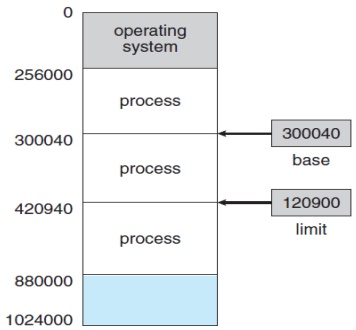
UNIT - III MEMORY MANAGEMENT

Background

- Memory is central to the operation of a modern computer system.
- The part of the OS that manages the memory hierarchy is called the memory manager.
    - to keep track of which parts of memory are in use and which parts are not in use,
    - to allocate memory to processes when they need it and deallocate it when they are done,
    - to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- 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.
- The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
- Accordingly, we can ignore how a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

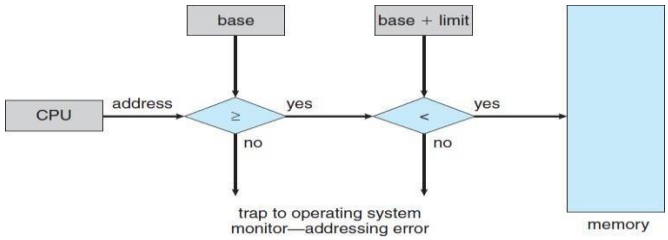
Basic Hardware

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses.
- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- 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 (processor stalls).
- The remedy is to add fast memory between the CPU and main memory (cache memory).
- Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure correct operation has to protect the OS from access by user processes and, in addition, to protect user processes from one another.



- This protection must be provided by the hardware. We first need to make sure that each process has a separate memory space.
- We can provide this protection by using two registers, usually a base and a limit.
  - 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).
- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the OS, 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 OS or other users.

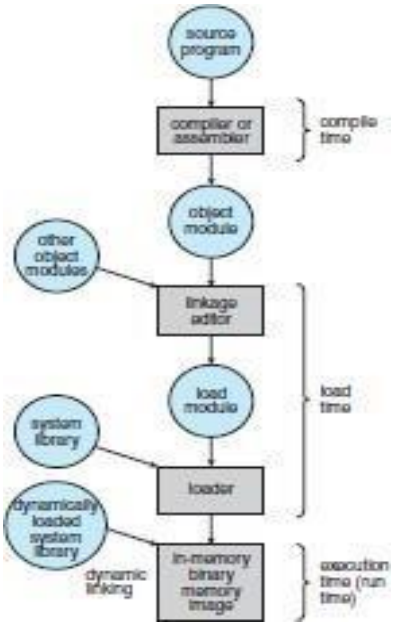
Hardware address protection with base and limit regsters



Address Binding

- The process of associating program instructions and data to physical memory addresses is called address binding, or relocation.
- A user program will go through several steps -some of which may be optional-before being executed

Multistep processing of a user program



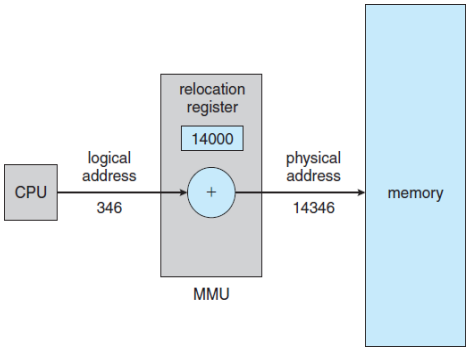
Addresses may be represented in different ways during these steps.

- Addresses in the source program are generally symbolic
- 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.
- Classically, the binding of instructions and data to memory addresses can be done at any step along the way:
    - **Compile time.** The compiler translates symbolic addresses to absolute addresses. If you know at compile time where the process will reside in memory, then absolute code can be generated (Static).
    - **Load time**. The compiler translates symbolic addresses to relative (relocatable) addresses. The loader translates these to absolute addresses. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code (Static).
    - **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. The absolute addresses are generated by hardware. Most general-purpose OSs use this method (Dynamic).
- Static-new locations are determined before execution. Dynamic-new locations are determined during execution.

Logical Versus Physical Address Space

- An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit -that is, 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. In this case, we usually refer to the logical address as a virtual address.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).



  - The Base register is 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
  - For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.
- The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses -all as the number 346.
- The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses.
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Dynamic Loading

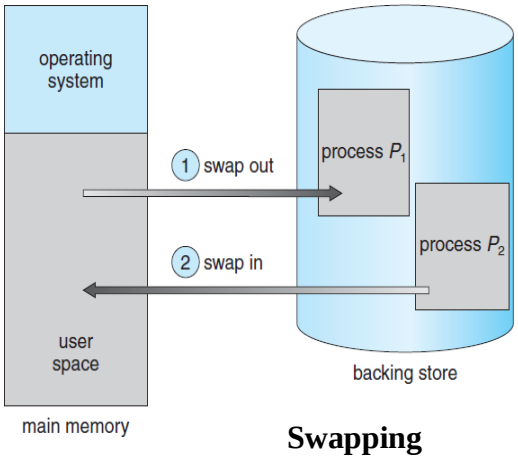
- To obtain better memory-space utilization, we can use dynamic loading.
  - With dynamic loading, a routine is not loaded until it is called.
  - All routines are kept on disk in a relocatable load format.
  - The main program is loaded into memory and is executed. When a routine needs to call another 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 Linking and Shared Libraries

- 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.
- 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.

