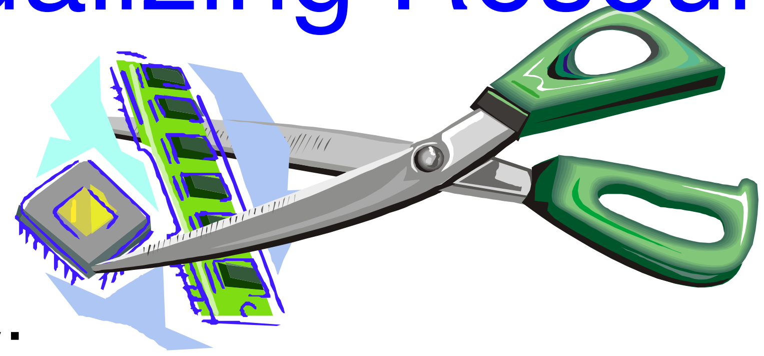
Main Memory

Goals for Today

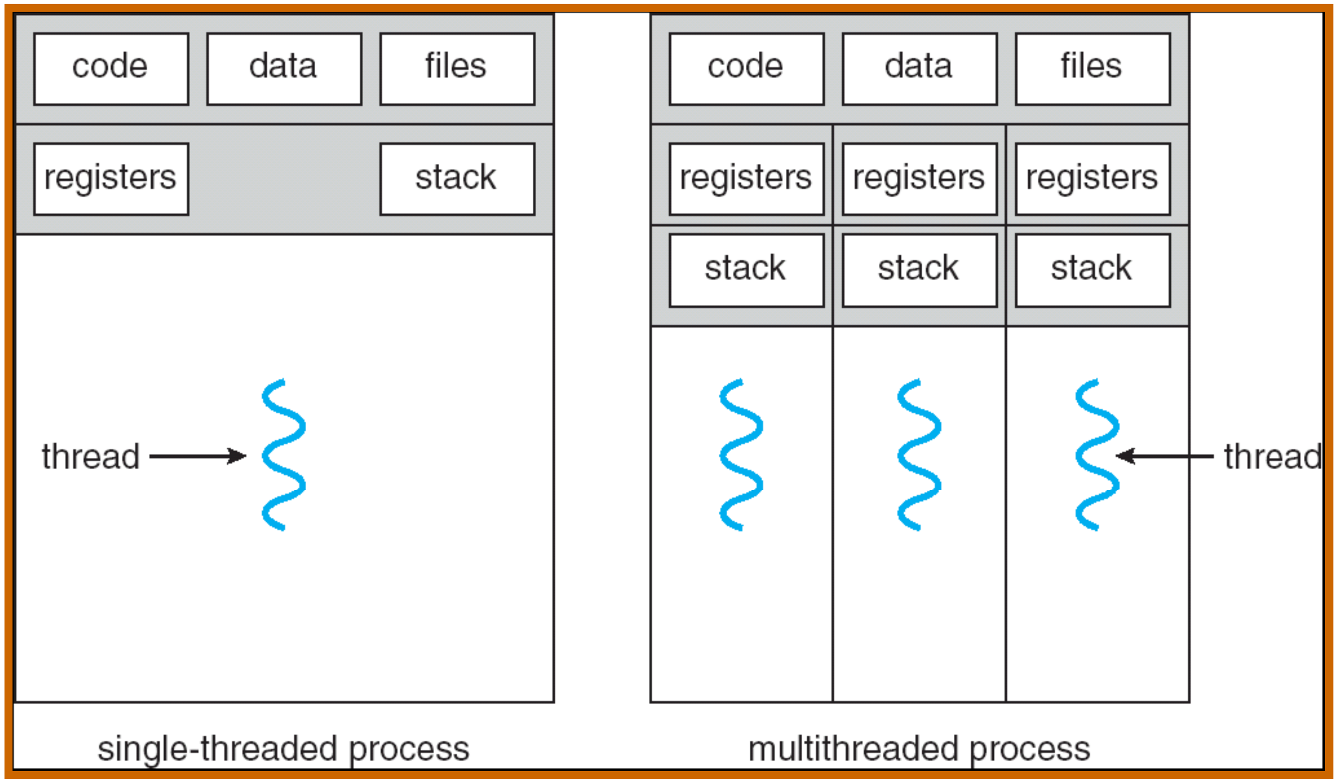
- Protection: Address Spaces
  - What is an Address Space?
  - How is it Implemented?
- Address Translation Schemes
  - Segmentation
  - Paging
  - Multi-level translation
  - Inverted page tables

Virtualizing Resources



- Physical Reality:
  Different Processes/Threads share the same hardware
  - Need to multiplex CPU (temporal)
  - Need to multiplex use of Memory (spatial)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Probably don’t want different threads to even have access to each other’s memory (protection)

Recall: Single and Multithreaded Processes



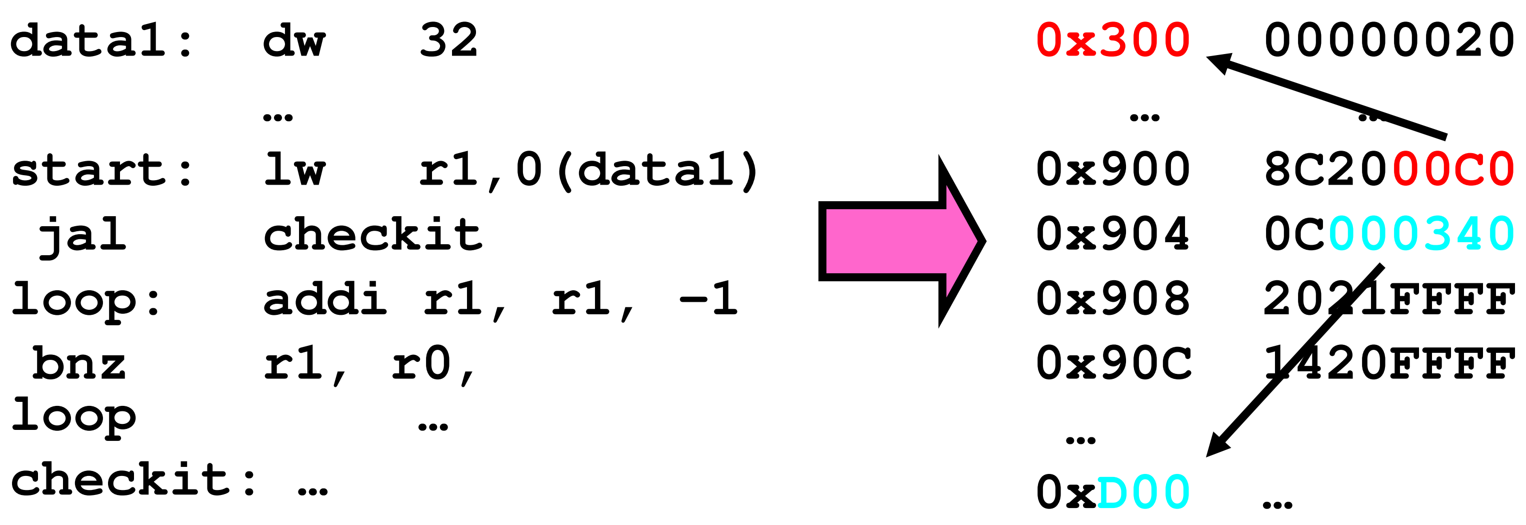
- Threads encapsulate concurrency
  - "Active" component of a process
- Address spaces encapsulate protection
  - E.g. Keeps buggy program from trashing the system

Important Aspects of Memory Multiplexing

- Isolation
  - Separate state of processes should not collide in physical memory.
    - Obviously, unexpected overlap causes chaos!
- Sharing
  - Conversely, would like the ability to overlap when desired
    - for communication
- Virtualization
  - Create the illusion of more resources than there exist in the underlying physical system

Binding of Instructions and Data to Memory

- Binding of instructions and data to addresses:
  - Choose addresses for instructions and data from the standpoint of the processor



