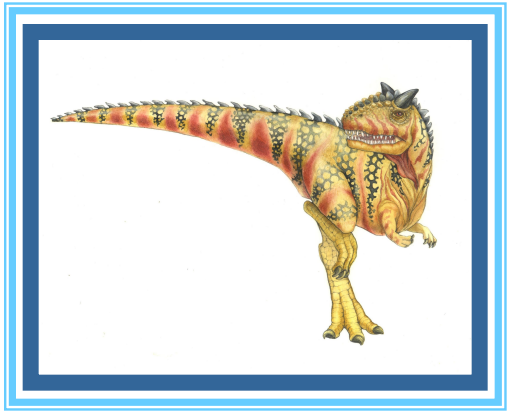
Chapter 9: Main Memory



Silberschatz, Galvin and Gagne ©2018

Chapter 9: Memory Management

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

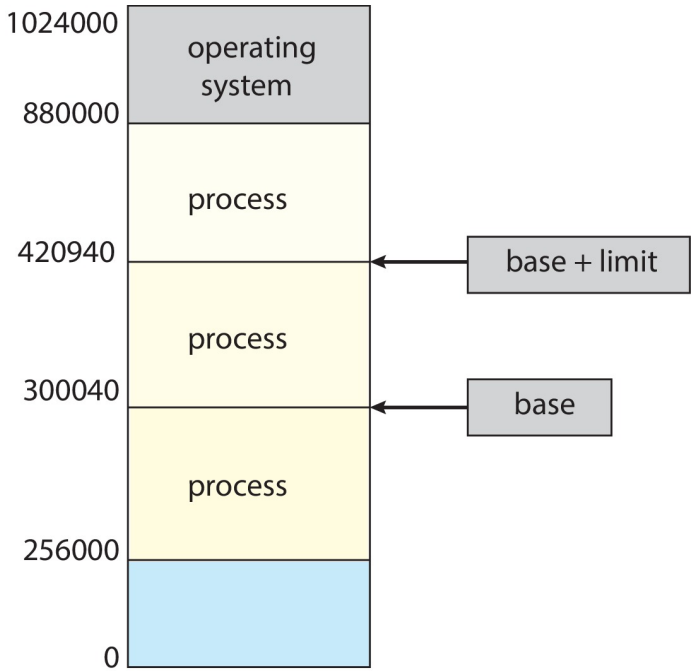
- Memory is central to the operation of a modern computer system with a large array of bytes, each with its own address.
- Main memory and registers are only storage CPU can access directly. Program must be brought (from disk) into memory and placed within a process for it to be run
- These instructions may cause additional loading from and storing to specific memory addresses.
- A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- There are machine instructions that take memory addresses as arguments, and not 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.

- Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. Whereas Completing a memory access may take many cycles of the CPU clock. In such a case main memory can take many cycles, causing a **stall.**
- This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. **Cache** sits between main memory and CPU registers

- we also must ensure correct operation so protection of memory required to ensure correct operation
  - the operating system from access by user processes
  - On multiuser systems, we must additionally protect user processes from one another.

Protection

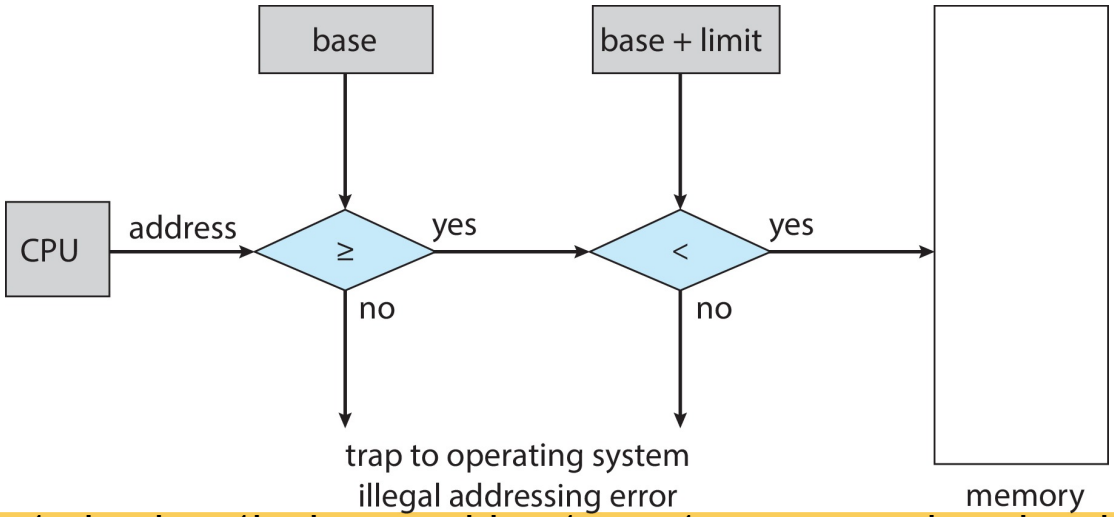
- each process has a separate memory space
- Need to ensure that a process can access only access those addresses in it address space and it is fundamental to having multiple processes loaded in memory for concurrent execution.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Hardware Address Protection

- 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 operating system, which treats the attempt as a fatal error
-



- the instructions to loading the base and limit registers are privileged and are done by the OS executed in the kernel mode.
  - prevents user programs from changing the registers' contents

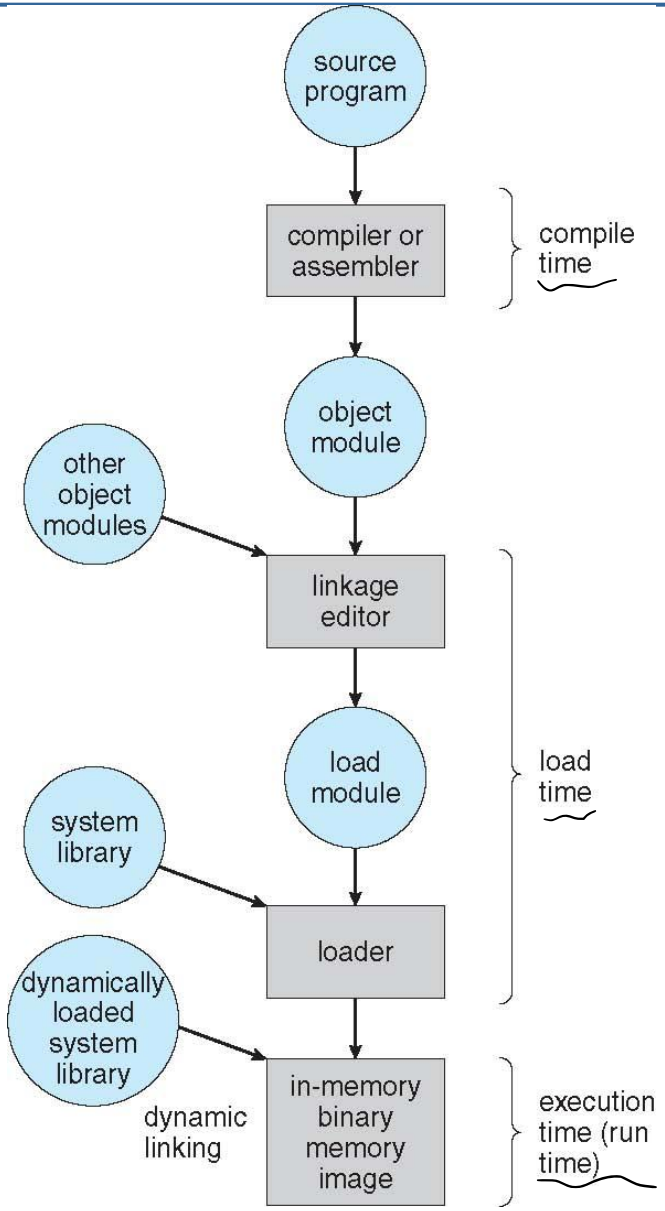
Address Binding

- 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.. The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.
- The normal single-tasking 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, though the address space of the computer may start at 00000, the first address of the user process need not be 00000.
- Addresses in the source program are generally symbolic
- A compiler typically **binds** these symbolic addresses to relocatable addresses, i.e. "14 bytes from beginning of this module"
- The linkage editor or loader in turn binds 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

- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile time**. If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that location and extend up from there. 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.
  - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



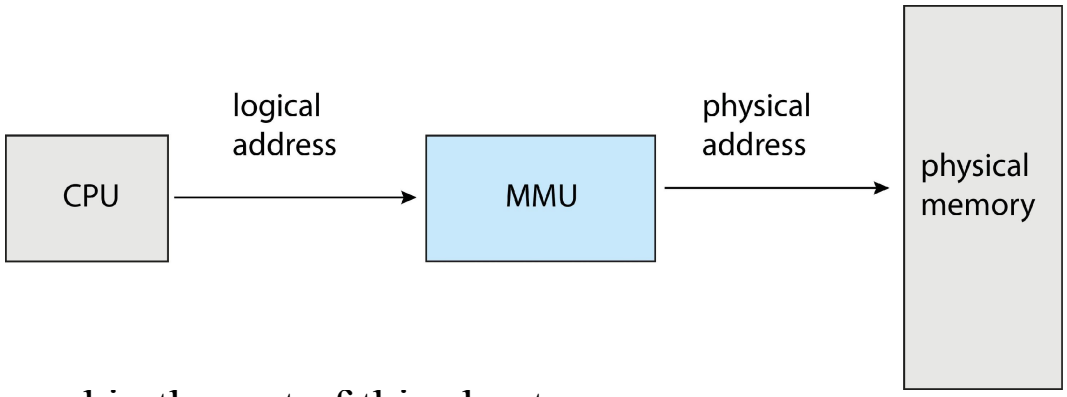
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit, the one loaded into the **memory-address register** of the memory
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