Swapping

- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store (disk) and then brought back into memory for continued execution.



Swapping

- A round-robin CPU-scheduling algorithm; when a quantum expires
    - The memory manager will start to swap out the process that just finished
    - and to swap another process into the memory space that has been freed.
    - In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.
    - When each process finishes its quantum, it will be swapped with another process. Backing store is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users.
- It must be capable of providing direct access to these memory images.

- Major time consuming part of swapping is transfer time.
- Total transfer time is directly proportional to the amount of memory swapped.

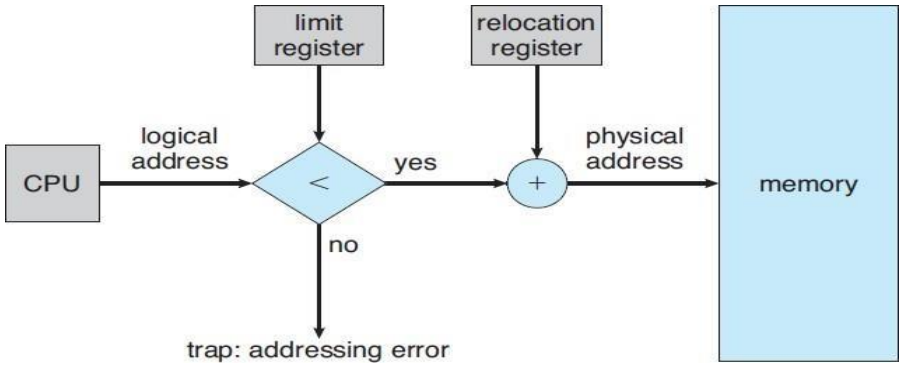
Contiguous Memory Allocation

- The memory is usually divided into two partitions:
  - one for the resident OS
  - one for the user processes.
- We can place the OS in either low memory or high memory (depends on the location of the interrupt vector).
- 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 this contiguous memory allocation, each process is contained in a single contiguous section of memory.

Memory Mapping and Protection

- 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.
- The relocation-register scheme provides an effective way to allow the OS size to change dynamically.

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.

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 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.

Fixed-partition scheme

- The OS 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.

- At any given time, we have a list of available block sizes and the input queue. The OS can order the input queue according to a scheduling algorithm.
- 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.
- This procedure is a particular instance of the general dynamic storage-allocation problem, which concerns how to satisfy a request of size from a list of free holes. There are many solutions to this problem.

    - **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.

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.
- **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.
- **Internal Fragmentation** Memory block assigned to process is bigger than requested. The difference between these two numbers is internal fragmentation; memory that is internal to a partition but is not being used.
- 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.

- 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.
- 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 non-contiguous, thus allowing a process to be allocated physical memory wherever the latter is available.

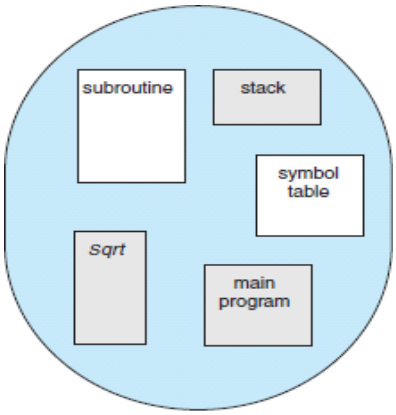
Segmentation

- An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory.
- The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory.

Basic Method

- Users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments

Programmer's view of memory



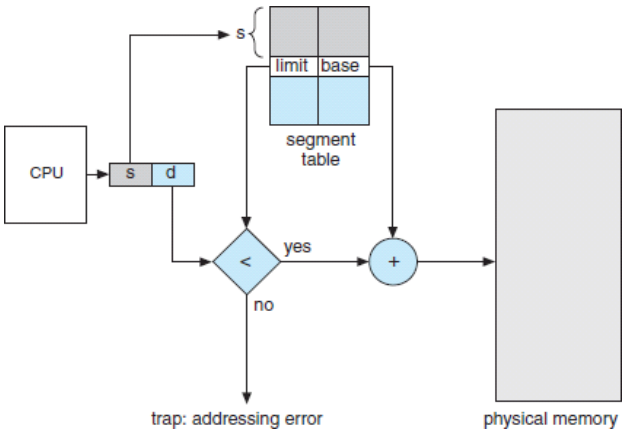
- Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions.
- Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments.
- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities:
    - a segment name
    - 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>

