Main Memory

Memory Management

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

Objectives

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

Background

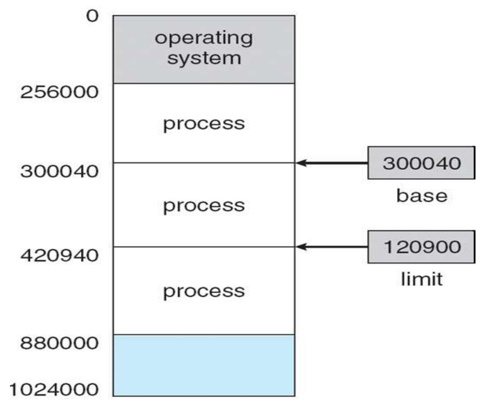
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

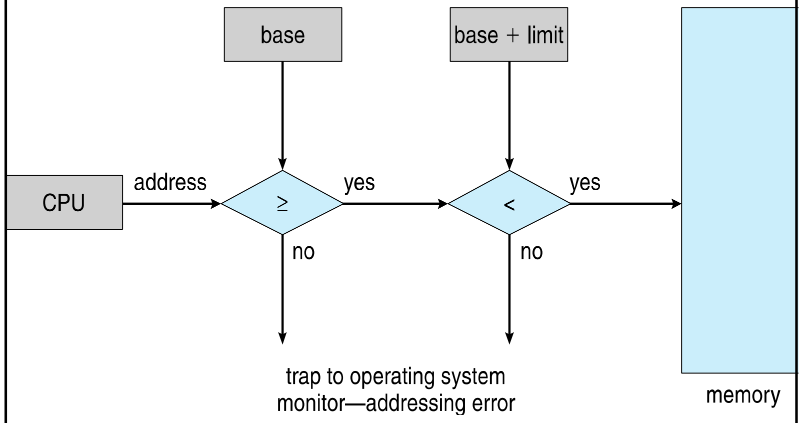
- We first need to make sure that each process has a separate memory space.
- Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.
- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 1.
- 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 the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



Address Binding

- To be executed, the program must be brought into memory and placed within a process.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution.
- The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.
- The normal 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.

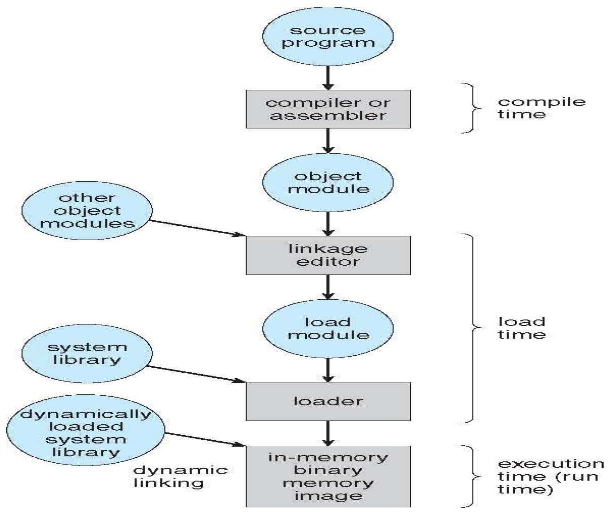
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps 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 memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



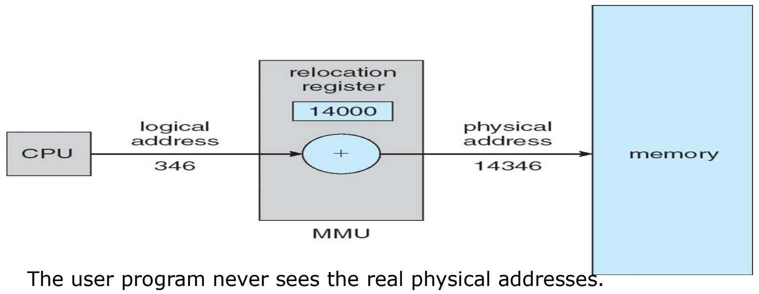
Logical vs. Physical Address Space

- 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
- 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

Memory-Management Unit (MMU)

- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.
- For the time being, we illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme
- The base register is now called a **relocation register**.
- The value in the relocation register is added to every address generated by a user process at the time the address 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 location14000; an access to location346 is mapped to location 14346.