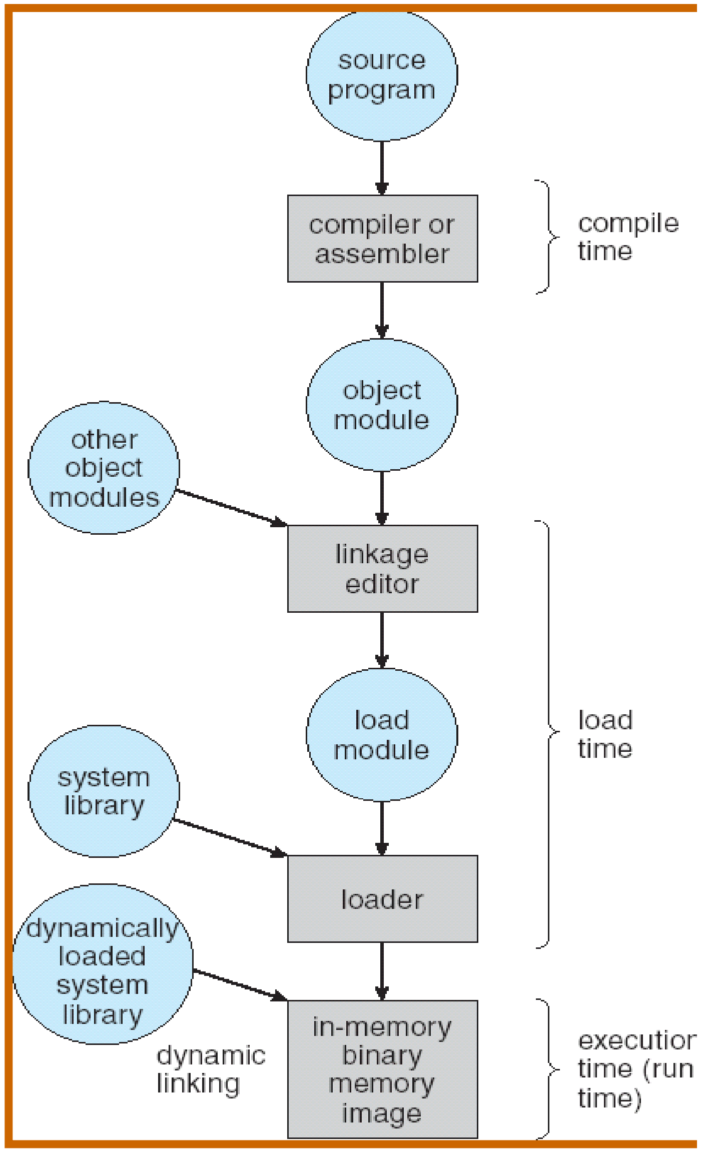
```
data1:   dw    32
         …
start:   lw    r1,0(data1)
 jal     checkit
loop:    addi r1, r1, -1
 bnz     r1, r0,
loop           …
checkit: …
```

```
0x300   00000020
  …        …
0x900   8C2000C0
0x904   0C000340
0x908   2021FFFF
0x90C   1420FFFF
 …
0xD00   …
```

  - Could we place `data1`, `start`, and/or `checkit` at different addresses?
    - Yes
    - When? Compile time/Load time/Execution time

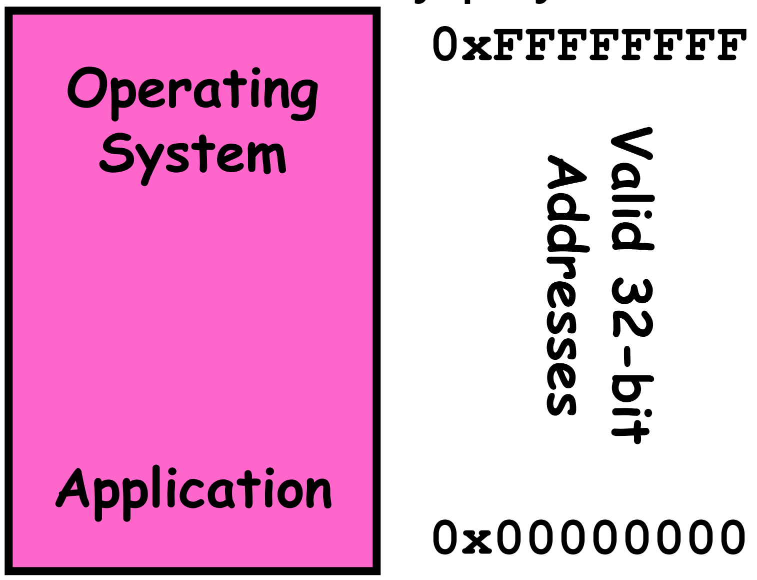
Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at
  - Compile time (i.e. “gcc”)
  - Link/Load time (unix “ld” does link)
  - Execution time (e.g. dynamic libs)
- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system
- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Recall: Uniprogramming

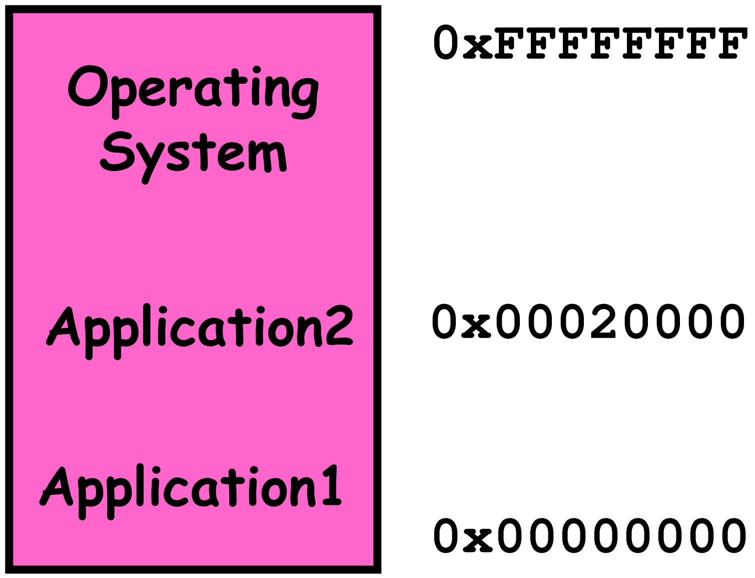
- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address



  - Application given illusion of dedicated machine by giving it reality of a dedicated machine
- Of course, this doesn't help us with multithreading

Multiprogramming (First Version)

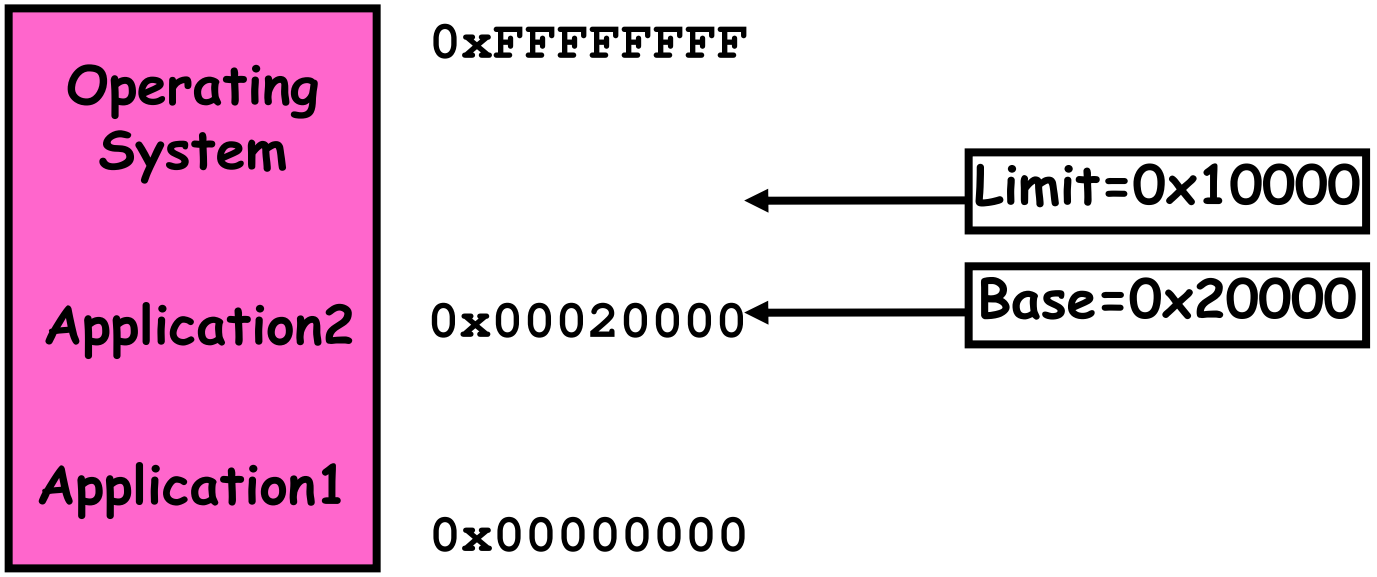
- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads



  - Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - Everything adjusted to memory location of program
    - Translation done by a linker-loader
    - Was pretty common in early days
- With this solution, no protection
  - bugs in any program can cause other programs to crash or even the OS

Multiprogramming (Version with Protection)

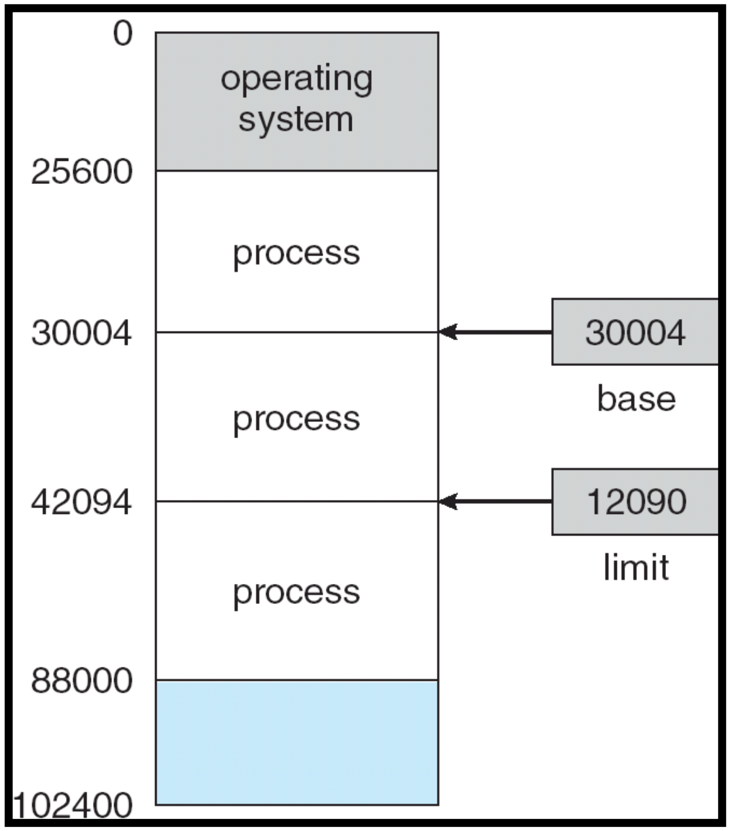
- Can we protect programs from each other without translation?



  – Yes: use two special registers *base* and *limit* to prevent user from straying outside designated area
    - If user tries to access an illegal address, cause an error
  – During switch, kernel loads new base/limit from TCB
    - User not allowed to change base/limit registers

Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space



Multiprogramming (Translation and Protection v. 2)

- Problem: Run multiple applications in such a way that they are protected from one another
- Goals:
  - Isolate processes and kernel from one another
  - Allow flexible translation that:
    - Doesn't lead to fragmentation
    - Allows easy sharing between processes
    - Allows only part of process to be resident in physical memory
- (Some of the required) Hardware Mechanisms:
  - General Address Translation
    - Flexible: Can fit physical chunks of memory into arbitrary places in users address space
    - Not limited to small number of segments
    - Think of this as providing a large number (thousands) of fixed-sized segments (called "pages")
  - Dual Mode Operation

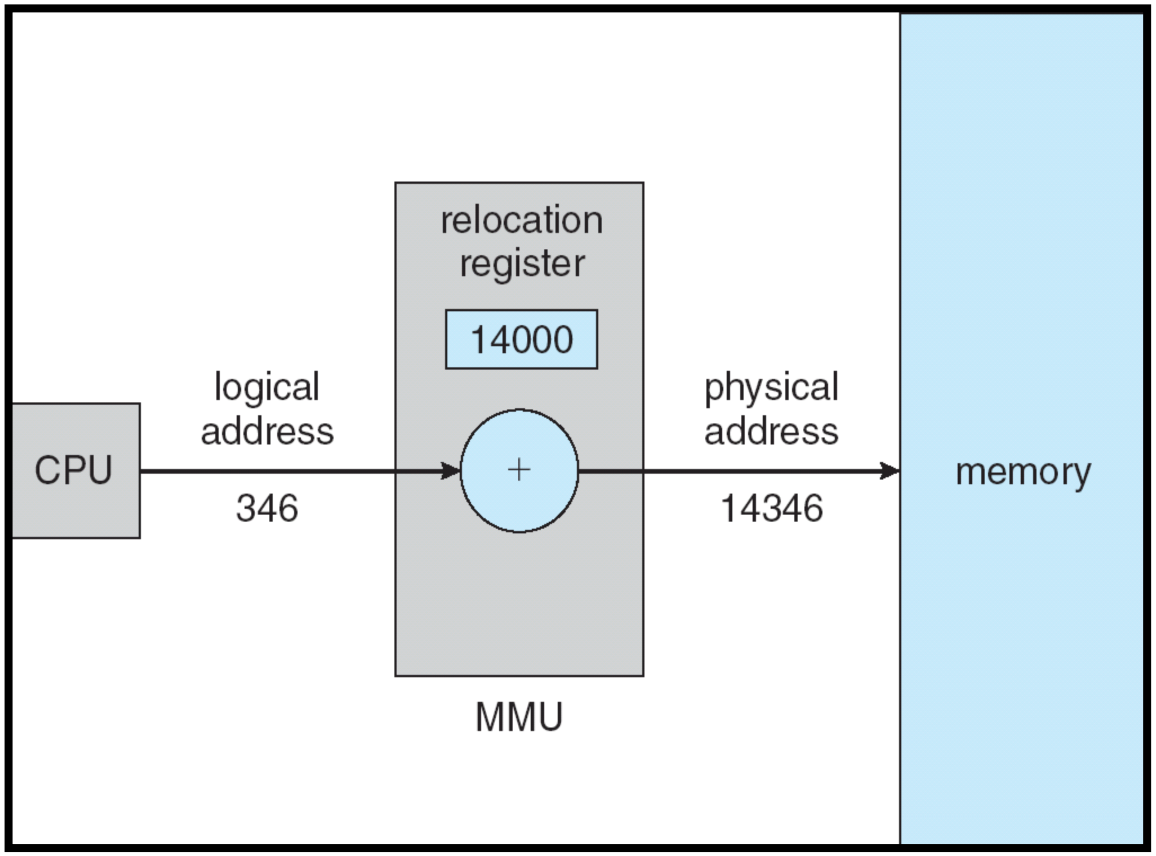
Memory Background

- 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
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU 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
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

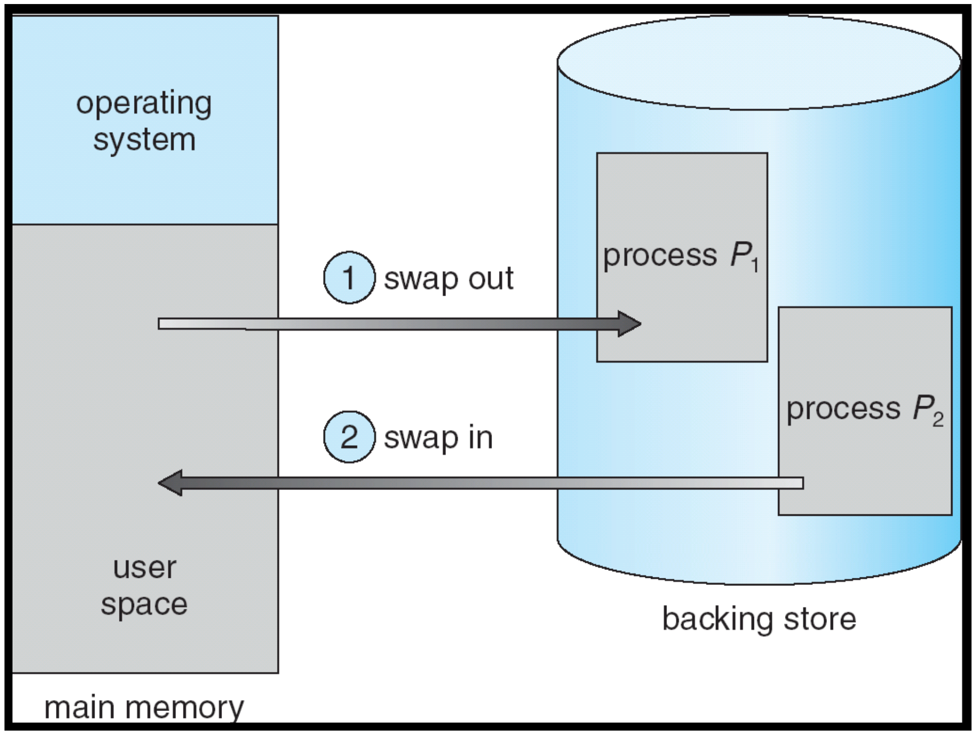
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases (error handling)
- No special support from the OS needed

