**Unit 4**

**Chapter 8: Main Memory**

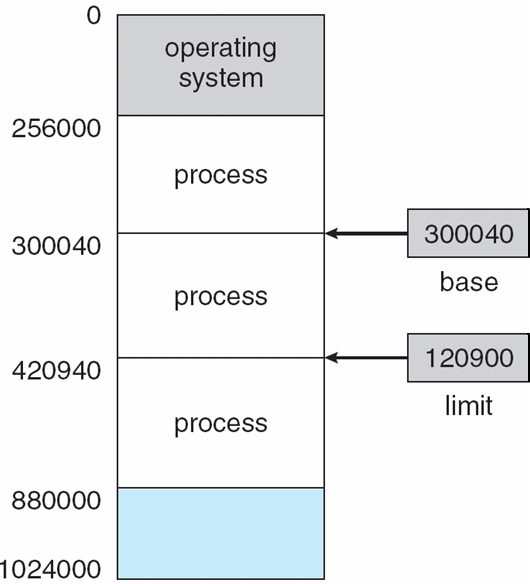
* Background
* Swapping
* Contiguous Memory Allocation
* Paging

**Background**

Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

* **Basic Hardware**
* Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
* Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved (from disk) there before the CPU can operate on them
* Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
* The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Memory access may take many cycles of the CPU clock to complete its execution.
* This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a cache
* Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation, has to protect the operating system from access by user processes and, in addition, to protect user processes from one another.
* This protection must be provided by the hardware. It can be implemented in several ways
* **Base and Limit Registers**

We first need to make sure that each process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit



A base and a limit register define a logical address space

The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.

For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).

* **Hardware Address Protection with base and limit registers**
* CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

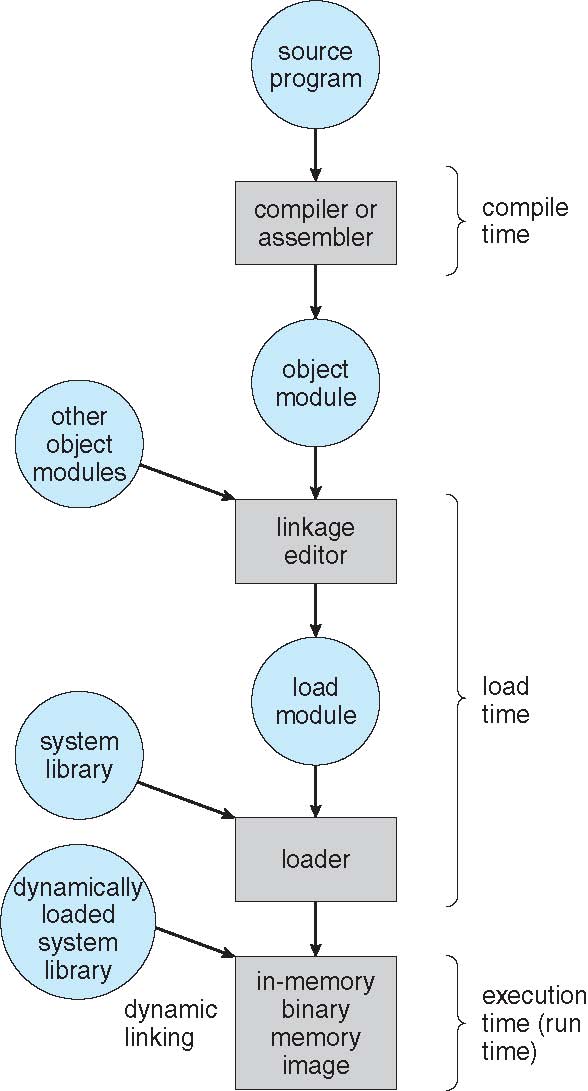
8.02.pdf

* Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error
* This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.
* The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode therefore only the operating system can load the base and limit registers.
* This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.
* The operating system, executing in kernel mode, is given unrestricted access to both operating system and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, and so on.

