CS 537
Lecture Notes, Part 8
Segmentation

Previous More About Paging
Next Disks
Contents

Segmentation

[Silberschatz, Galvin, and Gagne, Section 9.5]

In accord with the beautification principle, paging makes the main memory of the computer look more “beautiful” in several ways.

- It gives each process its own *virtual memory*, which looks like a private version of the main memory of the computer. In this sense, paging does for memory what the process abstraction does for the CPU. Even though the computer hardware may have only one CPU (or perhaps a few CPUs), each “user” can have his own private virtual CPU (process). Similarly, paging gives each process its own virtual memory, which is separate from the memories of other processes and protected from them.
- Each virtual memory looks like a linear array of bytes, with addresses starting at zero. This feature simplifies relocation: Every program can be compiled under the assumption that it will start at address zero.
- It makes the memory look bigger, by keeping infrequently used portions of the virtual memory space of a process on disk rather than in main memory. This feature both promotes more efficient sharing of the scarce memory resource among processes and allows each process to treat its memory as essentially unbounded in size. Just as a process doesn't have to worry about doing some operation that may block because it knows that the OS will run some other process while it is waiting, it doesn't have to worry about allocating lots of space to a rarely (or sparsely) used data structure because the OS will only allocate real memory to the part that's actually being used.

Segmentation caries this feature one step further by allowing each process to have multiple “simulated memories.” Each of these memories (called a *segment*) starts at address zero, is independently protected, and can be separately paged. In a segmented system, a memory address has two parts: a *segment number* and a *segment offset*. Most systems have some sort of segementation, but often it is quite limited. Unix has exactly three segments per process. One segment (called the *text* segment) holds the executable code of the process. It is generally[1] read-only, fixed in size when the process starts, and shared among all processes running the same program. Sometimes read-only data (such as constents) are also placed in this segment. Another segment (the *data* segment) holds the memory used for global variables. Its protection is read/write (but usually *not* executable), and is normally not shared between processes.[2] There is a special system call to extend the size of the data segment of a process. The third segment is the *stack* segment. As the name implies, it is used for the process' stack, which is used to hold information used in procedure calls and returns (return address, saved contents of registers, etc.) as well as local variables of procedures. Like the data segment, the stack is read/write but usually not executable. The stack is automatically extended by the OS whenever the process causes a fault by referencing an address beyond the current size of the stack (usually in the course of a procedure call). It is not shared between processes. Some variants of Unix have a fourth segment, which contains part of the OS data structures. It is read-only and shared by all processes.

Many application programs would be easier to write if they could have as many segments as they liked. As an example of an application program that might want multiple segments, consider a compiler. In addition to the usual text, data, and stack segments, it could use one segment for the source of the program being compiled, one for the symbol table, etc. (see Fig 9.18 on page 287). Breaking the address space up into segments also helps sharing (see Fig. 9.19 on page 288). For example, most programs in Unix include the library program `printf`. If the executable code of `printf` were in a separate segment, that segment could easily be shared by multiple processes, allowing (slightly) more efficient sharing of physical memory.[3]

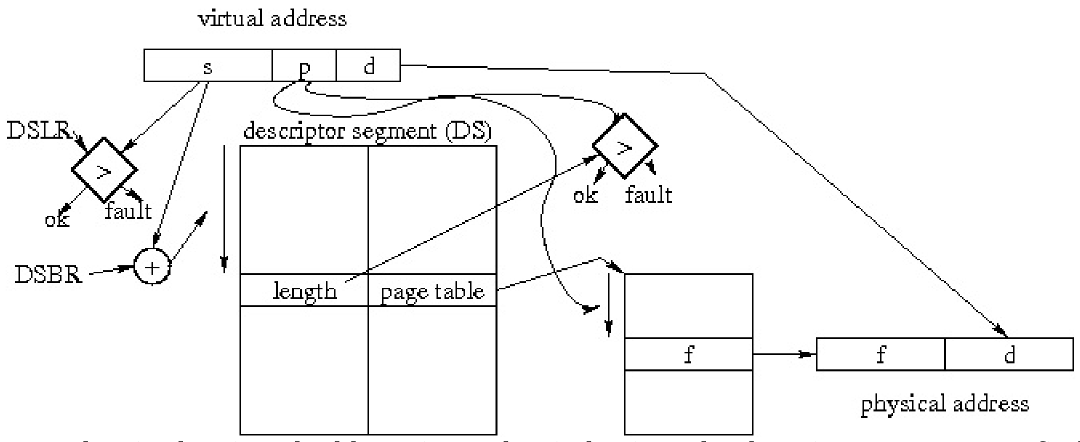
If you think of the virtual address as being the concatenation of the segment number and the segment offset, segmentation looks superficially like paging. The main difference is that the application programmer is aware of the segment boundaries, but can ignore the fact that the address space is divided up into pages.

The implementation of segmentation is also superficially similar to the implementation of paging (see Fig 9.17 on page 286). The segment number is used to index into a table of “segment descriptors,” each of which contains the length and starting address of a segment as well as protection information. If the segment offset not less than the segment length, the MMU traps with a segmentation violation. Otherwise, the segment offset is added to the starting address in the descriptor to get the resulting physical address. There are several differences between the implementation of segments and pages, all derived from the fact that the size of a segment is variable, while the size of a page is “built-in.”