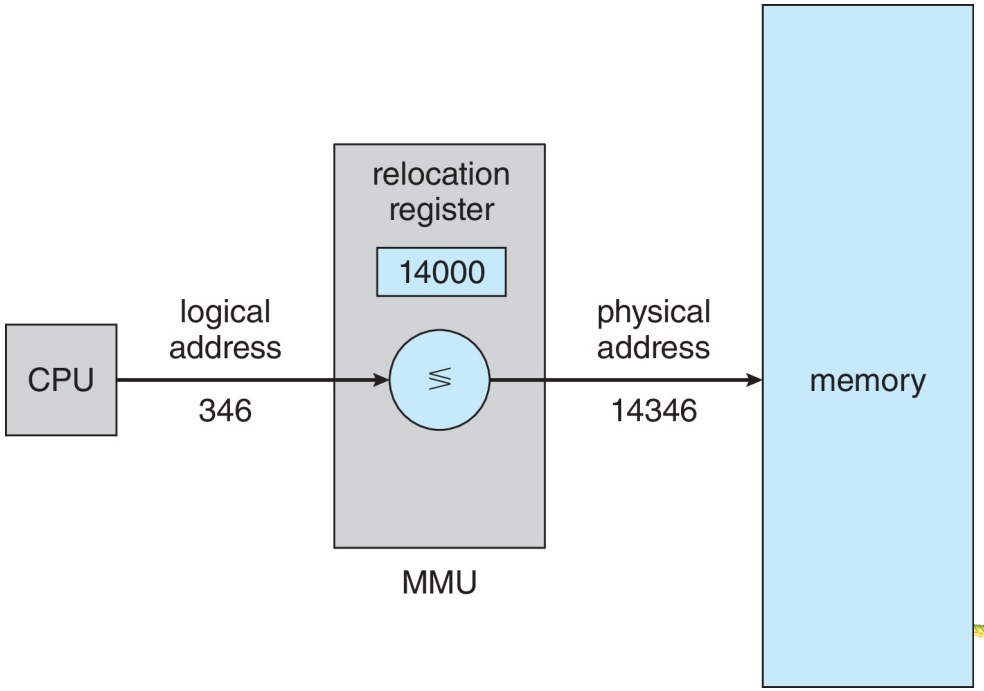
Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



- Many methods possible, covered in the rest of this chapter



Memory-Management Unit (Cont.)

- We illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme
- The base register now called **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 user program deals with logical addresses; it never sees the real physical addresses. It can create a pointer to the location.
- Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register by the memory-mapping hardware
  - Execution-time binding occurs when reference is made to location in memory
- The final location of a referenced memory address is not determined until the reference is made.
- 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 concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Dynamic Loading

- It has been necessary for the entire program and all data of a process to be in physical memory for the process to execute but the size of a process has thus been limited to the size of physical memory
- We can choose to that Routine is not loaded until it is called
- Better memory-space utilization can be done by using **dynamic loading**; unused routine is never loaded
- All routines kept on disk in 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 it has 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.
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - It is the responsibility of either the users to design their programs
  - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- **Static linking** – system libraries are treated like any other object module and are combined by the loader into the binary program image. **Disadvantage -** Each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement wastes both disk space and main memory.
- **Dynamically linked libraries** are system libraries that are linked to user programs when the programs are run – linking 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 it is not, the program loads the routine into memory.
- Stub replaces itself with the address of the routine, and executes the routine
- Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
- Also all processes that use a language library execute only one copy of the library code.

Shared Libraries

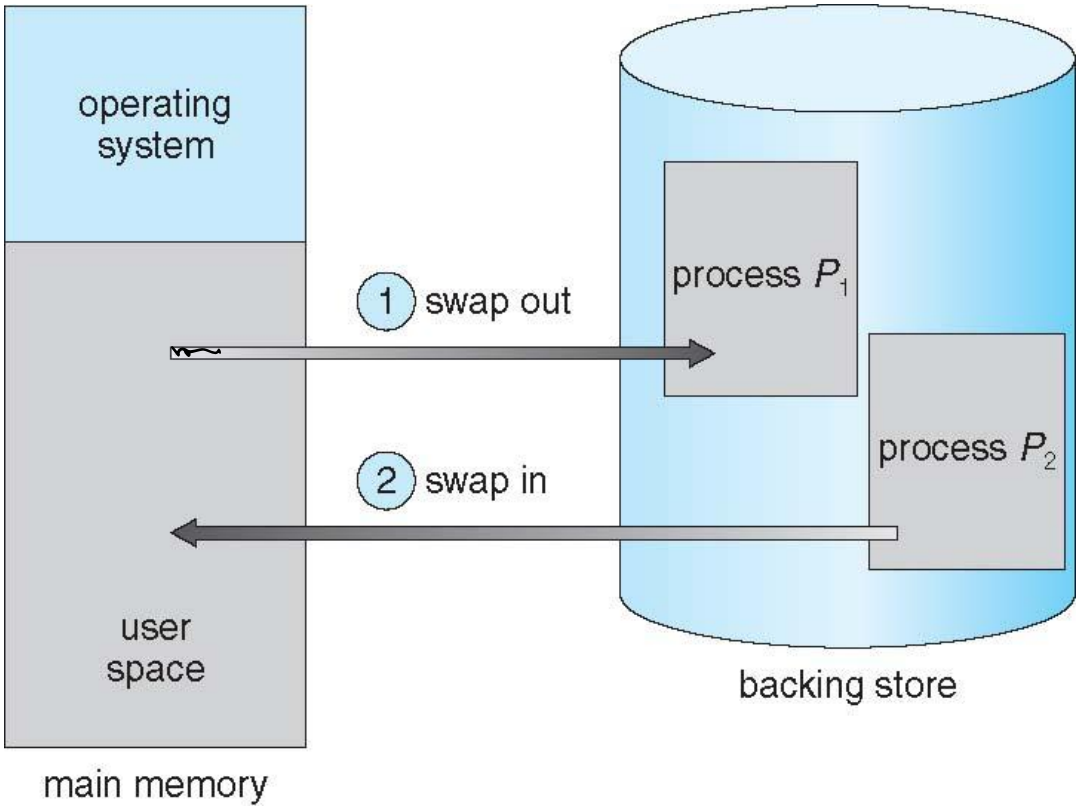
- This feature can be extended to library updates (such as bug fixes).A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
- Without dynamic linking, More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use.
- Versions with minor changes retain the same version number, whereas versions with major changes increment the number. Thus, only programs that are compiled with the new library version are affected by any 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.

Swapping

- A process must be in memory to be executed.
- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Makes it possible for total physical memory space of processes can exceed physical memory thus increasing the degree of multiprogramming in a system.
- **Standard swapping** involves moving processes between main memory and a backing store.
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk or in memory.
- **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

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space

Schematic View of Swapping



Swapping (Cont.)

- Major part of swap time is transfer time; 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.
    4. System calls to inform OS of memory use via `request_memory()` and `release_memory()`
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

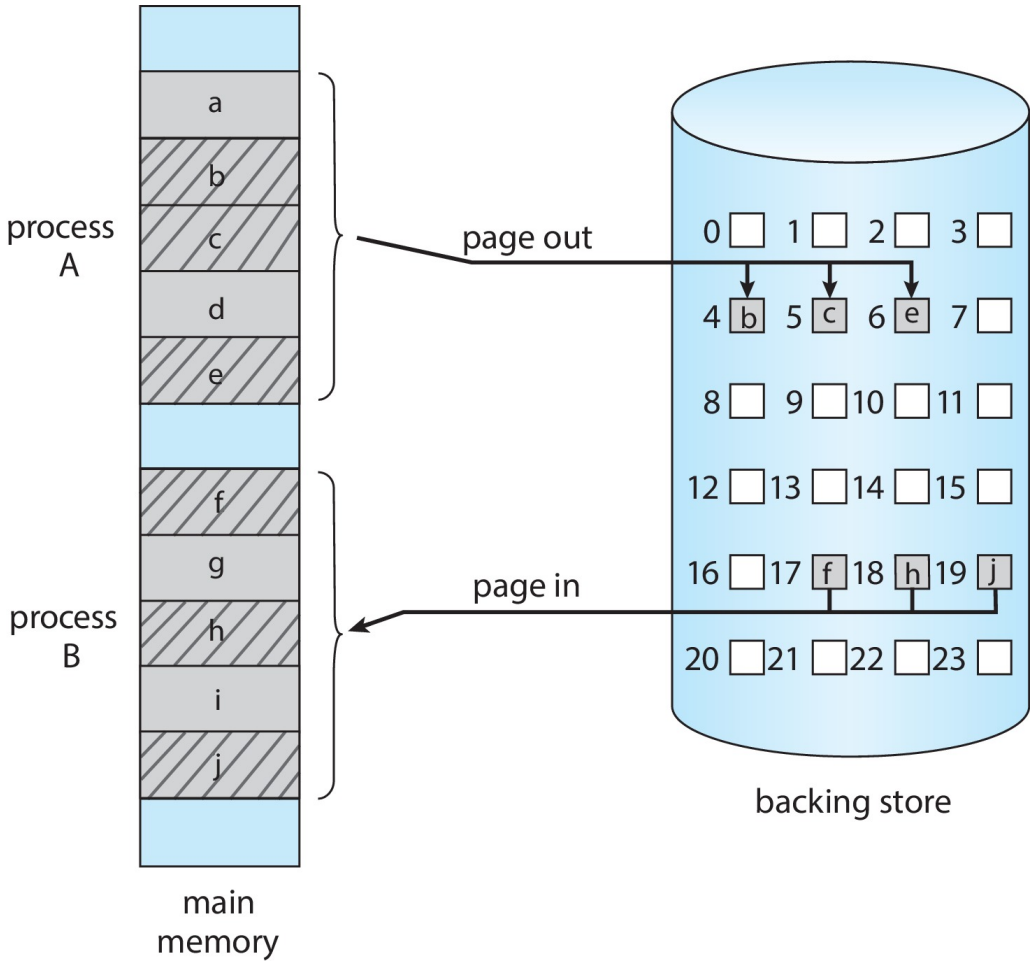
Context Switch Time including Swapping

- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time in such a swapping system can 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)