Hardware

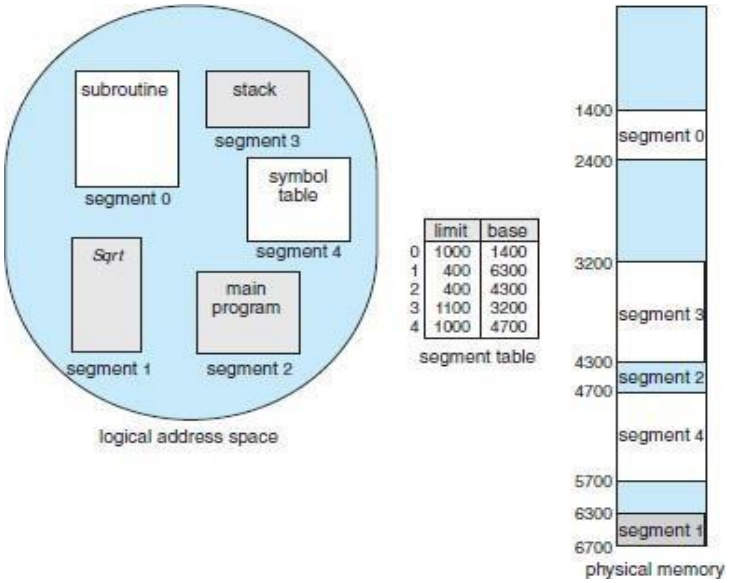
- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes.
- Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.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

Segmentation Hardware



- A logical address consists of two parts: a segment number, sand 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 OS (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.

Example of segmentation



- 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 would result in a trap to the OS, as this segment is only 1,000 bytes long.

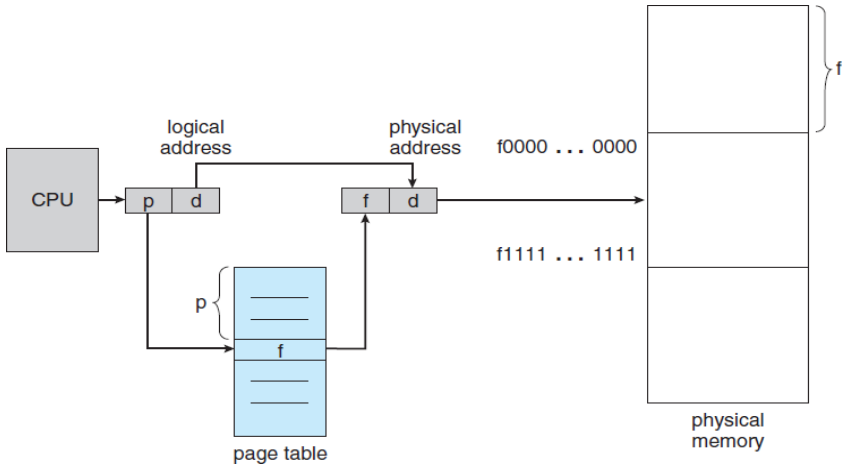
Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.

Basic Method

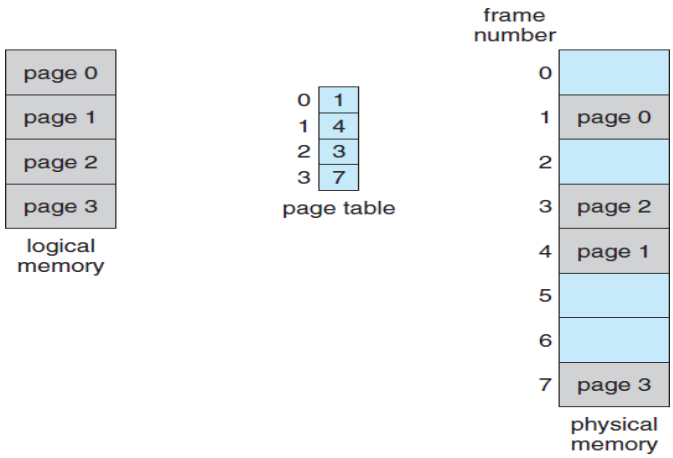
- The basic method for implementing paging involves
  - breaking physical memory into fixed-sized blocks called frames
  - breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.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

Paging Hardware



Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d).

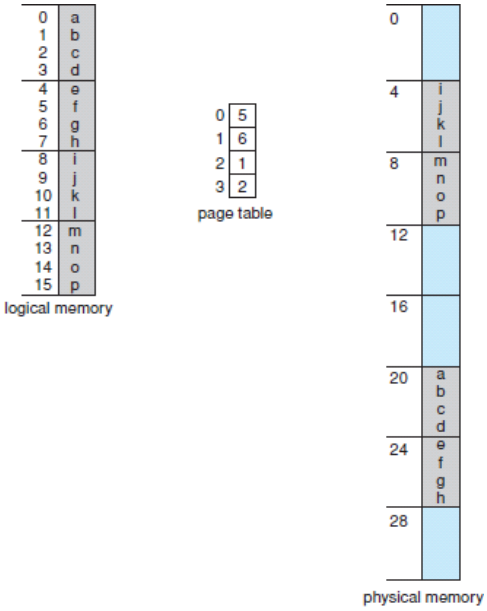
- The page number is used as an index into a page table.
- The page table contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



Paging model of logical and physical memory.