**Address Binding**

* Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.
* Depending on the memory management in use, the process may be moved between disk and memory during its execution.
* The processes on the disk that are waiting to be brought into memory for execution form the input queue.
* The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.
* Most systems allow a user process to reside in any part of the physical memory.
* Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000. This approach affects the addresses that the user program can use.

In most cases, a user program will go through several steps—some of which maybe optional-—before being executed. Addresses may be represented in different ways during these steps.



Multistep processing of a user program.

* Addresses in the source program are generally symbolic (such as count).
* A compiler will typically **bind** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module'').
* The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014).
* Each binding is a mapping from one address space to another

**Binding of Instructions and Data to Memory**

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

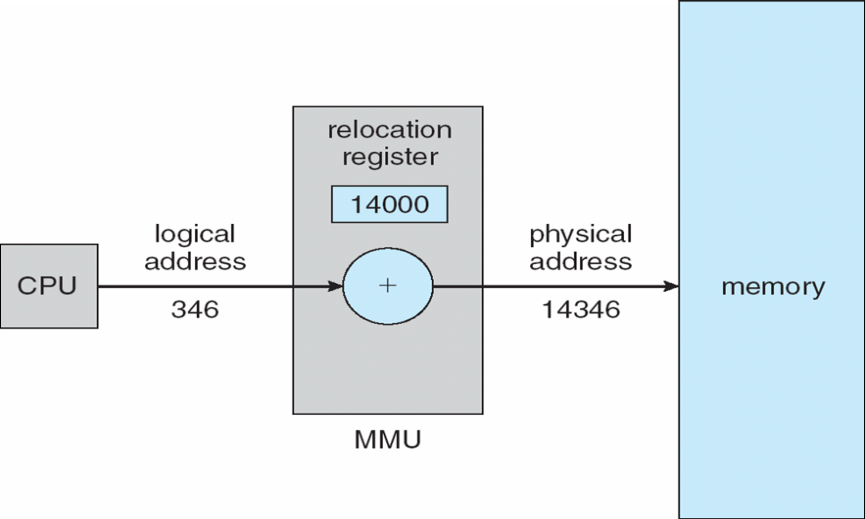
* **Compile time:** If you know at compile time where the process will reside in memory, then **absolute code** can be generated. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.
* **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code. In** this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
* **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work (e.g., base and limitregisters).Most general-purpose operating systems use this method.

**Logical vs. Physical Address Space**

* The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
* An address generated by the CPU is commonly referred to as a **logical address,** In this case, we usually refer to the logical address as a **virtual address.**
* Whereas an address seen by the memory unit i.e. the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address.**
* The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses.
* The set of all logical addresses generated by a program is a **logical address space;**
* The set of all physical addresses corresponding to these logical addresses is a **physical address space.**
* Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

**Memory-Management Unit (MMU)**

* The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU).**
* We can choose from many different methods to accomplish such mapping
* For the time being, we illustrate this mapping with a simple MMU scheme, which is a generalization of the base-register scheme



Dynamic relocation using a relocation register

* The base register is now called a **relocation register.** The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
* The MS-DOS operating system running on. the Intel 80x86 family of processors uses four relocation registers when loading and running processes.
* The user program deals with logical addresses; it never sees the real physical addresses
* The memory-mapping hardware converts logical addresses into physical addresses.
* The final location of a referenced memory address is not determined until the reference is made. Execution-time binding occurs when reference is made to location in memory
* We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R).
* The user generates only logical addresses and thinks that the process runs in locations 0 to max. The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used.

