. Each of the elements such as the stack, the heap, the standard C library, et cetera, are known as *segments*.

Programmers do not necessarily give much thought to whether variables are allocated on the stack or the heap, or where program instructions appear in memory. In many cases it does not matter, though C programmers are well advised to know the difference.

A full program has a collection of segments, each of which may be different lengths. Normally the compiler is responsible for constructing the various segments; perhaps one for each of the following [SGG13]:

1. The code (instructions).
2. Global variables.
3. The heap.
4. The stack (one per thread).
5. The standard C library.

From the programmer's perspective, memory may simply be various blocks, as below:



One way programmers might look at memory [SGG13].

Rather than thinking about memory as just a pure address, we can think of it as a tuple: $\langle \text{segment}, \text{offset} \rangle$. Given that, we need an implementation to map these tuples into memory addresses. The mapping has a segment table; each entry in the table contains two values: the base (starting address of the segment) and the limit (the length of the segment). So there will be some addition involved as well as a comparison to see if the address lies within that limit. As is typically the case, memory accesses are such a common operation that we will need another rescue from the hardware folks to make this not painfully slow.



Segmentation hardware [SGG13].

With segmentation, memory need longer be contiguous; we can allocate different parts of the program in different segments; different segments can be located in different areas of memory.

Paging

Fixed and variable sized partitions suffer from fragmentation, whether external or internal. Let us divide memory up into small, fixed-size chunks of equal size, called *frames*. Each process's memory is divided up into chunks the same size as a frame, called *pages*. Then a page can be assigned to a frame. A frame may be empty or may have exactly one page in it.

Imagine, as an analogy, a simple picture frame. The frame may be empty or it may contain a picture. If the picture frame is empty, all that is necessary is to put a picture in it. To put in a different picture, you would first need to take out the picture that is already there. Taking out the picture to empty the frame is allowed, too. A picture is always aligned so that it is completely in one frame; not half in and half out. Now expand this scheme by having a very long row of picture frames, each of which can contain one picture at a time, at most, and a picture can be in at most one frame at a time.

When a process starts, it is loaded into memory, and has some initial memory requirements (e.g., the stack and global variables), so it will require some number of pages. The number of pages can and will change over time as memory is allocated and freed. A process may also be swapped out to disk, but to run it will need to be swapped back in. Either way, a process will take up a certain number of pages in memory at any given time.

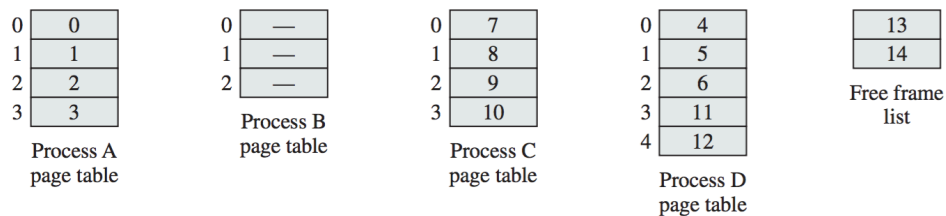
Pages provide the benefit of separating the logical address from the physical address: programmers may pretend the address space of the computer is 2^{64} bytes rather than however many GB of memory are in the physical machine. Any pages that do not fit into frames right now can be kept in a secondary storage medium (that is, on disk) until they are swapped in.

Consider the diagram below, in which there are 15 free frames initially. Then we load process *A*, which consists of 4 frames; then process *B* with 3 frames, *C* with another 3 frames. Eventually, *B* is swapped out (put back on disk) and *D* is loaded. Process *D* has five frames, but the frames do not have to be contiguous (and although they are shown in order, they do not necessarily have to be).



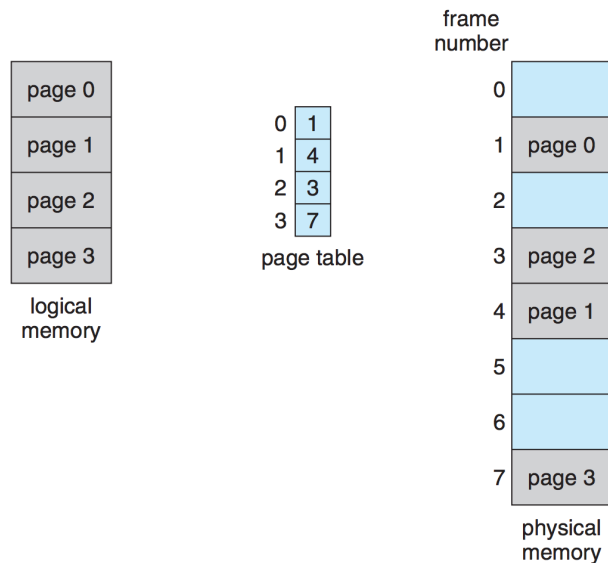
Assignment of process pages into free frames [Sta14].

