Chapter 8: Main Memory

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Objectives

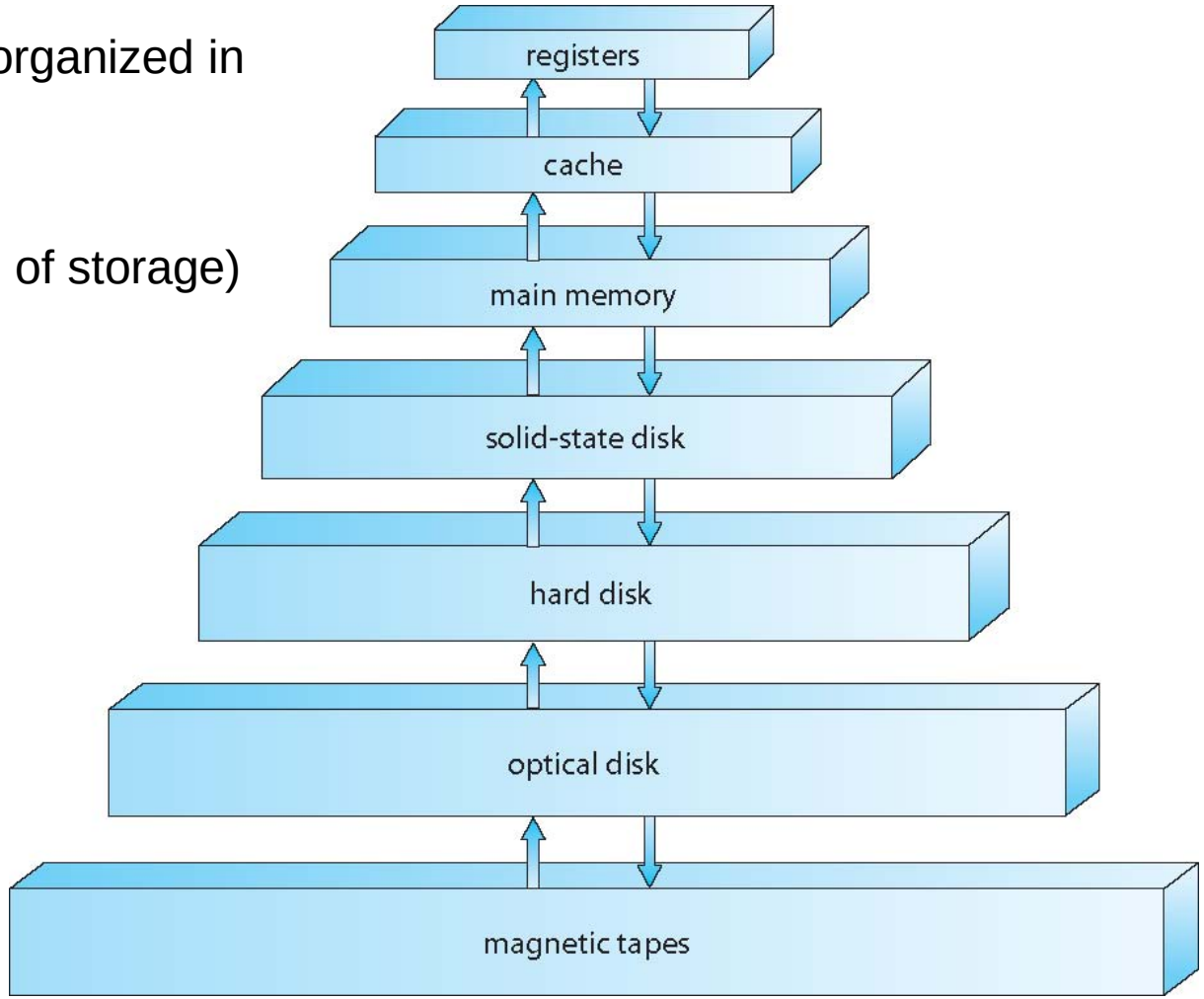
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including:
  - Paging, and
  - Segmentation

Background

- For it to be run, a program must be brought (from disk) into memory and placed within a process
- **Input queue –** stores which programs on disk are ready to be brought into memory to execute
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees:
  - read requests + the correspond addresses
  - write requests + the correspond addresses and data

Recap: Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost (per byte of storage)
  - Volatility



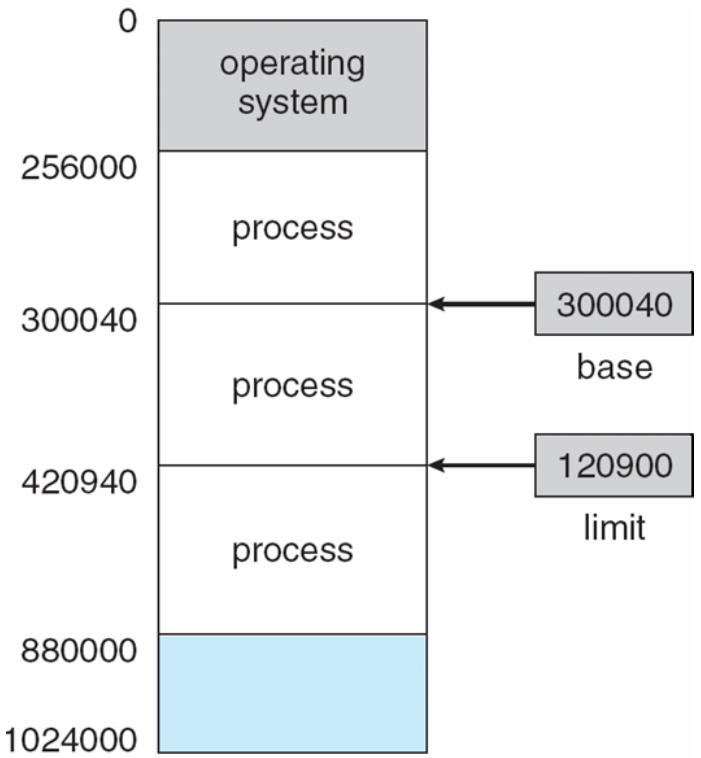
Recap: Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

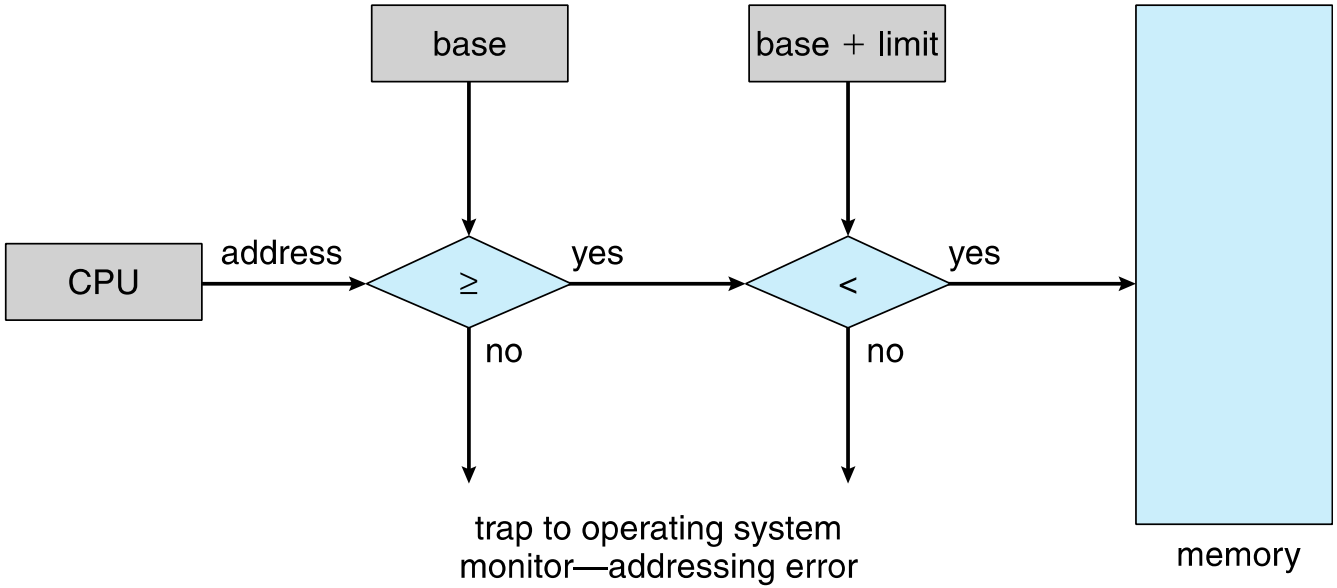
- Register access in 1 CPU clock (or less)
  - 0.25 – 0.50 ns (1 nanosec = 10^{-9} sec)
- Main memory can take **many cycles**, causing the CPU to **stall**
  - 80-250 ns (160-1000 times slower)
  - How to solve? -- **caching**
- **Cache** sits between main memory and CPU registers

Base and Limit Registers

- Protection of memory is required to ensure correct operation
  - Protect OS from processes
  - Protect processes from other processes
- How to implement memory protection?
  - Example of one simple solution using basic hardware
  - 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

- In most cases, a user program goes through several steps before being executed
  - Compilation, linking, executable file, loader creates a process
  - Some of which may be optional
- Addresses are represented in different ways at different stages of a program's life
  - Each binding maps one address space to another
- Source code -- addresses are usually symbolic
  - E.g., variable `count`
- A compiler typically binds these symbolic addresses to relocatable addresses
  - E.g., "14 bytes from beginning of this module"
- Linker or loader will bind relocatable addresses to absolute addresses
  - E.g., 74014

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at 3 different stages:

1. **Compile time**:
   - If you know at compile time where the process will reside in memory, then absolute code can be generated.
   - Must recompile the code if starting location changes
   - Example: MS DOS .com programs
2. **Load time**:
   - Compiler must generate **relocatable code** if memory location is not known at compile time
   - If the starting address changes, we need only reload the user code to incorporate this changed value.
3. **Execution time**:
   - If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time
   - Special hardware must be available for this scheme to work
   - **Most general-purpose operating systems use this method**

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address (=virtual address)** – generated by the CPU
    - This is what a process sees and uses
  - **Physical address** – address seen by the memory unit
    - Virtual addresses are mapped into physical addresses by the system
- **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

Logical vs. Physical Address Space

- Logical addresses and physical addresses
  - Are the same for
    - compile-time and load-time address-binding schemes
  - Different for
    - execution-time address-binding scheme