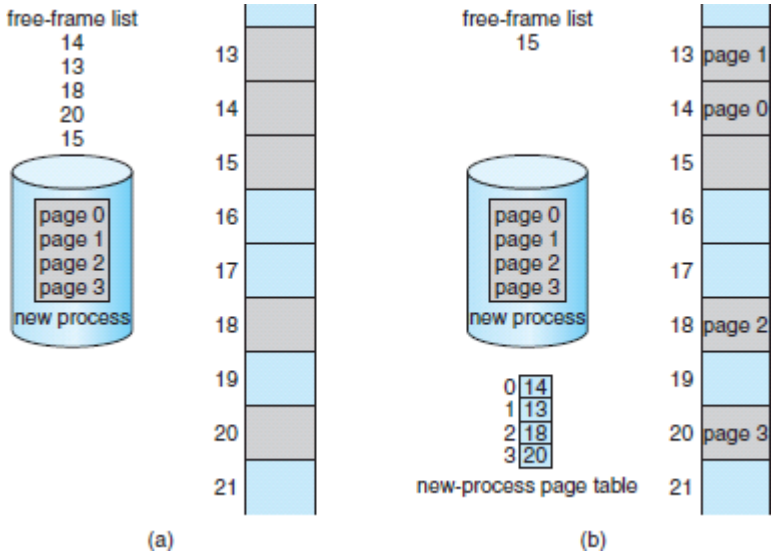
- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.

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



- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages). It is shown that how the user's view of memory can be mapped into physical memory.
    - Logical address 0 is page O, offset O. 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 (= (6 x 4) + 0).
    - Logical address 13 maps to physical address 9.
- 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, and its frame number is put into the page table, and so on

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



- 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.
- The logical addresses are translated into physical addresses by the address-translation hardware. This mapping is hidden from the user and is controlled by the OS.
- The user process has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.
- Since the OS 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.

HardwareSupport

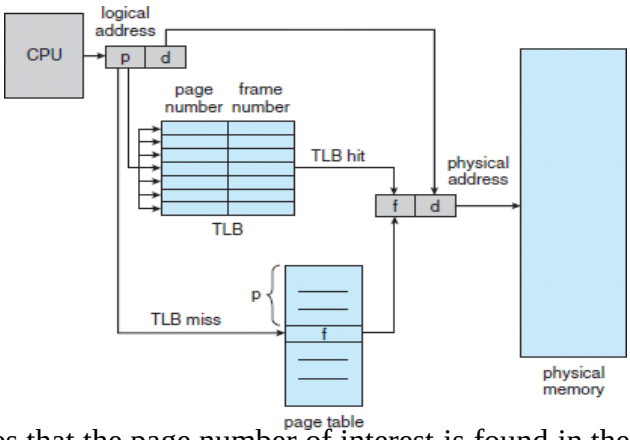
Implementation of Page Table

- Page table is kept in main memory
- Page-table base register (PTBR) points to the page table
- Page-table length register (PRLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. 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)
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

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 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 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.
- If the page number is found, its frame number is immediately available and is used to access memory.

Paging Hardware with TLB



Hit Ratio - The percentage of times that the page number of interest is found in the TLB is called the hit ratio.

Effective Memory Access Time

- 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.

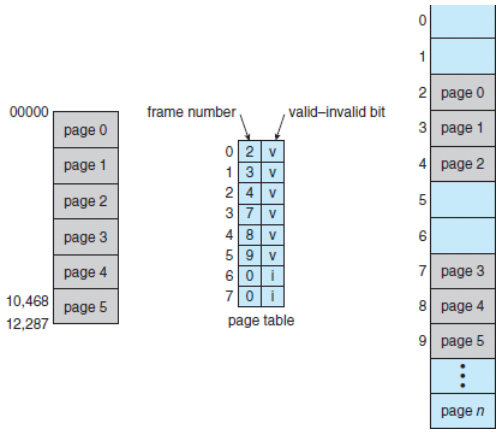
  effective access time = 0.80 × 100 + 0.20 × 200
  = 120 nanoseconds
- For a 99-percent hit ratio, which is much more realistic, we have

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

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.
- Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- 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 OS sets this bit for each page to allow or disallow access to the page.