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- OS checks if routine is in processes' memory address
- Also known as **shared libraries (e.g. DLLs)**

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution



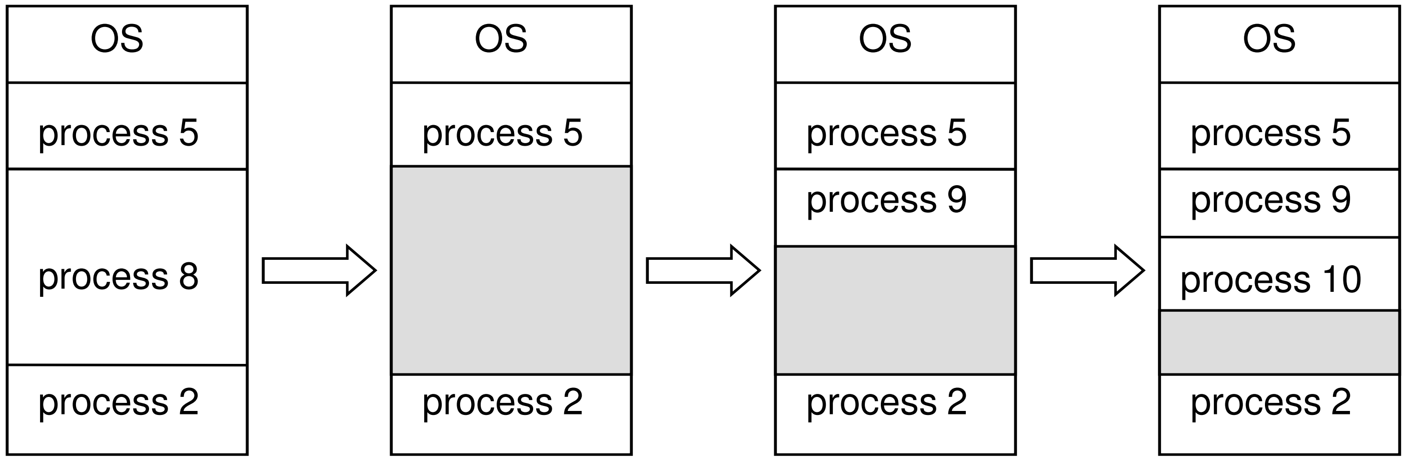
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

Contiguous Allocation

- Main memory usually into two partitions:
  - Resident OS, usually held in low memory with interrupt vector
  - User processes then held in high memory
- 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
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - 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
  - Operating system maintains information about: a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

- **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

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

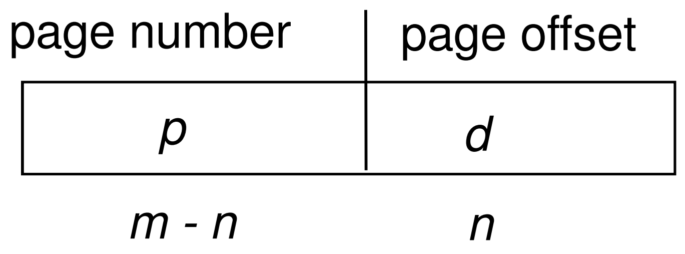
Paging

Paging - overview

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- 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
- What sort of fragmentation?

