Paging: Non-Contiguous Memory Allocation

With hardware support to manipulate address bits in different ways, it became possible to allocate non-contiguous memory blocks to a program. Paging is an extension of fixed partitions, where more than one partition is allocated to a process. Physical memory is divided into fixed size blocks (2^N, for addressing convenience) called frames. Similarly, program is divided into fixed size blocks (2^N, as of frame size) called pages. Program pages are placed in frames and information is maintained in a table called page map table. In fixed partitions, list of free partitions is maintained to allocate a partition, in paging a list of free frames (frame table) is maintained and by searching this list a frame is allocated. In paging every address generated by the CPU is divided into two parts: page number (P) and a page displacement (offset). The page number is used as an index into a page table for physical address translation as shown in Figure-1. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents.

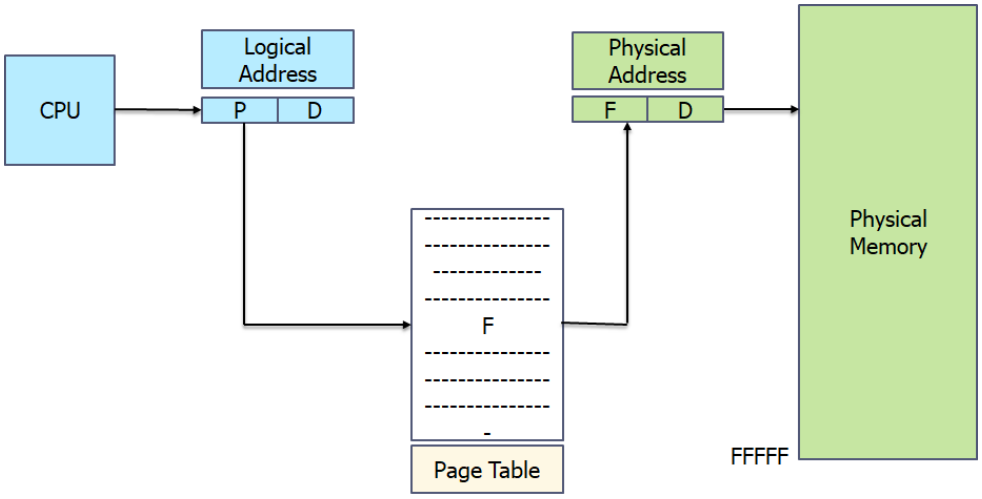


Figure-1: Page table for address translation.

By using a simple example of 16 bytes of memory, we explain how frames/pages are created and how address bits are manipulated along with page table to generate a physical address. You may recall that for a 16 bytes memory (address range 0-15) we need 4 bits to address any location. First, we divide memory into fixed size blocks (frame) of 4 bytes. We have 4 frames (0-3) as shown in Figure-2.

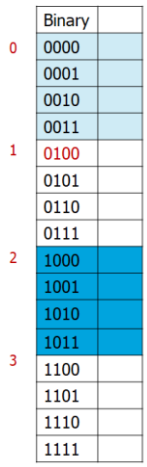


Figure-2: Memory divided into frames.

Since frame size is 4 bytes, within frame 4 bytes are addressed from 0-3. Once memory is divided into blocks, we address physical memory as frame number and displacement within frame. For contiguous memory we translate address of memory location by interpreting 4 address bits value. When memory is divided into fixed size blocks, we can translate address using frame number and displacement within frame. To further clarify address translation, we use a program of 8 bytes, along with addresses in decimal(binary) and divided into pages of 4 byte as shown in Figure-3. For this program we have 2 pages (page-0, and page-1).

0 (0000)	A	0
1 (0001)	B	
2 (0010)	C *	
3 (0011)	D	
4 (0100)	E	1
5 (0101)	F	
6 (0110)	G	
7 (0111)	H	

Figure-3: Program divided in pages

Now we have pages and frames, we can place the program in memory. In paging we place a page in frame one by one. For our memory of 4 frames in Figure-2, we assume, that frame-0 and frame-2 are allocated, and frame-1 and frame-3 are available. We place page-0 in frame-3 and enter the frame number in page table. We place page-1 in frame-1 and enter frame number in page table. Now we have program placed in memory and page table as shown in Figure-4.

