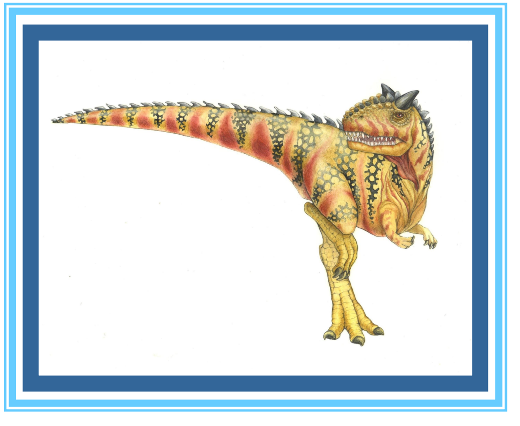
Memory Management



Slides mostly borrowed from Galvin, with some modifications by us

Memory Management: Topics

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

Multiple processes share the memory

- By "Memory", we refer to "Main memory"
- Program must be brought (from disk) into memory and placed within a process for it to be run

- Multiple programs (processes) share the memory
- Protection of memory required to ensure correct operation of various processes that share the memory

What a programmer wants from memory

- Memory addresses will start from 0
- Memory will be contiguous
- I should be able to dynamically increase the memory
- I should have as much memory as I want

In systems where multiple programs (processes) are in memory at the same time, EACH program has the above expectations

Address Binding

- We want to write programs without worrying about where in memory our program will be loaded during execution (we want to assume that memory will start from address zero)
  - Need a mechanism by which user processes can be stored in different areas of the memory
  - Need mechanisms to map addresses generated by a user program/process to actual physical memory addresses
- Address Binding: mapping addresses from one address space to another

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
  - **Logical address space** is the set of all logical addresses generated by a program (**i.e., the CPU**)

- **Physical address** – address seen by the memory unit
  - **Physical address space** is the set of all physical addresses generated by a program (**i.e., accessed in main memory**)

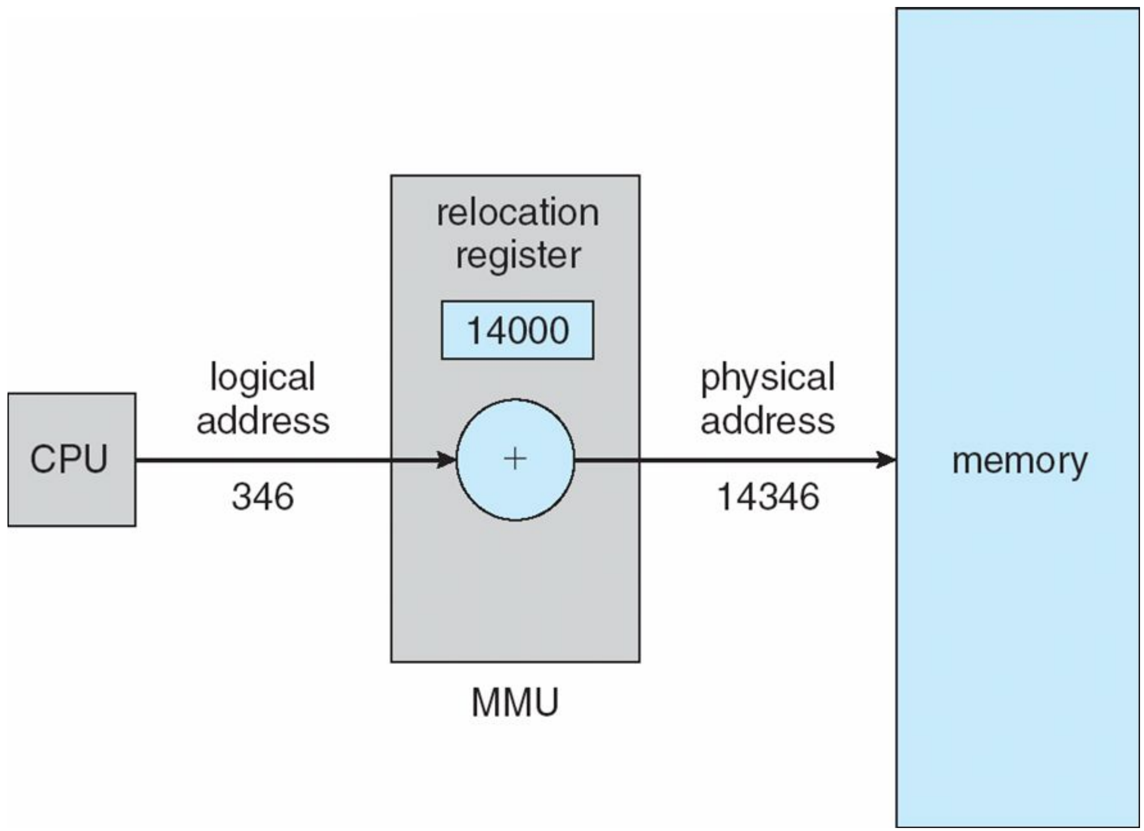
Memory-Management Unit (MMU)