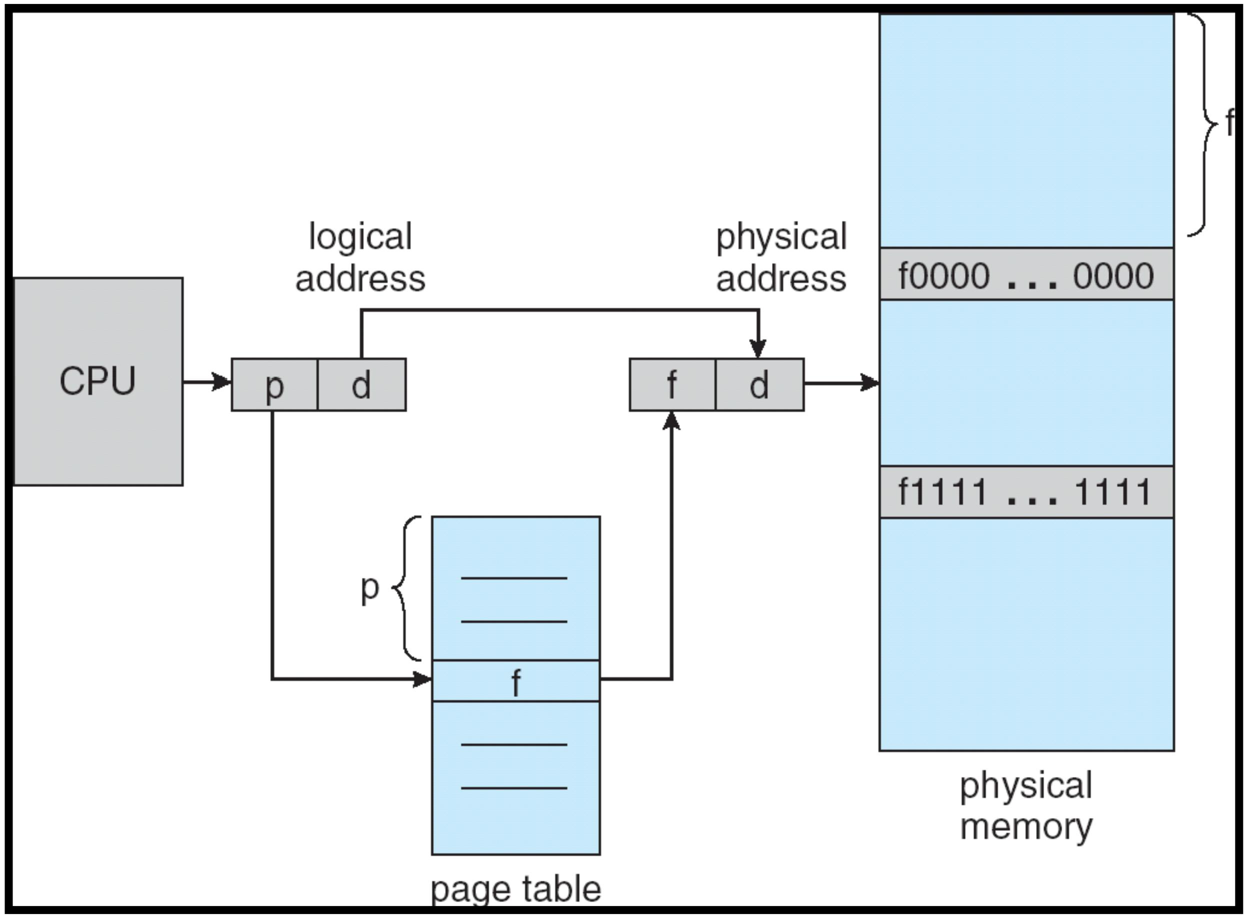
Address Translation Scheme

- 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

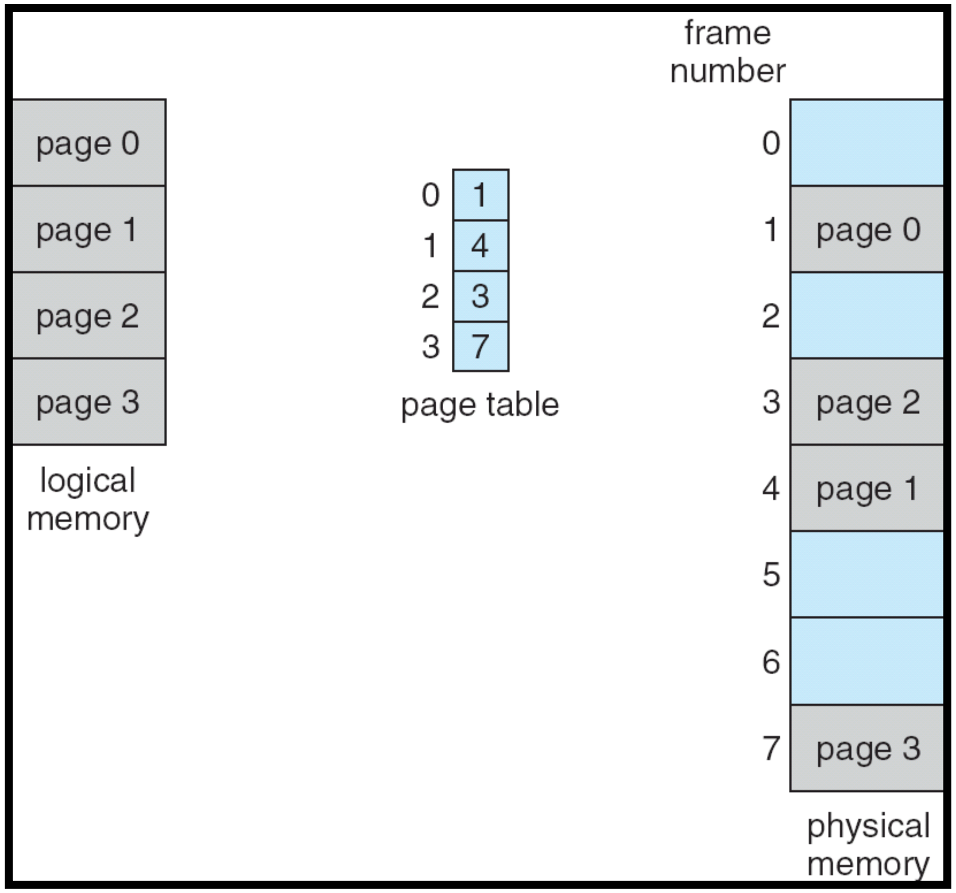


  - For given logical address space 2^m *and page size* 2^n

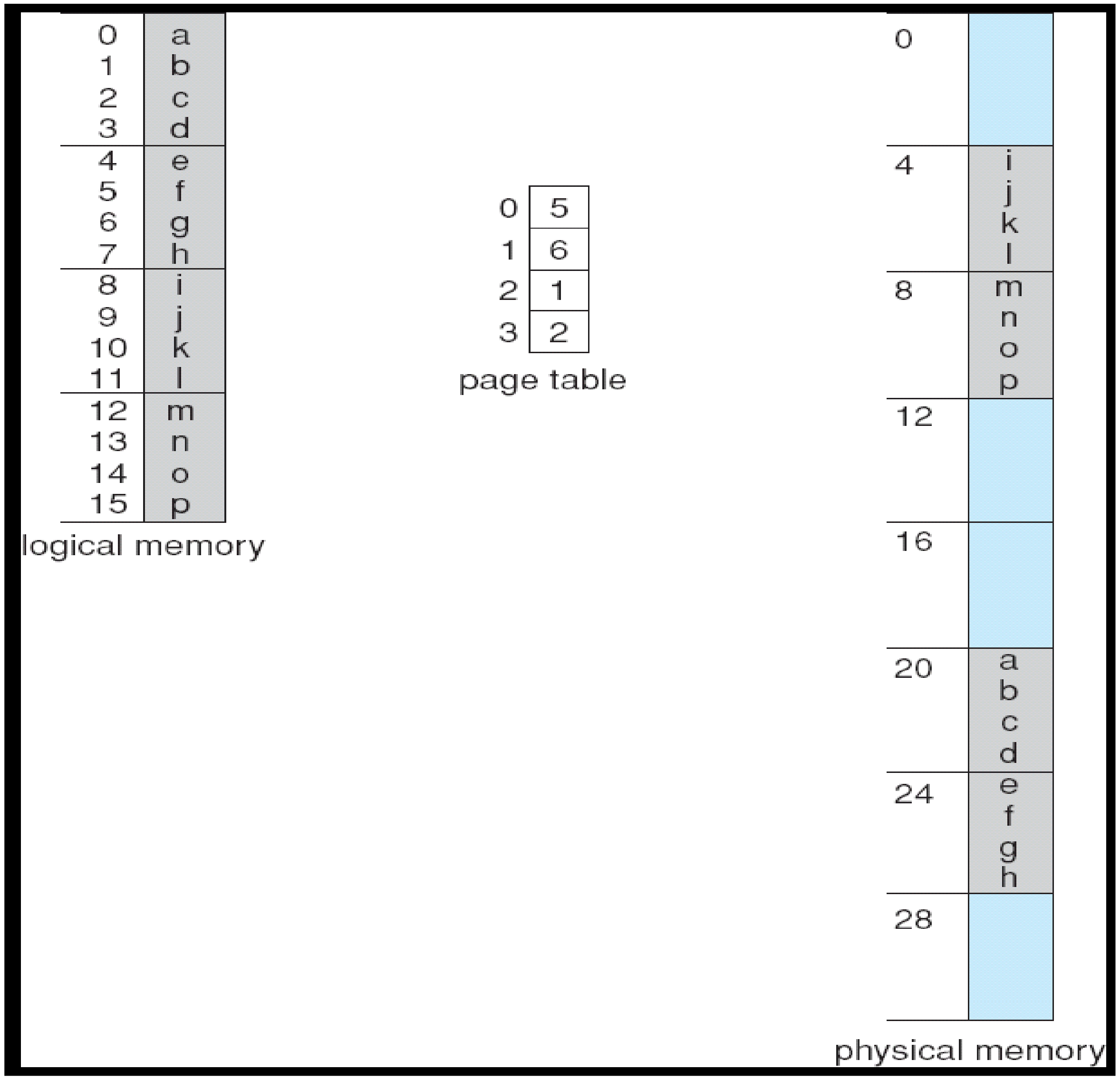
Paging Hardware



Paging Model of Logical and Physical Memory

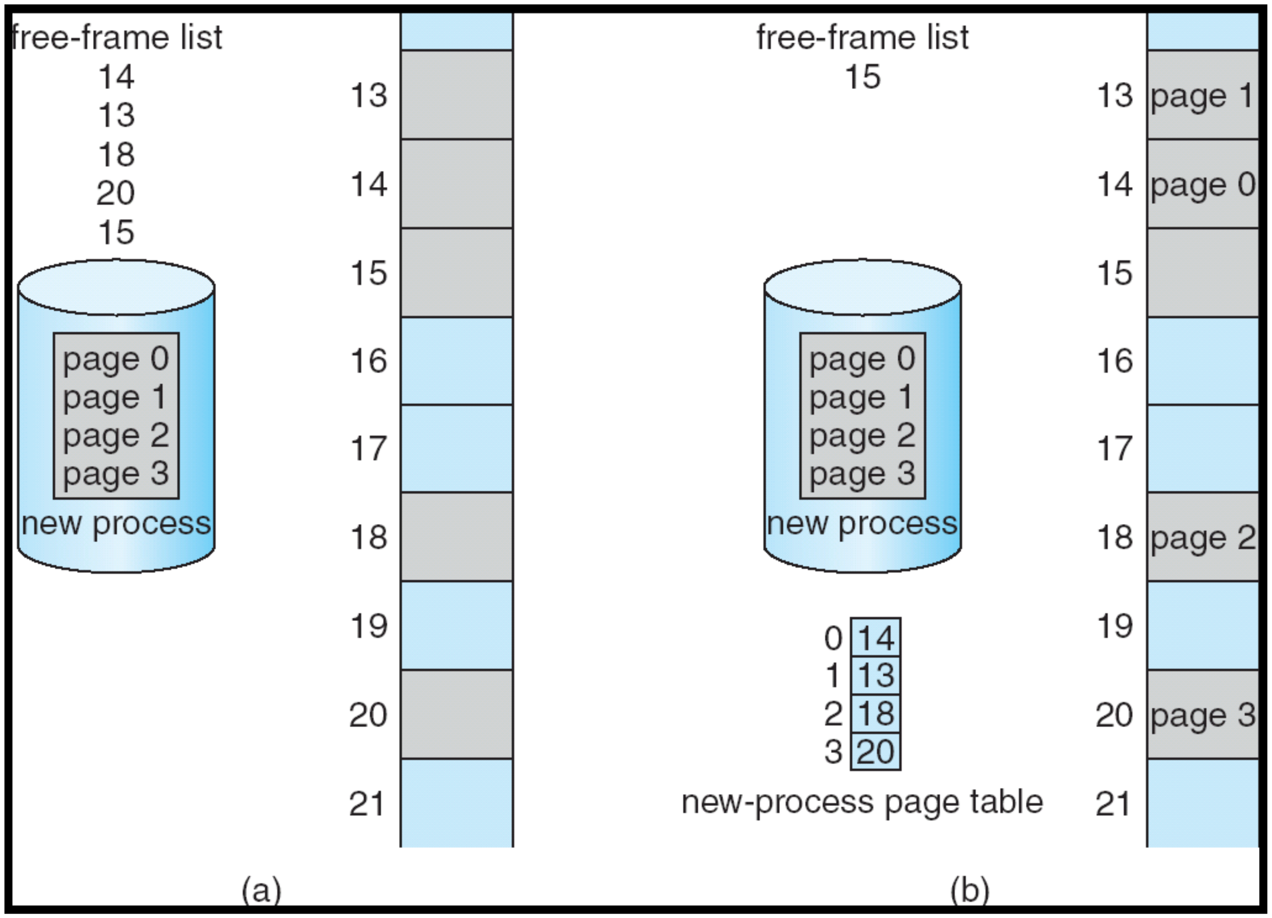


Paging Example



32-byte memory and 4-byte pages

Free Frames



Before allocation

After allocation