**Dynamic Loading**

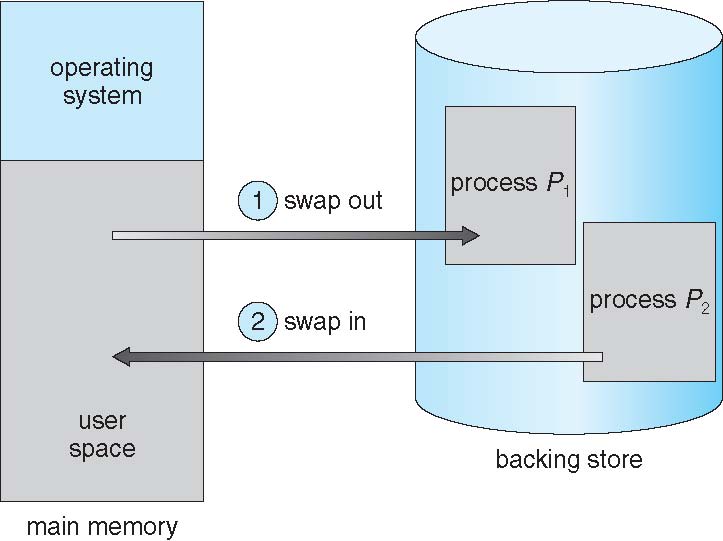
* The entire program and all data of a process must be in physical memory for the process to execute. The size of a process is thus limited to the size of physical memory.
* To obtain better memory-space utilization, we can use **dynamic loading.** With dynamic loading, a routine is not loaded until it is called.
* All routines are kept on disk in a relocatable load format.
* The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
* If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.
* The advantage of dynamic loading is that an unused routine is never loaded.
* Dynamic loading does not require special support from the operating system.
* It is the responsibility of the users to design their programs to take advantage of such a method.
* Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

**Dynamic Linking and Shared Libraries**

* Some operating systems support only **static linking,** in which system language libraries are treated like any other object module and are combined by the loader into the binary program image.
* The concept of dynamic linking is similar to that of dynamic loading. Here, though, linking, rather than loading, is postponed until execution time.
* With dynamic linking, a stub is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
* When the stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine.
* This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
* So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library.
* Thus, only programs that are compiled with the new library version are affected by the incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries.**
* Unlike dynamic loading, dynamic linking generally requires help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

**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.
* For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When each process finishes its quantum, it will be swapped with another process.
* **Roll out, roll in**
* A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process.
* When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in.**
* Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This restriction is dictated by the method of address binding.
* If binding is done at assembly or load time, then the process cannot be easily moved to a different location.
* If execution-time binding is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.



Swapping of two processes using a disk as a backing store

* **Backing store**
* Swapping requires a backing store. The 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.
* The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.
* 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.

**Context Switch Time including Swapping**

* The context-switch time in such a swapping system is fairly 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)
* To reduce swapping time we would need to swap only what is actually used, reducing swap time i.e. by knowing how much memory really being used
* 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. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped.
* 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.
* Generally, swap space is allocated as a chunk of disk, separate from the file system, so that its use is as fast as possible.
* A modification of swapping is used in many versions of UNIX. Swapping is normally disabled but will start if many processes are running and are using a threshold amount of memory. Swapping is again halted when the load on the system is reduced.

**Swapping on Mobile Systems**

Not typically supported

* Flash memory based
  + - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform

Instead use other methods to free memory if low

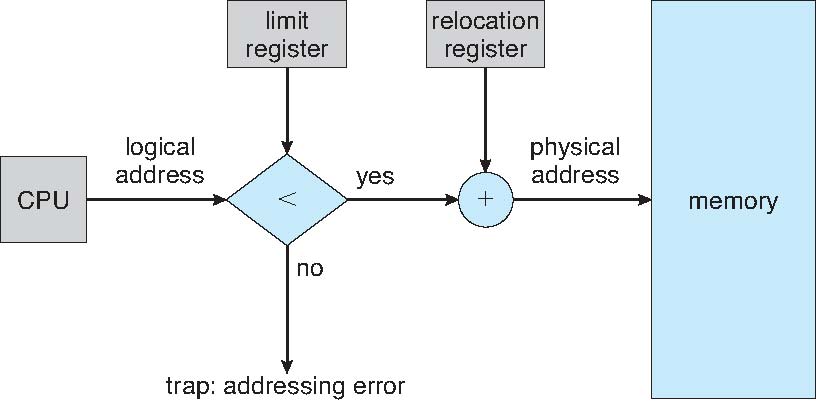
* iOS ***asks*** apps to voluntarily relinquish allocated memory
* Read-only data thrown out and reloaded from flash if needed
* Failure to free can result in termination
* Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
* Both OSes support paging as discussed below