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping - to swap a process, we must be sure that it is completely idle.
  - Pending I/O – can't swap out as I/O would occur to wrong process - if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped.
  - Or always execute I/O operations only into operating-system buffers. A transfers between operating-system buffers and process memory then occur only when the process is swapped in, Known as **double buffering,** adds overhead
- Standard swapping not used in modern operating systems as It requires too much swapping time and provides too little execution time to be a reasonable memory
- But modified version common
  - swapping is normally disabled but will start if the amount of free memory falls below a threshold amount. Swapping is halted when the amount of free memory increases.
  - swapping portions of processes—rather than entire processes—to decrease swap time. Typically, these modified forms of swapping work in conjunction with virtual memory,

Swapping with Paging



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

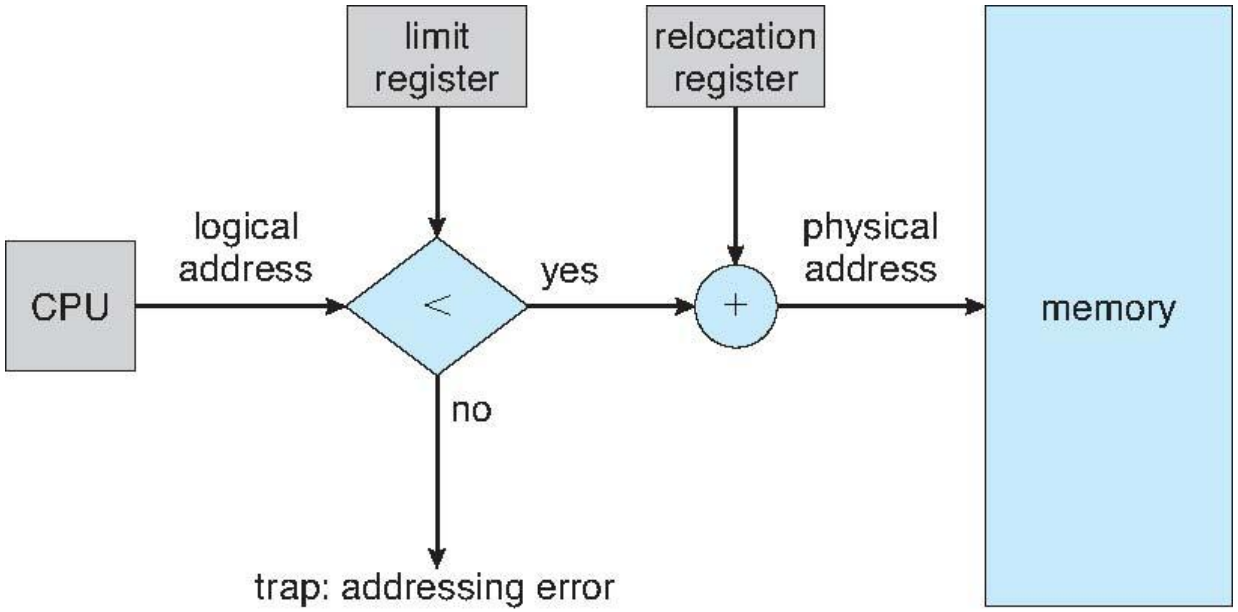
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, so need to allocate main memory in the most efficient way possible
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - One for Resident operating system, usually held in either low memory along with interrupt vector
  - User processes then held in high 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.

Contiguous Allocation (Cont.)

- 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 and Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address dynamically by adding the value in the relocation register and is sent to memory.
- 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.

Hardware Support for Relocation and Limit Registers



Memory Allocation Methods

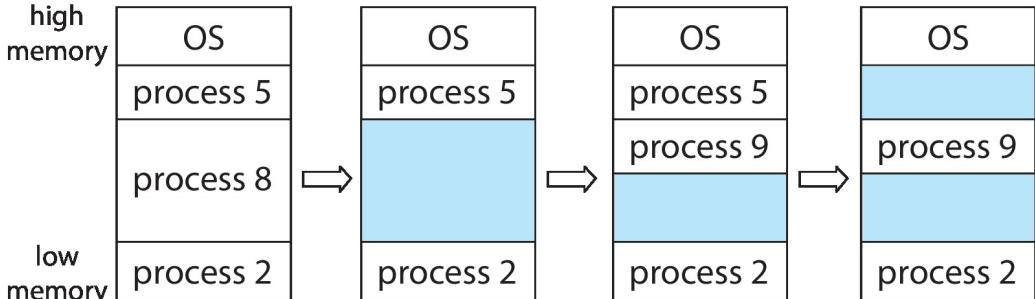
- **Fixed Partition Allocation** - Divide memory into several fixed-sized partitions. Each partition may contain exactly one process.
- **Multiple-partition allocation** - when a partition is free, a process is selected from the input queue and is loaded into the free partition
  - When the process terminates, the partition becomes available for another process.
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition Allocation -** sizes for efficiency (sized to a given process' needs)
  - the operating system keeps a table indicating which parts of memory are available and which are occupied.
  - **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 (hole)

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. 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.**



Fragmentation

- 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** 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.
  - 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.
  - 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?)
- First fit's statistical analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation (Cont.)

- Internal Fragmentation - 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.

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - If relocation is static and is done at assembly or load time, compaction cannot be done. Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - 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 be combined as well.

Segmentation

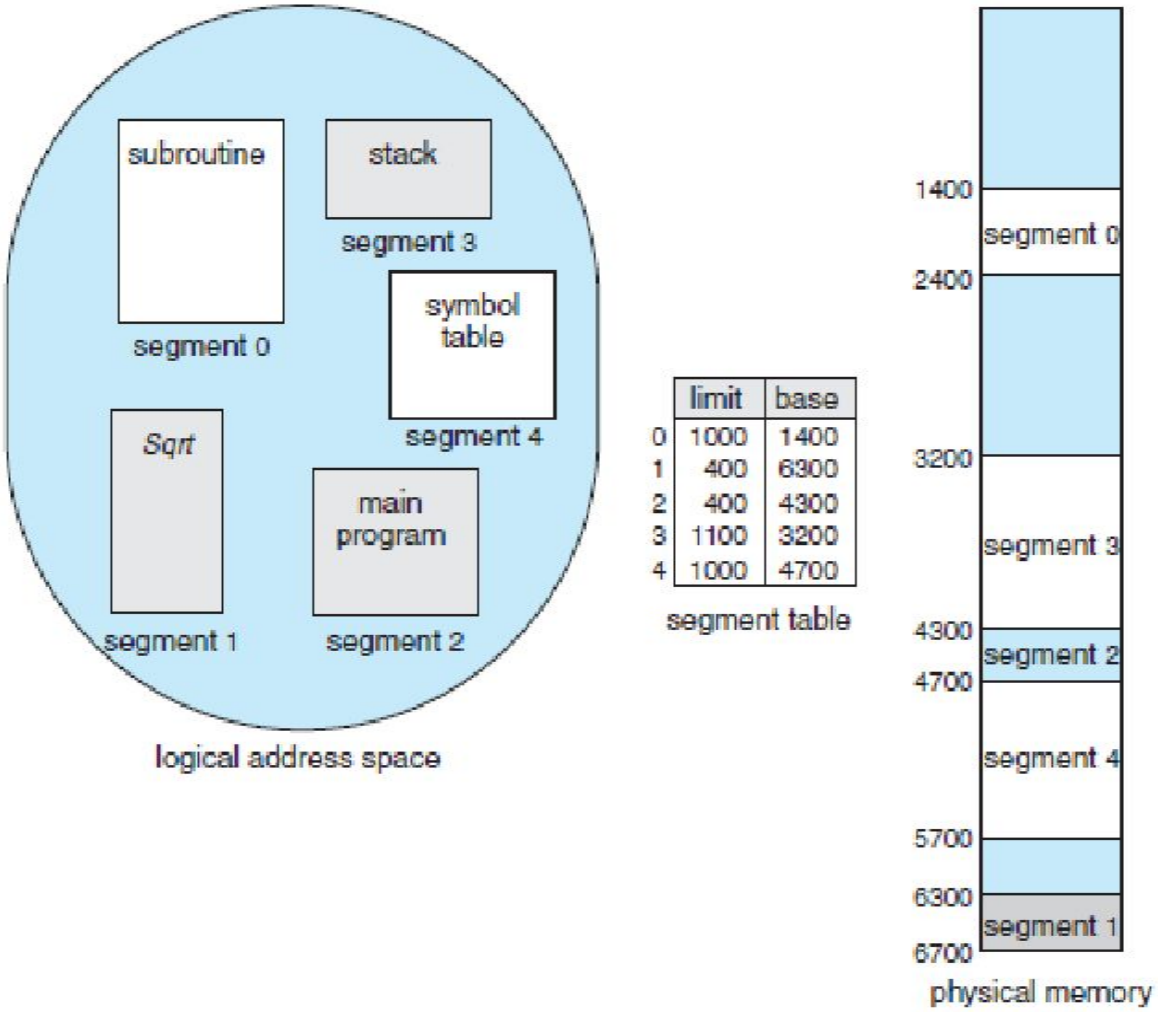


Figure 8.9 Example of segmentation.

Paging

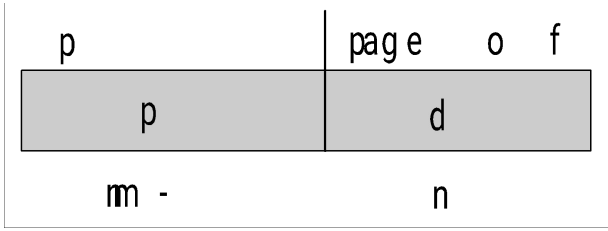
- Segmentation permits the physical address space of a process to be noncontiguous. But does not resolve the problem of external fragmentation completely.
- **Paging** is another memory-management scheme that offers this advantage.
- Paging avoids external fragmentation and the need for compaction.
- Paging is implemented through cooperation between the operating system and the computer hardware.
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- 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.