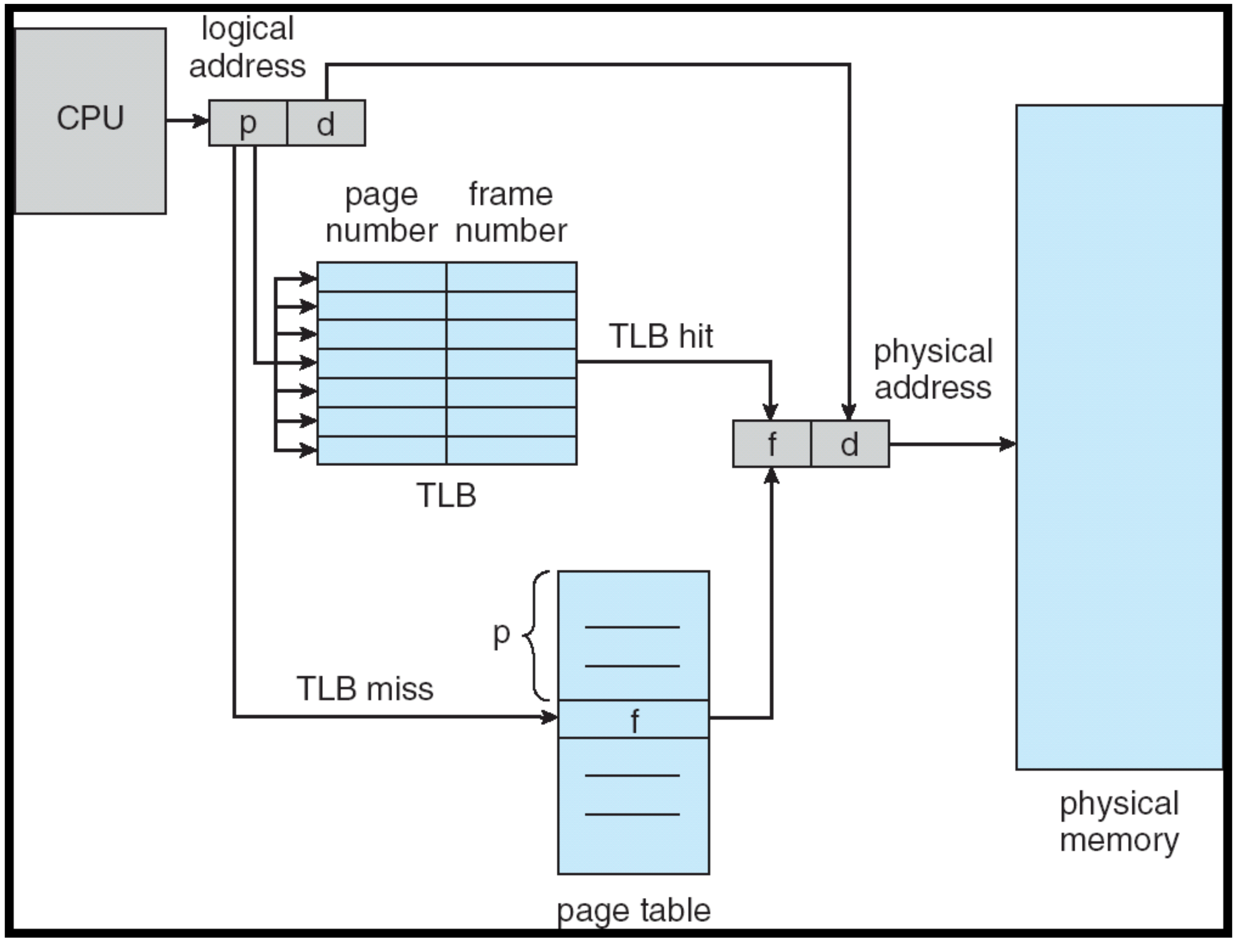
Implementation of Page Table

- Page table can be kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

Translation look-aside buffers (TLBs)

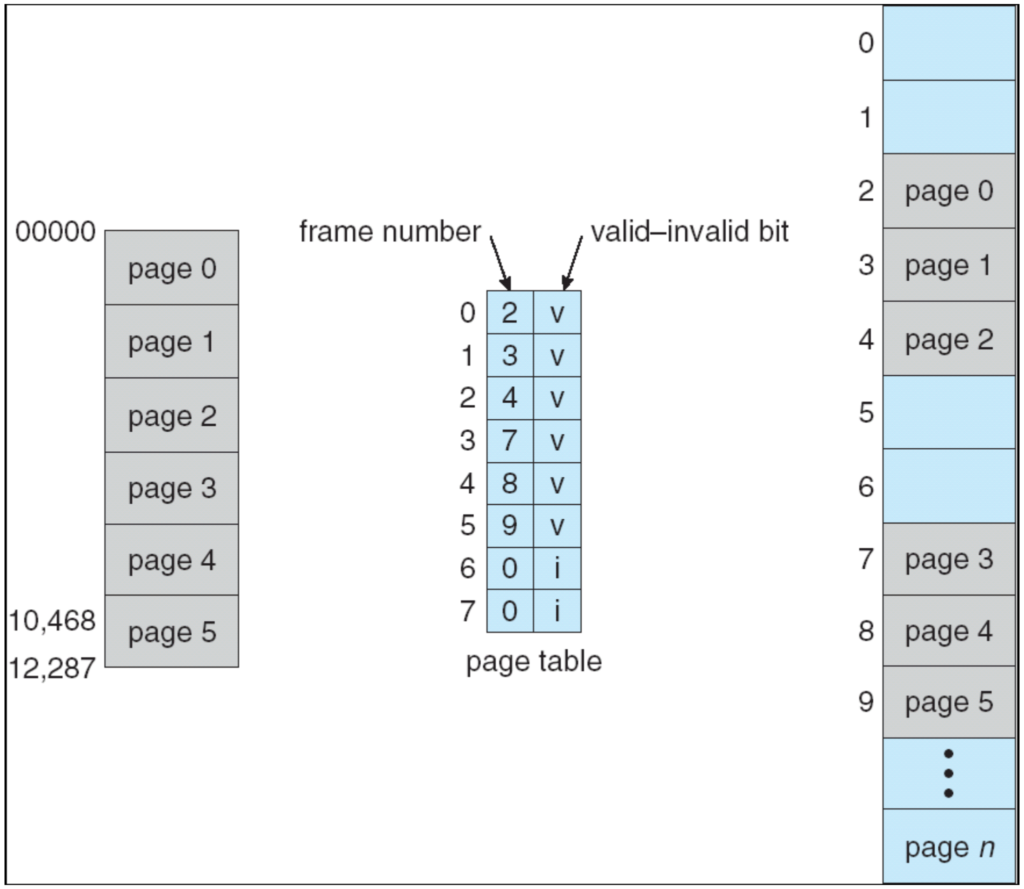
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache (an **associative memory)**
- Allows parallel search of all entries.
- Address translation (p, d)
  - If p is in TLB get frame # out (quick!)
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Memory Protection

