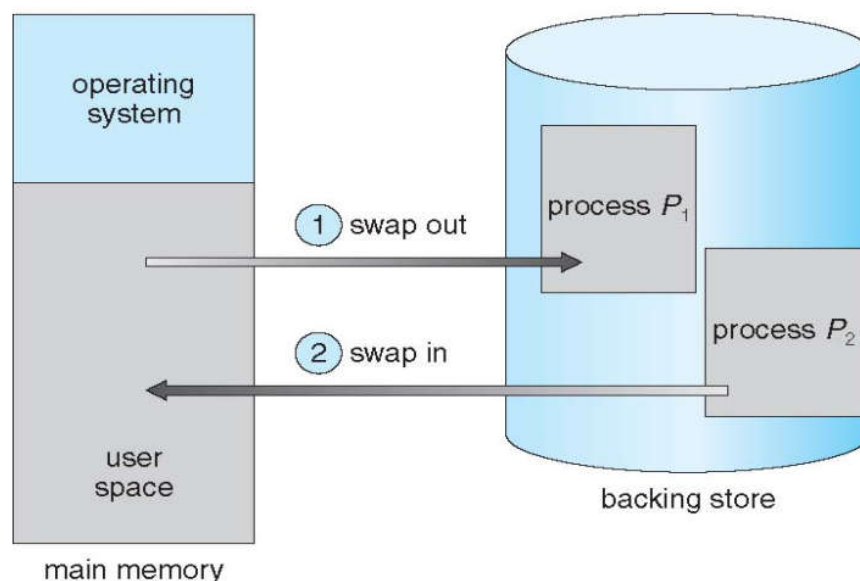❖ **Swapping**

- ▪ A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution.
- ▪ Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.
- o Standard **swapping** involves moving processes between main memory and a backing store.
- o **Backing store:** *is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.*
- o **Roll out, roll in:** swapping variant used for priority-based scheduling algorithms (Lower-priority process is swapped out so higher-priority process can be loaded and executed).
- o Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- o The system maintains a *ready queue* consisting of all ready-to-run processes whose memory images are on the backing store (disk) or in memory and are ready to run.
- o Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high. To get an idea of the context-switch time, let's assume that the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50 MB per second. The actual transfer of the 100-MB process to or from main memory takes:

$$100 \text{ MB}/50 \text{ MB per second} = 2 \text{ seconds}$$

The swap time is 200 milliseconds. Since we must swap both out and in, the total swap time is about 4,000 milliseconds.

Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.

If we have a computer system with 4 GB of main memory and a resident operating system taking 1 GB, the maximum size of the user process is 3 GB. However, many user processes may be much smaller than this—say, 100 MB. A 100-MB process could be swapped out in 2 seconds, compared with the 60 seconds required for swapping 3 GB. Clearly, it would be useful to know exactly how much memory a user process *is* using, not simply how much it *might* be using. Then we would need to swap only what is actually used, reducing *swap* time. For this method to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls (request memory( ) and release memory( )) to inform the operating system of its changing memory needs.

Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O. A process may be waiting for an I/O operation when we want to swap that process to free up memory. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the I/O operation is queued because the device is busy. If we were to swap out process $P1$ and swap in process $P2$, the I/O operation might then attempt to use memory that now belongs to process $P2$. There are two main solutions to this problem: never swap a process with pending I/O, or execute I/O operations only into operating-system buffers.

Transfers between operating-system buffers and process memory then occur only when the process is swapped in. Note that this **double buffering** itself adds

overhead. We now need to copy the data again, from kernel memory to user memory, before the user process can access it.

Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution. Modified versions of swapping, however, are found on many systems, including UNIX, Linux, and Windows. In one common variation, swapping is normally disabled but will start if the amount of free memory (unused memory available for the operating system or processes to use) falls below a threshold amount. Swapping is halted when the amount of free memory increases. Another variation involves swapping portions of processes— rather than entire processes—to decrease swap time. Typically, these modified forms of swapping work in conjunction with virtual memory.

- Does the swapped out process need to swap back in to same physical addresses?
    - o Depends on address binding method
        - ▪ Plus must consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows). A common variation:
    - o Swapping is normally disabled
    - o Swapping is started if the amount of free memory (unused memory available for the operating system or processes to use) falls below a given threshold.
    - o Swapping is disabled again once memory demand reduced below the threshold
- Another variation. Swapping portions of processes--rather than entire processes--to decrease swap time.
- Typically, these modified forms of swapping work in conjunction with virtual memory (covered soon).

❖ **Context Switch Time including Swapping**
- If next processes to be located a CPU (say A) is not in memory and there is not enough physical memory to accommodate A, then we need to swap out one of the processes in memory (say B) and swap in process A.

- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce context switch time by knowing how much memory is really being used. System calls to inform OS of memory use:
  - request_memory() and release_memory()
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

❖ Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation in which the operating system resides in low memory. The development of the other situation is similar.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Before discussing memory allocation further, we must discuss the issue of memory protection. We can prevent a process from accessing memory it does not own by combining two ideas previously discussed. If we have a system with a relocation register, together with a limit register, we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient** operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

- Main memory must support both OS and user processes
- Limited resource -- must allocate efficiently
- Contiguous allocation is one early method
- Main memory is usually divided into two **partitions**:
    - Resident operating system, usually held in low memory with interrupt vector
    - User processes are held in high memory
    - Each process contained in single contiguous section of memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
    - Base register contains value of smallest physical address

- o Limit register contains range of logical addresses – each logical address must be less than the limit register
- o MMU maps logical address *dynamically*
- o Can then allow actions such as kernel code being **transient** – comes and goes as needed**.** Thus, kernel can change size dynamically.
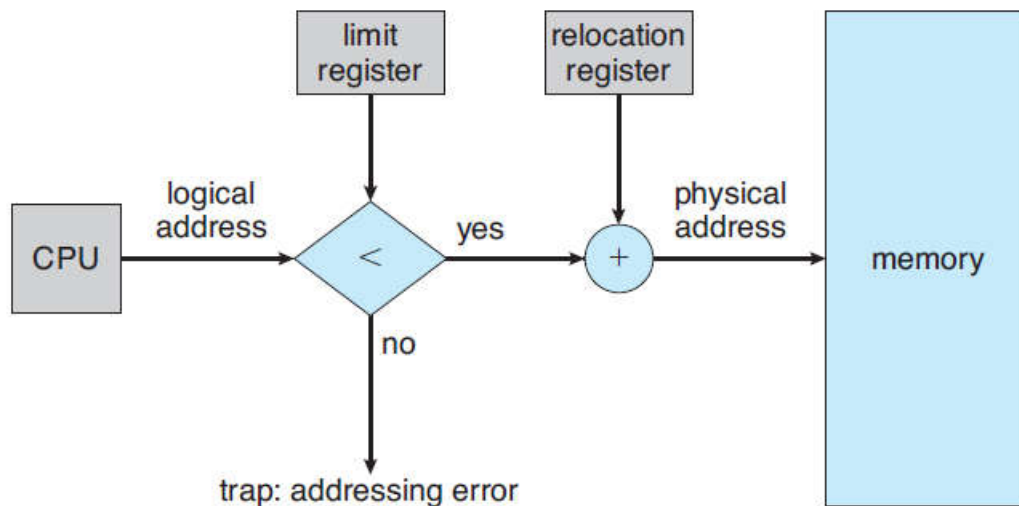
Hardware Support for Relocation and Limit Registers

❖ **Multiple-partition allocation**

Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this **multiple-partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use.