**Contiguous Memory Allocation**

* The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate the parts of the main memory in the most efficient way possible.
* 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.
* In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

**Memory Mapping and Protection**

* Relocation registers used to protect user processes from each other, and from changing operating-system code and data
* The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses.
* With relocation and limit registers, each logical address must be less than 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



Hardware support for relocation and limit registers.

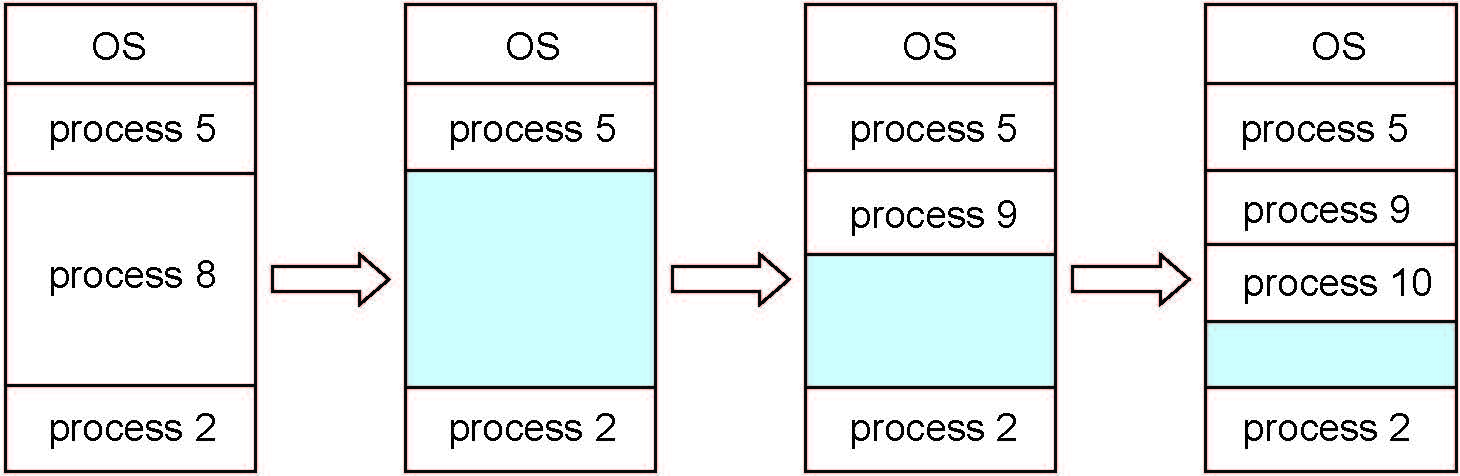
* When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
* Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.
* The relocation-register scheme provides an effective way to allow the operating-system size to change dynamically. This flexibility is desirable in many situations.
* Some 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.

**Memory Allocation**

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.

**Multiple-partition allocation**

* Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions.
* In the fixed-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.**
* When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.
* In general, at any given time we have 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

**Dynamic Storage-Allocation Problem**

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.

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 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.

**Fragmentation**

**Disadvantages of continuous memory allocation**

* **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.
* 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**.
* **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
* 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.
* Compaction 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.
* 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.