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

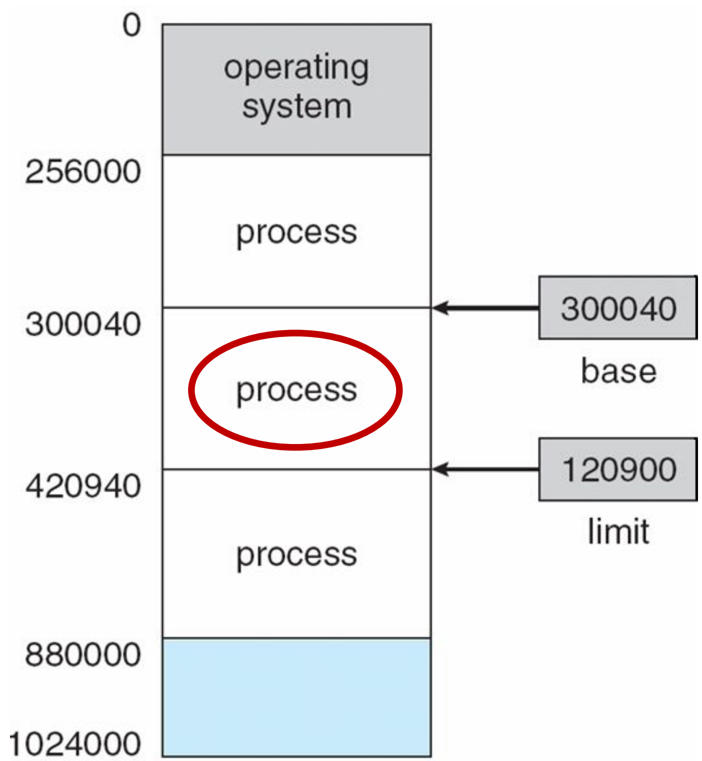
- Consider a simple scheme where the value in a **relocation register** is added to every address generated by a user process at the time it is sent to memory

- Relocation register also sometimes called a base register

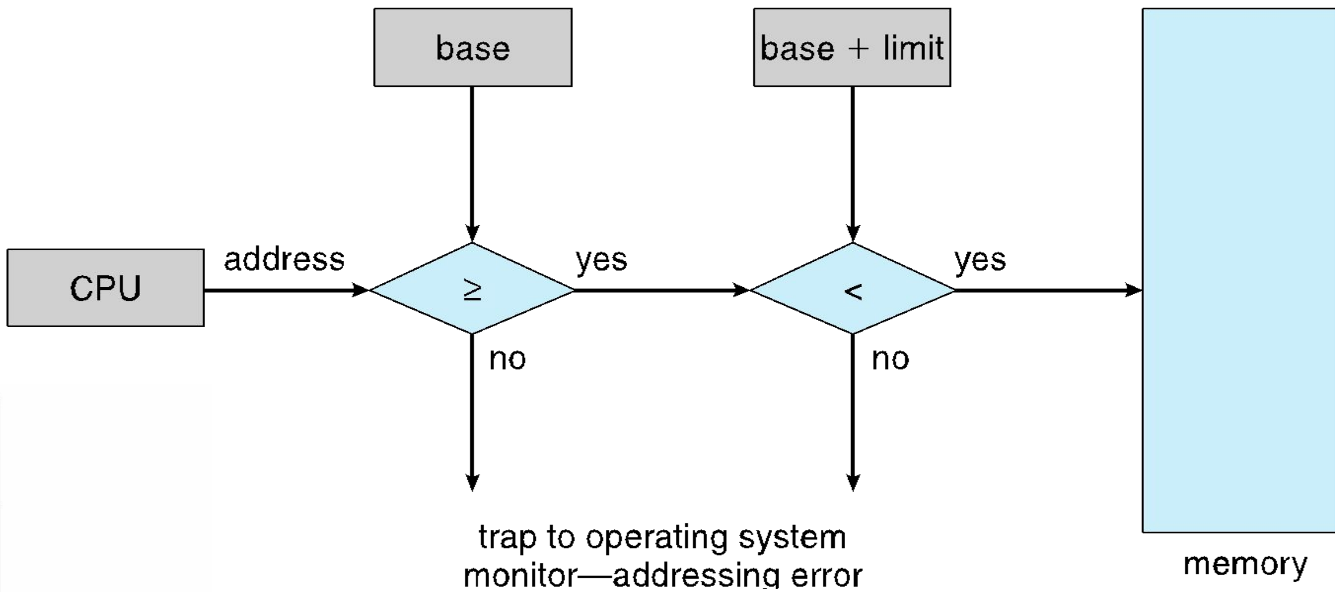


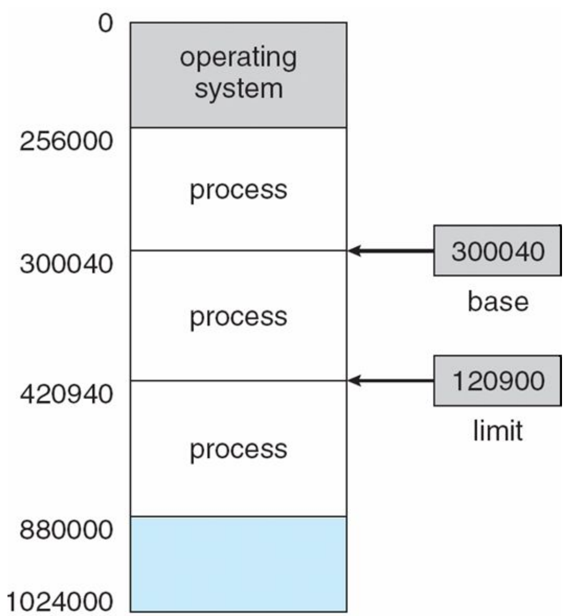
Base and Limit Registers

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



Hardware Address Protection





What a programmer wants from memory

- Memory addresses will start from 0

Satisfied:
Relocation register

- Memory will be contiguous
- I should be able to dynamically increase the memory
- I should have as much memory as I want