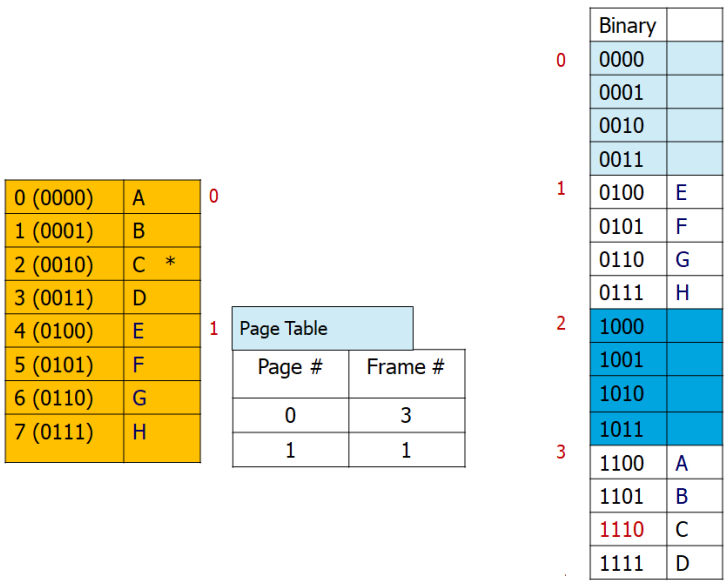


Figure-4. Program, Page Table and Memory

Now to translate logical address into physical address we split logical address bits (4 bits) into ‘P’ and ‘D’ and physical address bits into ‘F’ and ‘D’. For our example memory of 16 bytes, we need 4 bits to address 0-15 bytes. We have frame size of 4 bytes, so we need 2 bits to represent 4 bytes (0-3). We have 4 frames; we need 2 bits to represent 4 frame numbers (0-3). Similarly, we need 2 bits to represent pages (0-3) of logical address space.

Since page and frame size is 4 bytes, we need 2 bits to represent offset/displacement of pages as shown in Figure-5.

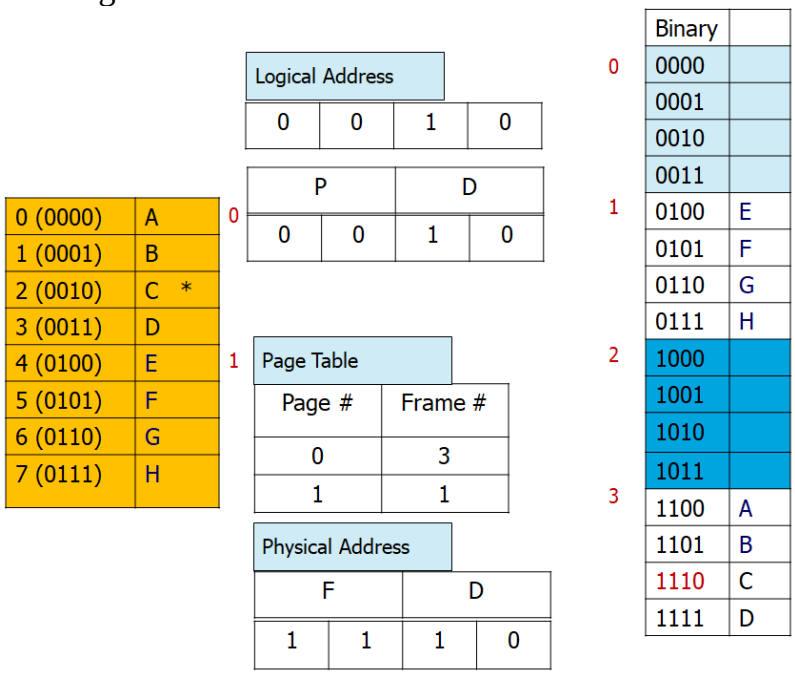


Figure-5: Logical to Physical Address

We will generate physical address to access 'C' by using page table in Figure-5. Logical address of 'C' is '0010'. We interpret these 4 bits of logical address as P and D shown in Figure-5. In this case P is '00' using page table we get frame number '11'. Whereas D in logical address is '10' We combine F and D as most and least significant bits respectively and get physical address '1110', which can be verified by looking at physical memory. You can compute physical address of any logical address of the program in Figure-5 by following the above stated steps.
Exercise: Using physical memory of 16 bytes (Figure-2) and program of 8 bytes (Figure-3). With frame/page size of 2 bytes, create pages and frames. Assume program is placed in same memory locations as in Figure-4. Modify page table with page and frames values and verify address translation to access 'A', 'D', 'E', and 'G'.

Memory in bytes can be extended to KB, MB and GB. Same process of splitting address bits into P/F and D is applied. if we have memory of 64KB and frame size is 2KB, we split 16 bits of address space as shown in below.

16 bits	
F	D
5 bits	11 bits

For a memory of 32MB, and frame size of 16KB, we will have 2048 frames. We split 25 bits address space as shown below.

25 bits	
F	D
11 bits	14 bits

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation.

Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. If process size is independent of page size, we expect internal fragmentation to an average one-half page per process.

Page Table Implementation

Different operating systems use different methods for storing page tables. Some allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. In the simplest case, the page table is implemented as a set of dedicated high-speed registers. The CPU dispatcher reloads these registers, just as it reloads the other registers at context switch time. If the frame/page size is small, then for a process number of page table entries will be large and a large number of these registers are required, and context switch time will be increased for loading these registers.

Some operating systems use main memory to store page table, and a page-table base register (PTBR) points to the page table. Changing page tables requires changing only this one register, which reduces context-switch time. In this approach the main problem is the time required to access a memory location. Every data/instruction access requires two memory accesses: one for the page table and another for the data/instruction. Thus, memory access is slowed by a factor of 2.

The standard solution to this problem is to use a special, small, fast lookup hardware cache called a Translation Look-aside Buffer (TLB). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made (may be done automatically in hardware or via an interrupt to the operating system). When the frame number is obtained, we can use it to access memory along with adding the page number and frame number to the TLB as shown in Figure-6.

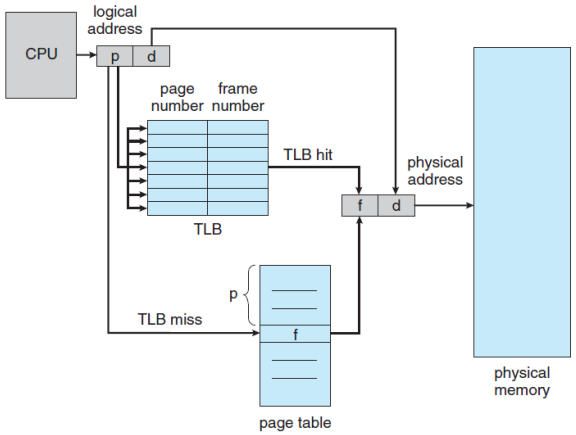


Figure-6: Translation Look-aside Buffer

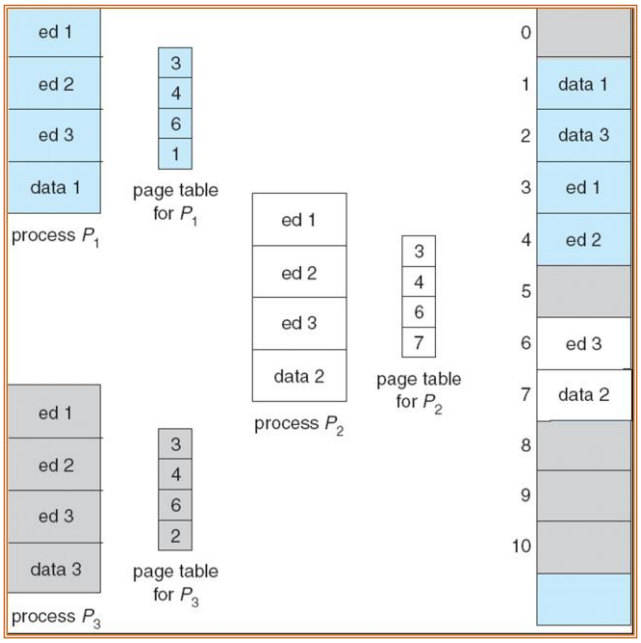
Memory Protection in Paging

In a paged environment memory is protected by using protection bits in the page table for each frame. One bit can define a page to be read-write or read-only. Every reference to memory uses the page table to find the correct frame number. While the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system. In fixed partitions, limit register is used to restrict a process to its logical address space, some systems provide hardware, in the form of a Page-Table Length Register (PTLR), to indicate the size of the page table. PTLR value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

Sharing in Paging

Paging facilitates sharing of common code, which is particularly important in a time-sharing environment. For example, a time-sharing system that supports 40 users, each of whom executes a text editor. If the text editor consists of 128 KB of code and 48 KB of data space, we need 7,040 KB to support the 40 users. If the code is reentrant code (or pure code), however, it can be shared. *Reentrant code* is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (128KB), plus 40 copies of the 48KB of data space per user. The total space required is now 2,048KB instead of 7,040 KB.

Assume frame/page size is 16KB, with no code sharing, we need 440 frames, and with code sharing where only one copy of the editor is kept in memory, we need only 128 frames. In paging heavily used programs like compilers, run-time libraries, and database systems can also be shared. Only condition for a code to be sharable, is that of reentrant code. With



sharing of code, memory utilization is improved, and with higher degree of multiprogramming utilization of other resources is also improved along improved throughput of the system. For sharing pages, code should be multiples of frame size, otherwise last frame allocated will have internal fragmentation.