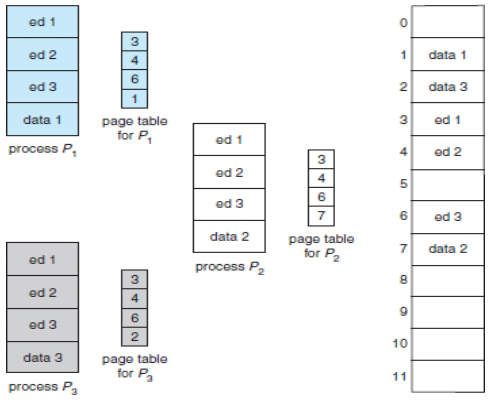
Valid (v) or invalid (i) bit in a page table.



    - Item Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
    - Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the OS

Shared Pages

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment
    - **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.



Sharing of code in a paging environment

- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different.
- Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

Structure of the Page Table

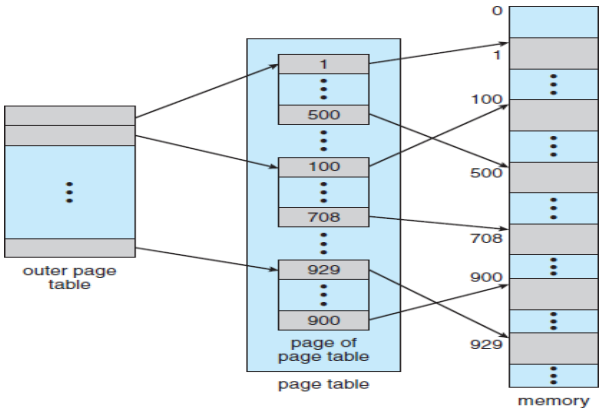
The most common techniques for structuring the page table are

Hierarchical paging
Hashed page tables
Inverted page tables.

Hierarchical Paging

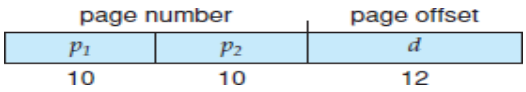
The page table itself becomes large for computers with large logical address space (232 to 264).
Example:

- Consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (212), then a page table may consist of up to 1 million entries (232/212).
- Assuming that each entry consists of 4 bytes,each process may need up to 4 MB of physical address space for the page table alone.
- The page table should be allocated contiguously in main memory.
- The solution to this problem is to divide the page table into smaller pieces.
- One way of dividing the page table is to use a two-level paging algorithm, in which the page table itself is also paged as in the figure:



For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.

Thus, a logical address is as follows:

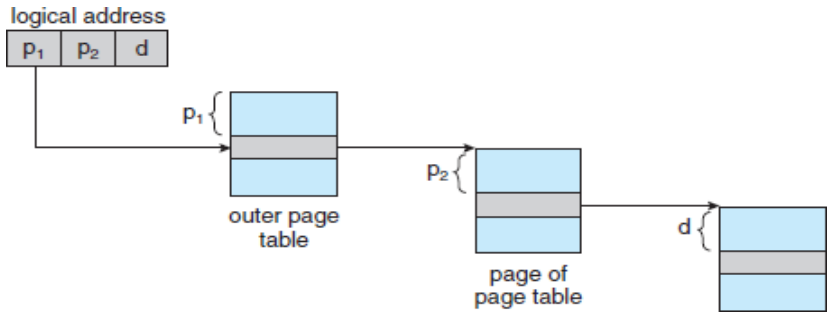


where

*p*1 - an index into the outer page table

*p*2 - the displacement within the page of the inner page table.

The address-translation method for this architecture is shown in the figure. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table.**

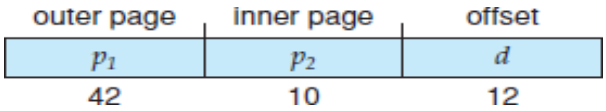


Address translation for a two- level 32-bit paging architecture

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate.

- Suppose that the page size in such a system is 4 KB (212).
  - In this case, the page table consists of up to 252 entries.
  - If a two-level paging scheme is used, then the inner page tables can conveniently be one page long, or contain 210 4-byte entries.

The addresses look like this:

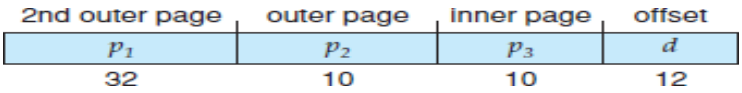


The outer page table consists of 242 entries, or 244 bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces.

The outer page table can be divided in various ways.

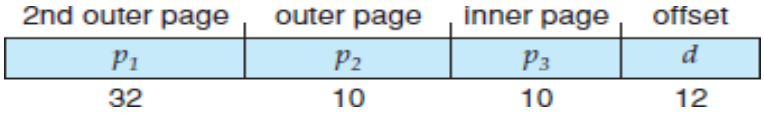
- we can page the outer page table, giving us a three-level paging scheme. Suppose that the outerpage table is made up of standard-size pages (210 entries, or 212 bytes).

In this case, a 64-bit address space is as follows:



The outer page table is still 234 bytes (16 GB) in size.

- The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth.