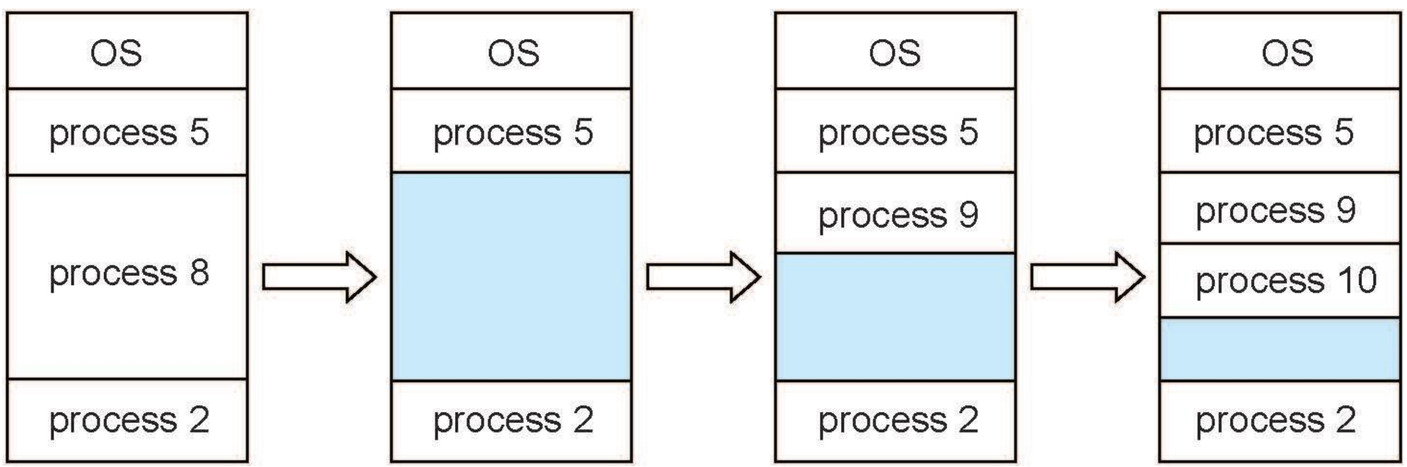
In systems where multiple programs (processes) are in memory at the same time, EACH program has the above expectations

Memory Management: Topics

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

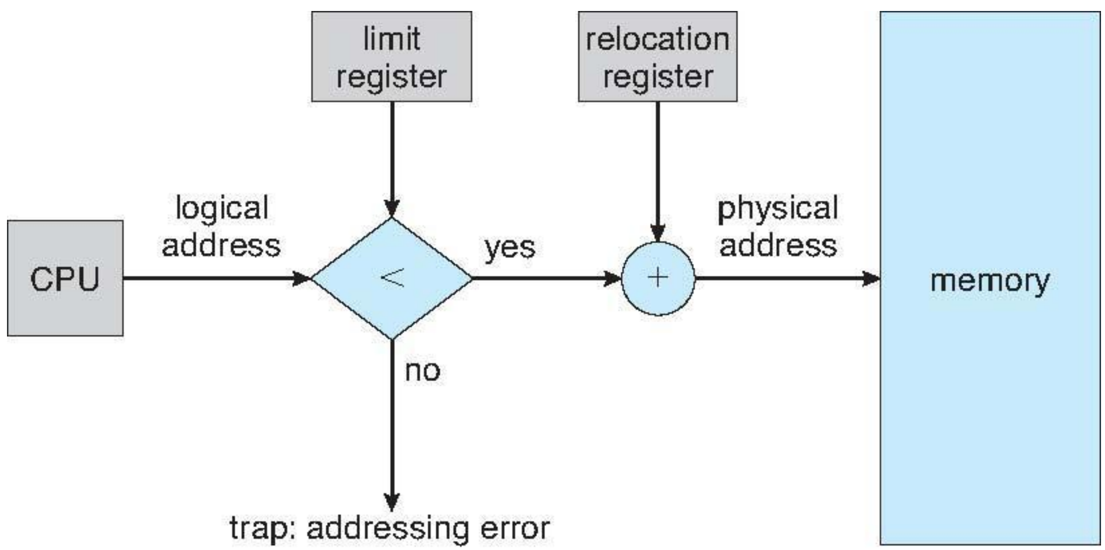
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- **Contiguous allocation** is one early method
- Main memory usually is split into two partitions:
  a) Resident OS, usually held in lower address portion of memory
  b) User processes then held in higher address portion of memory
- Each process contained in single contiguous section of memory