**Problems**

Example 1 : 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 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

1. Fixed size partitioning.

* First-fit:

212K is put in 500K partition

417K is put in 600K partition

112 is put into 200k partition

426 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 300K partition

426K must wait

* Which algorithm makes the most efficient use of memory?
* Ans : best fit

1. Variable size partitioning

* First-fit:

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

417K is put in 600K partition (600-417= 183)

112K is put in 288K partition (288-112=176)

426K must wait

* Best-fit:

212K is put in 300K partition (300-212=88)

417K is put in 500K partition (500-417=83)

112K is put in 200K partition (200-112=88)

426K is put in 600K partition

* Worst-fit:

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

417K is put in 500K partition (500-417=83)

112K is put in 388K partition

426K must wait

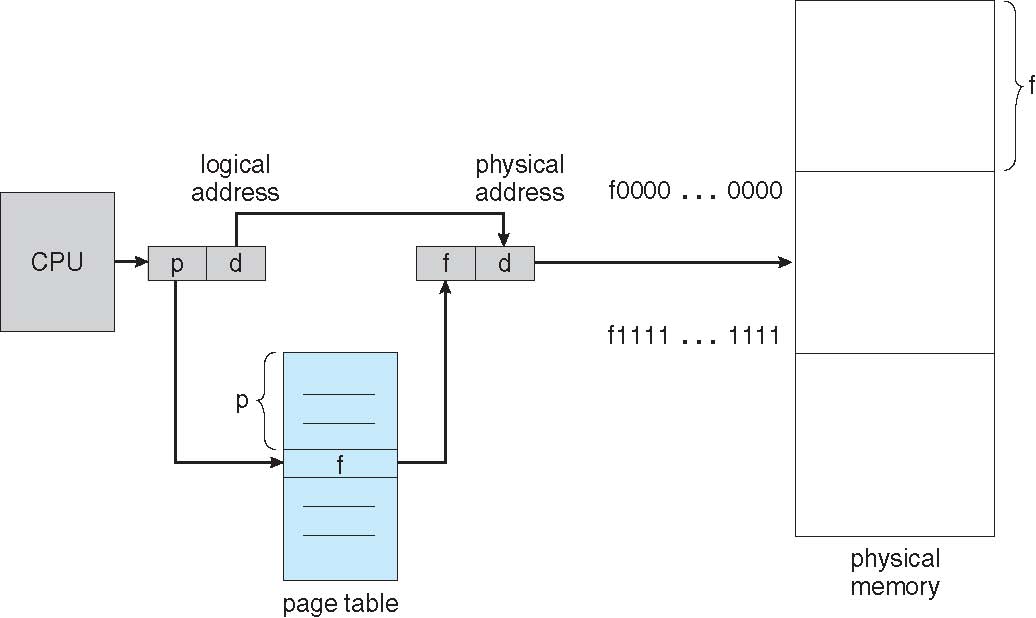
* Which algorithm makes the most efficient use of memory?
* Ans : best fit

**Paging**

* Paging is a memory-management scheme that permits the physical address  
  space of a process to be noncontiguous.
* Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store;
* The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store also has the fragmentation problems discussed in connection with main memory; except that access is much slower, so compaction is impossible.
* Because of its advantages over earlier methods, paging in its various forms is commonly used in. most operating systems.

**Basic Method**

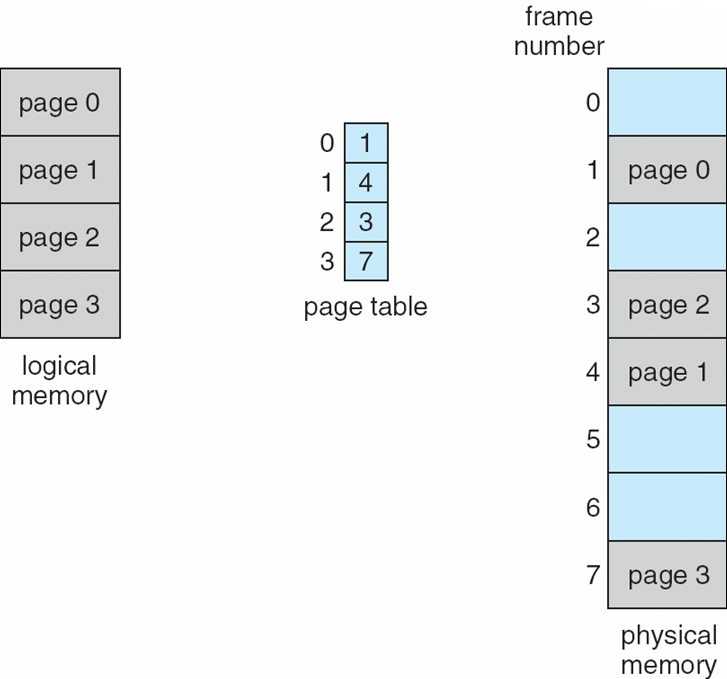
* The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages.**
* When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