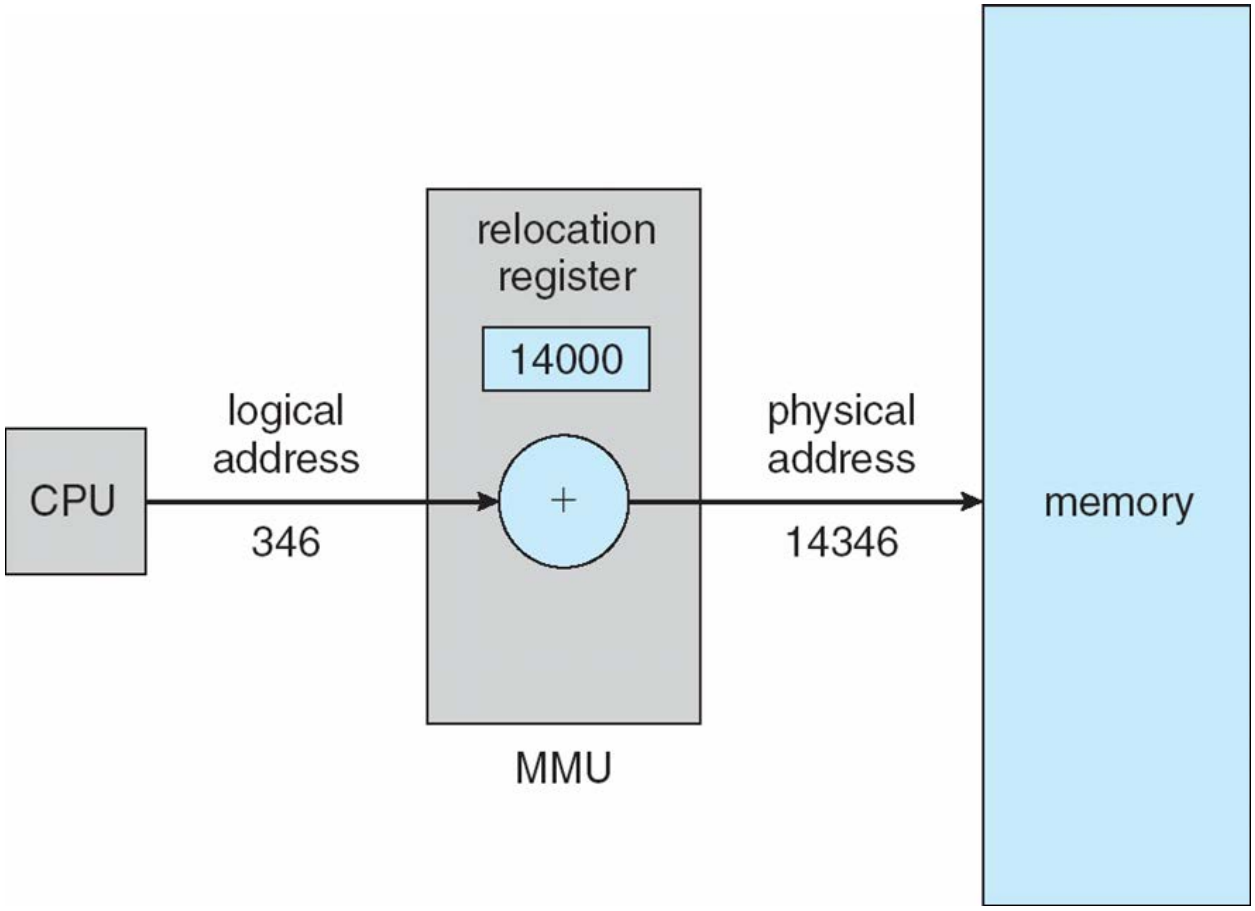
Memory-Management Unit (MMU)

- Memory-Management Unit (MMU)
  - Hardware device that (at run time) maps virtual to physical address
  - Many methods for such a mapping are possible
    - Some are considered next

- To start, consider simple scheme:
  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register is now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers

- The user program deals with *logical* addresses
  - It never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

- In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute.
- Dynamic Loading -- routine is not loaded (from disk) until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- Some OS'es support only static linking
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking
- Linking is postponed until execution time
- Similar to dynamic loading, but linking, rather than loading, is postponed
- Usually used with system libraries, such as language subroutine libraries
- Without this, each program must include a copy of its language library (or at least the routines referenced by the program) in the executable image.
- This wastes both disk space and main memory

Dynamic Linking (cont. 1)

- Dynamically linked libraries are system libraries that are linked to user programs when the programs are run
- 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
- 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.
  - Under this scheme, all processes that use a language library execute only 1 copy of the library code

Dynamic Linking (cont. 2)

- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
  - Extensions to handle 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
    - No relinking is necessary
- Versioning may be needed
  - In case the new library is incompatible with the old ones
  - More than one version of a library may be loaded into memory
    - 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.

Dynamic Linking (cont. 3)

- Unlike dynamic loading, dynamic linking and shared libraries generally require help from the OS.
  - If the processes in memory are protected from one another, then the OS is the only entity that can check to see whether the needed routine is in another process's memory space
  - or that can allow multiple processes to access the same memory addresses
  - We elaborate on this concept when we discuss paging

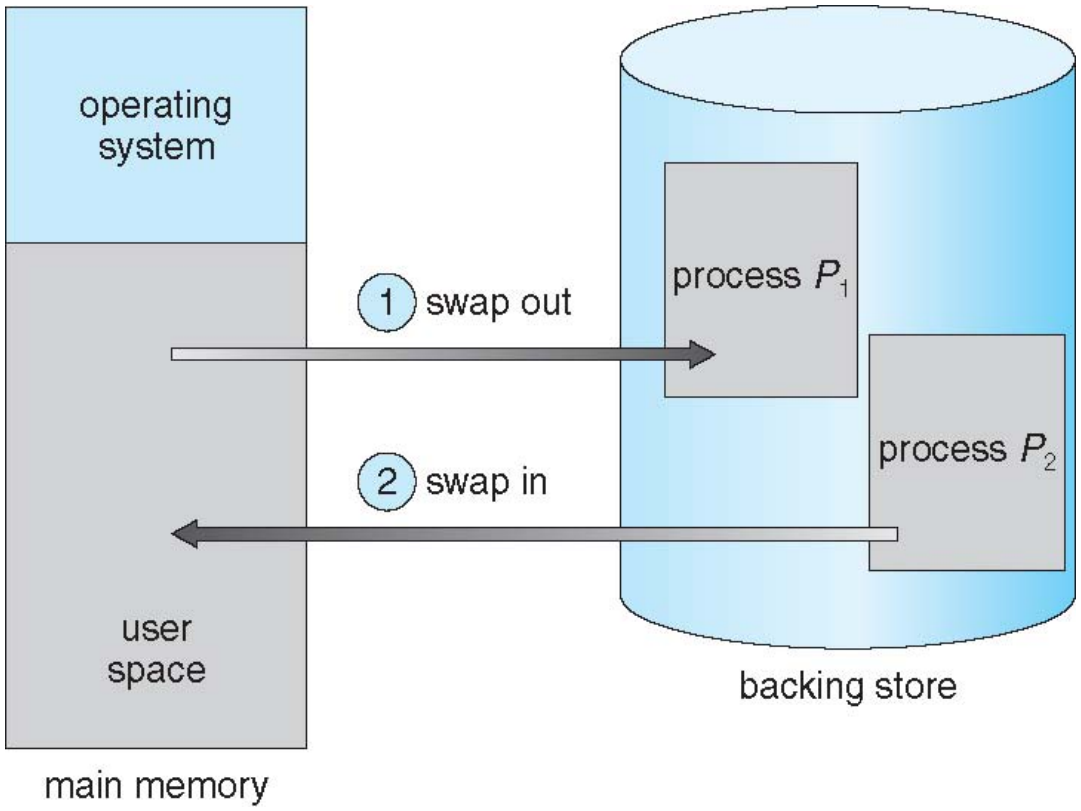
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
  - This increases 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
- 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
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

Swapping (Cont.)

- Modified versions of swapping are found on many systems
  - For example, UNIX, Linux, and Windows
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Contiguous Allocation

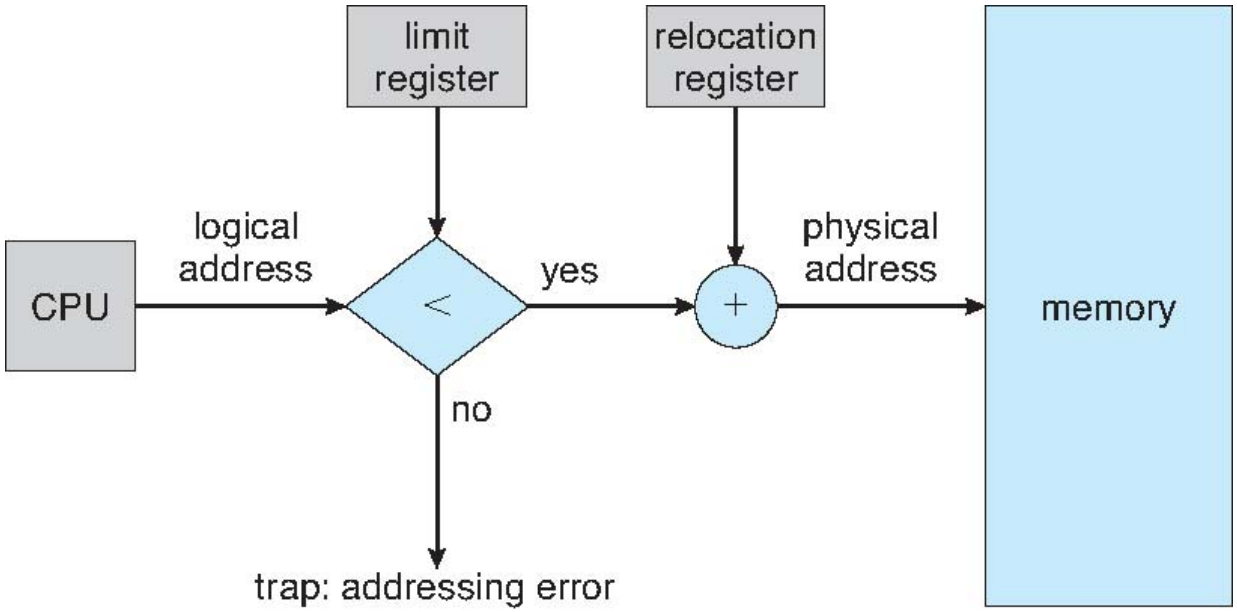
- Main memory must support both OS and user processes
  - Limited resource, must allocate efficiently
  - How? -- Many methods

- Contiguous memory allocation is one early method
  - each process is contained in a single section of memory that is contiguous to the section containing the next process.
- Main memory is usually divided into 2 **partitions**:
  1. Resident operating system
     - usually held in low memory
  2. User processes
     - usually held in high memory
     - each process contained in single contiguous section of memory

Contiguous Allocation: Memory Protection

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains the value of the smallest physical address for the process
  - Limit register contains range of logical addresses for the process
    - each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers



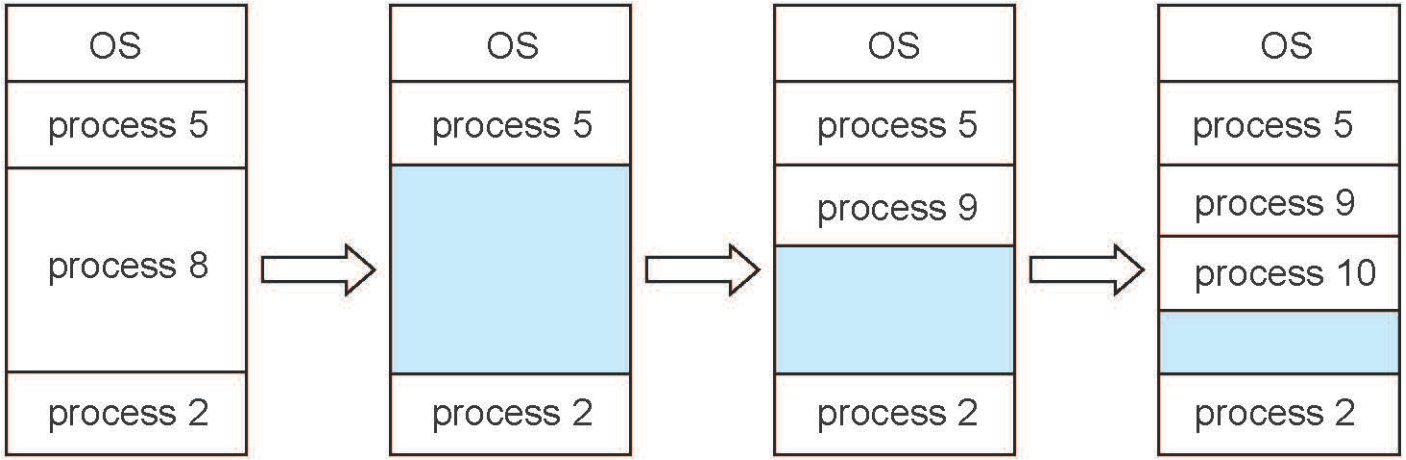
Contiguous Allocation: Memory Allocation

- Multiple-partition allocation
  - One of the simplest methods
  - Originally used by the IBM OS/360 operating system (called MFT)
    - no longer in use.
  - Divide memory into several fixed-sized partitions
  - Each partition may contain exactly 1 process
    - Thus, the degree of multiprogramming is limited by the number of partitions
  - 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: Memory Allocation

- Variable-partition scheme
  - Generalization of the previous method
  - **Idea: Variable-partition** sizes for efficiency
    - Sized to a given process' needs
  - Initially, all memory is available for user processes and is considered one large block of available memory (a hole).
  - **Hole** – block of available memory;
    - Holes of various size are scattered throughout memory
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (holes)
  - 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

Contiguous Allocation: Memory Allocation



Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

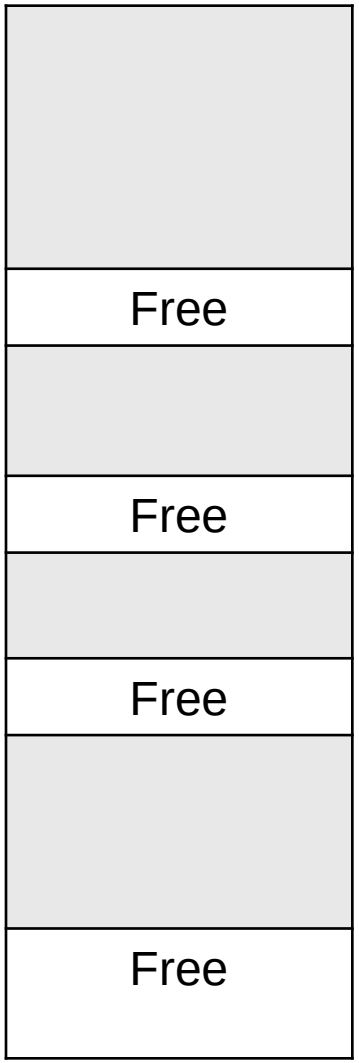
- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole
  - Must also search entire list
  - Produces the largest leftover hole
- First-fit and best-fit are better than worst-fit in terms of speed and storage utilization

Fragmentation

- Memory allocation can cause fragmentation problems:
  1. **External Fragmentation**
  2. **Internal Fragmentation**

External 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** – total memory space exists to satisfy a request, but it is not contiguous
  - If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.
- Analysis of the first-fit strategy reveals that, given *N* blocks allocated, another 0.5 *N* blocks will be lost to fragmentation
  - That is, 1/3 of memory may be unusable!
  - This is known as **50-percent rule**



Fragmentation (Cont.)

- Some of the solutions to external fragmentation:
  1. **Compaction**
  2. **Segmentation**
  3. **Paging**

Fragmentation: Compaction

- Compaction
  - Shuffle memory contents to place all free memory together in 1 large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
    - If addresses are relocated dynamically, relocation requires only:
      1. moving the program and data, and then
      2. changing the base register to reflect the new base address.
  - I/O can cause problems
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

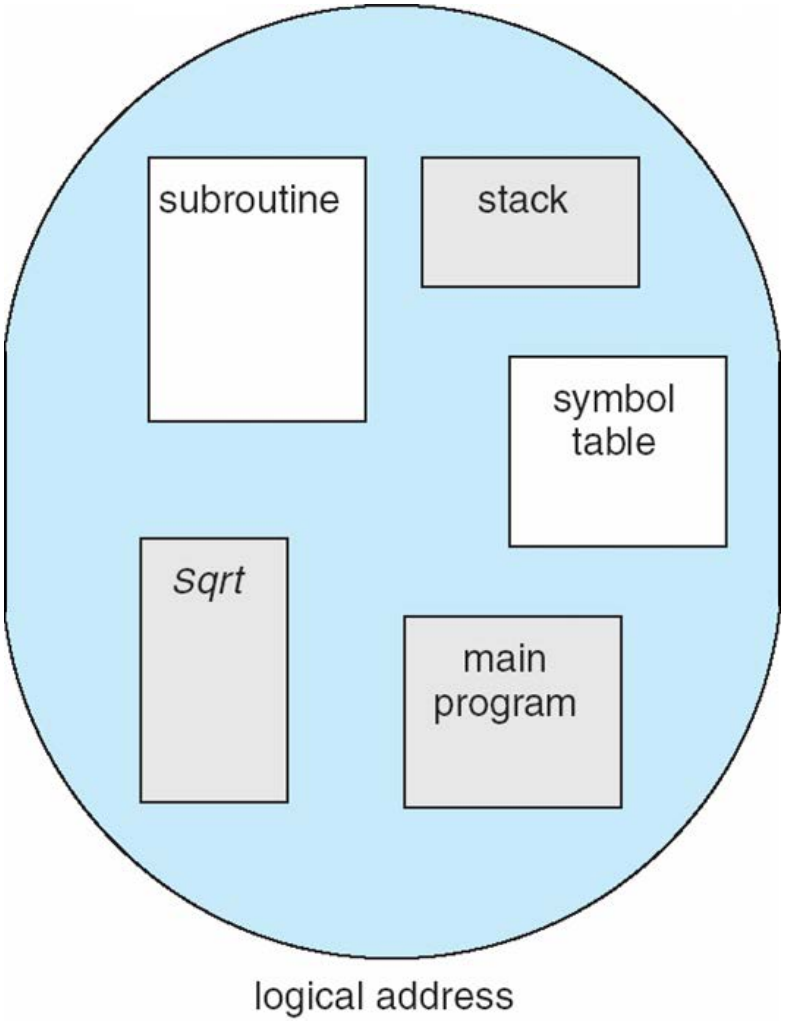
Internal Fragmentation

- Example:
  - Consider a multiple-partition allocation scheme with a hole of 10,002 bytes
  - Suppose that the next process requests 10,000 bytes.
  - If we allocate exactly the requested block, we are left with a hole of 2 bytes.
  - **Problem:** The overhead to keep track of this hole will be substantially larger than the hole itself.
- **Solution:**
  - Break the physical memory into fixed-sized **blocks**
  - Allocate memory in units based on block size.
- **Issue:** 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.
  - Example:
    - Block size is 4K
    - Process request 16K + 2Bytes space
    - 5 blocks will be allocated, (4K – 2) bytes are wasted in the last block

Segmentation: a Memory-Management Scheme

- **Motivation:** Do programmers think of memory as a linear array of bytes, some containing instructions and others containing data?
  - Most programmers would say “no.”
  - Rather, they prefer to view memory as a collection of variable-sized segments,
    - with no necessary ordering among the segments
  - The programmer talks about “the stack,” “the math library,” and “the main program” without caring what addresses in memory these elements occupy.
  - The programmer is not concerned with whether the stack is stored before or after the `sqrt()` function.
  - What if the hardware could provide a memory mechanism that mapped the programmer’s view to the actual physical memory?

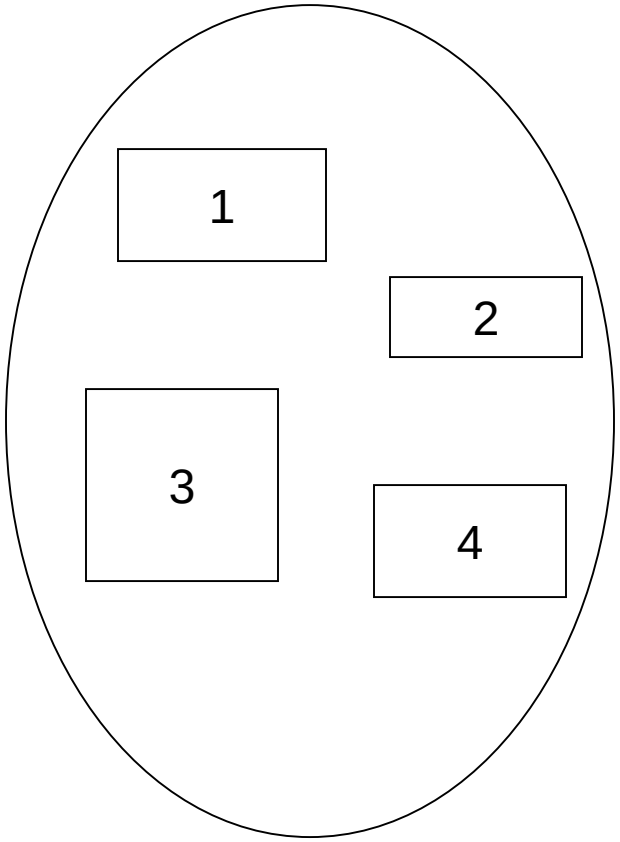
User's View of a Program



Segmentation

- **Solution:** Segmentation – a memory-management scheme that supports “programmer/user view” of memory
- A logical address space 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
- Each segment has a name and a length.
- The addresses specify both
  - the segment name, and
  - the offset within the segment.
- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.