The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses— to translate each logical address.

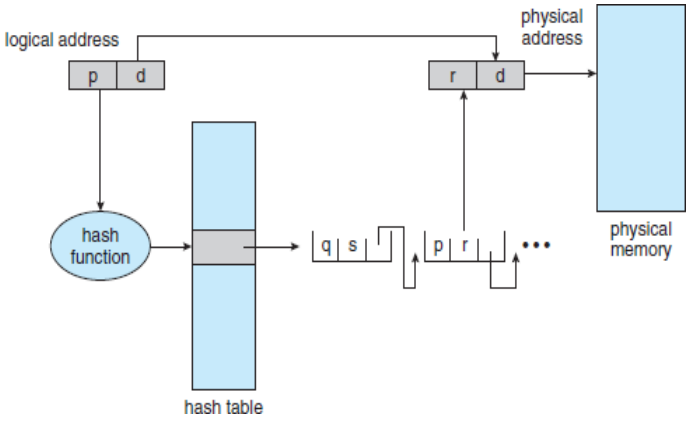
Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).
- Each element consists of three fields:
- the virtual page number
- the value of the mapped page frame
- a pointer to the next element in the linked list.

Algorithm:

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

This scheme is shown below:



- A variation of this scheme that is useful for 64-bit address spaces has been proposed.
- This variation uses clustered page tables, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page.Therefore, a single page-table entry can store the mappings for multiple physical-page

frames.

- Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space.

Inverted Page Tables

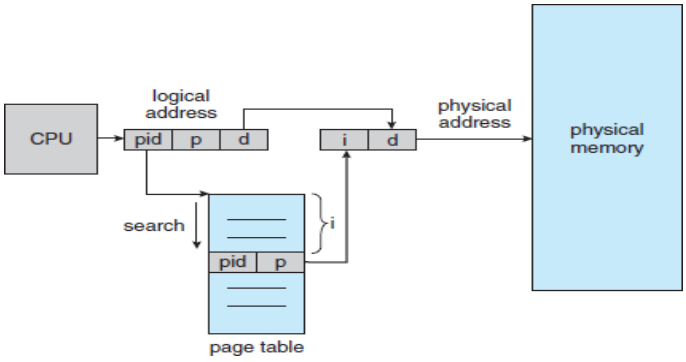
- Each process has an associated page table.
- The page table has one entry for each page that the process is using. This table representation is a natural one, since processes reference pages through the pages' virtual addresses.
- The operating system must then translate this reference into a physical memory address.
- Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly.

Drawbacks of this method

- Each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used. To solve this problem, we can use an inverted page table.

An inverted page table has one entry for each real page (or frame) of memory.

- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page.
- Thus, only one page table is in the system, and it has only one entry for each page of physical memory.



- The operation of an inverted page table is shown above:
- Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory.
- Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

To illustrate this method, a simplified version of the inverted page table used in the IBM RT is described. For the IBM RT, each virtual address in the system consists of a triple:

<Process-id, page-number, offset>.

Where

process-id - the role of the address-space identifier.

- When a memory reference occurs, part of the virtual address, consisting of <process-id, pagenumber>, is presented to the memory subsystem.

- The inverted page table is then searched for a match.
- If a match is found—say, at entry *i*—then the physical address <*i*, offset> is generated.
- If no match is found, then an illegal address access has been attempted.

- Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs.
- Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long.
- To alleviate this problem, a hash table is used, to limit the search to one—or at most a few—page-table entries.
- Each access to the hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table.
- Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses that are mapped to one physical address.
- This standard method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses.
- A simple technique for addressing this issue is to allow the page table to contain only one mapping of a virtual address to the shared physical address. This means that references to virtual addresses that are not mapped result in page faults.

Example: Intel 32 and 64-bit Architectures

- Intel later produced a series of 32-bit chips —the IA-32—which included the family of 32-bit Pentium processors. The IA-32 architecture supported both paging and segmentation.
- More recently, Intel has produced a series of 64-bit chips based on the x86-64 architecture.
- Currently, all the most popular PC operating systems run on Intel chips, including Windows, Mac OS X, and Linux .

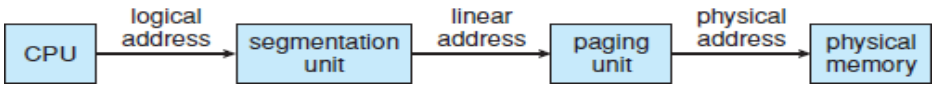
IA-32 Architecture:

- Memory management in IA-32 systems is divided into two components

— segmentation and paging.

It works as follows:

- The CPU generates logical addresses, which are given to the segmentation unit.
- The segmentation unit produces a linear address for each logical address.
- The linear address is then given to the paging unit, which in turn generates the physical address in main
- memory.
- Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU).
- This scheme is shown in Figure.

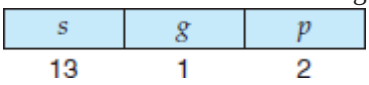


Logical to physical address translation in IA-32.