****

**Paging Hardware**

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
* **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit

****

**Paging model of logical and physical memory**.

**Address Translation Scheme**

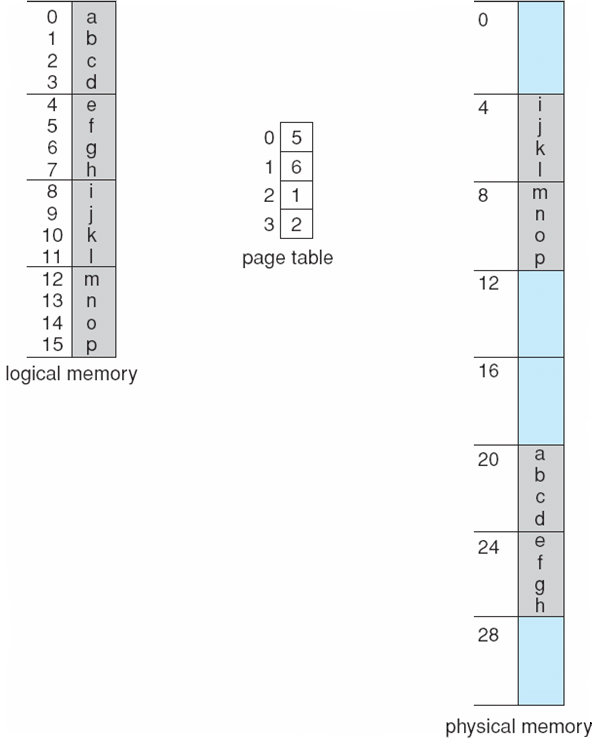
* The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.
* The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.
* If the size of logical address space is 2'" and a page size is 2" addressing units (bytes or words), 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:



* Where *p* is an index into the page table and *d* is the displacement within the page.

**Paging example for a 32-byte memory with 4-byte pages**

As an example, consider the memory in Figure



* Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user'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 x 4) + 0).
* Logical address 3 (page 0, offset 3) maps to physical address 23 *=* ( 5 x 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 (= (6x4) + 0).
* Logical address 13 maps to physical address 9.