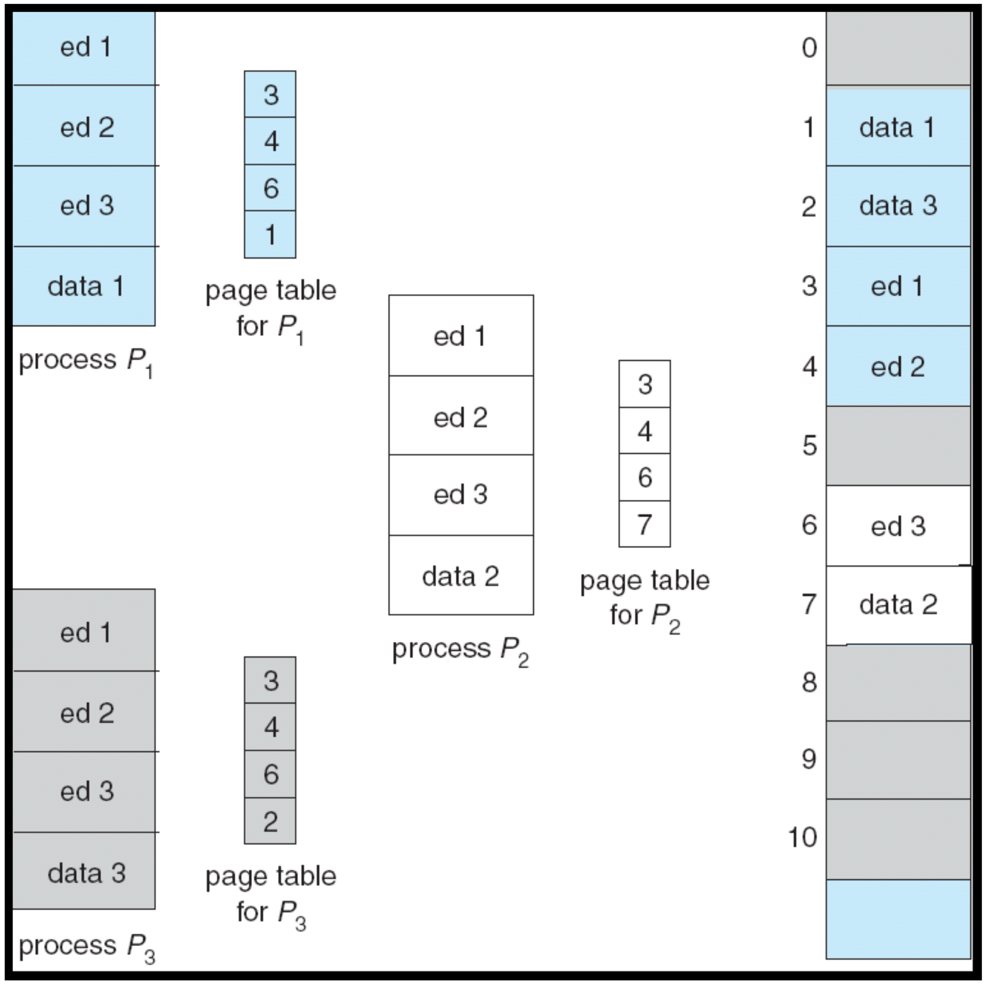
- Implemented by associating protection bit with each frame



Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- **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



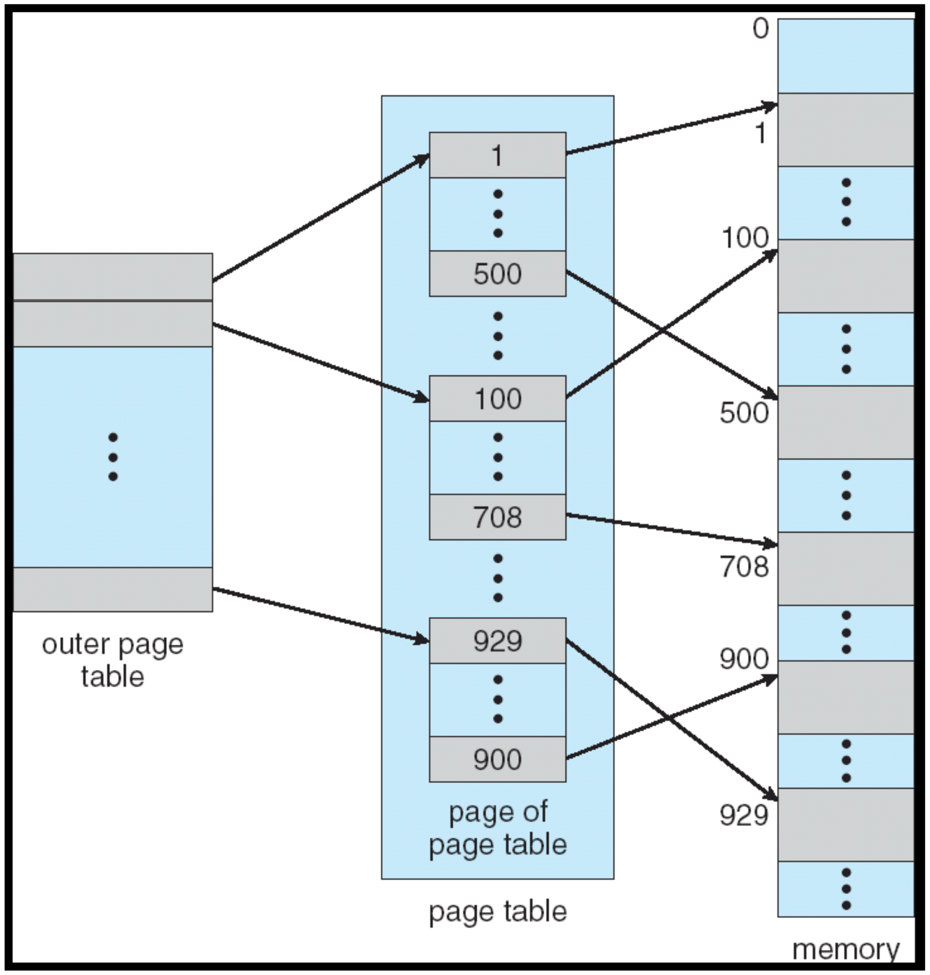
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

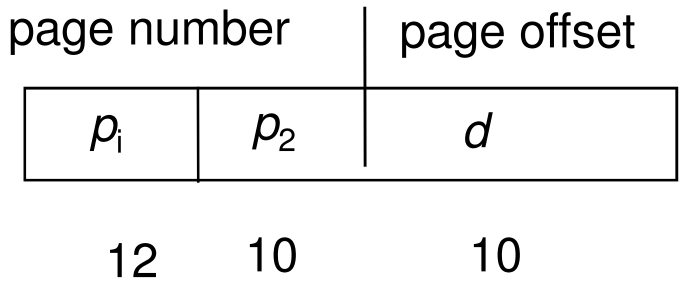
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