IA-32 Segmentation

- The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 K.
- The logical address space of a process is divided into two partitions.
  - The first partition consists of up to 8 K segments that are private to that process.
  - The second partition consists of up to 8 K segments that are shared among all the processes.

- Information about the first partition is kept in the local descriptor table (LDT);
- Information about the second partition is kept in the global descriptor table (GDT).
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.



Where S- Segment Number

g- Segment in the GDT or LDT

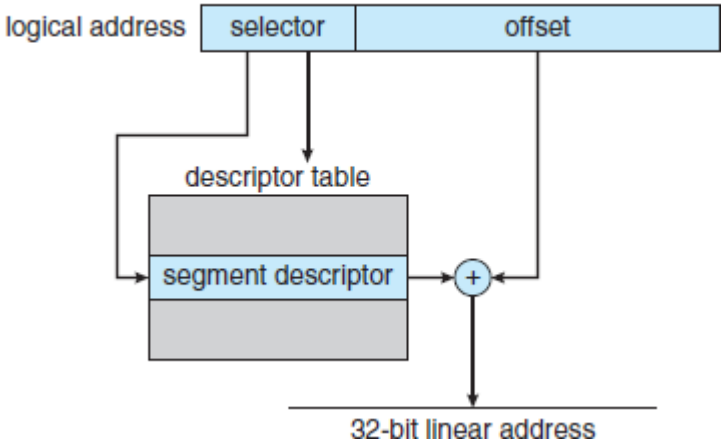
p- Prtoection

The offset is a 32-bit number specifying the location of the byte within the segment in question.

- The machine has six segment registers, allowing six segments to be addressed at any one time by a process.
- It also has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or GDT.
- This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference.

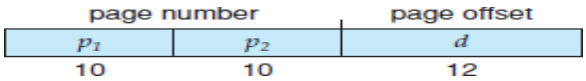
The linear address on the IA-32 is 32 bits long and is formed as follows.

- The segment register points to the appropriate entry in the LDT or GDT.
- The base and limit information about the segment in question is used to generate a linear address.
- First, the limit is used to check for address validity.
- If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. \
- This is shown in Figure .

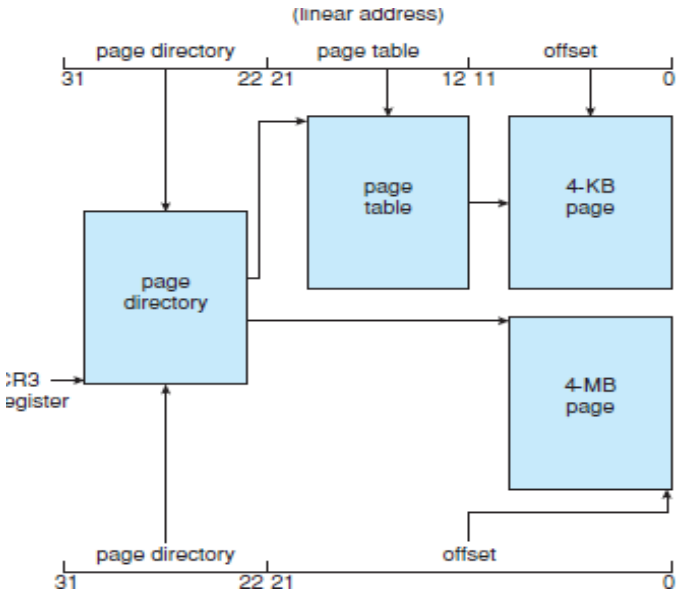


IA-32 Paging

The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



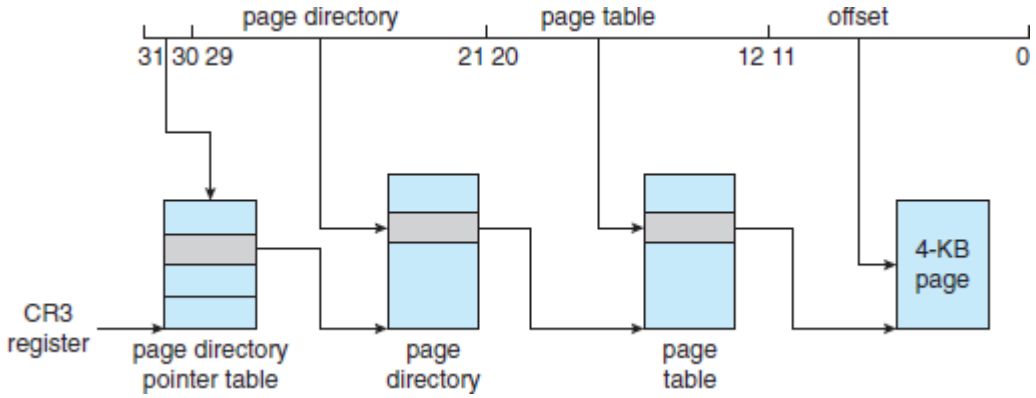
The address-translation scheme for this architecture is similar to the scheme shown in Figure. The IA-32 address translation is shown in Figure.