Calculating internal fragmentation

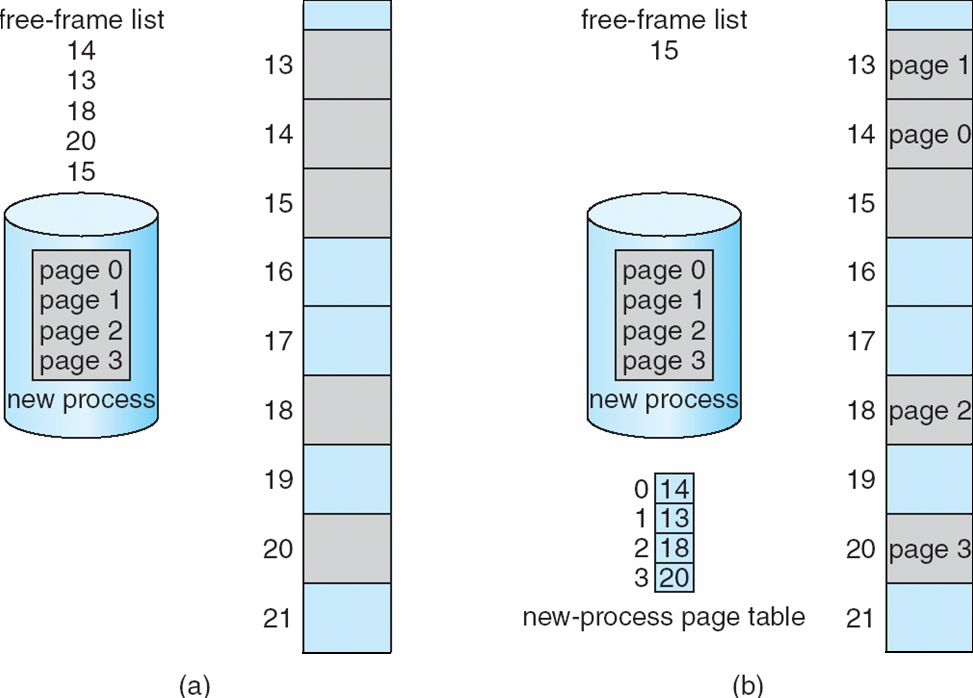
Page 0 in frame 5: a-> 5X4= 20+ 0= 20 b-> 5X4=20+1=21

Page 1 in frame 6: e-> 6X4= 24+0 = 24 f-> 6X4=24+1= 25

Page 2 in frame 1 i-> 1X4= 4+0= 4 j-> 1X4=4+1=5

**Free Frames**

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.



**Free frames (a) before allocation and (b) after allocation**

* When we use a paging scheme, we have no external fragmentation: Ay 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.
* 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.

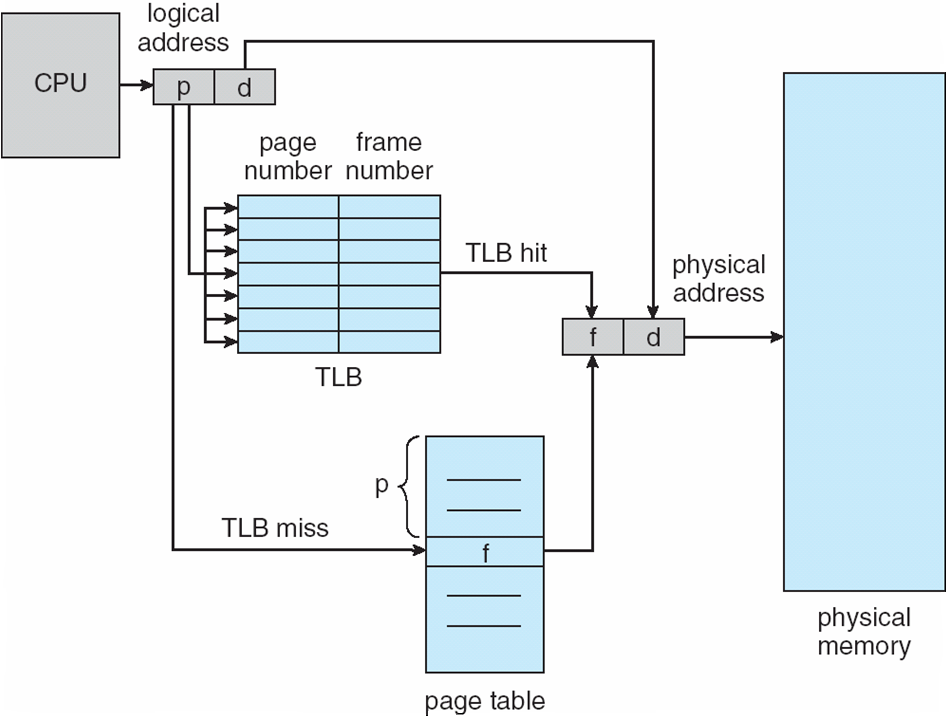
An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.

The user program 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.

This is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system.