Logical View of Segmentation



user space

physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

  `<segment-number, offset>`

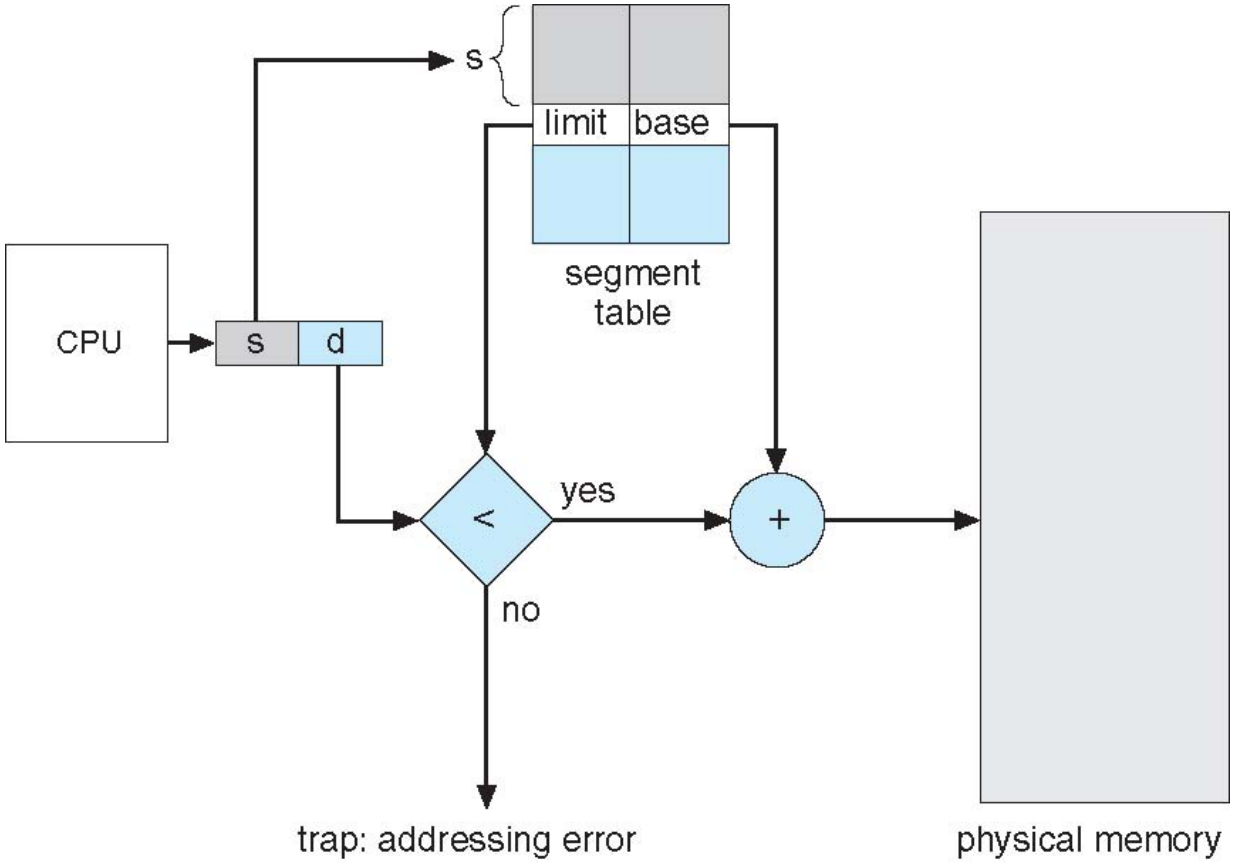
- How to map this 2D user-defined address into 1D physical address?

- **Segment table** – 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



Paging

- Segmentation
  - **Pros:** permits the physical address space of a process to be noncontiguous.
  - **Cons:** can suffer from external fragmentation and needs compaction
    - Any alternatives to it?
    - Paging -- another memory-management scheme

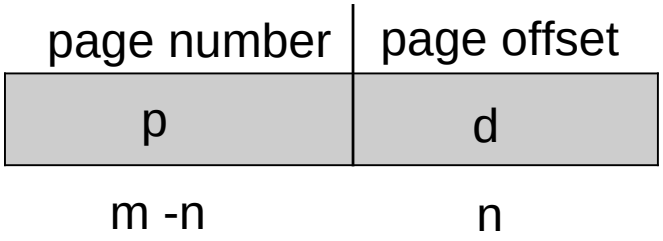
- Benefits of Paging
  - Physical address space of a process can be noncontiguous
  - Process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks

Main Idea of 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
  - One table per process
- Still have internal fragmentation