Paging

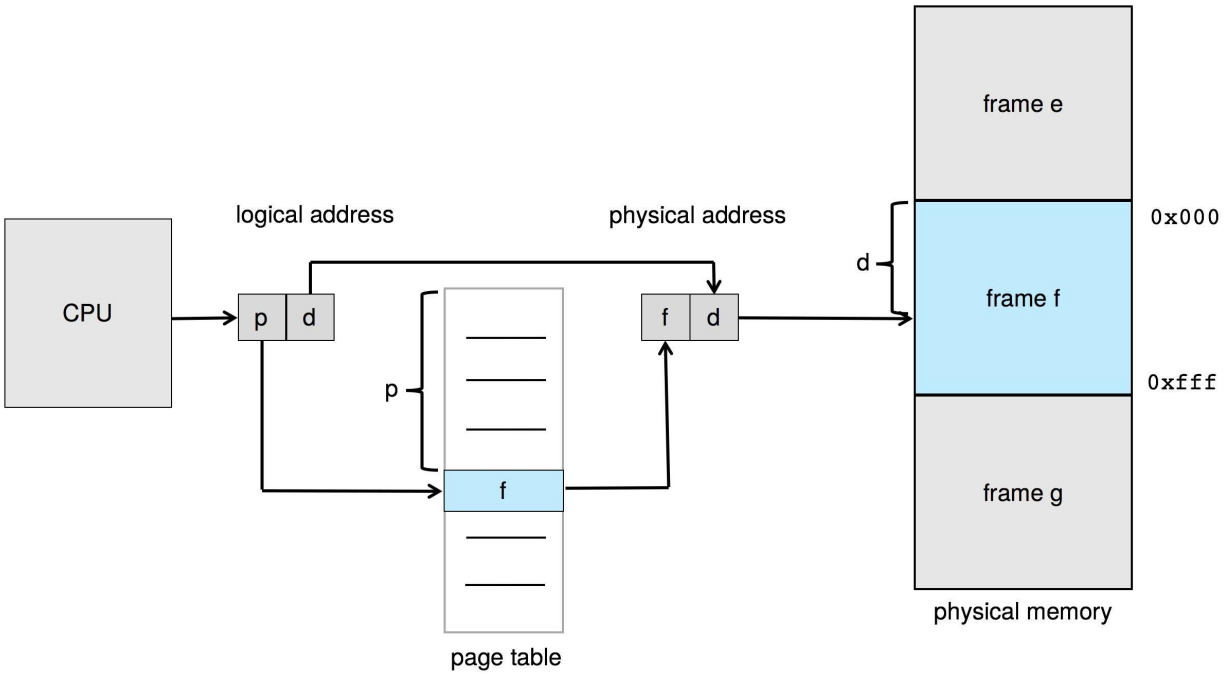
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Still have Internal fragmentation
- 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.
- 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 2^{64} bytes of physical memory.

Address Translation Scheme and Paging Hardware

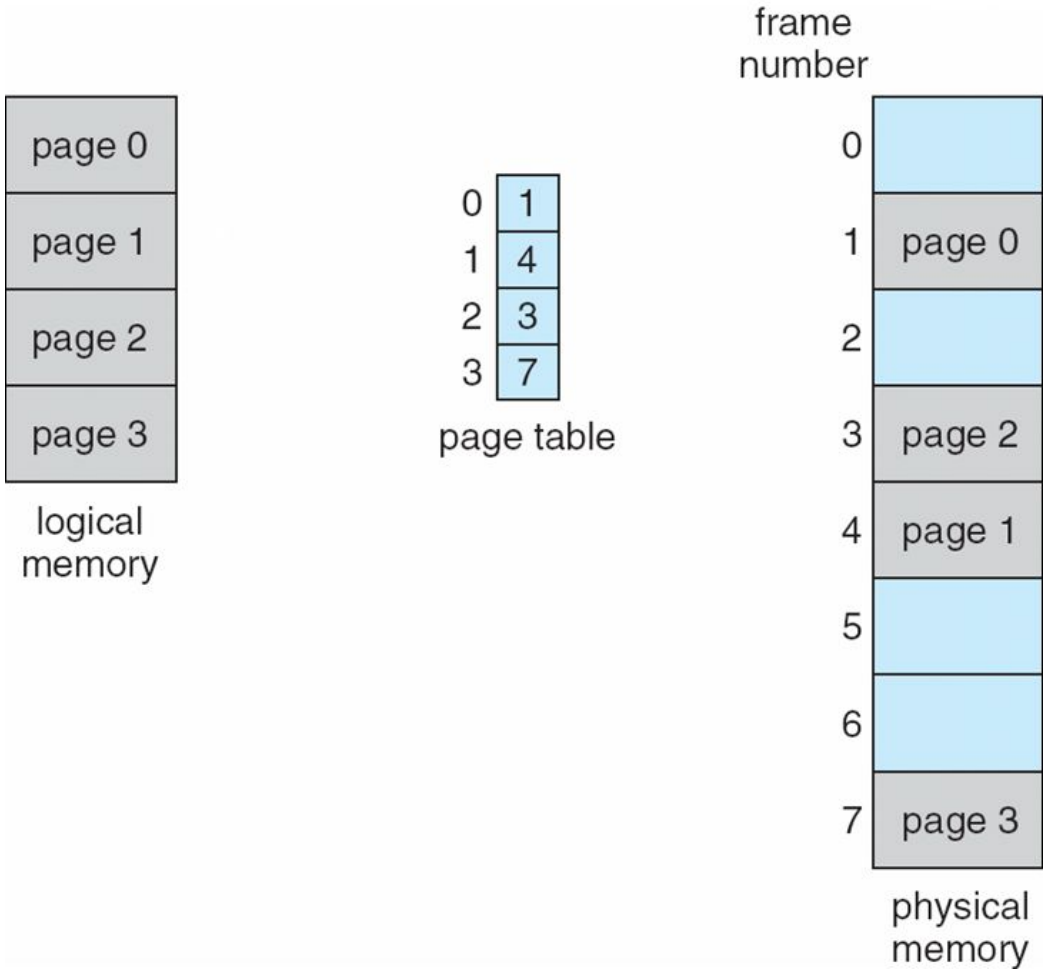
- Address generated by CPU is divided into:
  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit
  - For given logical address space 2^m and page size 2^n



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

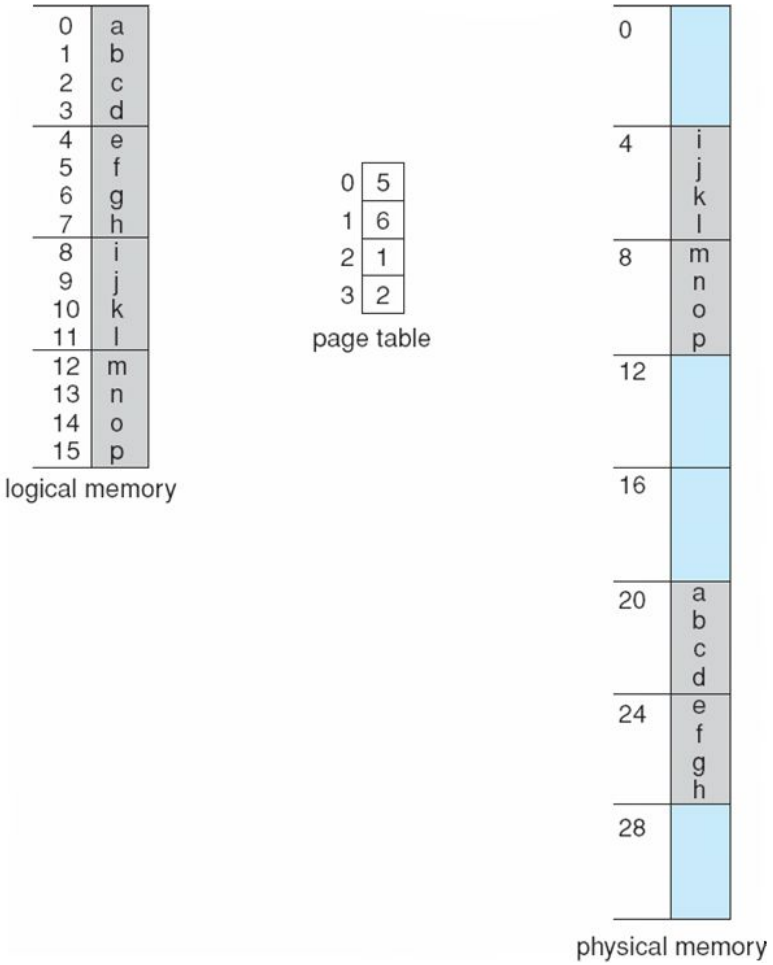


Paging Model of Logical and Physical Memory



Paging Example

- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

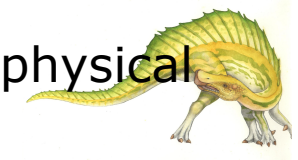


Logical address 0 is page 0, offset 0. Indexing into the page table - page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0].

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.