The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management.

In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. Eventually, as you will see, memory contains a set of holes of various sizes.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, then, we have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, as mentioned, the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general **dynamic storage-allocation problem**, which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best fit. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst fit. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

- **Variable-partition:** sized to a given process' needs.
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions   b) free partitions (holes)

❖ **Dynamic Storage-Allocation Problem**
- How to satisfy a request of size n from a list of free holes?
  - **First-fit**: Allocate the *first* hole that is big enough
  - **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless the list is ordered by size.
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the *largest* hole; must also search entire list, unless the list is ordered by size
    - Produces the largest leftover hole
- First-fit and best-fit are better than worst-fit in terms of speed and storage utilization
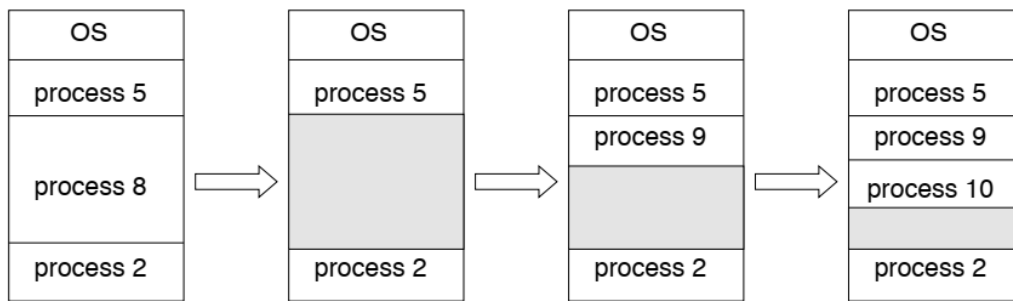
Examples of Dynamic Memory allocation

Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit:

212K is put in 500K partition

417K is put in 600K partition

112K is put in 288K partition (new partition 288K = 500K - 212K)

426K must wait

Best-fit:

212K is put in 300K partition

417K is put in 500K partition

112K is put in 200K partition

426K is put in 600K partition

Worst-fit:

212K is put in 600K partition

417K is put in 500K partition

112K is put in 388K partition (new partition 388K = 600K - 212K)

426K must wait

In this example, best-fit turns out to be the best.

Example: Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of (a) 12KB, (b) 10KB, (c) 9KB for First Fit?

First Fit		Best Fit		Worst Fit	
12k → 20k		12k → 12k		12k → 20k	
10k → 10k		10k → 10k		10k → 18k	
9k → 18k		9k → 9k		9k → 15k	

❖ **Fragmentation**
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous and therefore cannot be used.
  o First fit analysis reveals that given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation
    ▪ 1/3 of memory may be unusable -> **50-percent rule**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory.
  o Can happen if there is a hole of size 15,000 bytes and a process needs 14,900 bytes; keeping a hole of size 100 bytes is not worth the effort so the process is allocated 15,000 bytes.
  o The size difference of 100 bytes is memory internal to a partition, but not being used
- ➢ Reduce external fragmentation by **compaction**
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem: cannot perform compaction while I/O is in progress involving memory that is being compacted.
  o Latch job in memory while it is involved in I/O
  o Do I/O only into OS buffers

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for

others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5 N$ blocks will be lost to fragmentation.

That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the **hole** itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. Two complementary techniques achieve this solution: segmentation and paging. These techniques can also be combined.

❖ **Segmentation**
- Memory-management scheme that supports user's view of memory
- A program is a collection of segments -- a logical unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local variables, global variables
  - common block
  - stack
  - symbol table
  - arrays
- Each segment can to reside in different parts of memory. Way to circumvent the contiguous allocation requirement.

- Logical address consists of a two tuple:
  - <segment-number, offset>
- Need to map two-dimensional logical addresses to a one-dimensional physical address. Done via **Segment table**:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- Segment table is kept in memory
  - **Segment-table base register (STBR)** points to the segment table's location in memory
  - **Segment-table length register (STLR)** indicates number of segments used by a program;
    - segment number *s* is legal if *s* < **STLR**

Do programmers think of memory as a linear array of bytes, some containing instructions and others containing data? Most programmers would say "no." Rather, they prefer to view memory as a collection of variable-sized segments, with no necessary ordering among the segments.

When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. The programmer talks about "the stack," "the math library," and "the main program" without caring what addresses in memory these elements occupy. She is not concerned with whether the stack is stored before or after the Sqrt() function. Segments vary in length, and the length of each is intrinsically defined by its purpose in the program. Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the Sqrt(), and so on.

Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*
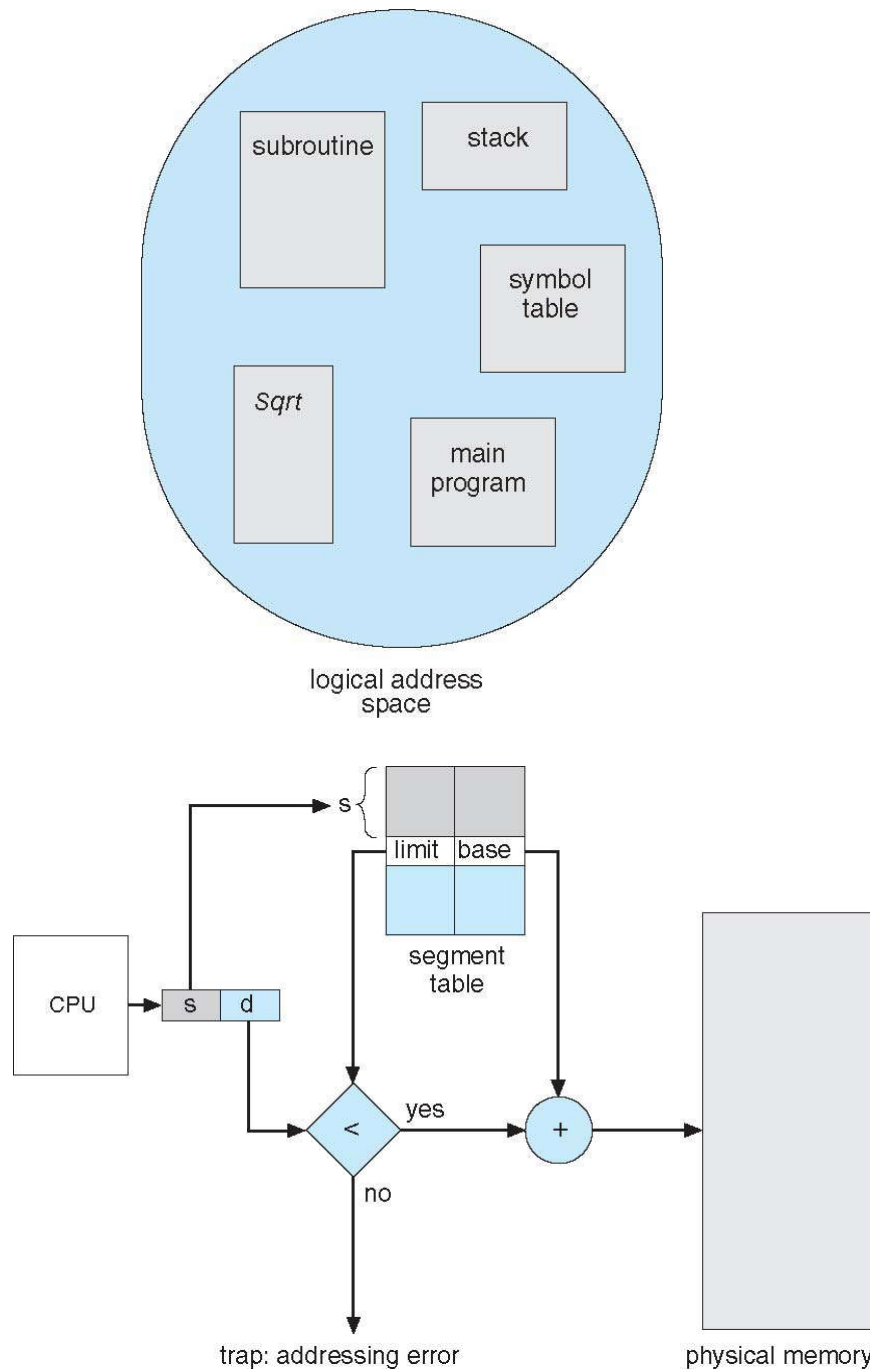
<segment-number, offset>.

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

subroutine

stack

symbol table

Sqrt

main program

logical address space

s

limit | base

segment table

CPU

s | d

<

yes

+

no

trap: addressing error

physical memory

❖ Segmentation Hardware

Although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.

14

This mapping is effected by a **segment table**. Each entry in the segment table has a **segment base** and a **segment limit**. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 8.8. A logical address consists of two parts: a segment number, *s,* and an offset into that segment, *d.* The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base–limit register pairs.

As an example, consider the situation shown in Figure 8.9. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

❖ **Paging**

Segmentation permits the physical address space of a process to be noncontiguous. **Paging** is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. Most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems, from those for mainframes through those for smart phones. Paging is implemented through cooperation between the operating system and the computer hardware.

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 264 bytes of physical memory. The hardware support for paging is illustrated in Figure 8.10. Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 8.11.

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset

particularly easy. If the size of the logical address space is 2*m*, and a page size is 2*n* bytes, then the high-order *m* - *n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset. Thus, the logical address is as follows:

page number	page offset
p	d
$m - n$	n
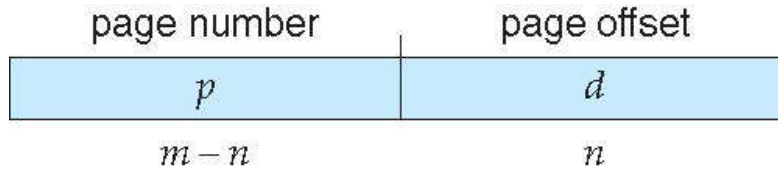
where *p* is an index into the page table and *d* is the displacement within the page. As a concrete (although minuscule) example, consider the memory in Figure 8.12. Here, in the logical address, *n*= 2 and *m* = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

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. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 - 1,086 = 962 bytes. In the worst case, a process would need *n* pages plus 1 byte. It would be allocated *n* + 1 frames, resulting in internal fragmentation of almost an entire frame.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount data being transferred is larger (Chapter 10). Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages.

Researchers are now developing support for variable on-the-fly page size. Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 232 physical page frames. If frame size is 4 KB (212), then a system with 4-byte entries can address 244 bytes (or 16 TB) of physical memory. We should note here that the size of physical memory in a paged memory system is different from the maximum logical size of a process. As we further explore paging, we introduce other information that must be kept in the page-table entries. That information reduces the number of bits available to address page frames. Thus, a system with 32-bit page-table entries may address less physical memory than the possible maximum. A 32-bit CPU uses 32-bit addresses, meaning that a given process space can only be 232 bytes (4 TB).

Therefore, paging lets us use physical memory that is larger than what can be addressed by the CPU's address pointer length. When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 8.13).

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the programmer's view of memory and the

actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a **frame table**. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes. In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. 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. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

- Physical address space of a process can be non-contiguous.
- Process is divided into fixed-size blocks, each of which may reside in a different part of physical memory.
- Divide physical memory into fixed-sized blocks called **frames**
  - Size of a frame is power of 2 between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size as frames called **pages**
- Backing store, where the program is permanently residing, is also split into storage units (called **blocks**), which are the same size as the frame and pages.
- Physical memory allocated whenever the latter is available
  - Avoids external fragmentation
 - Still have Internal fragmentation

- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program from backing store.
- Set up a **page table** to translate logical to physical addresses
- Page table is kept in memory.
    - **Page-table base register** (**PTBR**) points to the page table
    - **Page-table length register** (**PTLR**) indicates size of the page table
- Still have Internal fragmentation (more later)