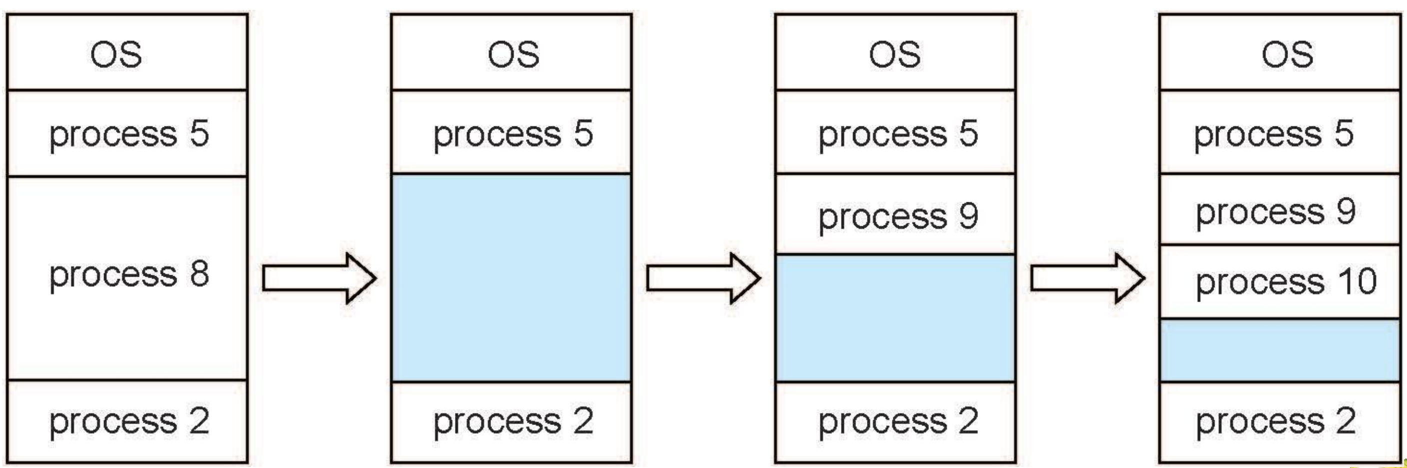
Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - **Base / relocation register** contains value of smallest physical address of a user process
  - **Limit register** contains range of logical addresses accessible by a user process – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Instructions to load these registers must be ***privileged***



Multiple-partition allocation

- Multiple-partition allocation
  - **Variable-size partitions** (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
  - Benefit: Produces the smallest leftover hole
  - Cost: Must search entire list, unless ordered by size
- **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

Fragmentation – a problem

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - Holes of sizes 10K, 15K and 25K exists
  - A process of size 35K arrives --- cannot be loaded

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - Allocated memory = 20K, but using only 18K
  - 2K wasted due to internal fragmentation

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory (i.e. holes) together in one large block
  - Compaction is usually very expensive
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time

- For the same reasons, dynamically increasing memory of a process is very expensive

What a programmer wants from memory

- Memory addresses will start from 0

Satisfied:
Relocation register

- Memory will be contiguous

Easy with contiguous allocation, but fragmentation, wastage of memory. Dynamic increase of mem impractical

- I should be able to dynamically increase the memory

- I should have as much memory as I want

In systems where multiple programs (processes) are in memory at the same time, EACH program has the above expectations