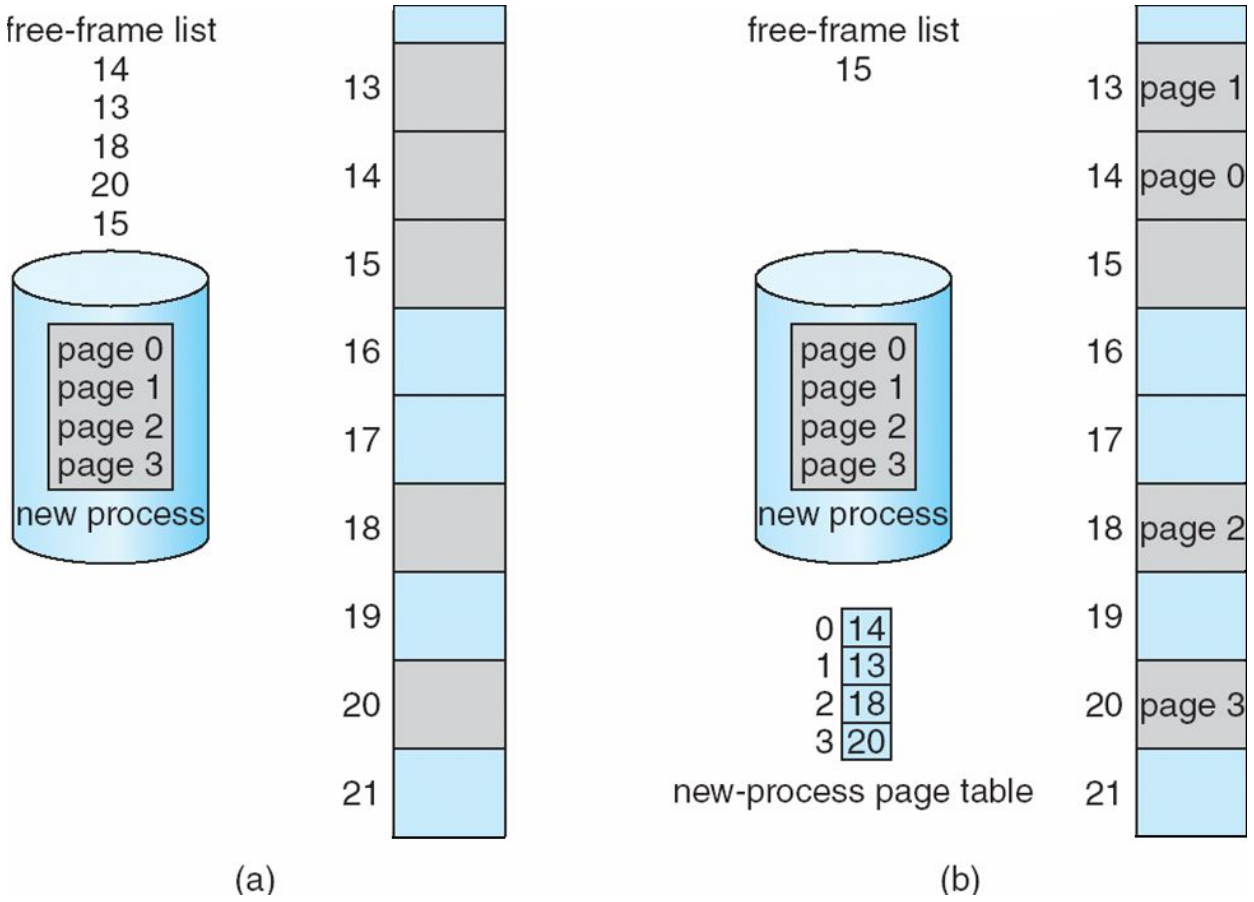


Paging -- Calculating internal fragmentation

- Paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address.
- Page size = 2,048 bytes Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes - It will be allocated 36 frames
- 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.
- So small frame sizes desirable? - 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.
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB
  - 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. A32-bit entry can point to one of 2^{32} physical page frames. If frame size is 4 KB (2^{12}), then a system with 4-byte entries can address 2^{44} 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.
- Further we will 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 2^{32} 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.

Free Frames



Before allocation

After allocation

- 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.

Hardware Implementation of Page Table

- 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.
- 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.
- 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.
- The page table is implemented as a set of dedicated registers given they are built with very high-speed logic to make the paging-address translation efficient as every access to memory must go through the paging map

Implementation of Page Table

- 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
- In this scheme every data/instruction access requires two memory accesses
  - 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.
- Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances.
- The two memory access problem can be solved by the use of a special, small and fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**/high-speed memory). It is typically between 32 and 1,024 entries in size.

Translation Look-Aside Buffer

- 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 TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system.
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - 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
  - Otherwise every time a new page table is selected with every context switch
  - 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.

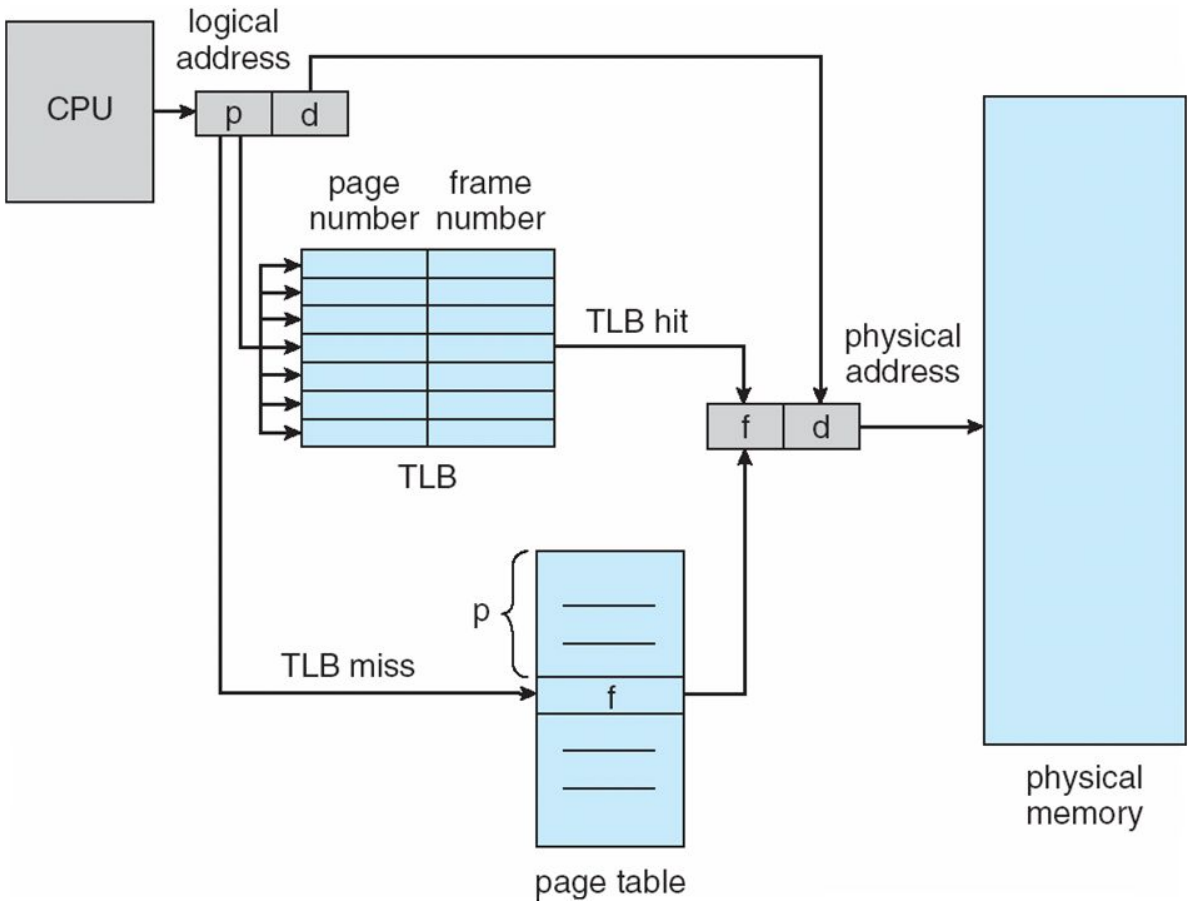
Hardware

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
- If we find the desired page in TLB then a mapped-memory access take 10 ns
- Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds
- Here we suffer a 20-percent slowdown in average memory-access time
- Consider a more realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

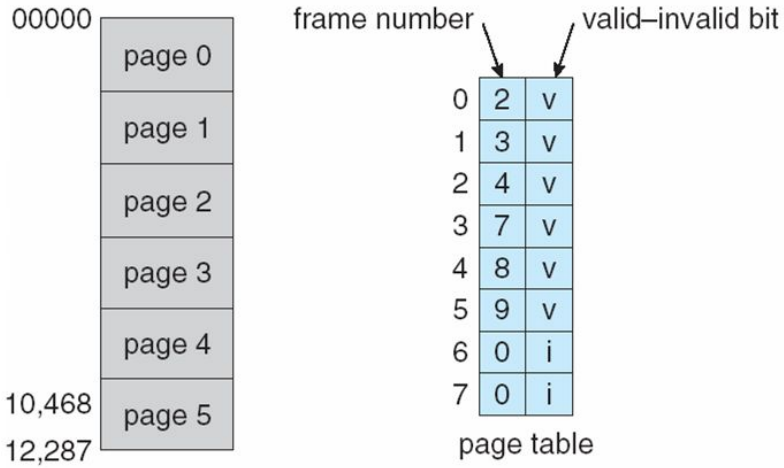
  implying only 1% slowdown in access time.
- CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated

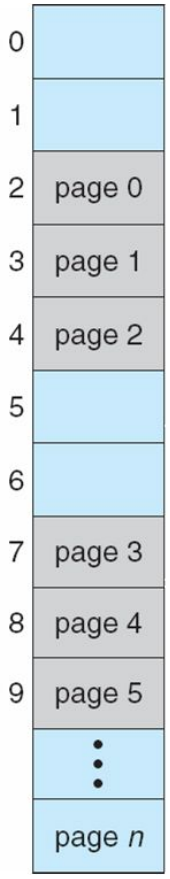
Memory Protection

- Memory protection implemented by associating protection bit with each frame kept in the page table.
  - One bit defines a if a page is read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. An attempt to write to a read-only page causes a hardware trap to the operating system.
  - Can also add more bits to indicate page execute-only, and can allow any combination of these accesses
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
- 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. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable memory space. 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. Failure of this test causes an error trap to the operating system.
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB. 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 operating system (invalid page reference).