- Assume the logical address space is 2^m. *(How is m determined?)*
- Assume page size is 2^n
- Address generated by CPU is divided into:
    - Page number (p) – used as an index into a page table which contains base address of each page in physical memory. Size of p is "$m - n$"
    - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit. Size of d is "n".
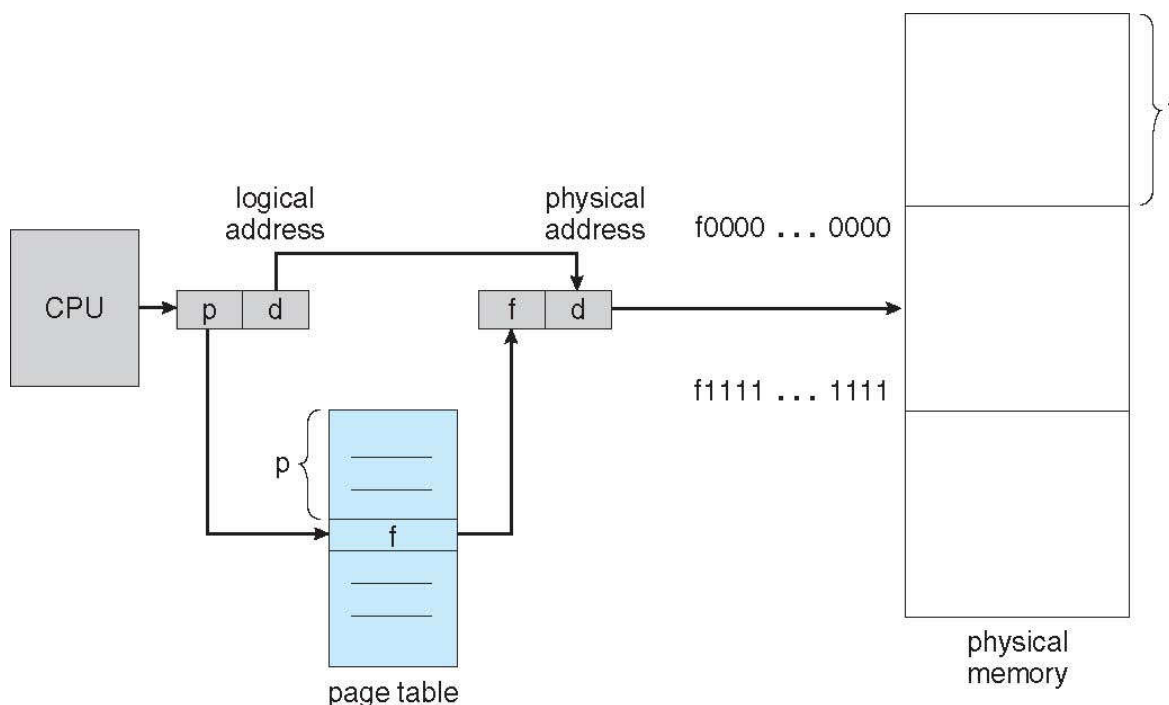
Assume *m* = 4 and *n* = 2 and 32-byte memory and 4-byte pages
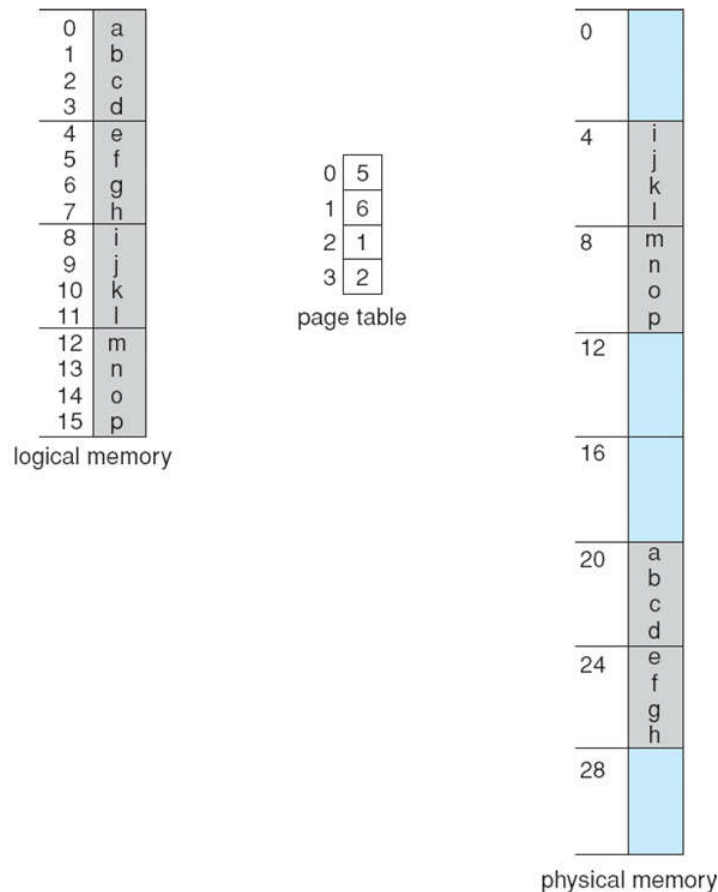
- Calculating internal fragmentation
    - Page size = 2,048 bytes
    - Process size = 72,766 bytes
    - 35 pages + 1,086 bytes
    - Internal fragmentation of 2,048 - 1,086 = 962 bytes
    - Worst case fragmentation = 1 frame – 1 byte
    - On average fragmentation = 1 / 2 frame size
    - So small frame sizes desirable?
    - But each page table entry takes memory to track
    - Page sizes growing over time
        - Solaris supports two page sizes – 8 KB and 4 MB
- By implementation process can only access its own memory