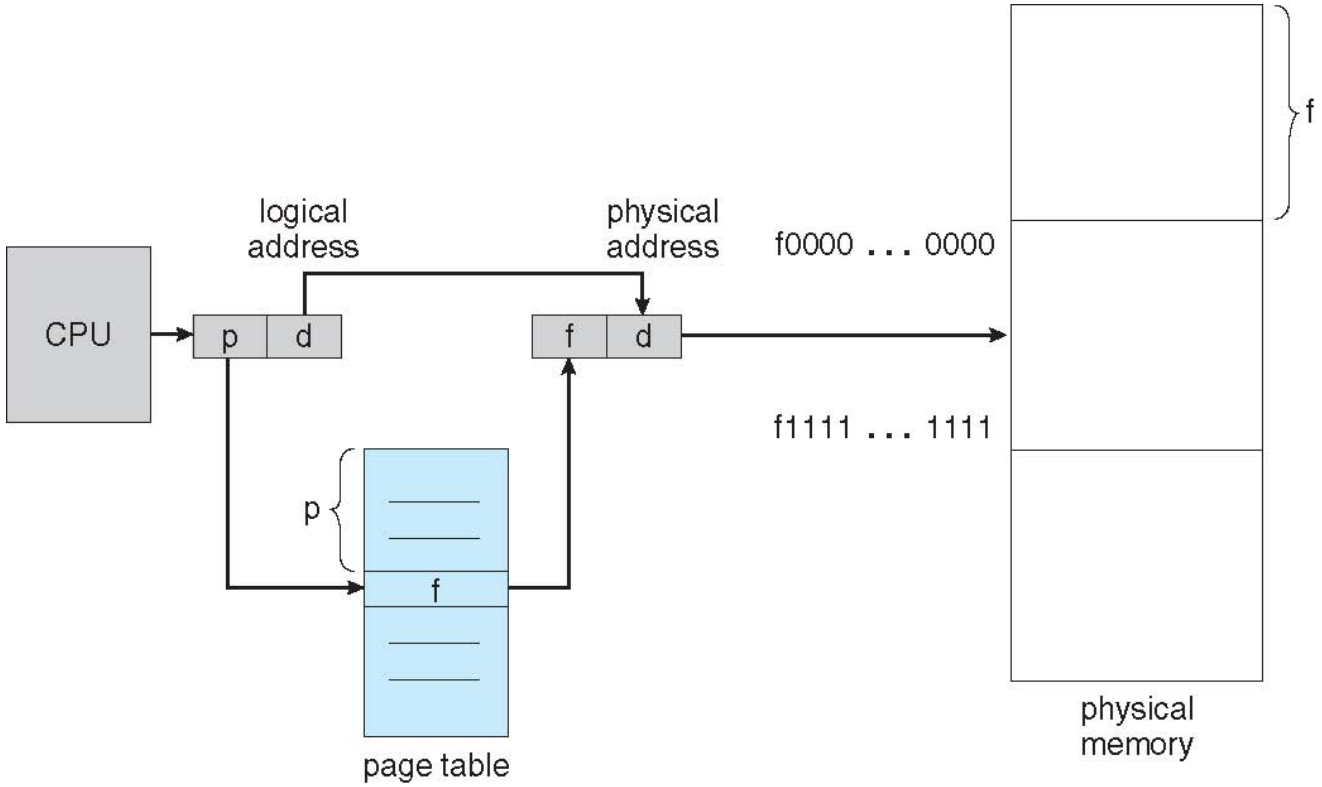
Address Translation Scheme

- 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

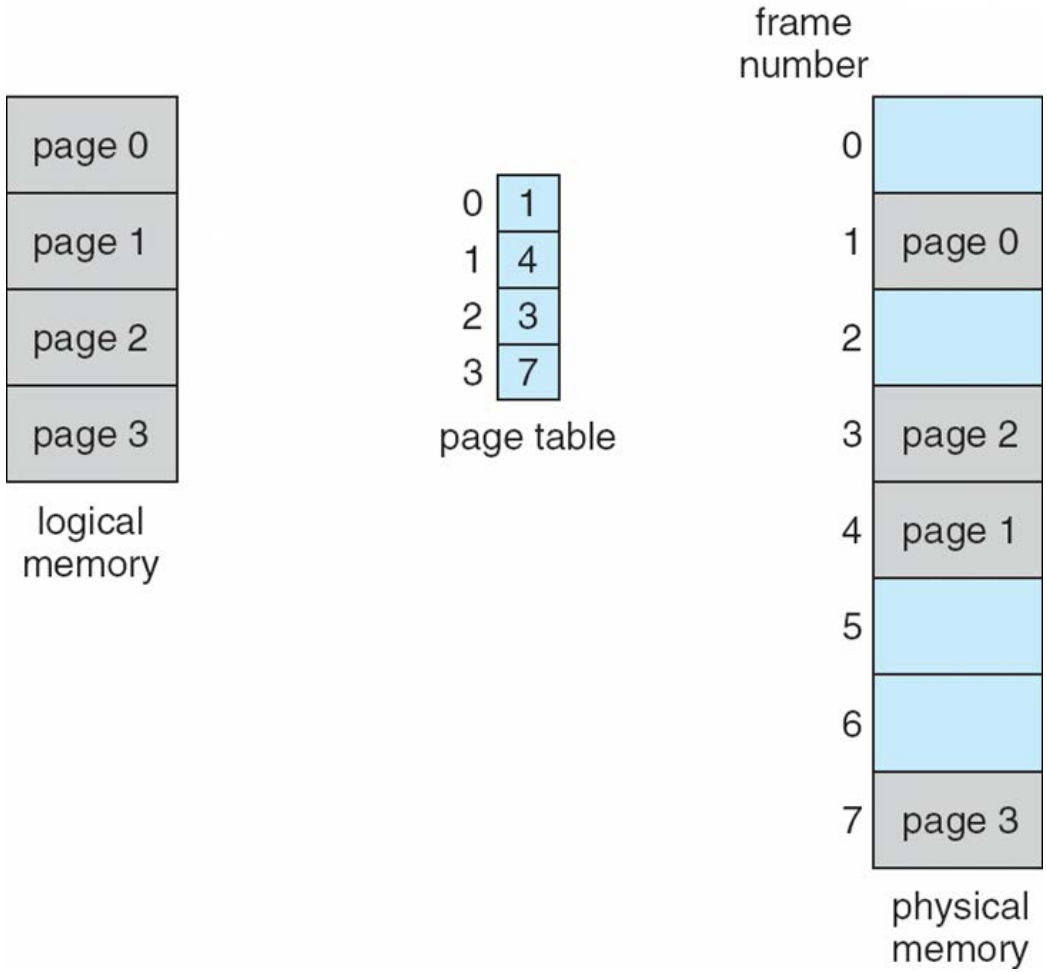


  - For given logical address space 2^m and page size 2^n

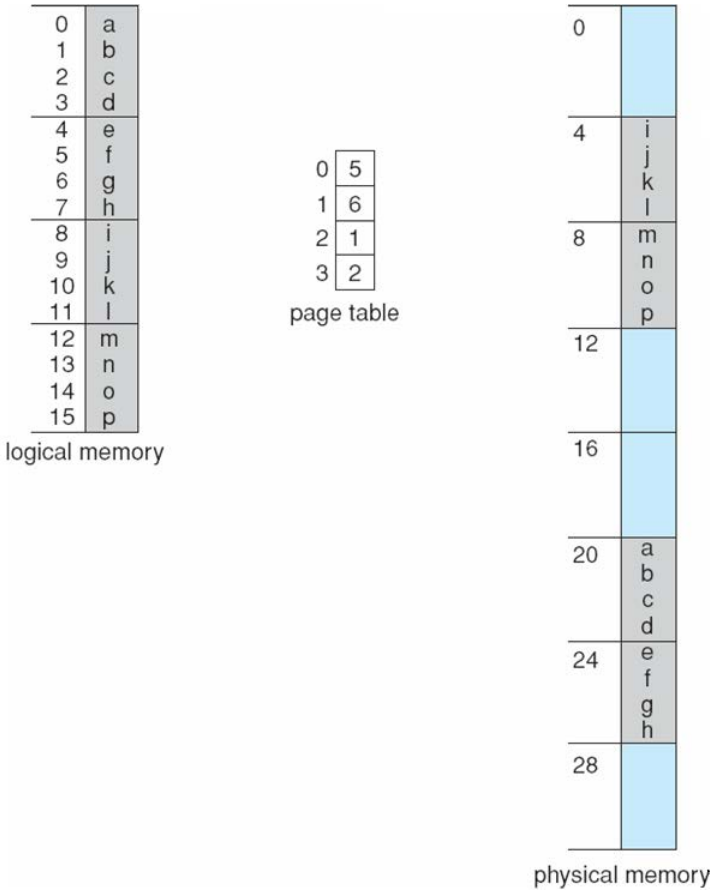
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



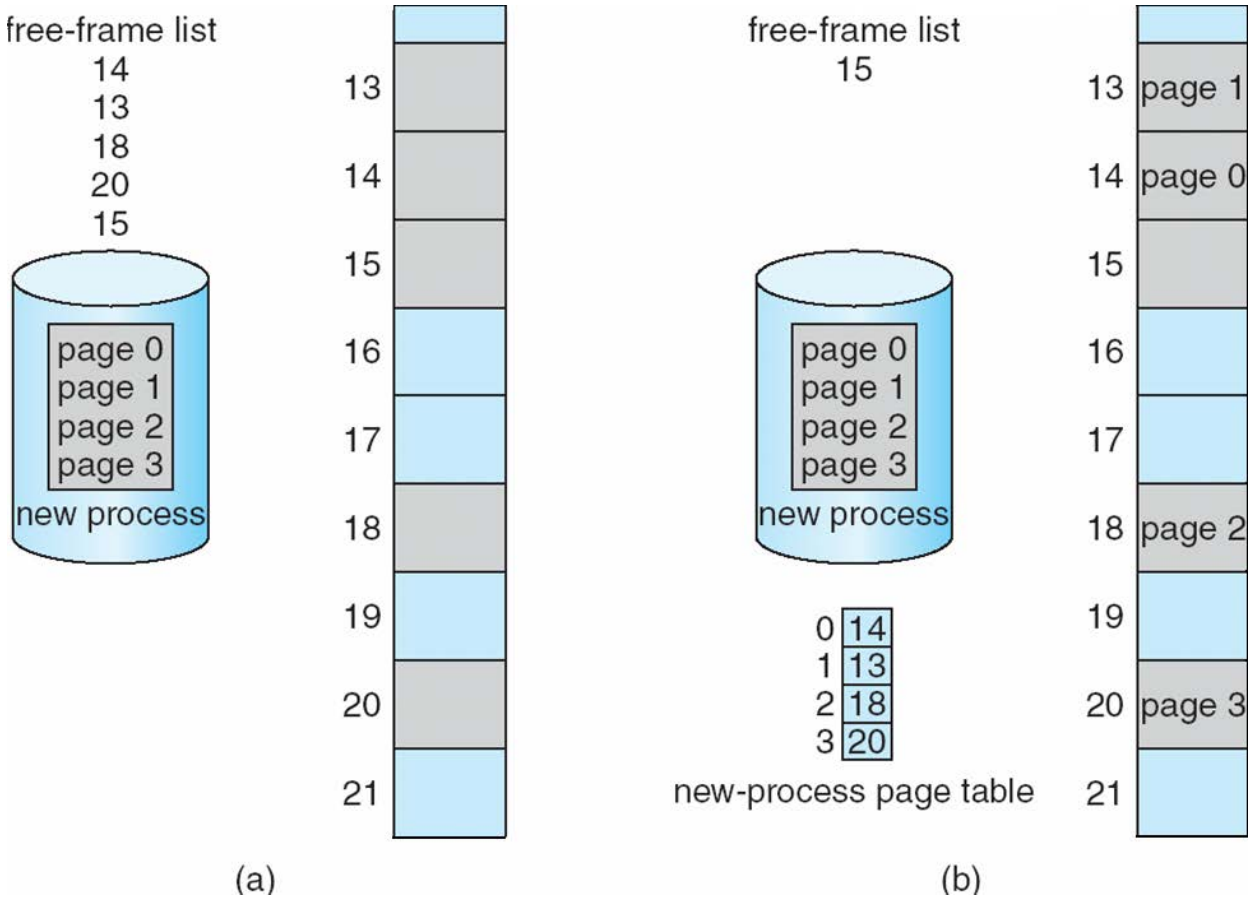
n=2 and *m*=4 32-byte memory and 4-byte pages

- Process view and physical memory now very different
- By implementation a process can only access its own memory

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 are desirable?
  - Not as simple, as each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames



Before allocation

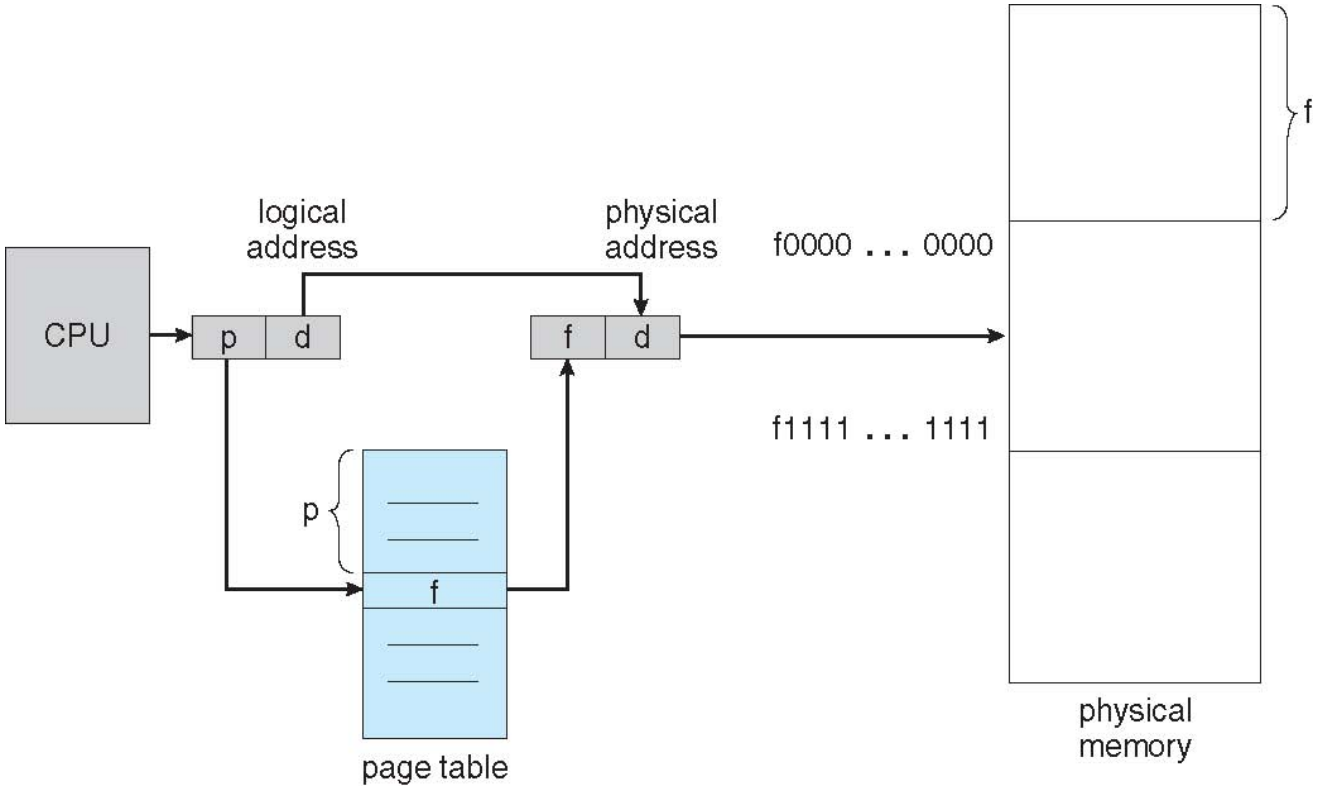
After allocation

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**)
  - Stores the physical address of page table
  - Changing page tables requires changing only this one register
    - Substantially reduces context-switch time
- **Page-table length register** (**PTLR**)
  - Indicates the size of the page table

The 2 memory access problem

- **Problem:**
  - In this scheme every data/instruction access requires 2 memory accesses:
    1. One for the page table and (to get the frame number)
    2. One for the data / instruction
  - Efficiency is lost



The 2 memory access problem

- **Solution: (Use of TLB's)**
  - Use of a special fast-lookup hardware cache, called:
    - **associative memory,** or
    - **translation look-aside buffers** (**TLBs**)
- TLB
  - Caches (p,f) tuples for frequently used pages
    - That is, the mapping from p to the corresponding f
  - Small
  - Very fast

Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - **If** p is in TLB **then** get the frame # out
    - Hardware searches in parallel all entries at the same time
    - Very fast
  - **else** get the frame # from page table in memory