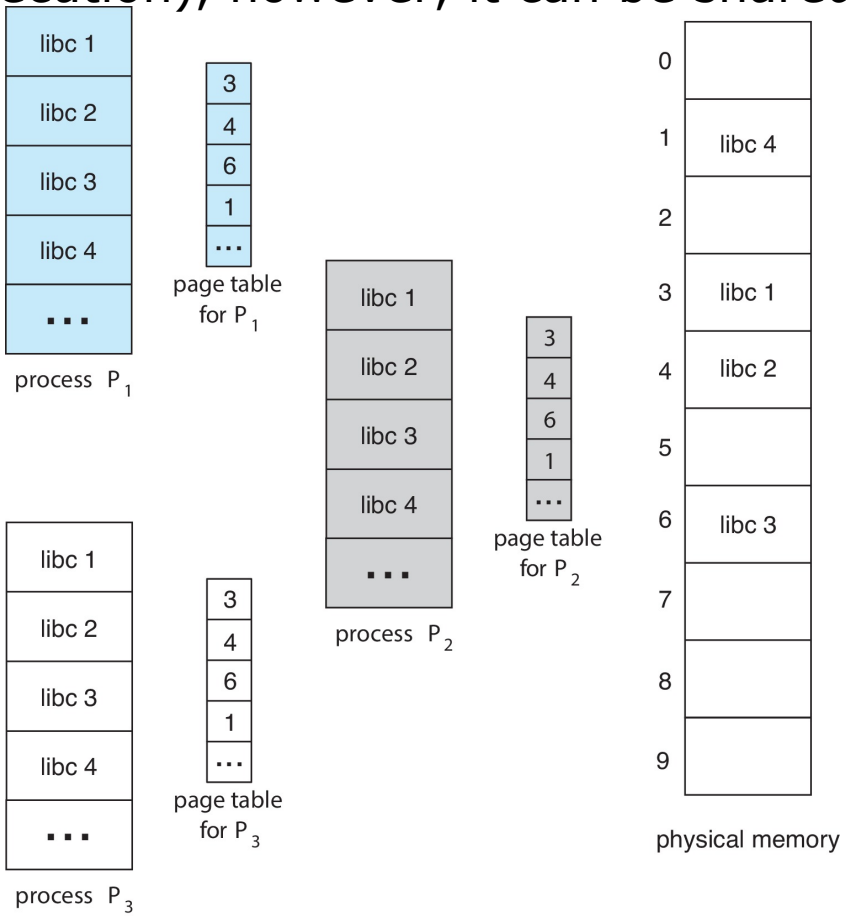


Shared Pages

- Considering a time-sharing environment, paging is the possibility of sharing common code.
- **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 interprocess communication if sharing of read-write pages is allowed
  - Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages.
- **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

Shared Pages Example

Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code (or pure code - non-self-modifying code: it never changes during execution), however, it can be shared. Each process has its own data page.



Each process has its own copy of registers and data storage to hold the data for the process's execution

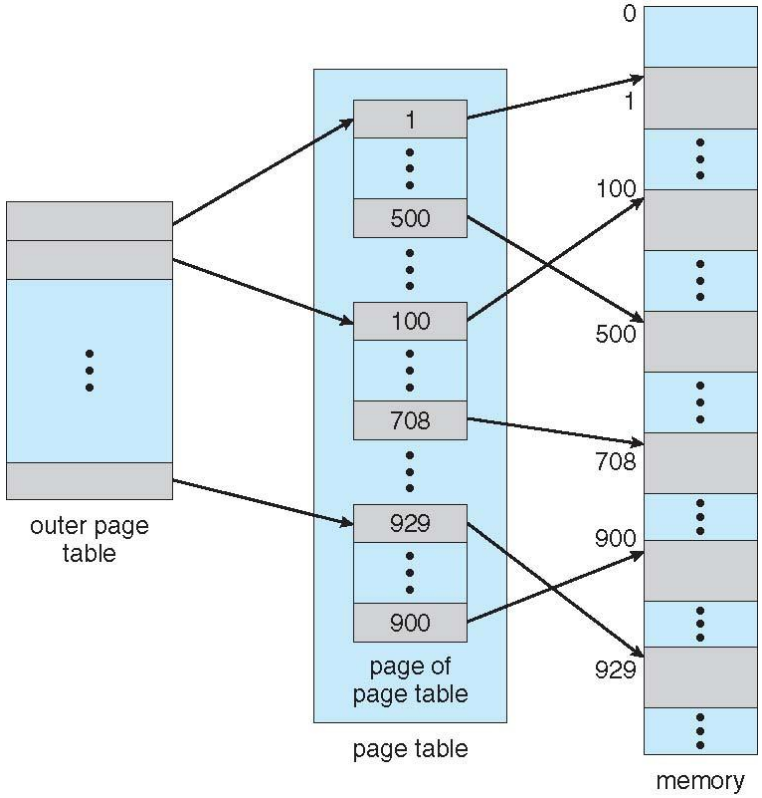
Two or more processes can execute the same code at the same time.

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes □ each process 4 MB of physical address space for the page table alone
    - Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - 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 in which the page table itself is also paged
- We then page the page table

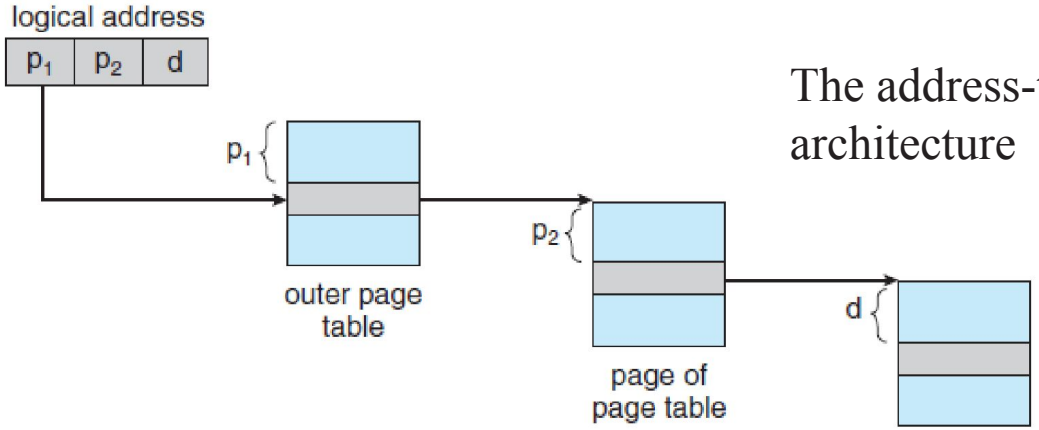


Two-Level Paging Example

- 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 10-bit page number
  - a 12-bit page offset

page number		page offset
p_1	p_2	d
10	10	12

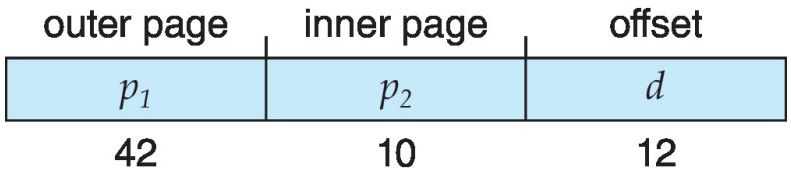
- Thus, a logical address is as follows:
  - where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.



The address-translation method for this architecture

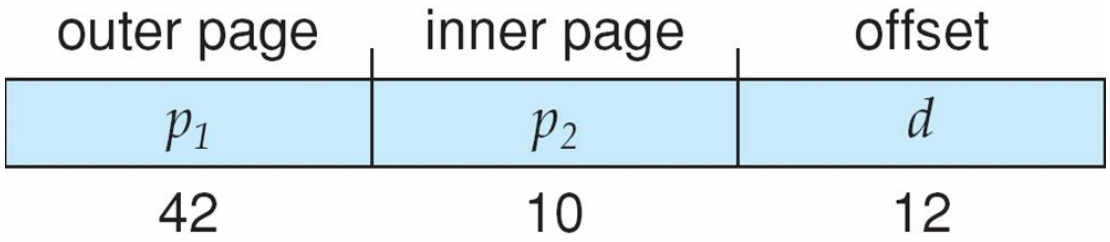
64-bit Logical Address Space

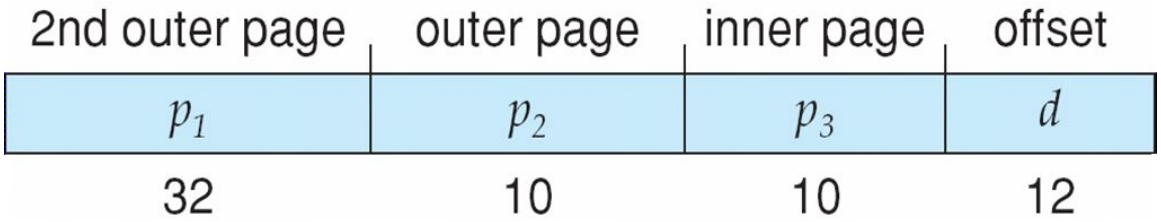
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme





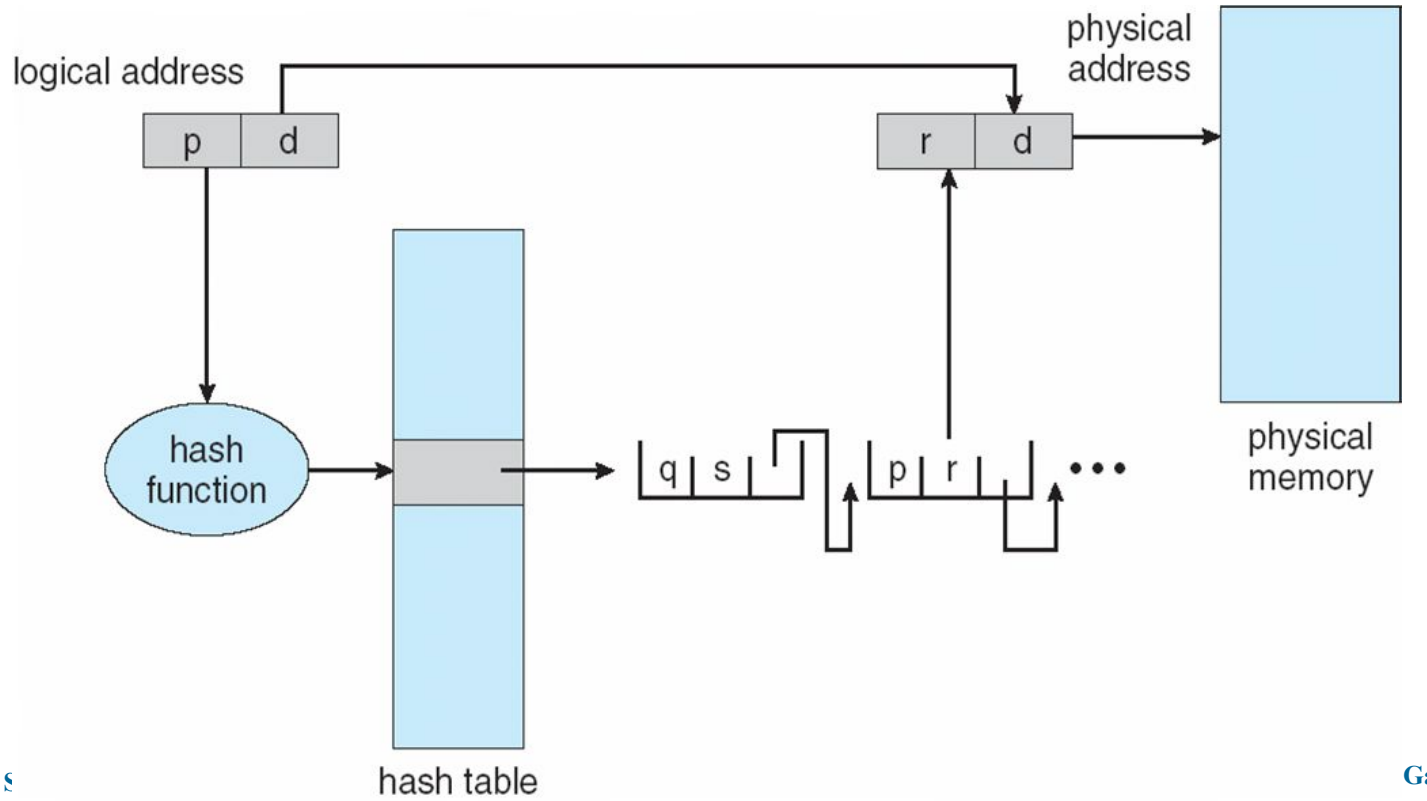
The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses—to translate each logical address.

Hashed Page Tables

- A hashed page table is common in handling address spaces > 32 bits 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).
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Therefore, a single page-table entry can store the mappings for multiple physical-page frames.
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered throughout the address space.)

Hashed Page Table

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.

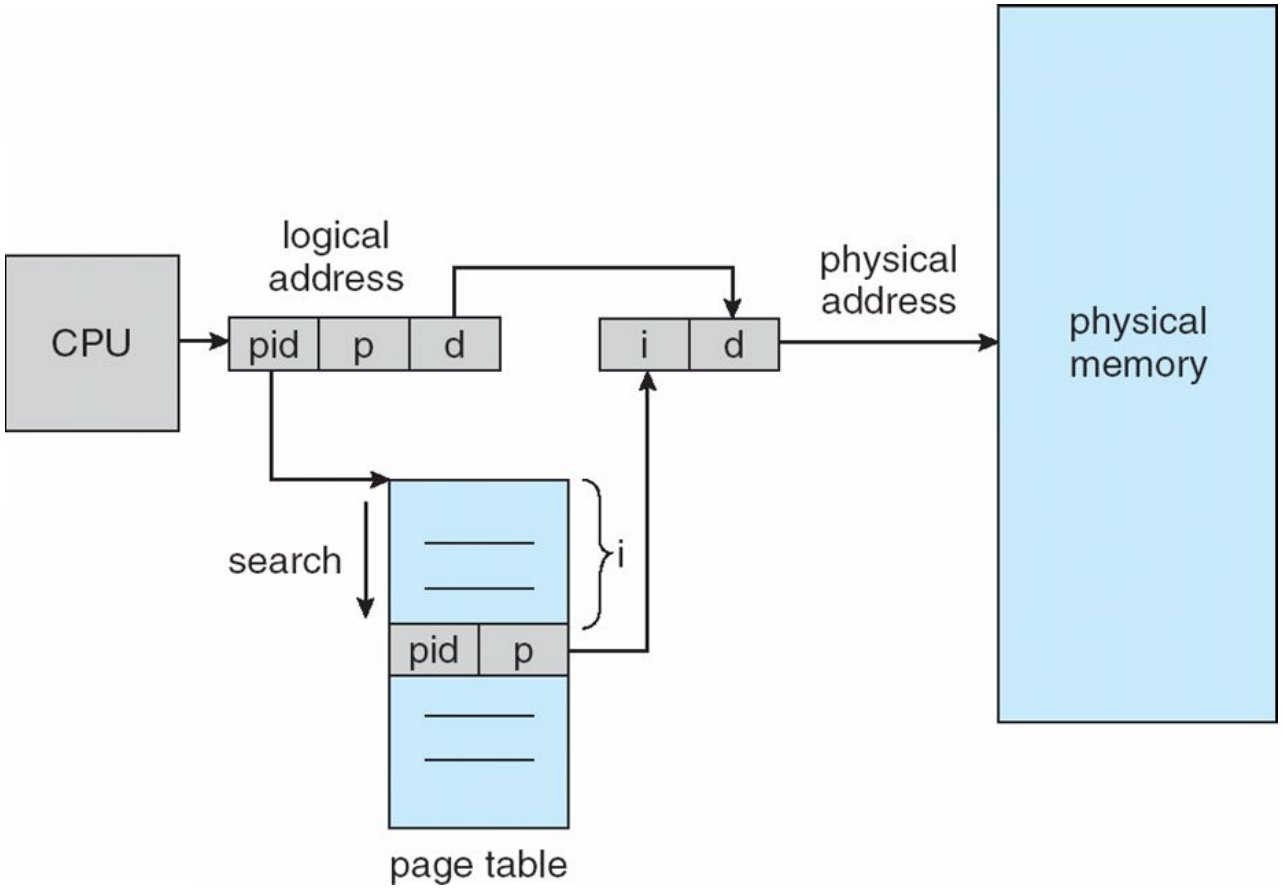


Galvin and Gagne ©2018

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

Example: The Intel 32 and 64-bit Architectures

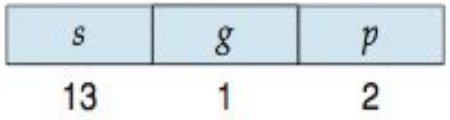
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)

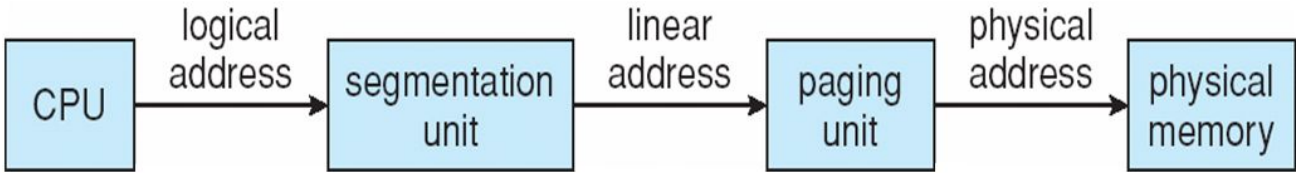
Example: The Intel IA-32 Architecture (Cont.)

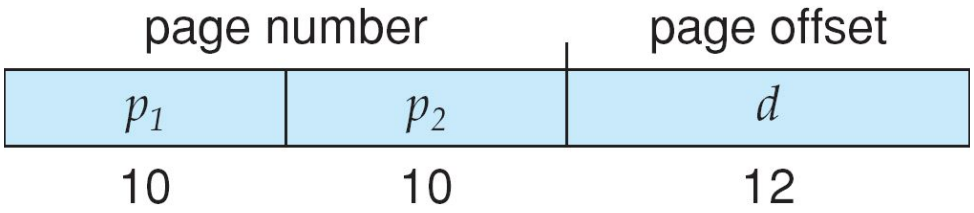
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



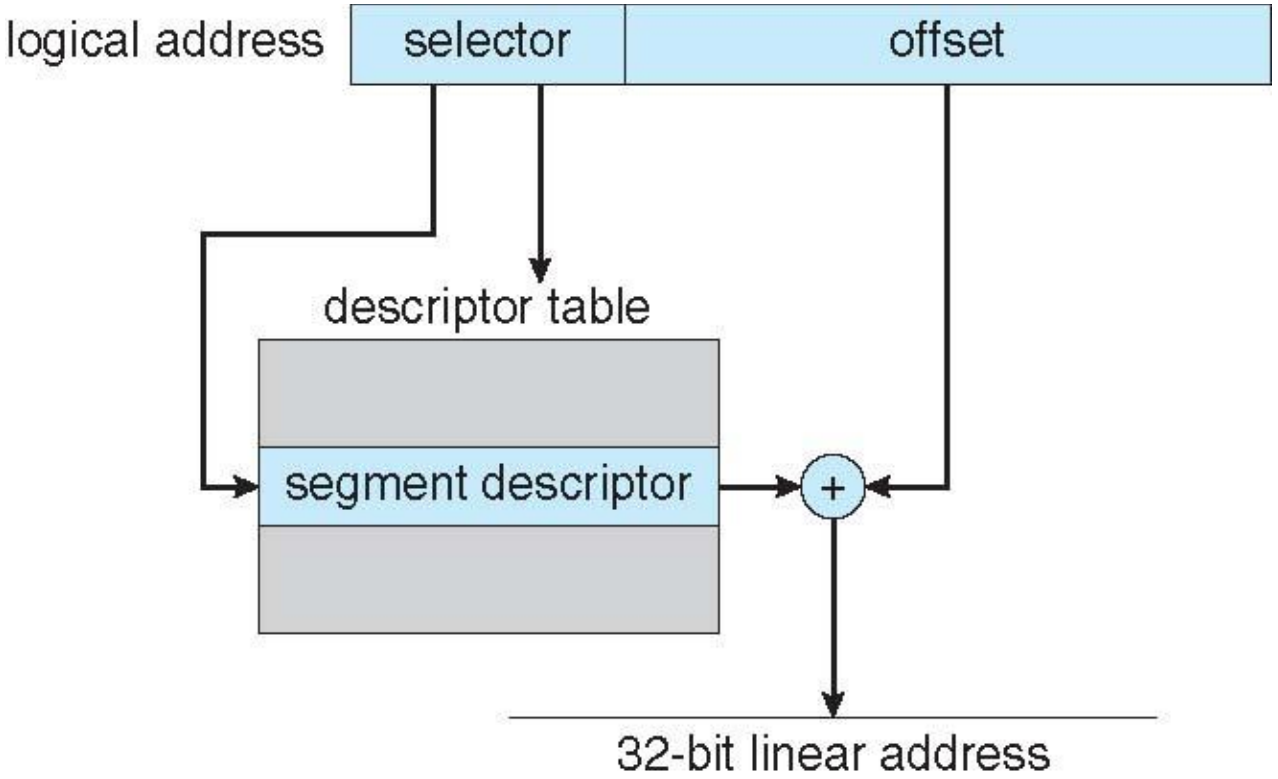
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32