Now that we have multiple segments for each process and they are no longer contiguous, it is insufficient to have a base address and a limit. Now instead, each process needs a page table, to keep track of which pages are located where in memory. A list of free frames is also necessary. See the diagram below:



The page tables as of (f) in the previous diagram [Sta14].

The page table is used to map logical memory to physical memory, as in the diagram below:



Mapping of logical memory to physical memory via the page table [SGG13].

For convenience, page size is usually a power of 2 (and the actual value is determined by hardware). The selection of a power of 2 makes translating a logical address into a tuple of the page number and offset easy. If the logical address space has size 2^m and the page size is 2^n bytes, n obviously smaller than m , then the high order $m - n$ bits of the logical address are the page number and the lower n bits are the page offset [SGG13].

External fragmentation is eliminated as a problem in this scheme, because pages are all the same size. That also means that compaction is not an issue. Compaction, when it's possible, is painful enough in memory; it is excruciating to do on disk. It is therefore desirable to avoid it entirely. We accept some internal fragmentation because a process gets a whole page at a time.

But how much internal fragmentation do we have to live with? Not very much. If the memory required aligns perfectly with a multiple of the page size, then no memory is wasted. If a new memory allocation comes in, then a new page is allocated and added to the logical memory space of the process. The last frame, however, may not be completely full. In the worst case scenario, a full page less one byte is wasted. However, internal fragmentation of one page is not very much in the grand scheme of things.

How big should page sizes be? If they are smaller, then less memory is wasted in internal fragmentation. However, having a large number of pages introduces a lot of overhead. The size of pages has tended to grow along with the size of main memory in computers [SGG13]. The key factor is actually disk: the disk operates on a certain block size and it is most efficient for the size of a page to be equal to a disk read/write size. That way when a page is to be swapped into or out of memory, it can be done in a single disk read or write. In a typical modern system, pages are 4 KB, but they can be bigger.

Now we finally have a good answer to why the application developer can treat memory as if it is infinitely large and unshared. The program is scattered across physical memory, but appears to the application developer and running application as if it is all contiguous.

We also get protection in this scheme: a program cannot access any address outside of its memory space. There is simply no way to make a memory request outside of the logical memory space. No matter what address is generated, it could only be inside the page table, and the page table has only entries of that process.

The operating system, however, can manage memory of all processes, so it will need another scheme. The OS will operate on the *frame table*, a listing of all the frames, indicating which page of which process a frame currently holds, if any.

Paging: Hardware Support

As we have repeatedly discussed, memory accesses are very frequent and often require additions and comparisons. After all, an operation as simple as adding two numbers requires fetching the add instruction, fetching the operands, and storing the result. To prevent abysmal performance, modern computers have hardware support, because hardware is much, much faster than doing these operations in software.

The simplest implementation of the page table is to use a set of dedicated registers. Registers are the fastest form of storage. When a process switch takes place, these registers, just as all other registers, are replaced with those of the process to run. The PDP-11 was an example of a system that had this architecture. Addresses were 16 bits and the page size was 8 KB. Yes, it was a very long time ago. The page table was therefore 8 entries and kept in fast registers. This might work if the number of entries in the page table is small (something like 256 entries). The page table can easily however, be something like 1 million entries, so it would be a little bit expensive to have that many registers. Instead, the page table is kept in main memory and a single register is used to point to the page table [SGG13].

Unfortunately, this solution comes with a big catch. To access a page from memory, we need to first figure out where it is, so that requires accessing the page table in main memory. Then after retrieving that, we can look in the page table to find the frame where the desired page is stored. Then we can access that page. So two memory accesses are required for every read or write operation. Remember that as far as the CPU is concerned, main memory already moves at a snail's pace. Doubling the amount of time it takes to do a read or write means it takes roughly forever. Thus, we will need to find a way to speed this up.

References

- [SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.
- [Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.