Dynamic relocation using a relocation register



- 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.
- Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register.
- The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses

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.
- The size of a process has thus been limited to the size of physical memory.
- To obtain better memory-space utilization, we can use **dynamic loading**.
- With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If 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.
- The advantage of dynamic loading is that a routine is loaded only when it is needed.
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

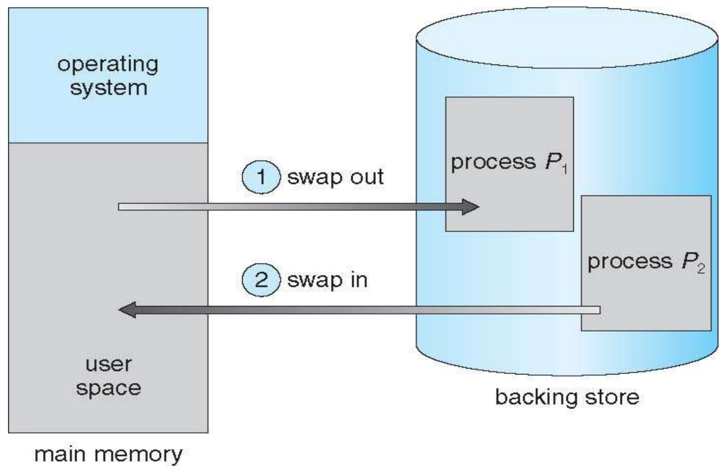
Dynamic Linking

- We know that for the operating system code is considered read only, and separate from data. It seems logical then that if programs can not modify code and have large amounts of common code, instead of replicating it for every executable it should be shared between many executables.
- With virtual memory this can be easily done. The physical pages of memory the library code is loaded into can be easily referenced by any number of virtual pages in any number of address spaces. So while you only have one physical copy of the library code in system memory, every process can have access to that library code at any virtual address it likes.
- Thus people quickly came up with the idea of a shared library which, as the name suggests, is shared by multiple executables. Each executable contains a reference essentially saying "I need library foo". When the program is loaded, it is up to the system to either check if some other program has already loaded the code for library foo into memory, and thus share it by mapping pages into the executable for that physical memory, or otherwise load the library into memory for the executable.
- This process is called dynamic linking because it does part of the linking process "on the fly" as programs are executed in the system.

Swapping

- A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution
- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **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
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



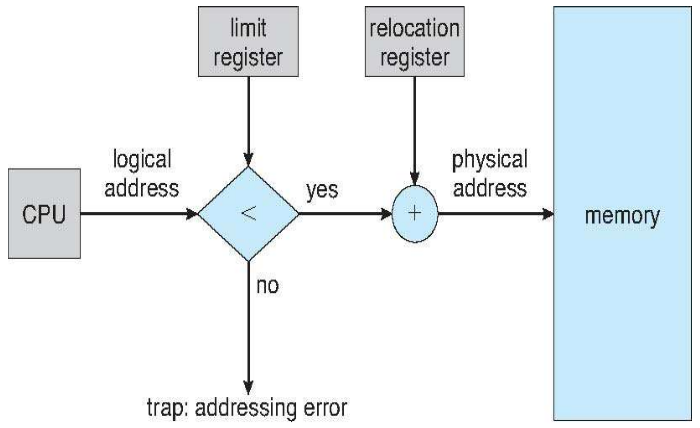
Memory Allocation

- Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**.
- Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions.
- In this **multiple partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

Contiguous Allocation (Cont.)

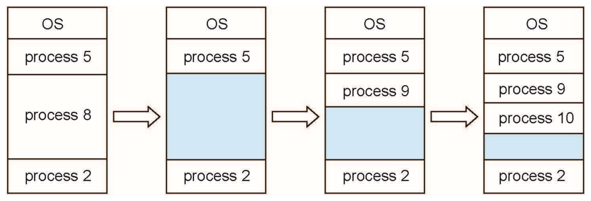
- In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**.
- Eventually, as you will see, memory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

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (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
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Strategies of Memory Allocation

- 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.

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation-External

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**.
- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- External fragmentation exists when there is enough total memory space to satisfy a request but the available

spaces are not contiguous: storage is fragmented into a large number of small

- holes.
- This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Internal Fragmentation

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

Compaction

- One solution to the problem of external fragmentation is compaction.
- The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however.
- If relocation is static and is done at assembly or load time, compaction cannot be done.
- It is possible only if relocation is dynamic and is done at execution time.
- If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.