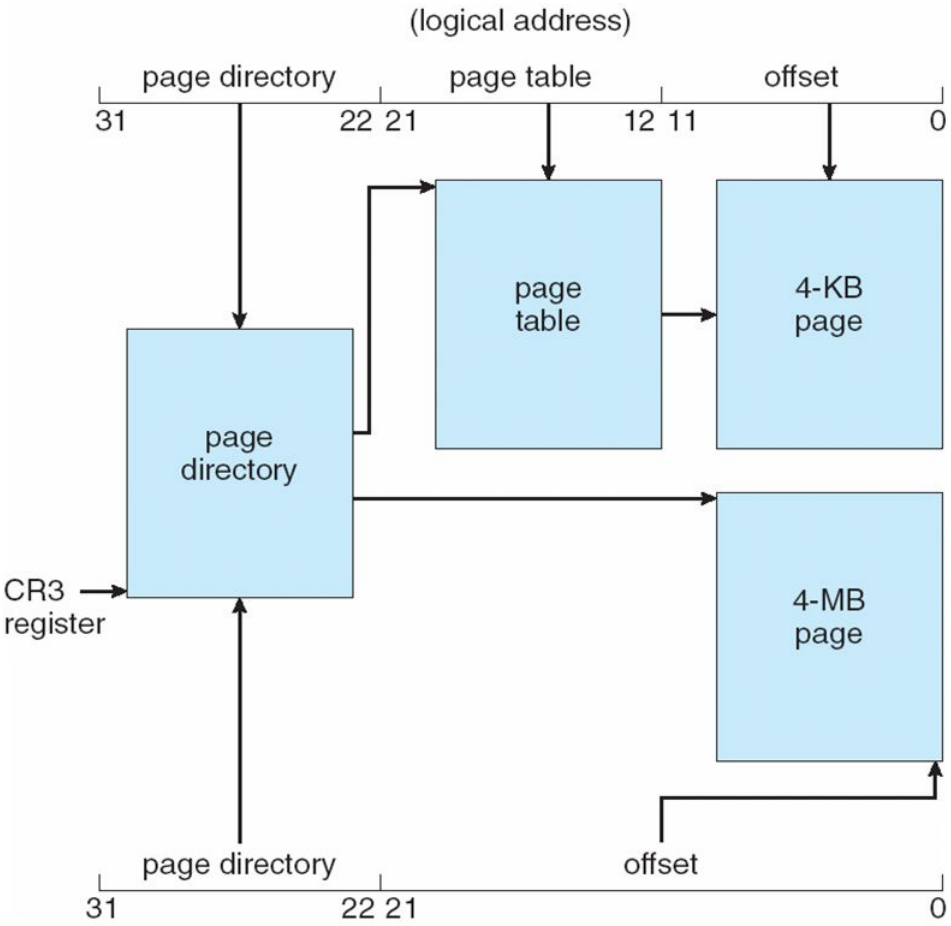




Intel IA-32 Segmentation

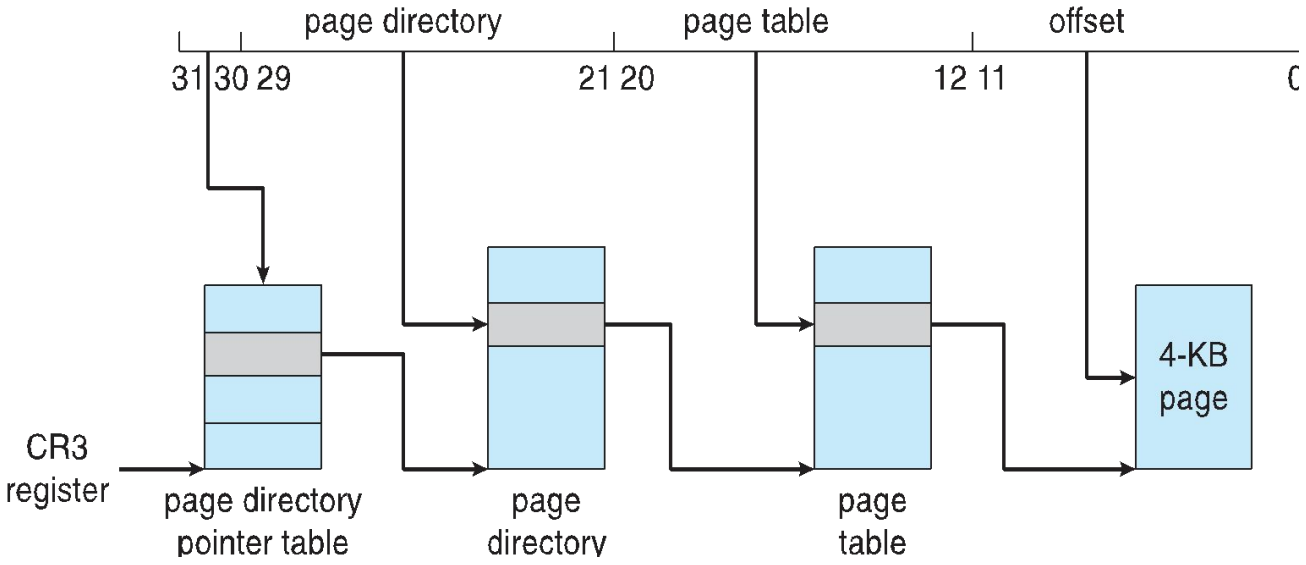


Intel IA-32 Paging Architecture



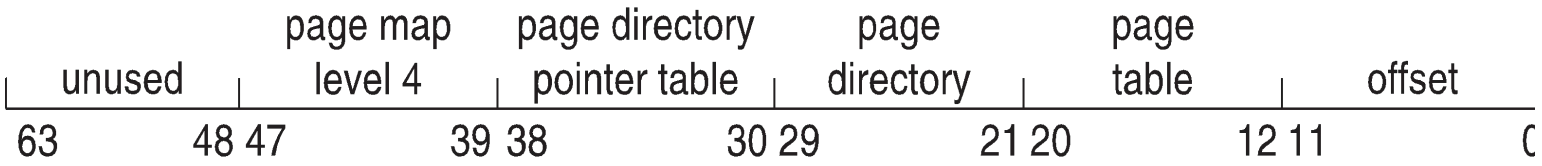
Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



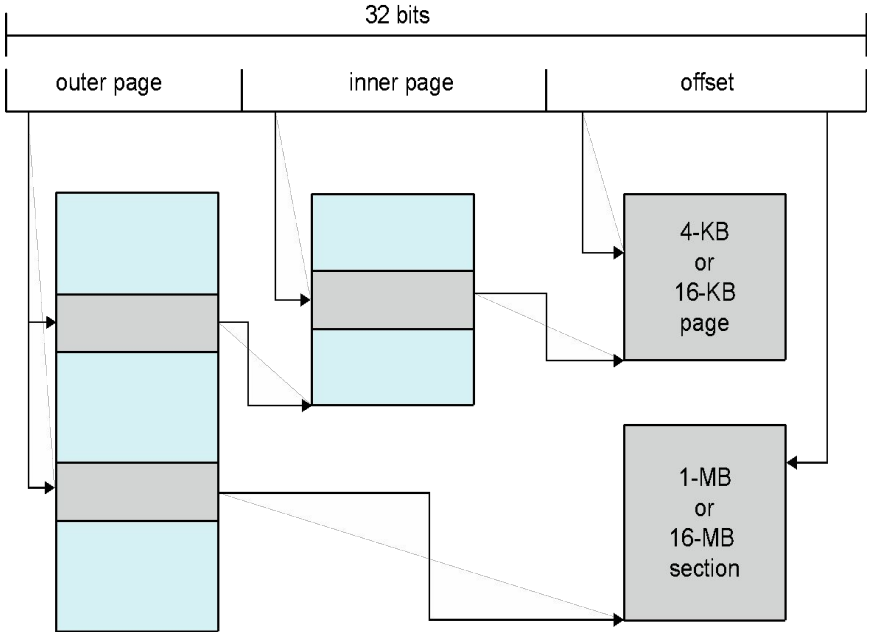
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits

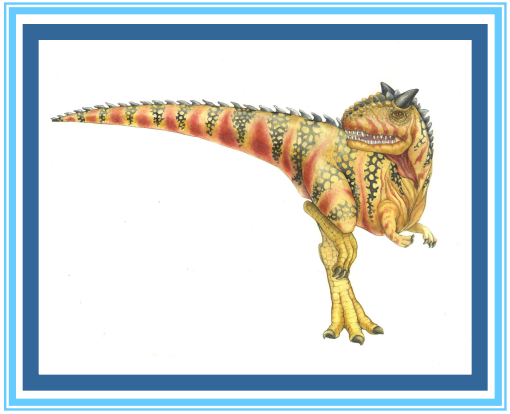


Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed sections)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



End of Chapter 9



Silberschatz, Galvin and Gagne ©2018