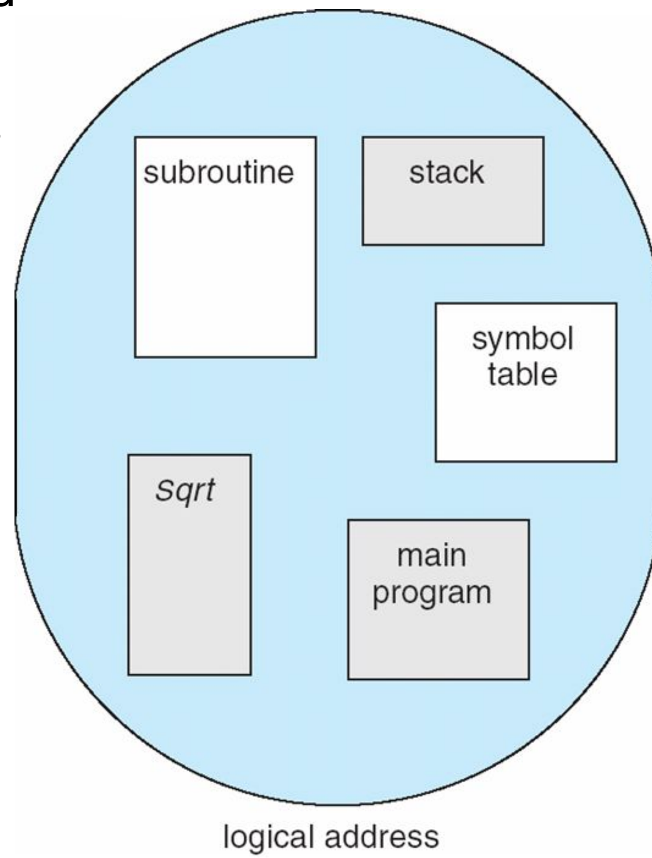
Memory Management: Topics

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

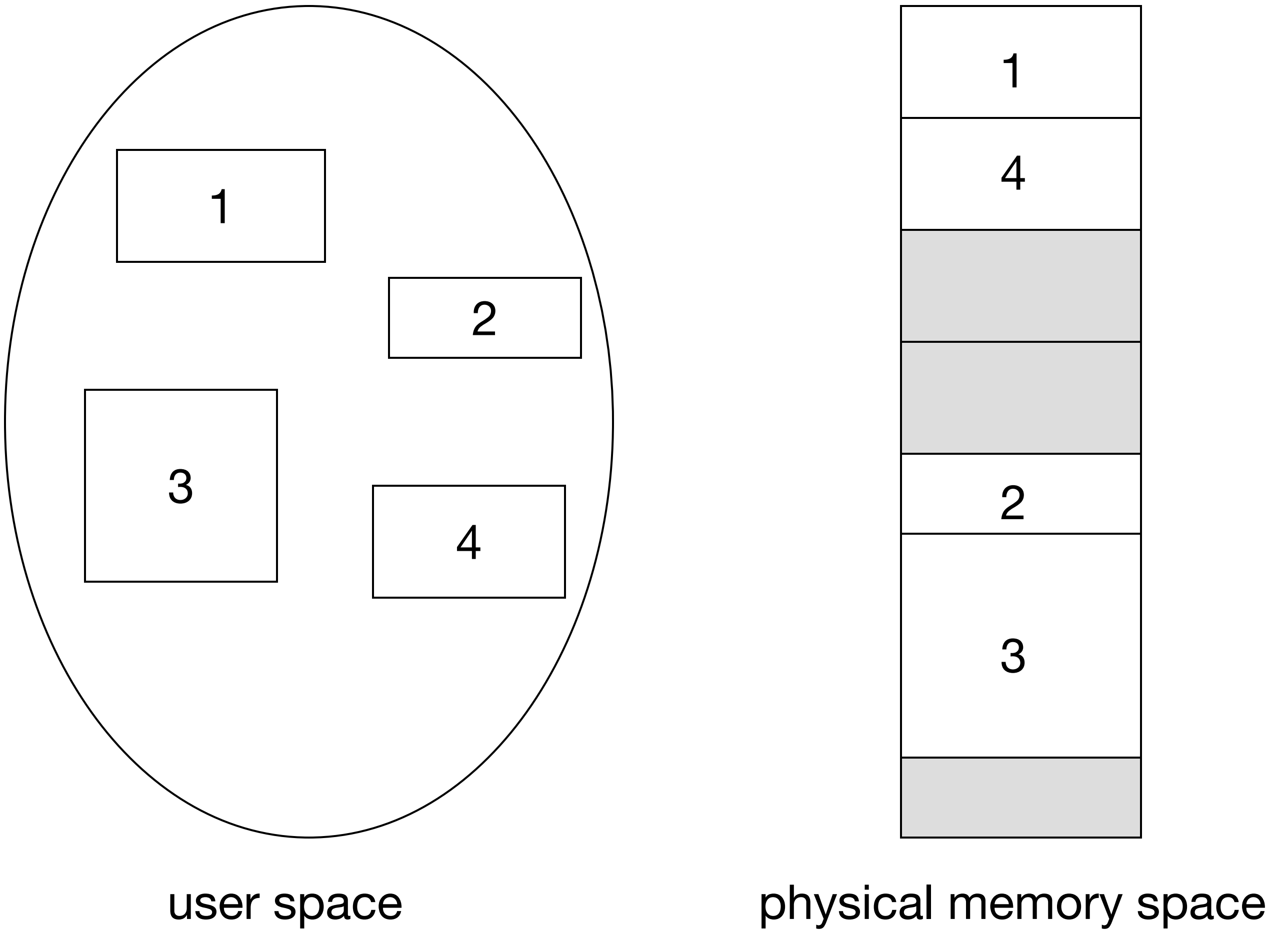
Segmentation

- Memory-management scheme that supports user-view of memory
- A program is a collection of segments, where a segment is a logical unit such as:
  - a function / procedure / method
  - an object
  - local variables, global variables
  - stack

User’s view of a program



Logical View of Segmentation



Note: we are shifting to non-contiguous memory allocation

Segmentation Architecture

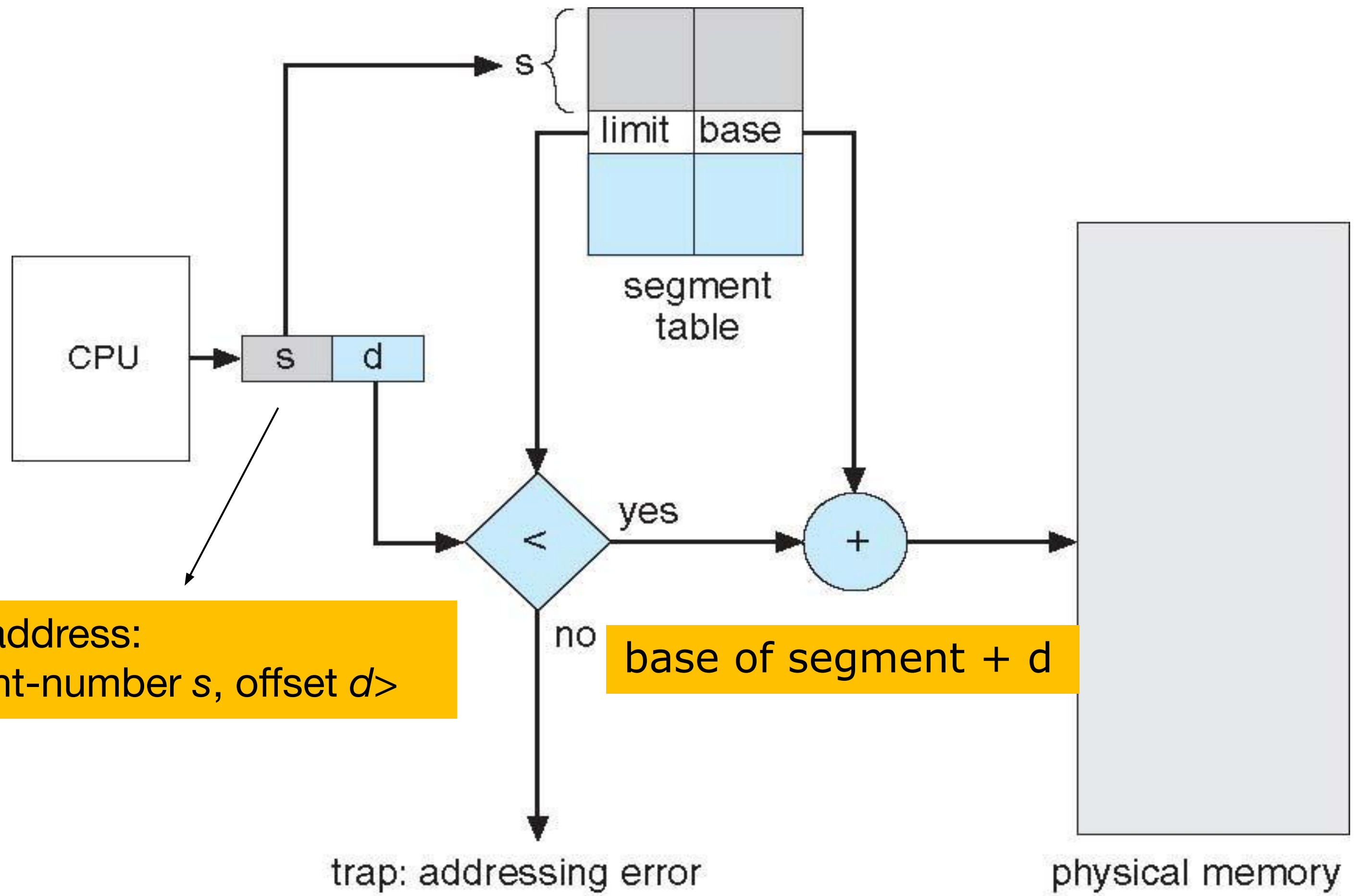
- Logical address consists of a two-tuple:

  < segment-number *s*, offset *d* >

- **Segment table** – used to map logical addresses to physical addresses

- Segment table has one entry for each segment of this process
- Each segment table entry has:
  - **base** – contains the starting physical address where the corresponding segment resides in memory
  - **limit** – specifies the length of the segment (*d* should be less than *limit*)
  - Other fields, such as protection bits, access privileges

Segmentation Hardware



Logical address:
$\langle$segment-number *s*, offset *d*$\rangle$

base of segment + d

Memory Management: Topics

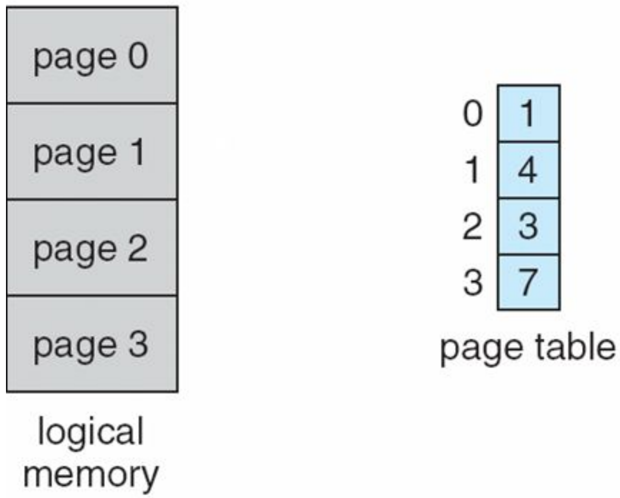
- Background
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

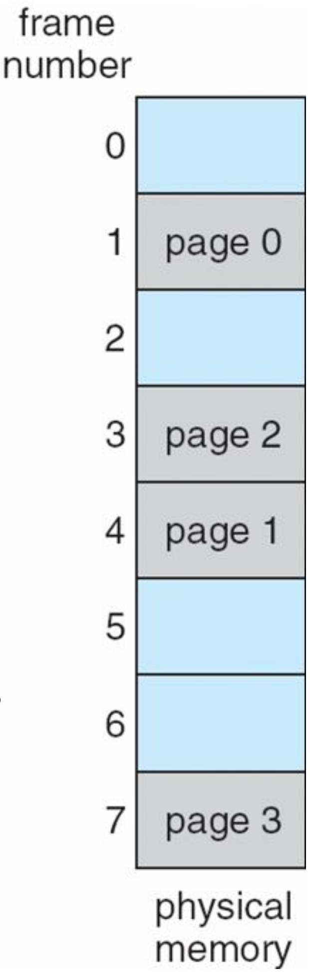
Paging

- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 MB
  - Typical size: 32KB, 4 MB
- Divide logical memory into blocks of same size called **pages**
- Physical address space of a process can be noncontiguous; allocated physical memory in units of a page wherever available (see next slide)

Paging Model of Logical and Physical Memory

Program's view: contiguous





Physically non-contiguous allocation

To run a program of size ***N*** pages, OS needs to find ***N*** free frames and load program (frames need not be contiguous)