Segmentation

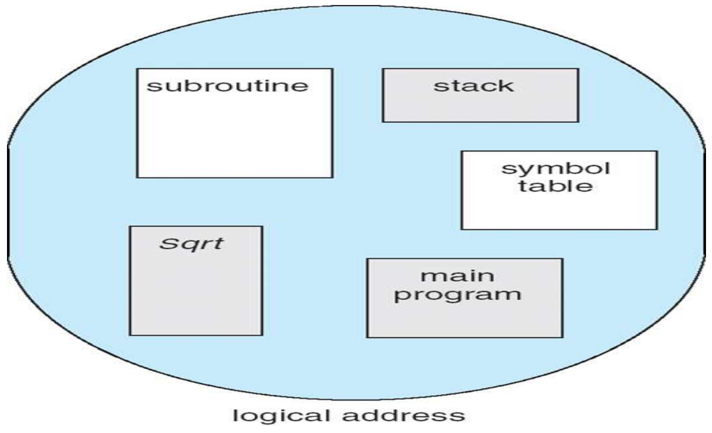
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays

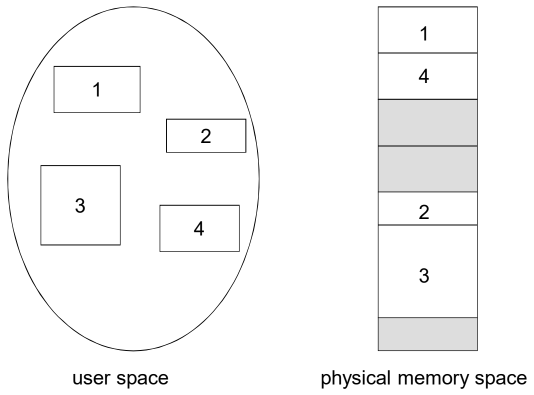
Examples

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

User’s View of a Program



Logical View of Segmentation



Segmentation Architecture

- **Segmentation** is a memory-management scheme that supports this programme rview of memory. A logical address space is a collection of segments.
- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.
- The programmer therefore specifies each address by two quantities: a segment name and an offset.
- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- Thus, a logical address consists of a *two tuple:*
- <segment-number, offset>.
- Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.
- A C compiler might create separate segments for the following:
- **1.** The code
- **2.** Global variables
- **3.** The heap, from which memory is allocated
- **4.** The stacks used by each thread
- **5.** The standard C library
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

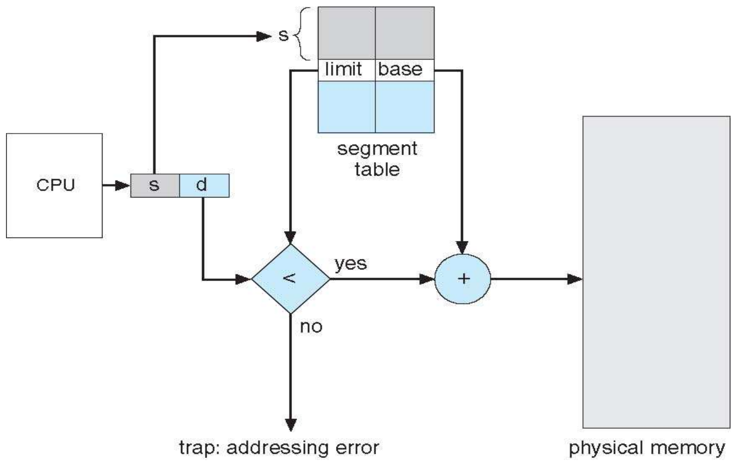
  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Hardware

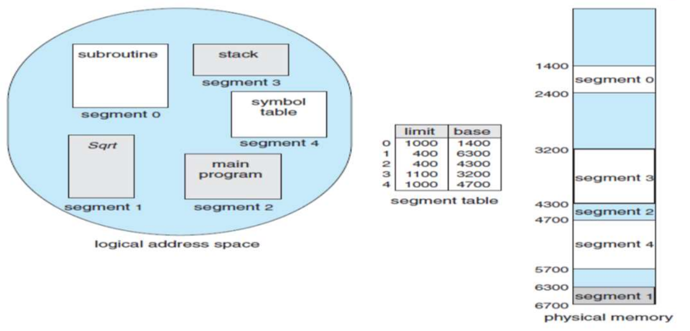
- Each entry in the segment table has a **segment base** and a **segment limit**. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.
- The use of a segment table is illustrated in Figure 8.8. A logical address
- consists of two parts: a segment number, *s,* and an offset into that segment, *d.*
- The segment number is used as an index to the segment table. The offset *d* of
- the logical addressmust be between 0 and the segment limit. If it is not,we trap
- to the operating system (logical addressing attempt beyond

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

Segmentation Hardware



8.5 Paging



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

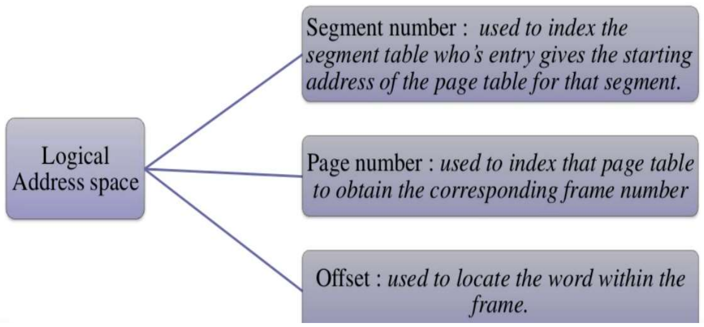
Figure 8.9 Example of segmentation.