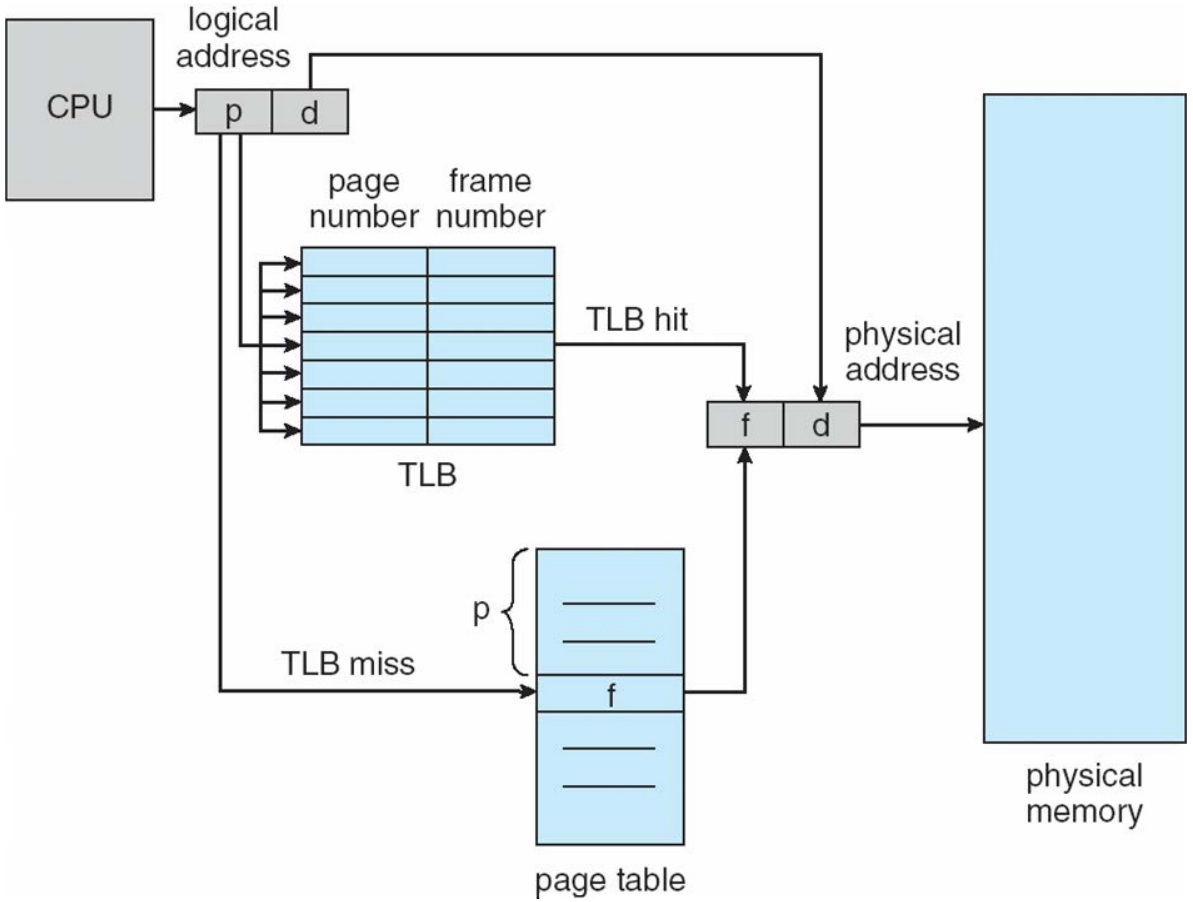
Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry
  - ASID uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch

PID	Page#	Frame#

- TLBs typically small (64 to 1,024 entries)
- 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

Paging Hardware With TLB



Effective Access Time

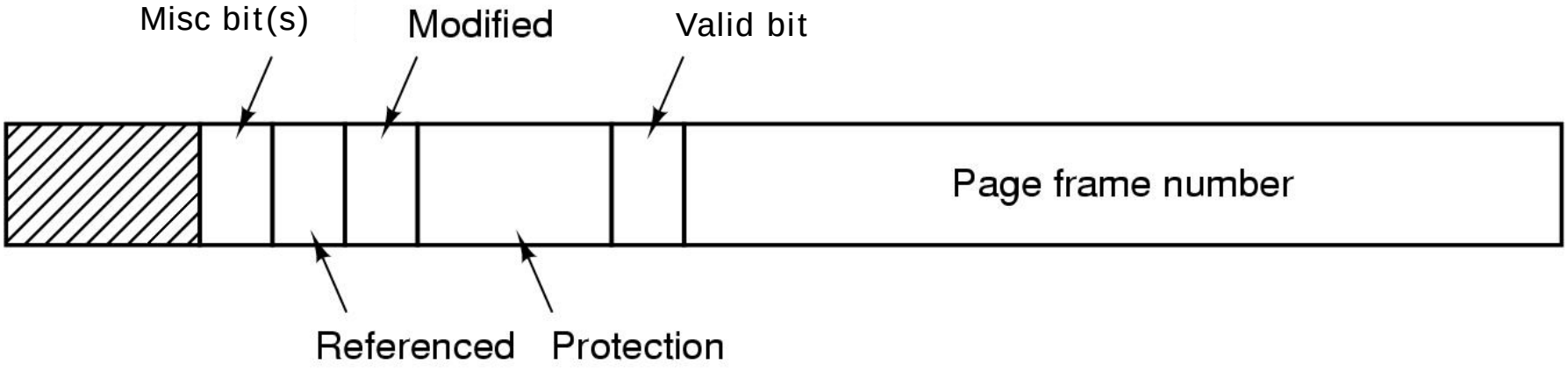
- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Percentage of times that a page number is found in the TLB
  - Hit ratio is related to number of associative registers in TLB
- **Effective Access Time** (**EAT**)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

// 1 memory access plus TLB access time, or 2 memory accesses +TLB

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140 ns
- Consider more realistic hit ratio α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 121 ns

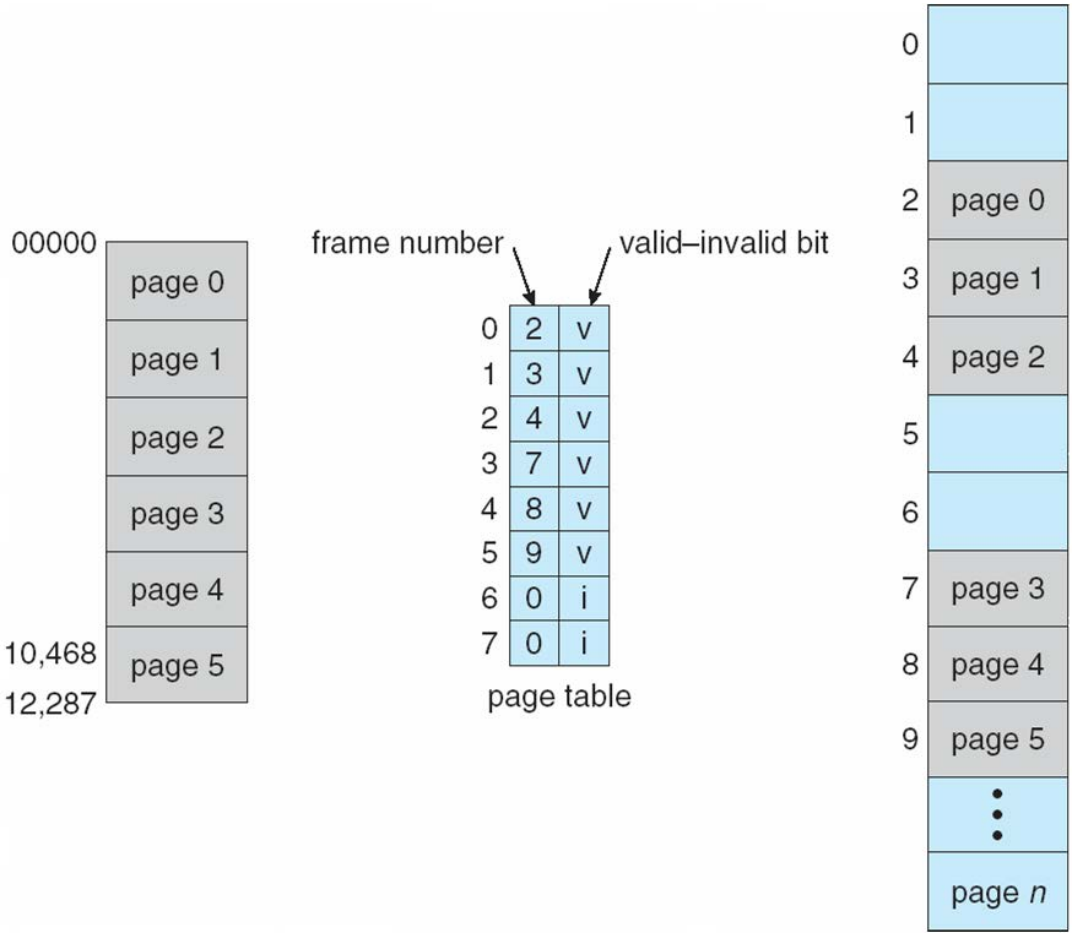
Typical Page Table Entry



Memory Protection

- Memory protection in a paged environment is accomplished by protection bits associated with each frame.
- For example, protection bit to indicate if
  - read-only or
  - read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- Normally, these bits are kept in the page table.
- **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
- Some system have **page-table length register** (**PTLR**)
  - Can also be used to check if address is valid
- Any violations result in a trap to the kernel