Two-Level Page-Table Scheme

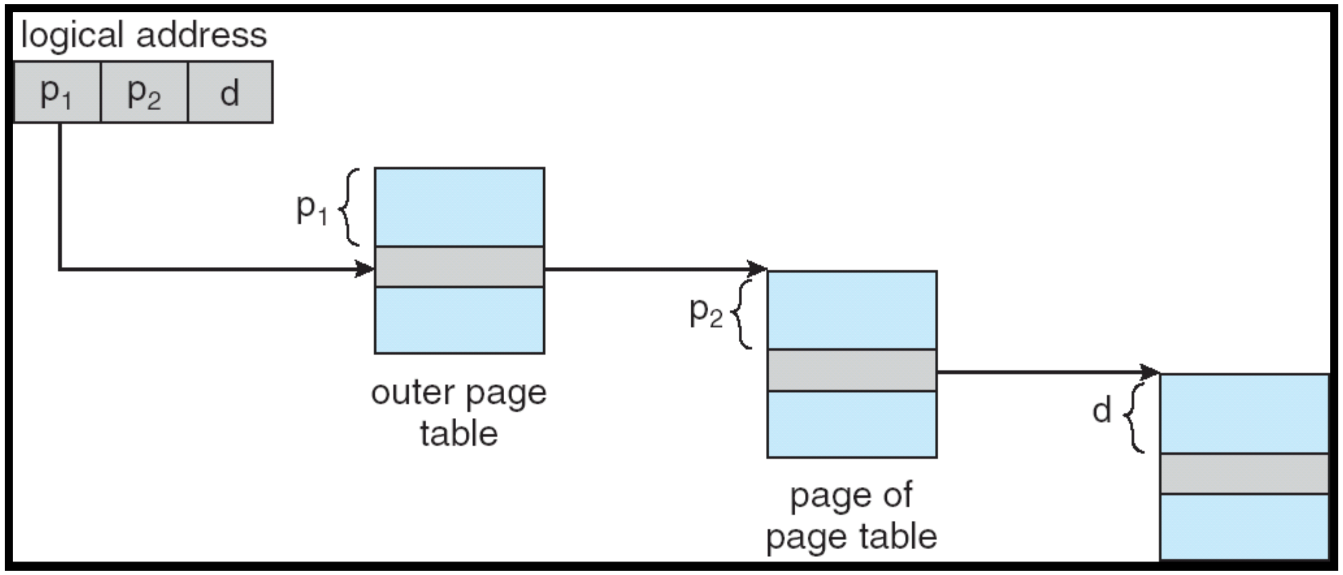


Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page offset of 10 bits (1024 = 2^10)
  - a page number of 22 bits (32-10)
- 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:



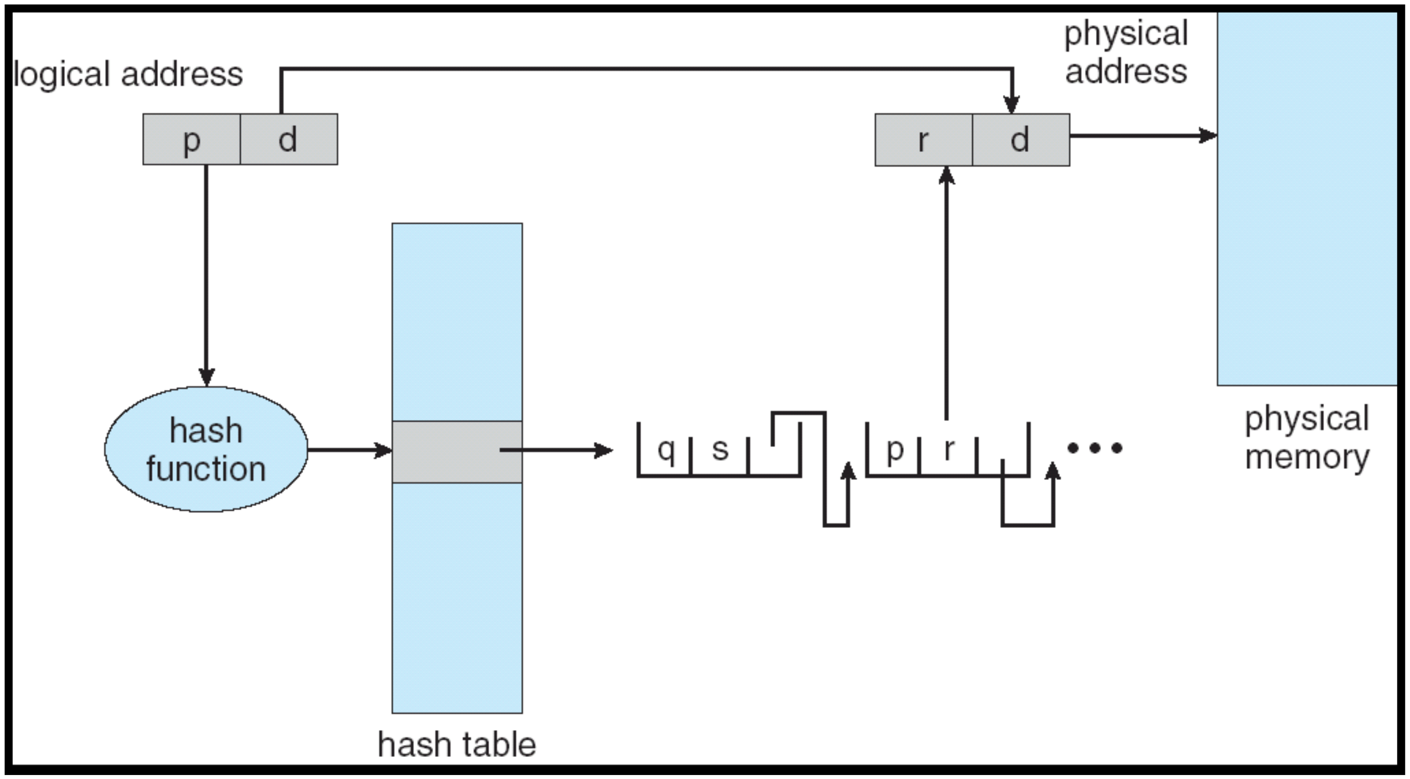
Address-Translation Scheme



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.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

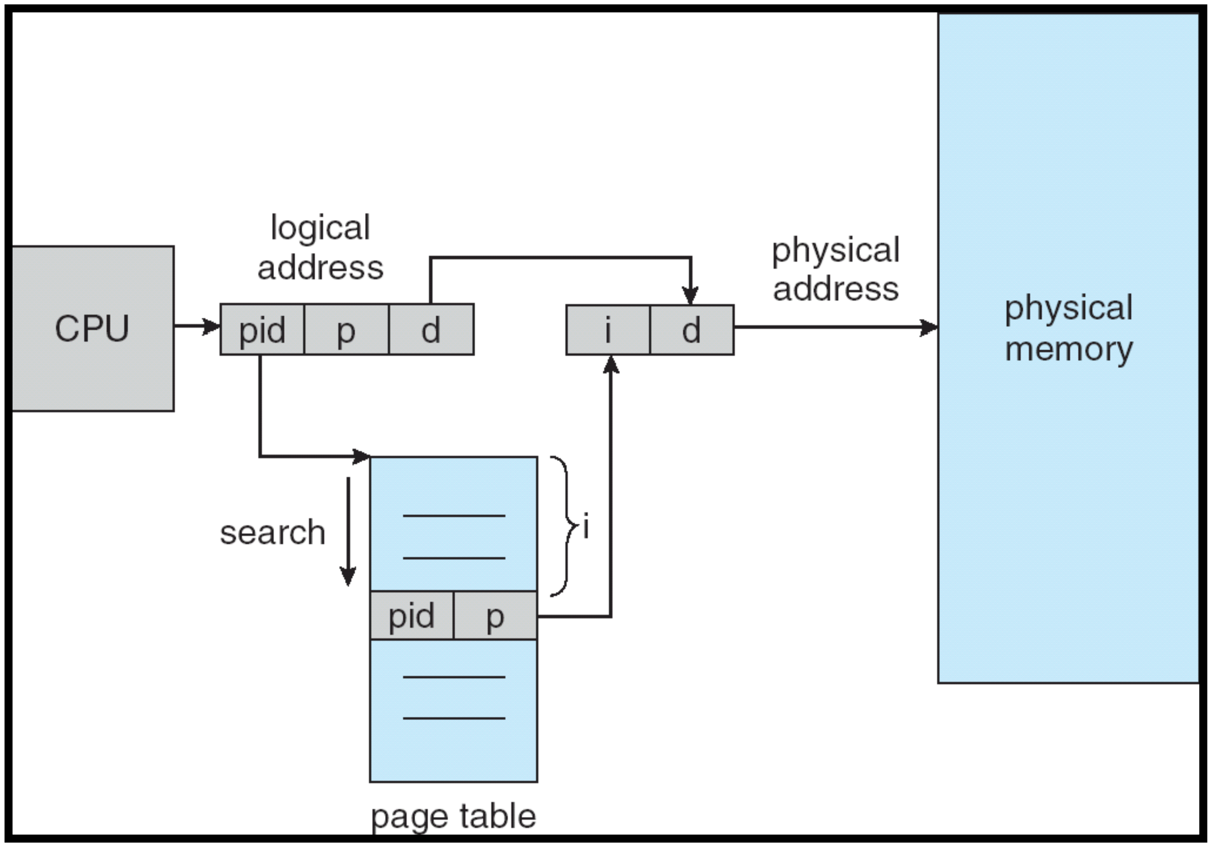
Hashed Page Table



Inverted Page Table

- 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

Inverted Page Table Architecture


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table