(a)   (b)
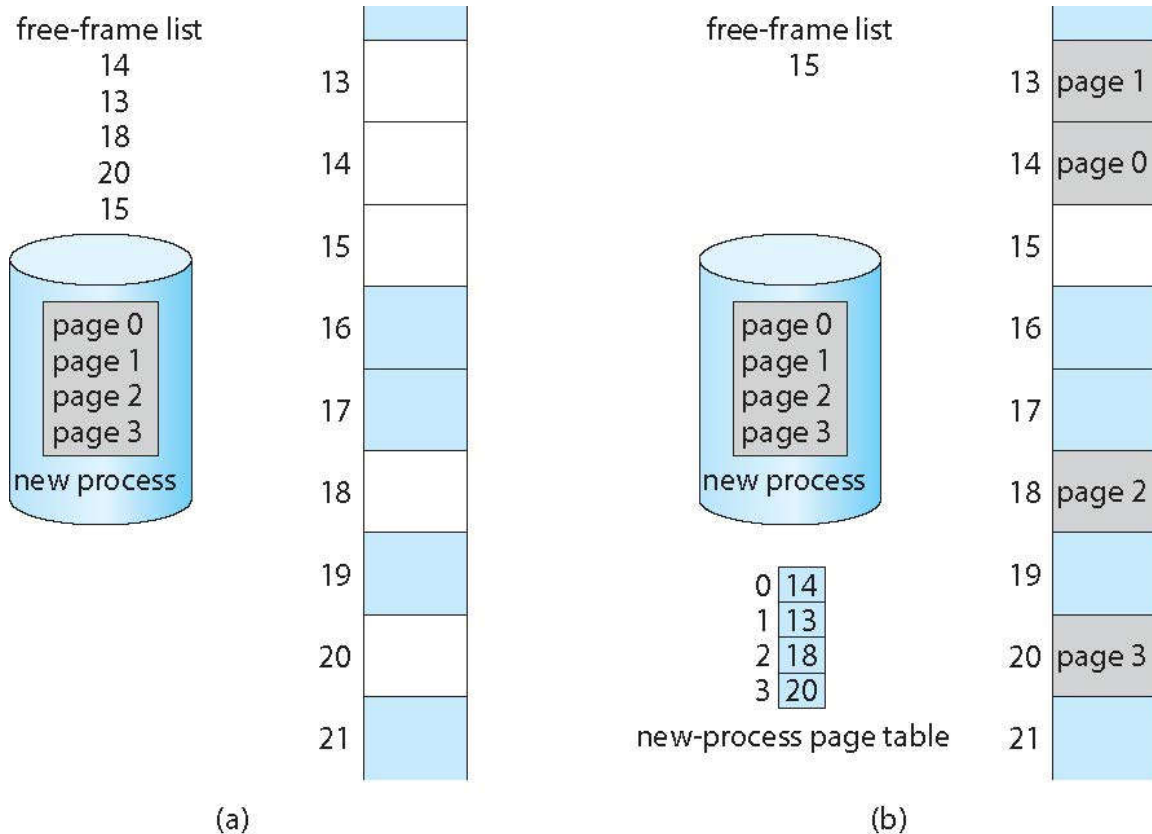
❖ Hardware Support

Each operating system has its own 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. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table. Other operating systems provide one or at most a few page tables, which decreases the overhead involved when processes are context-switched. The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map.

The DEC PDP-11 is an example of such an architecture. The address consists of 16 bits, and the page size is 8 KB. The page table thus consists of eight entries that are kept in fast registers. The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries).

For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register** (**PTBR**) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time. The problem with this approach is the time required to access a user memory location. If we want to access location *i,* we must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, *two* memory accesses are needed to access a byte (one for the page-able entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping!

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. Some CPUs implement separate instruction and data address TLBs. That can double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches. The TLB is used with page tables in the following way. 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. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging. If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. Depending on the CPU, this 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 (Figure 8.14). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds)

and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

effective access time = 0.80 × 100 + 0.20 × 200 = 120 nanoseconds

In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds). For a 99-percent hit ratio, which is much more realistic, we have:

effective access time = 0.99 × 100 + 0.01 × 200 = 101 nanoseconds

This increased hit rate produces only a 1 percent slowdown in access time. As we noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work. A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs). We will further explore the impact of the hit ratio on the TLB in Chapter 9. TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between generations of Intel CPUs) may necessitate a change in the paging implementation of the operating systems that use it.

❖ **TLB -- Associative Memory**
- If page table is kept in main memory every data/instruction access requires two memory accesses

- o One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)
- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
    - o If p is in associative register, get frame # out
    - o Otherwise get frame # from page table in memory
- TLB is typically small (64 to 1,024 entries)
- On a TLB miss, the value of the (missed page-table and frame-number), is loaded into the TLB for faster access next time that address is used.
    - o What if there is no free TLB entry? Replacement policies must be considered
    - o Some entries can be **wired down** for permanent fast access
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
    - o Otherwise need to flush TLB at every context switch
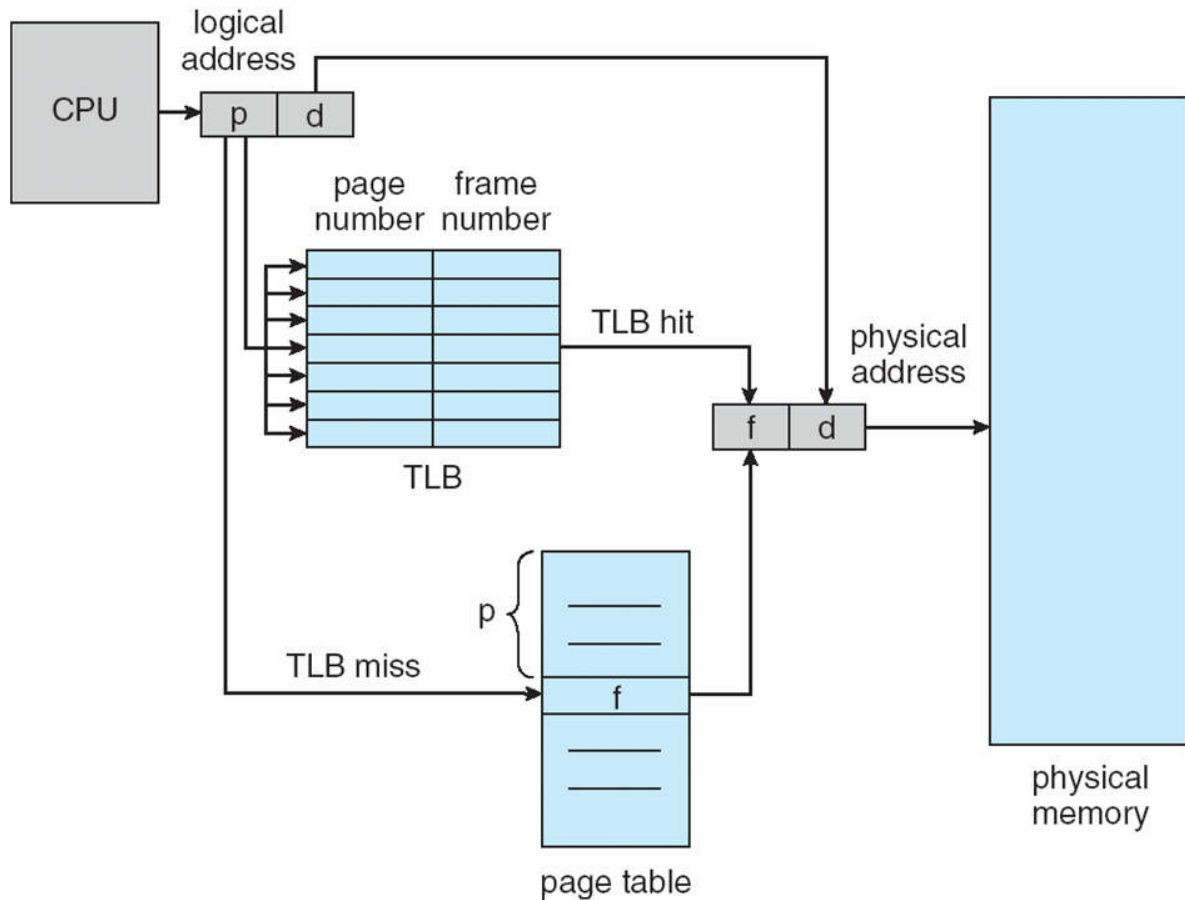
❖ Memory Protection

- Memory protection implemented by associating protection bits with each frame to indicate if "read-only " or "read-write" access is allowed
    - o Can also add more bits to indicate "execute-only" and so on
- **Valid-invalid** bit attached to each entry in the page table:
    - o "valid" indicates that the associated page is in the process' logical address space, and is thus is a legal page
    - o "invalid" indicates that the page is not in the process' logical address space
    - o Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