- The size of the segment is stored in the segment descriptor and compared with the segment offset. The size of a page need not be stored anywhere because it is always the same. It is always a power of two and the page offset has just enough bits to represent any legal offset, so it is impossible for the page offset to be out of bounds. For example, if the page size is 4k (4096) bytes, the page offset is a 12-bit field, which can only contain numbers in the range 0...4095.
- The segment descriptor contains the physical address of the start of the segment. Since all page frames are required to start at an address that is a multiple of the page size, which is a power of two, the low-order bits of the physical address of a frame are always zero. For example, if pages are 4k bytes, the physical address of each page frame ends with 12 zeros. Thus a page table entry contains a frame *number*, which is just the higher-order bits of the physical address of the frame, and the MMU *concatenates* the frame number with the page offset, as contrasted with *adding* the physical address of a segment with the segment offset.

Multics

One of the advantages of segmentation is that each segment can be large and can grow dynamically. To get this effect, we have to page each segment. One way to do this is to have each segment descriptor contain the (physical) address of a page table for the segment rather than the address of the segment itself. This is the way segmentation works in Multics, the granddaddy of all modern operating systems and a pioneer of the idea of segmentation. Multics ran on the General Electric (later Honeywell) 635 computer, which was a 36-bit word-addressable machine, which means that memory is divided into 36-bit words, with consecutive words having addresses that differ by 1 (there were no bytes). A virtual address was 36 bits long, with the high 18 bits interpreted as the segment number and the low 18 bits as segment offset. Although 18 bits allows a maximum size of 2^{18} = 262,144 words, the software enforced a maximum segment size of 2^{16} = 65,536 words. Thus the segment offset is effectively 16 bits long. Associated with each process is a table called the *descriptor segment*. There is a register called the Descriptor Segment Base Register (DSBR) that points to it and a register called the Descriptor Segment Length Register (DSLR) that indicates the number of entries in the descriptor segment.



First the segment number in the virtual address is used to index into the descriptor segment to find the appropriate descriptor. (If the segment number is too large, a fault occurs). The descriptor contains permission information, which is checked to see if the current process has rights to access the segment as requested. If that check succeeds, the memory address of a page table for the segment is found in the descriptor. Since each page is 1024 words long, the 16-bit segment offset is interpreted as a 6-bit page number and a 10-bit offset within the page. The page number is used to index into the page table to get an entry containing a valid bit and frame number. If the valid bit is set, the physical address of the desired word is found by concatenating the frame number with the 10-bit page offset from the virtual address.

Actually, I've left out one important detail to simplify the description. The “descriptor segment” really is a segment, which means it really is paged, just like any other segment. Thus there is another page table that is the page table for the descriptor segment. The 18-bit segment number from the virtual address is split into an 8-bit page number and a 10-bit offset. The page number is used to select an entry from the decriptor segment's page table. That entry contains the (physical) address of a page of the descriptor segment, and the page-offset field of the segment number is used to index into that page to get the descriptor itself. The rest of the translation occurs as described in the preceding paragraph. In total, each memory reference turns into four accesses to memory.

1. one to retrieve an entry from the descriptor segment's page table,
2. one to retrieve the descriptor itself,
3. one to retrieve an entry from the page table for the desired segment, and
4. one to load or store the desired data.

Multics used a TLB mapping the segment number and page number within the segment to a page frame to avoid three of these accesses in most cases.

Intel x86

[Silberschat, Galvin, and Gagne, Section 9.6]

The Intex 386 (and subsequent members of the X86 family used in personal computers) uses a different approach to combining paging with segmentation. A virtual address consists of a 16-bit *segment selector* and a 16 or 32-bit segment offset. The selector is used to fetch a *segment descriptor* from a table (actually, there are two tables and one of the bits of the selector is used to choose which table). The 64-bit descriptor contains the 32-bit address of the segment (called the *segment base*) 21 bits indicating its length, and miscellaneous bits indicating protections and other options. The segment length is indicated by a 20-bit *limit* and one bit to indicate whether the limit should be interpreted as bytes or pages. (The segment base and limit “fields” are actually scattered around the descriptor to provide compatibility with earlier version of the hardware.) If the offset from the original virtual address does not exceed the segment length, it is added to the base to get a “physical” address called the *linear address* (see Fig 9.20 on page 292). If paging is turned off, the linear address really is the physical address. Otherwise, it is translated by a two-level page table as described previously, with the 32-bit address divided into two 10-bit page numbers and a 12 bit offset (a page is 4K on this machine).

Previous More About Paging
Next Disks
Contents

---

[1] I have to say “generally” here and elsewhere when I talk about Unix because there are many variants of Unix in existence. Sometimes I will use the term “classic Unix” to decribe the features that were in Unix before it spread to many distinct dialects. Features in classic Unix are generally found in all of its dialects. Sometimes features introduced in one variant became so popular that they were widely immitated and are now available in most dialects.

[2] This a good example of one of those “popular” features not in classic Unix but in most modern variants: System V (an AT&T variant of Unix) introduced the ability to map a chunk of virtual memory into the address spaces of multiple processes at some offset in the data segment (perhaps a different offset in each process). This chunk is called a “shared memory segment,” but is not a segment in the sense we are using the term here. So-called “System V shared memory” is available in most current versions of Unix.

[3] Many variants of Unix get a similar effect with so-called “shared libraries,” which are implemented with shared memory but without general-purpose segmentation support.

---

solomon@cs.wisc.edu
Tue Jan 16 14:33:41 CST 2007