IMPLEMENTING SEGMENTATION

- Each process has:

  one segment table.

  several page tables : one page table per segment.



Logical Address space

Segment number : *used to index the segment table who's entry gives the starting address of the page table for that segment.*

Page number : *used to index that page table to obtain the corresponding frame number*

Offset : *used to locate the word within the frame.*

Paging

- Segmentation permits the physical address space of a process to be noncontiguous.
- **Paging** is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not.
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.
- When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store).
- The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications.
- For example, the logical address space is now totally separate fromthe physical address space, so a process can have a logical 64-bit address space even though the system has less than 264 bytes of physical memory.

- 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.

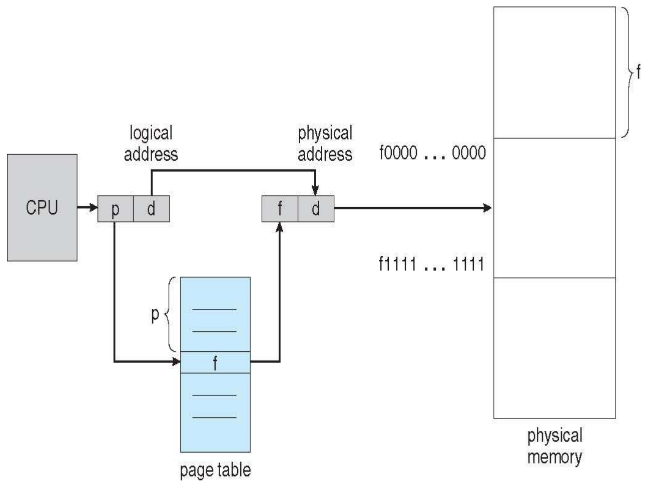
Address Translation Scheme

- 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

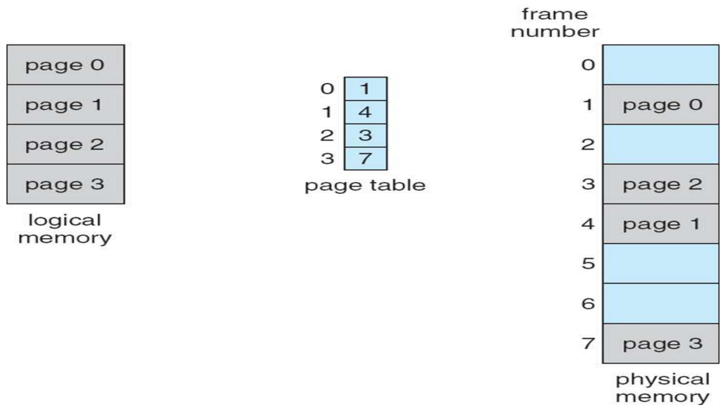
page number	page offset
p	d
m -n	n

  - For given logical address space 2^m and page size 2^n

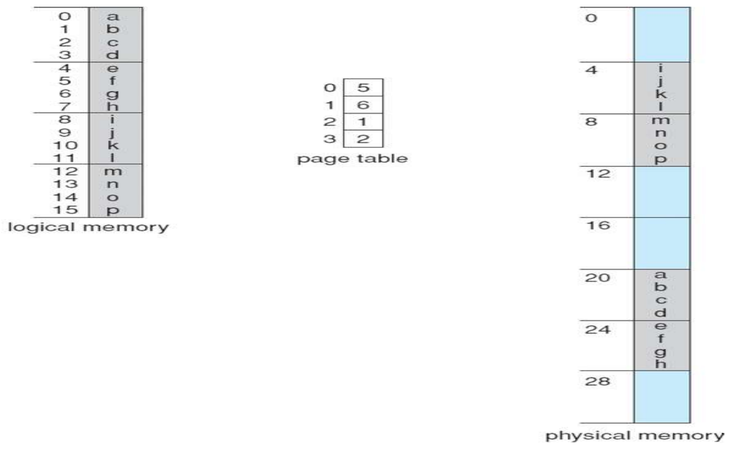
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example