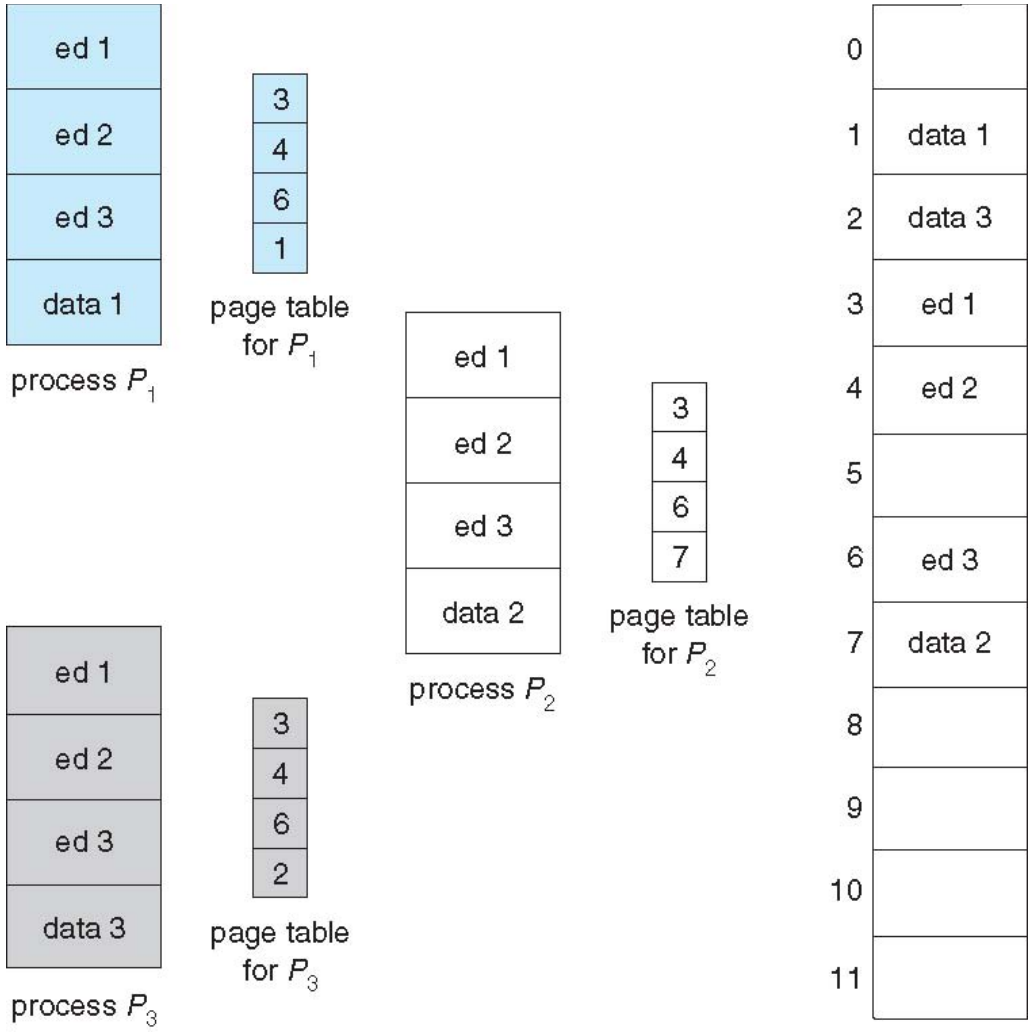
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- An advantage of paging is the possibility of sharing data and common code
- **Shared code**
  - Particularly important in a time-sharing environment
    - Ex: A system that supports 40 users, each executes a text editor
  - A single copy of read-only (**reentrant**) code shared among processes
    - For example, text editors, compilers, window systems
    - Reentrant code is non-self-modifying code: never changes during exec.
  - This is similar to multiple threads sharing the same process space
  - 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.
- **Shared data**
  - Some OSes implement shared memory suing shared 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

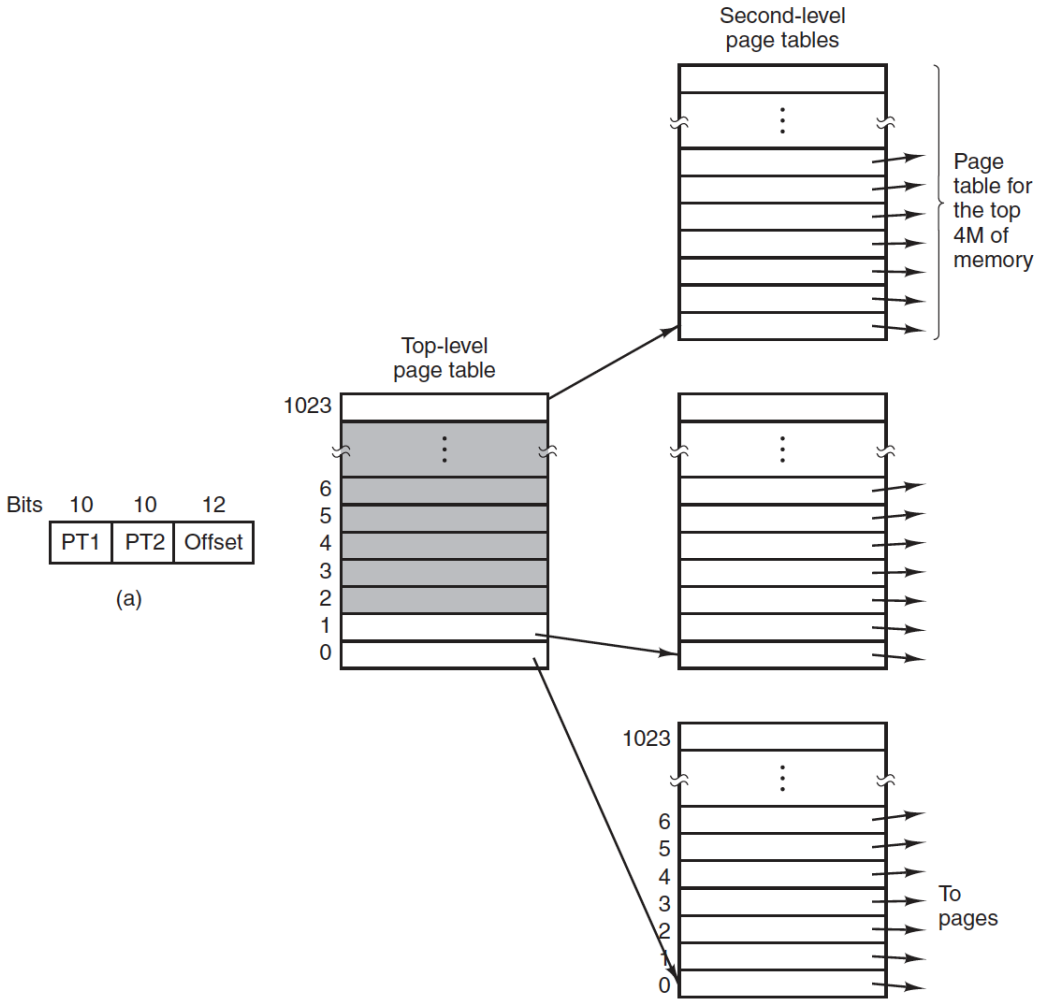


Structure of the Page Table

- Memory structures for paging can get huge using straightforward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 1 KB (2^{10})
  - Page table would have 4 million entries (2^{32} / 2^{10}= 2^{22})
  - **Problem:** If each entry is 4 bytes -> 16 MB of physical address space / memory for page table alone
    - This is **per process**
    - That amount of memory used to cost a lot
    - Do not want to allocate that contiguously in main memory
  - **Solution:** One simple solution to this problem is to divide the page table into smaller pieces.
  - We can accomplish this division in several ways, e.g.:
    - 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

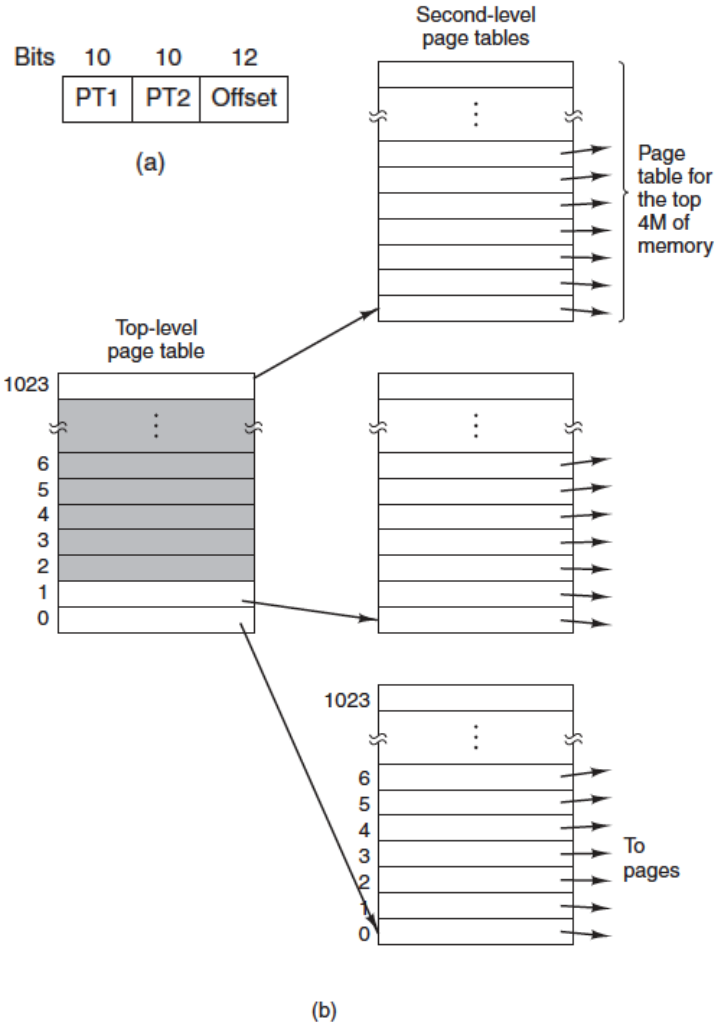


Two-Level Paging Example

- 32-bit virtual address is partitioned
  - 10-bit PT1 field
  - 10-bit PT2 field
  - 12-bit offset
- 12-bit offset => pages are 4K(=2^{12}) and 2^{20} of them
- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.
  - In particular, those that are not needed should not be kept around.
- Suppose, that a process needs 12 MB:
  - the bottom 4 MB of memory for program text,
  - the next 4 MB for data, and
  - the top 4 MB for the stack.
  - Hence, a gigantic hole that is not used
    - in between the top of the data and the bottom of the stack

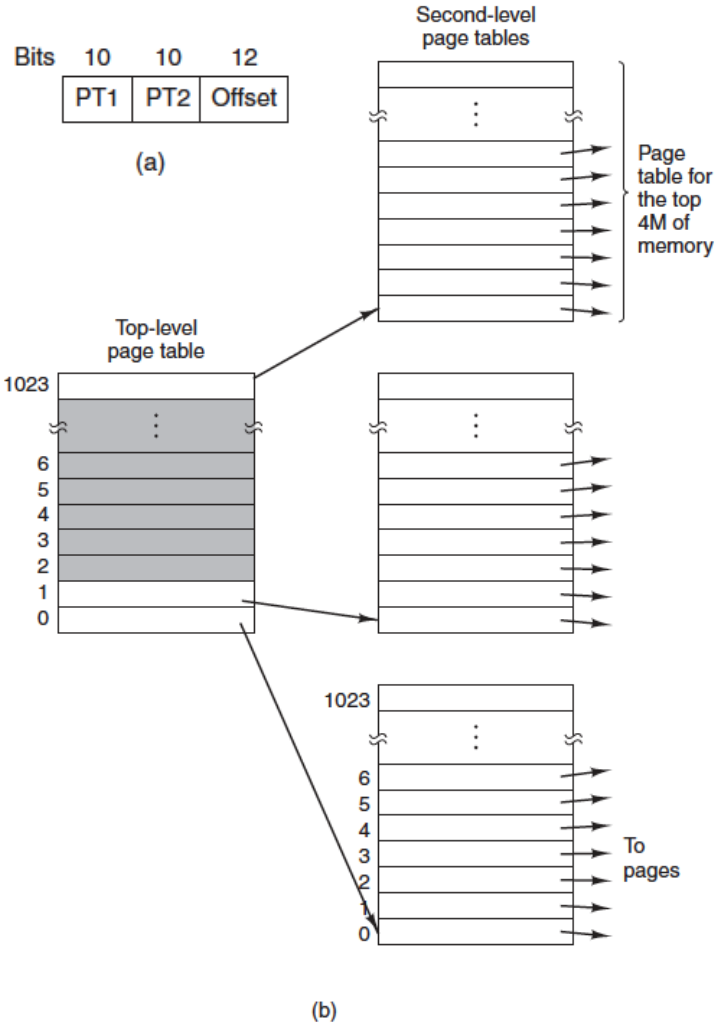
Two-Level Paging

- The top-level page table, with 1024 entries, corresponding to the 10-bit PT1 field.
- When a virtual address is presented to the MMU, it first extracts the PT1 field and uses this value as an index into the top-level page table.
- Each of these 1024 entries in the top-level page table represents 4 MB
  - 4 GB (i.e., 32-bit) virtual address space has been chopped into 1024 chunks
  - 4 GB / 1024 = 4 MB



Two-Level Paging

- The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table.
- Entry 0 of the top-level page table points to the page table for the program text,
- Entry 1 points to the page table for the data,
- Entry 1023 points to the page table for the stack.
- The other (shaded) entries are not used.
  - No need to generate page tables for them
  - Saving lots of space!
- The PT2 field is now used as an index into the selected second-level page table to find the page frame number for the page itself.