❖ **Valid (v) or Invalid (i) Bit In A Page Table**
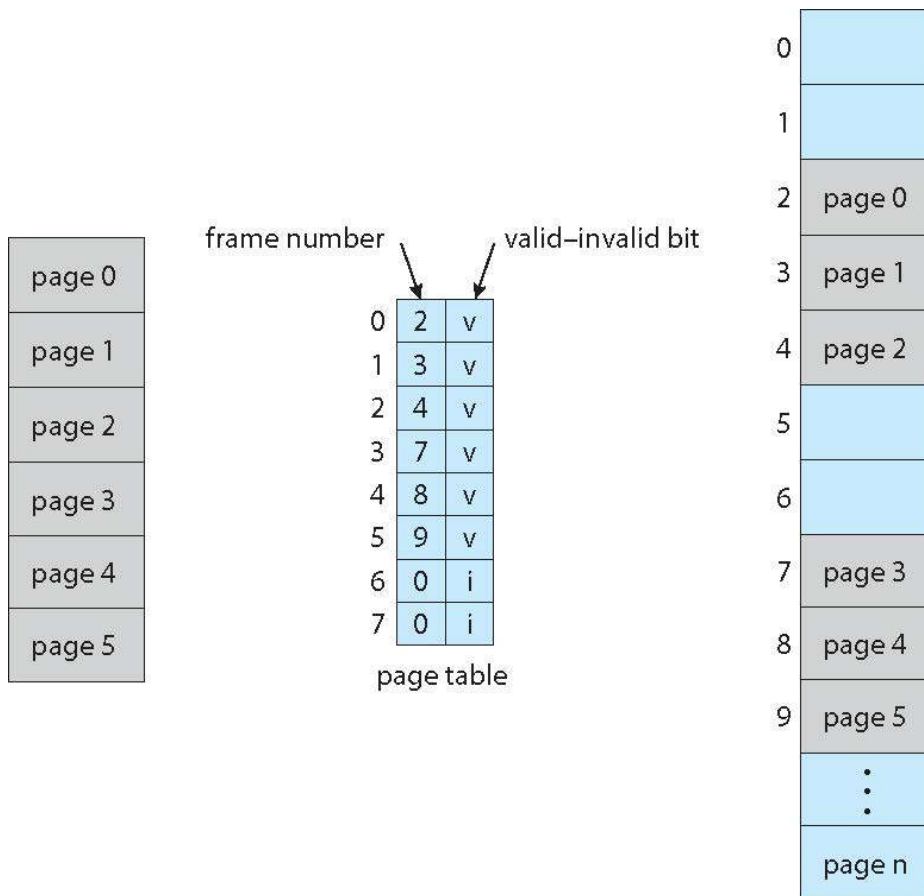
❖ **Shared Pages**
- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for inter-process communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

28

❖ **Shared Pages Example**
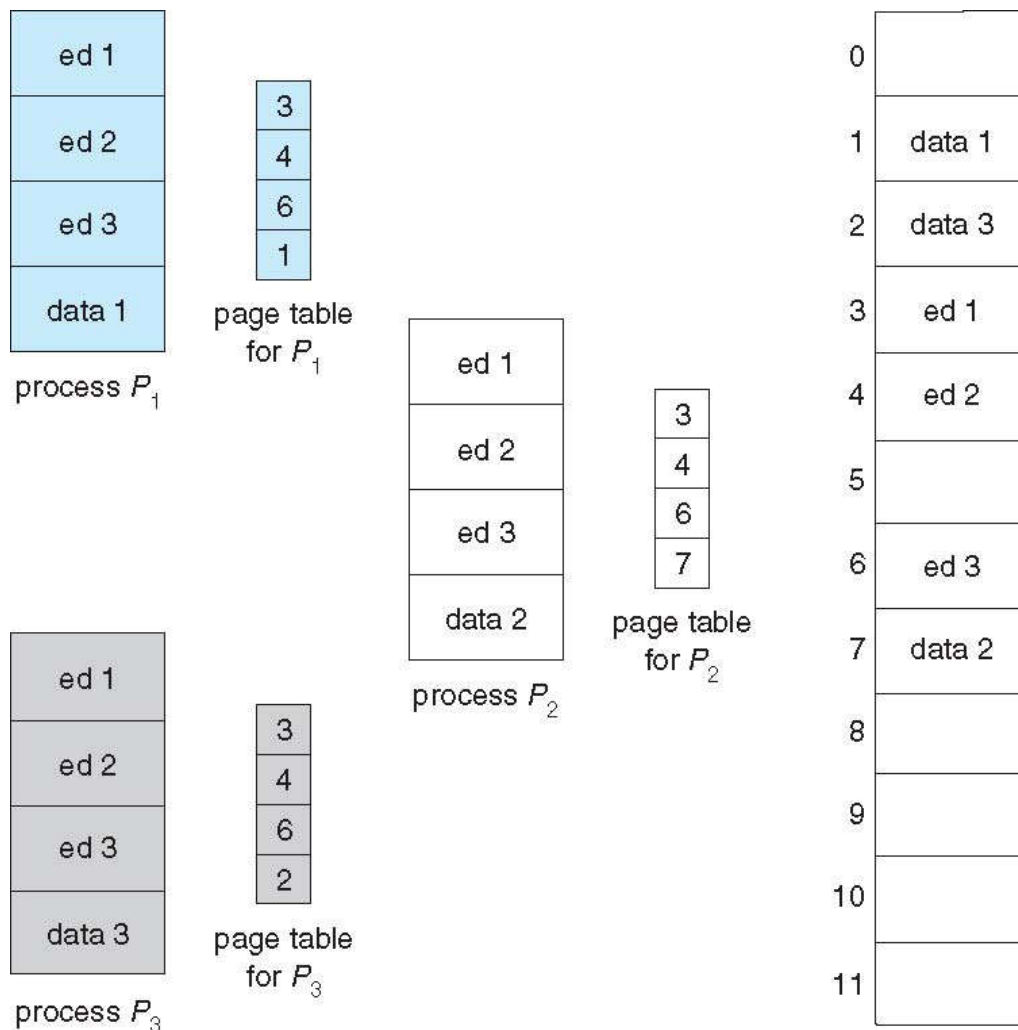
❖ **Structure of the Page Table**
- Memory structures for paging can get huge using straight-forward methods
  - ○ Consider a 32-bit logical address space
  - ○ Page size of 1 KB (2^{10})
  - ○ Page table would have 4 million entries ($2^{32} / 2^{10}$)
  - ○ If each entry is 4 bytes -> Page table is of size 16 MB
    - ▪ That amount of memory used to cost a lot.
    - ▪ Do not want to allocate that contiguously in main memory
- What about a 64-bit logical address space?
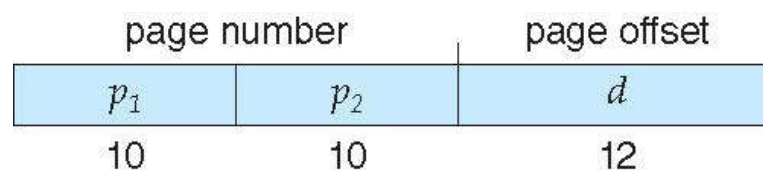
❖ **Page Table for Large address space**
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