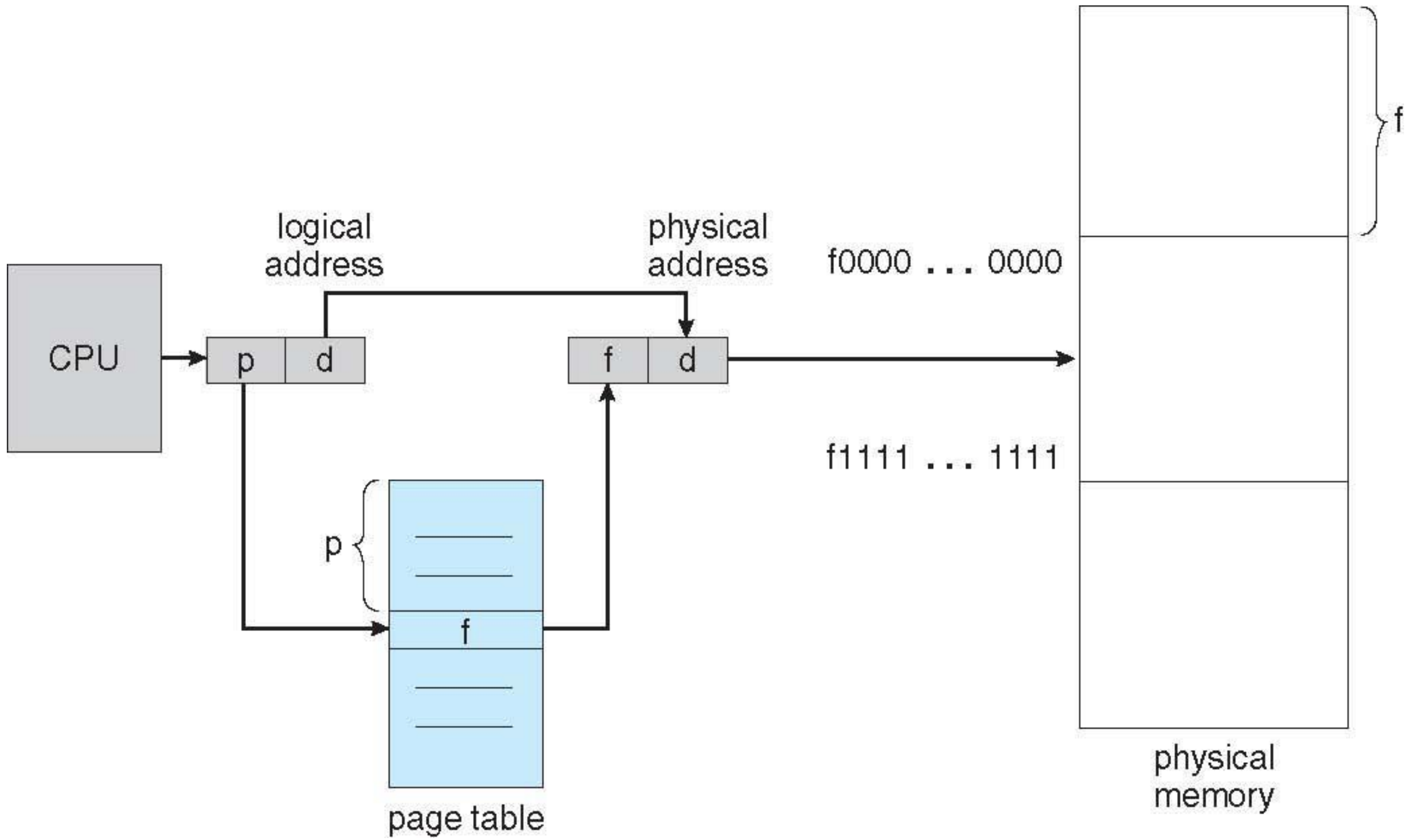
Set up a **page table** to translate logical to physical addresses

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

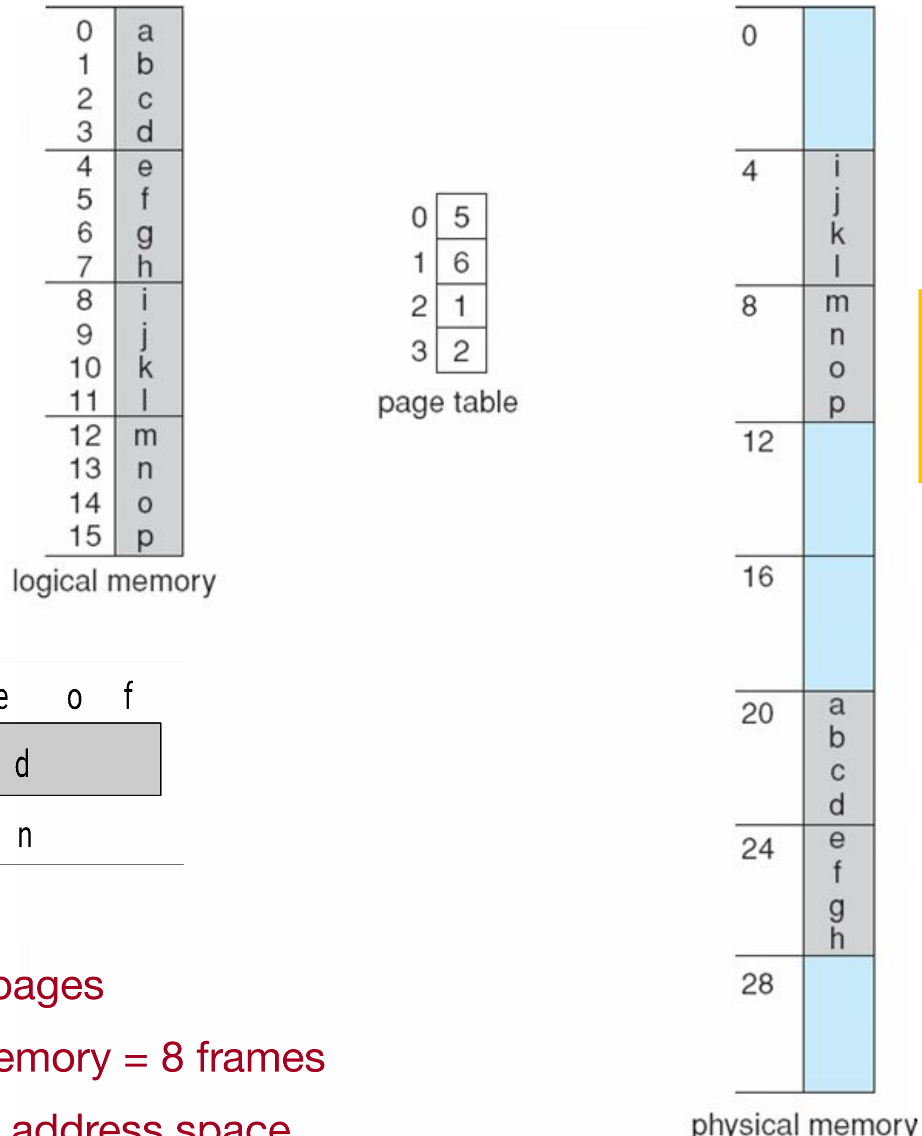
- For given logical address space 2^m and page size 2^n
  - Page offset is of n bits
  - Page number is $m - n$ bits