Two-Level Paging: Example

- Example: consider the 32-bit virtual address 0x00403004 (4,206,596 decimal),
  - which is 12,292 bytes into the data (4,206,596 – 4,194,304= 12,292).
  - Corresponds to PT1 = 1, PT2 = 3, and Offset = 4.
- The MMU first uses PT1 to index into the top level page table
  - It obtains entry 1, which corresponds to addresses 4M to 8M − 1.
  - It finds the corresponding 2-level page table for entry 1
- It then uses PT2 to index into the second-level page table just found
  - It extract entry 3,
  - which corresponds to addresses 12288(=3*4K) to 16383 within its 4M chunk
  - (i.e., absolute addresses 4,206,592 to 4,210,687).
- This entry contains the page frame number of the page containing virtual address 0x00403004.
- If that page is not in memory, the valid_bit =0, causing a page fault.
- If the page is present in memory, the page frame number taken from the second-level page table is combined with the offset (=4) to construct the physical address. This address is put on the bus and sent to memory.

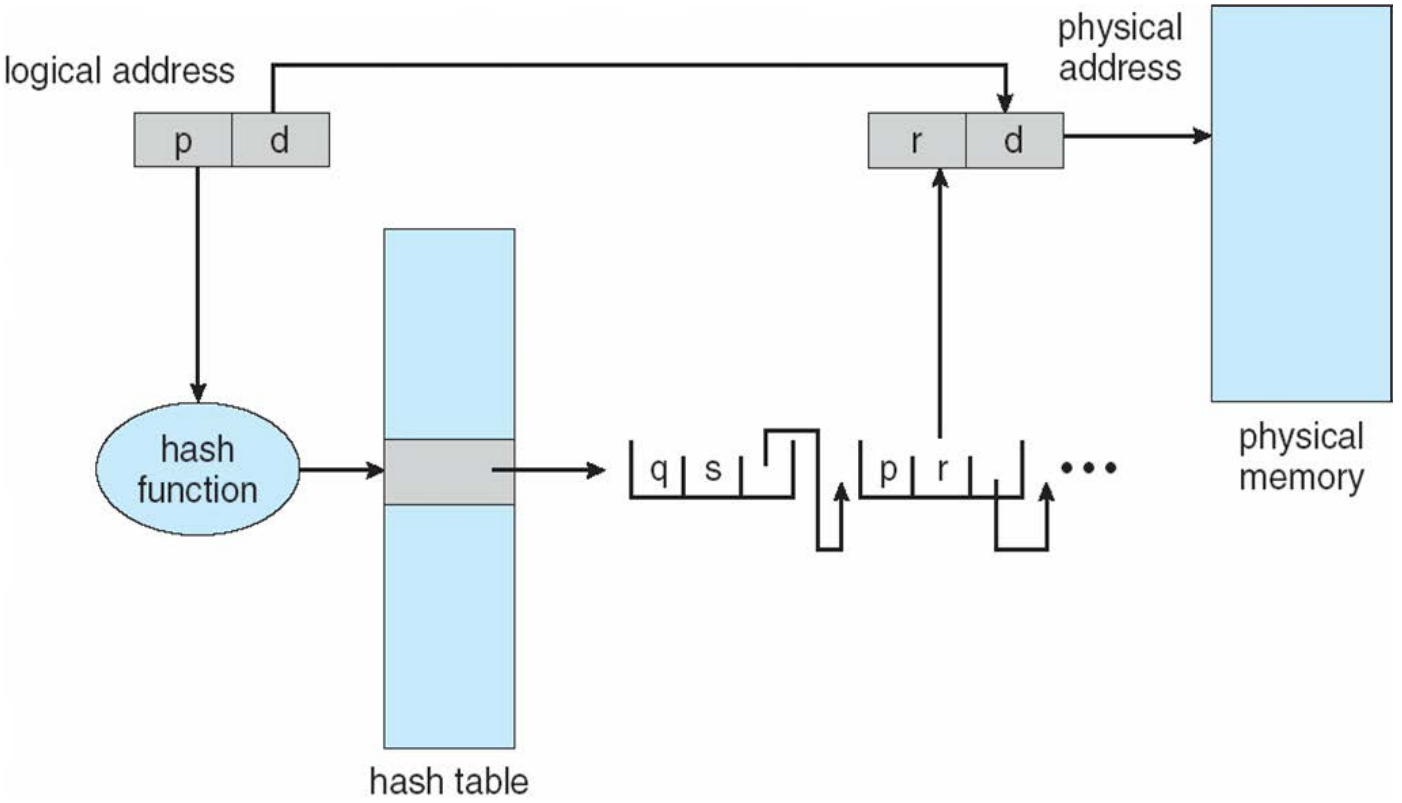
Two-Level Paging

- Notice, although the address space contains over a million pages, only 4 page tables are needed:
  1. the top-level table,
  2. the 2nd-level table for 0 to 4M (for the program text),
  3. the 2nd-level table 4M to 8M (for the data), and
  4. the 2nd-level table for the top 4M (for the stack).
- Valid bits in the remaining 1021 entries of the top-level page table are =0
  - forcing a page fault if they are ever accessed.
- In this example we have chosen round numbers for the various sizes and have picked PT1 equal to PT2,
  - but in actual practice other values are also possible, of course.
- The two-level page table system can be expanded to 3, 4, or more levels.
- Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.

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

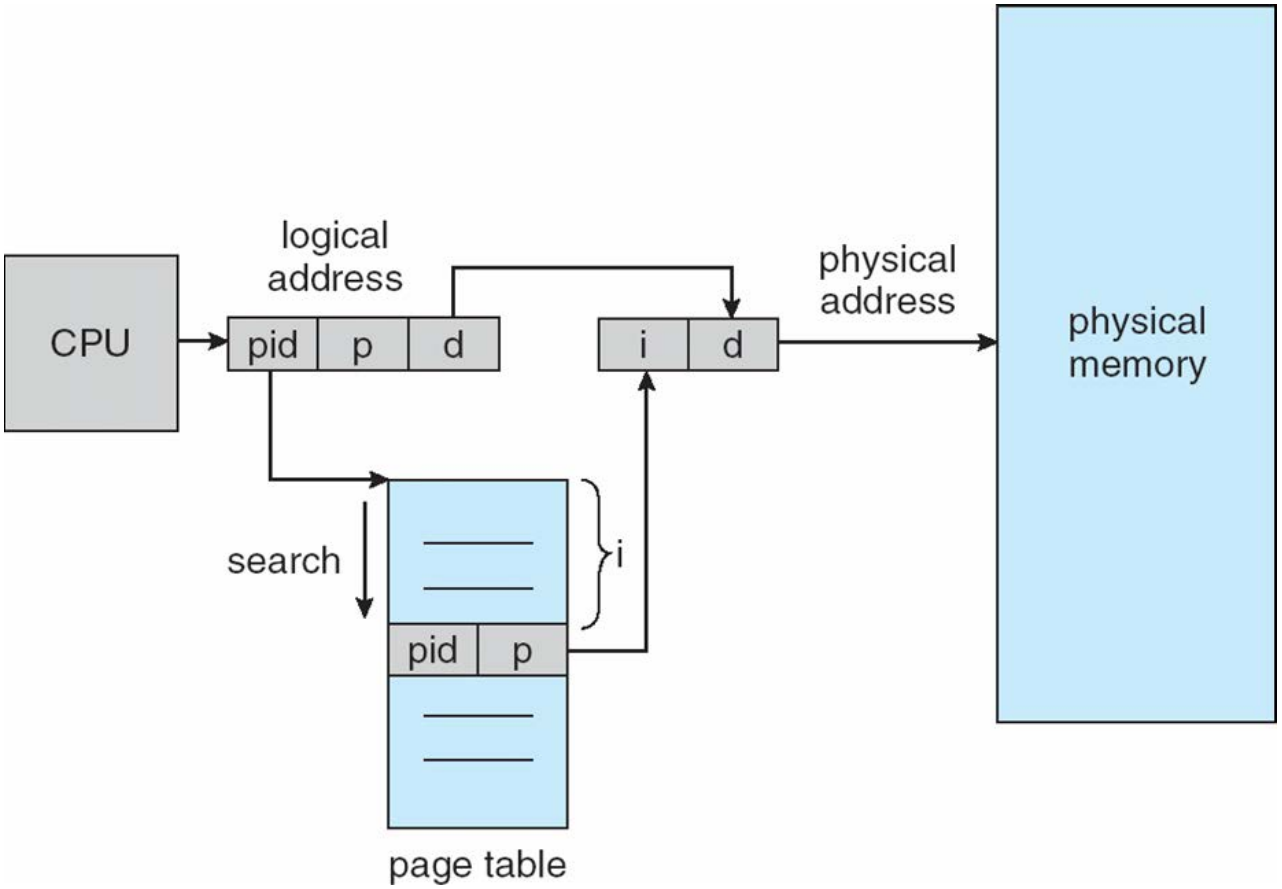
Hashed Page Table



Inverted Page Table

- **Motivation:** 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
  - The entry keeps track of which (process, virtual page) is located in the page frame.
- **Pros:** tends to save lots of space
- **Cons:** virtual-to-physical translation becomes much harder
- When process **n** references virtual page **p**, the hardware can no longer find the physical page by using **p** as an index into the page table.
- Instead, it must search the entire inverted page table for an entry **(n, p)**.
- Furthermore, this search must be done on every memory reference, not just on page faults.
- Searching a 256K table on every memory reference is slow
- Other considerations:
  - TLB and hash table (key: virtual address) is used to speed up accesses
  - Issues implementing shared memory when using inverted page table

Inverted Page Table Architecture



End of Chapter 8