❖ **Hierarchical Page Tables**
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
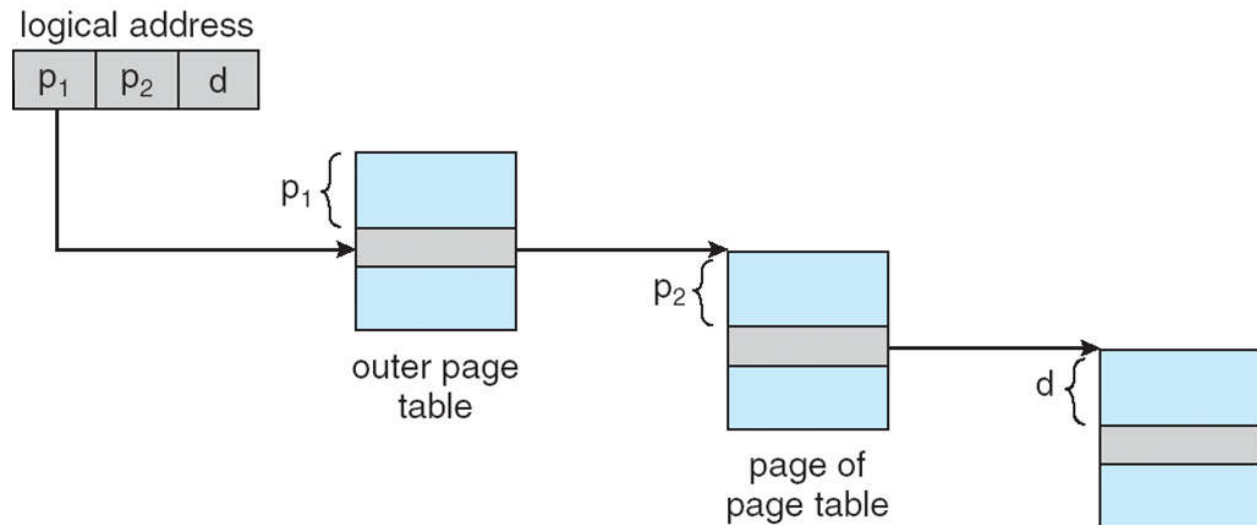- We then page the page table
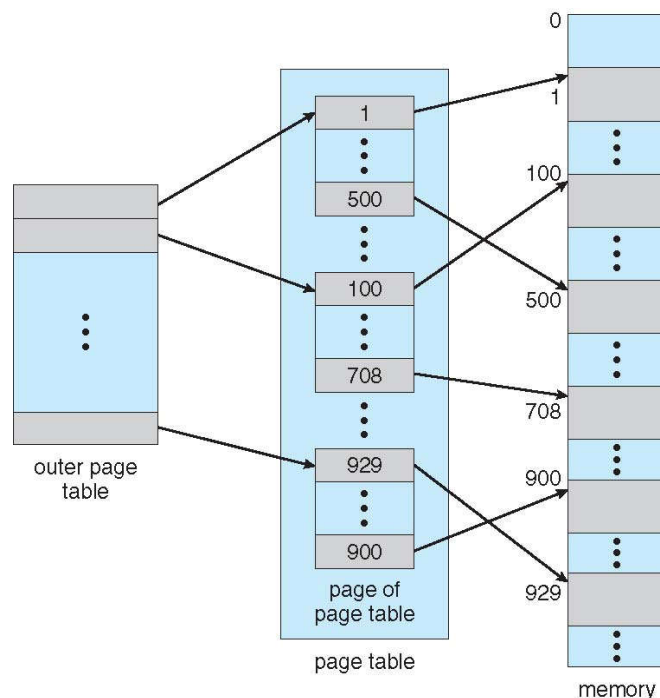
❖ **Two-Level Paging Example**
- A logical address (on 32-bit machine with 1K page size) is divided into:
    - a page number consisting of 22 bits
    - a page offset consisting of 10 bits
    - Since the page table is paged, the page number is further divided into:
    - a 12-bit page number
    - a 10-bit page offset
   - Thus, a logical address is as follows:

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

❖ **Address-Translation Scheme**

❖ **64-bit Logical Address Space**
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
    - ○ Then page table has 2^{52} entries
    - ○ If two level scheme, inner page tables could be 2^{10} 4-byte entries
    - ○ Address would look like

outer page	inner page	offset
p_1	p_2	d
42	10	12
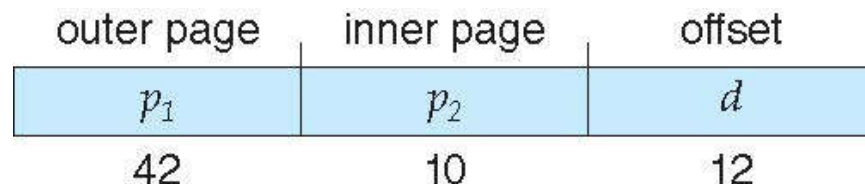
   ○ Outer page table has 2^{42} entries or 2^{44} bytes

- One solution is to divide the outer page table. Various ways of doing so. Example – three-level page table

| 2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

- o Even with 2nd outer page table, the outer-outer table is still 2^{34} bytes in size.
- o And possibly 4 memory access to get to one physical memory location.
- The next step would be four-level. But ….