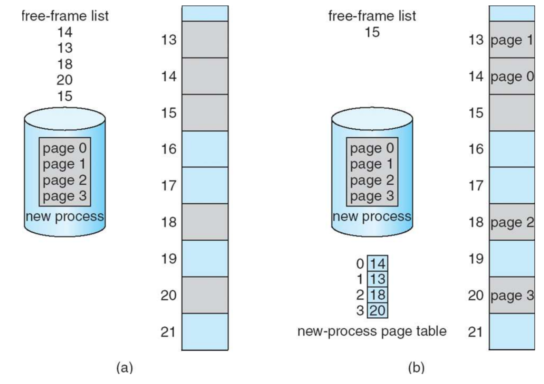


n=2 and *m*=4   32-byte memory and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames



Before allocation

After allocation

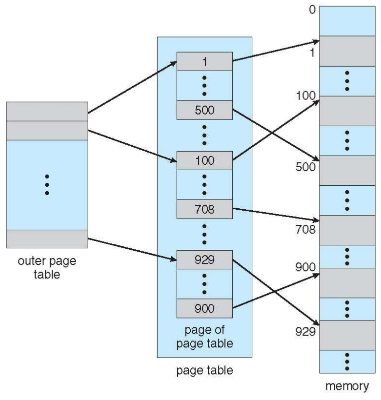
Structure of the Page Table

- 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 -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



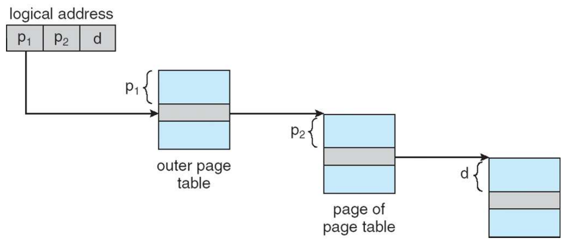
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
12	10	10

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

Address-Translation Scheme



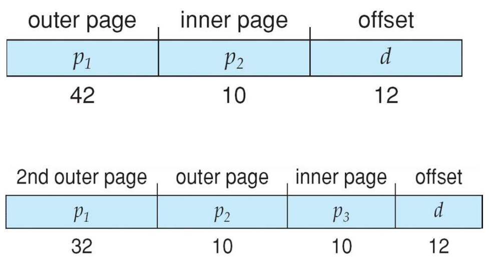
64-bit Logical Address Space

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

outer page	inner page	page offset
p_1	p_2	d
42	10	12

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

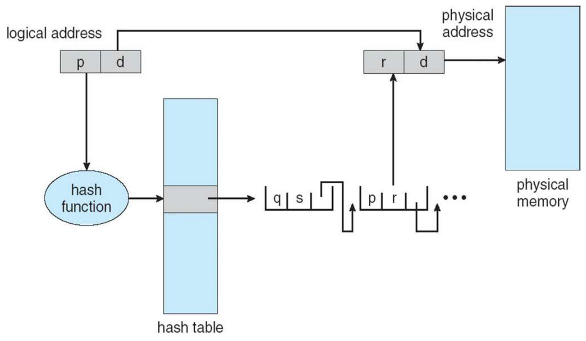
Three-level Paging Scheme



Hashed Page Tables

- Common in address spaces > 32 bits
- 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
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- 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
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

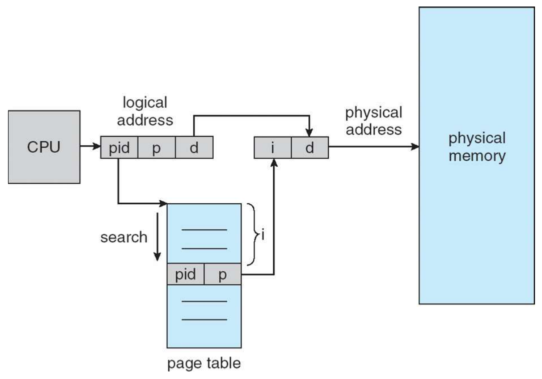
Hashed Page Table



Inverted Page Table

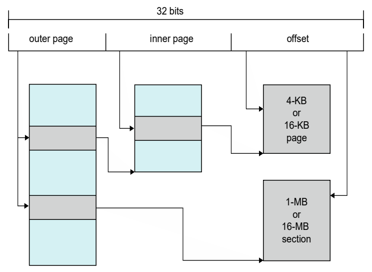
- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that 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



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


32 bits
outer page
inner page
offset
4-KB or 16-KB page
1-MB or 16-MB section