**Hardware Support**

* 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.
* **Implementation of Page Table**
* The page table is kept in main memory
* **Page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.
* **Page-table length register** (**PTLR**) indicates size of the page table
* With this scheme, twomemory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances.



**Paging hardware with TLB.**

The standard solution to this problem is to use a special, small, fastlookup hardware cache, called a **translation look-aside buffer (TLB)** or **associative memory**.

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; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

The TLB is used with page tables in the following way.

* The TLB contains only a few of the page-table entries (The most frequently used pages are stored in TLB).
* 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.
* The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.
* If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made.
* If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random.

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.
* **Effective Access Time**

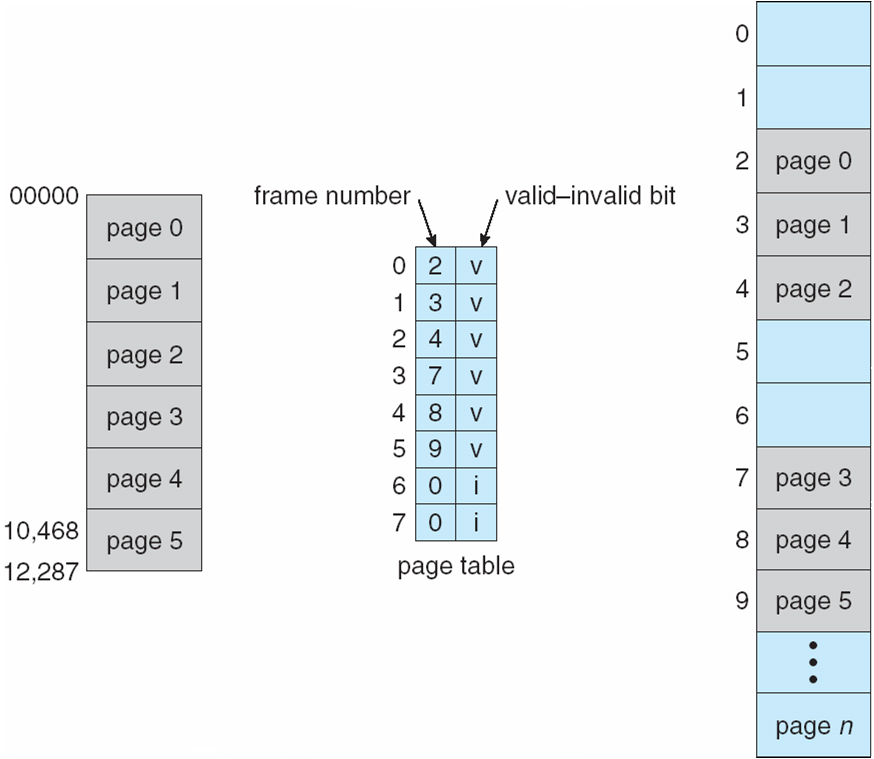
The percentage of times that a particular page number is found in the TLB is called the **hit ratio.**

Example: An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory,

* Then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.
* If we fail to find the page number in the TLB (20 nanoseconds), 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 220 nanoseconds.
* To find the **effective memory-access time,** we weight each case by its probability
* **Effective Access Time** (**EAT**)
  + - * EAT = 0.80X120+0.20X220 =140 ns
* In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).

**Memory Protection**

* Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.
* One bit can define a page to be read-write or read-only.
* At time of 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 (or memory-protection violation).
* We can create hardware to provide read-only, read-write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.



**Valid (v) or invalid (i) bit in a page table**

* One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit.
* When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page.
* When the bit is set to "invalid" the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid-invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.
* Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them.
* Some systems provide hardware, in the form of a **page-table length register** (PTLR), to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process.

**Shared Pages**

* An advantage of paging is the possibility of sharingcommon code. This consideration is particularly important in a time-sharing environment.
* If the code is **reentrant code** (or **pure code)** 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.
* Each user's page table maps onto the same physical copy, but data pages are mapped onto different frames.
* Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on.
* To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.
* The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads

**Shared Pages Example**