Several schemes for dealing with very large logical address space
- Hashed Page Table.
- Clustered Page Tables
- Inverted Page Table

- ❖ **Hashed Page Table**
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - o This page table contains a chain of elements hashing to the same location
- Each element contains:
  - o The virtual page number
  - o The value of the mapped page frame
  - o A pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - o If a match is found, the corresponding physical frame is extracted
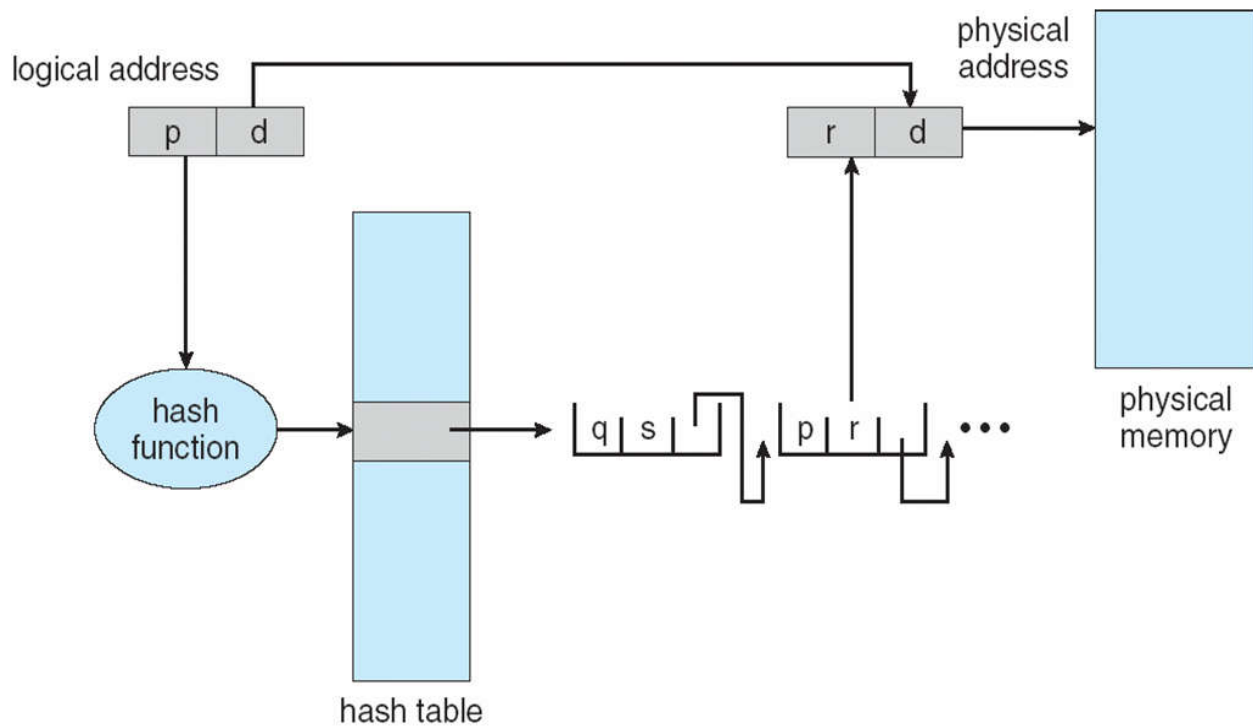
❖ **Inverted Page Table**
- Rather than each process having a page table and keeping track of all possible logical pages, track all the physical pages
- Use **inverted page-table**, which has one entry for each real page of memory
- An entry the inverted-page table consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- What is maximum size of the inverted page-table?
- Decreases memory needed to store each individual page table, but increases time needed to search the inverted page table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
    - o TLB can accelerate access
- But how to implement shared memory?
    - o One mapping of a virtual address to the shared physical address

logical address

CPU → pid | p | d

physical address

i | d → physical memory

search

page table

pid | p
}i
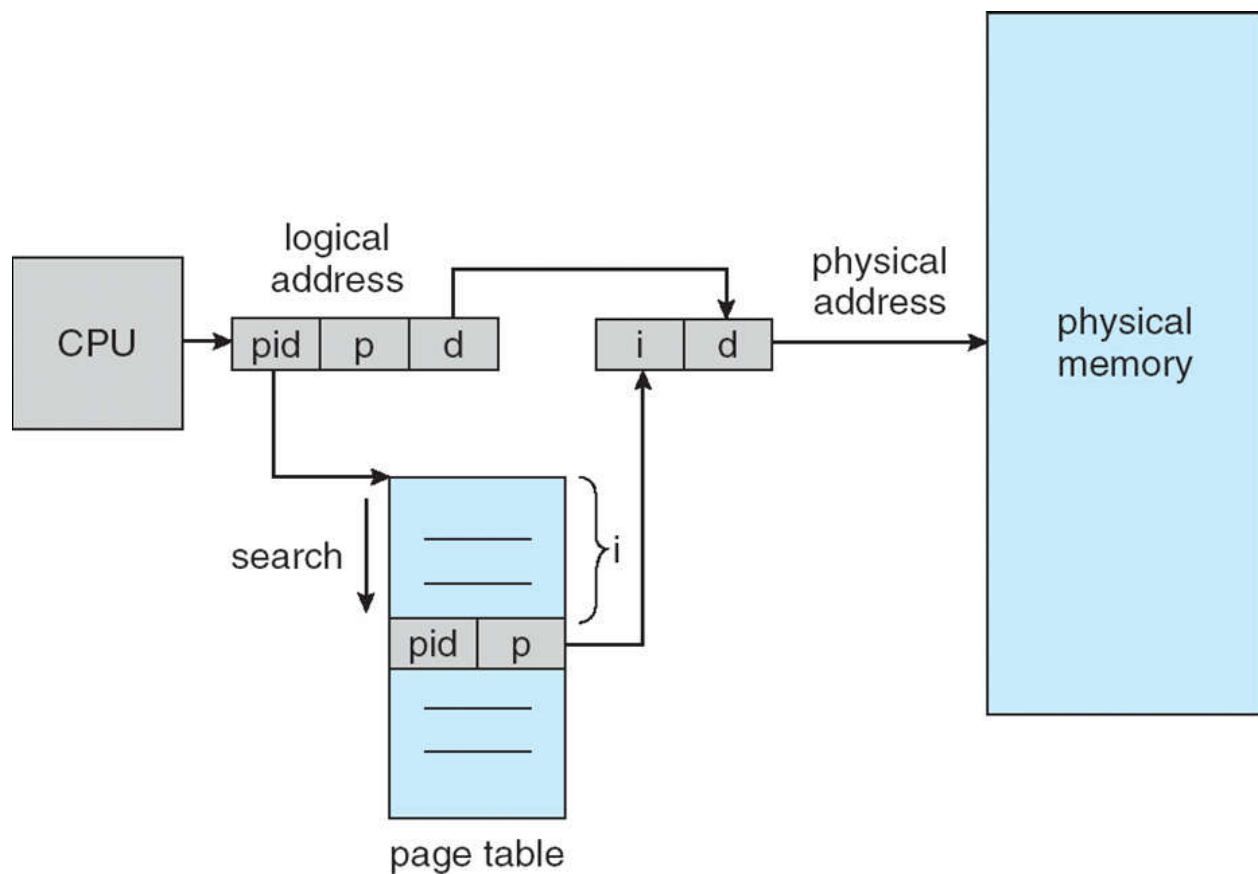
How does the 50 percent rule make 1/3rd of memory ... - Quora

Fifty percent rule : For every N blocks allocated, 0.5N blocks are lost to fragmentation. Unusable memory = (0.5N)/(N+0.5N)= 1/3 Hope this clears things up!