- The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the page directory. (The CR3 register points to the page directory for the current process.)

- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.
- Finally, the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.
- One entry in the page directory is the Page Size flag, which—if set— indicates that the size of the page frame is 4 MB and not the standard 4 KB.
- If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.
- To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk.
- In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk..
- If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table. The table can then be brought into memory on demand.

Intel adopted a page address extension (PAE), which allows 32-bit processors to access a physical address space larger than 4 GB. The fundamental difference introduced by PAE support was that paging went from a two-level scheme to a three-level scheme, where the top two bits refer to a page directory pointer table. Figure illustrates a PAE system with 4-KB pages. (PAE also supports 2-MB pages.)



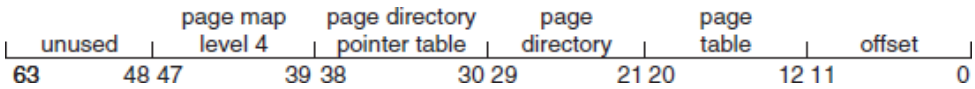
Page address extensions

PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to extend from 20 to 24 bits. Combined with the 12-bit offset, adding PAE support to IA-32 increased the address space to 36 bits, which supports up to 64 GB of physical memory.

x86-64

- The initial entry of Intel developing 64-bit architectures was the IA-64 (later named Itanium) architecture, but was not widely adopted.
- Meanwhile, AMD —began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set.
- The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances.
- Support for a 64-bit address space yields an astonishing 264 bytes of addressable memory—a number greater than 16 quintillion (or 16 exabytes).
- However, even though 64-bit systems can potentially address this much memory, in practice far fewer than 64 bits are used for address representation in current designs.
- The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes

of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy.

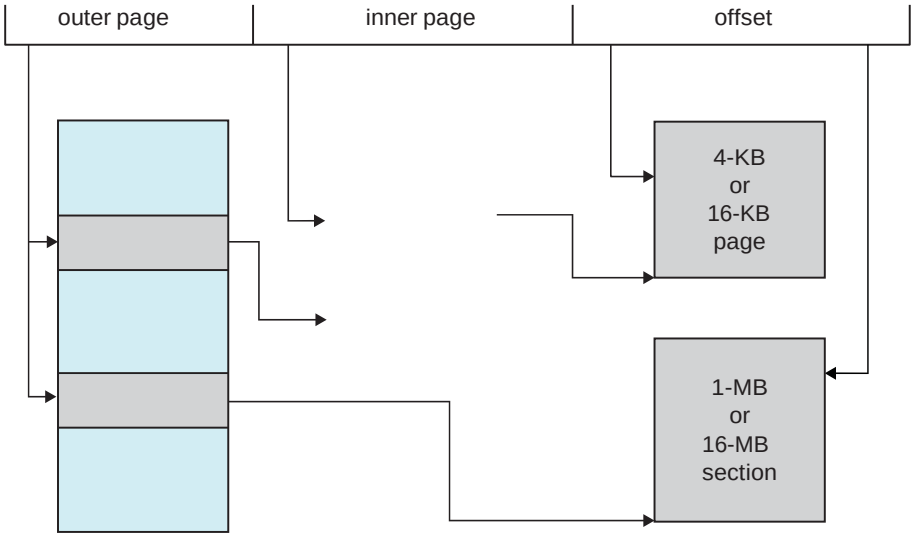


- The representation of the linear address appears in Figure

x86-64 linear address

Because this addressing scheme can use PAE, virtual addresses are 48 bits in size but support 52-bit physical addresses (4096 terabytes).

Example: ARM Architecture



Although Intel chips have dominated the personal computer market for over 30 years, chips for mobile devices such as smart phones and tablet computers often instead run on 32-bit ARM processors. Interestingly, whereas Intel both designs and manufactures chips, ARM only designs them. It then licenses its designs to chip manufacturers. Apple has licensed the ARM design for its iPhone and iPad mobile devices, and several Android-based smart phones use ARM processorsas well.

The 32-bit ARM architecture supports the following page sizes:

1. 4-KB and 16-KB pages
2. 1-MB and 16-MB pages

The paging system in use depends on whether a page or a section is being referenced. One-level paging is used for 1-MB and 16-MB sections; two-level paging is used for 4-KB and 16-KB pages. Address translation with the ARM MMU is shown in Figure 8.26.

The ARM architecture also supports two levels of TLBs. At the outer level are two micro TLBs a separate TLB for data and another for instructions. The micro TLB supports ASIDs as well. At the inner level is a single main TLB.Address translation begins at the micro TLB level. In the case of a miss, the main TLB is then checked. If both TLBs yield misses, a page table walk must beperformed in hardware.