Paging Hardware

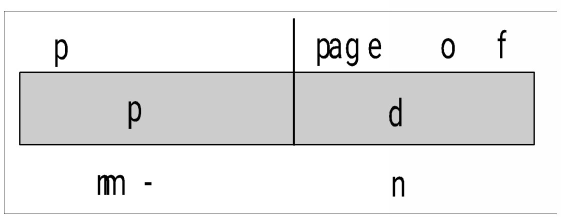


Paging Example

Program's view: contiguous



Physically non-contiguous allocation



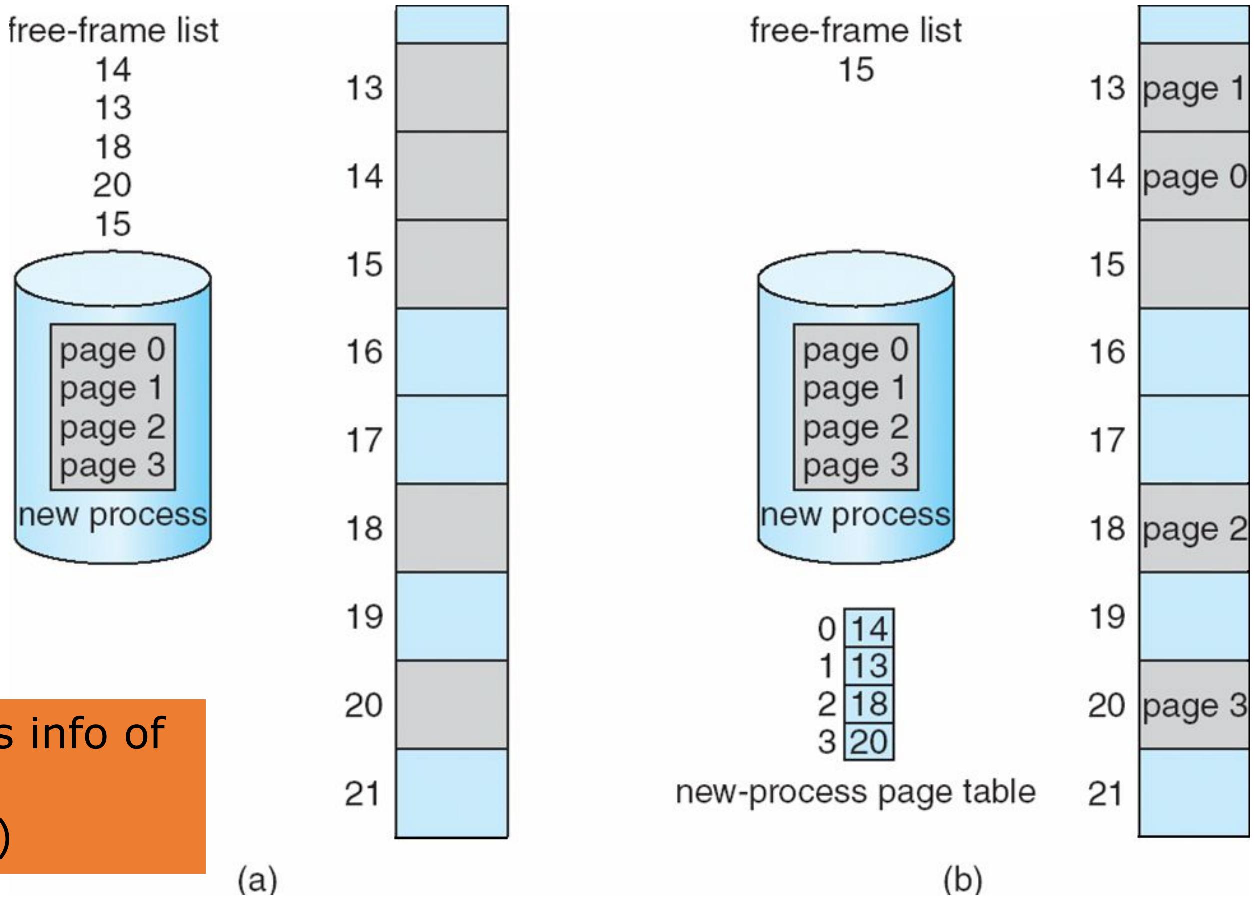
n=2 and *m*=4

n=2 means 4-byte pages

32-byte physical memory = 8 frames

2^4 = 16-byte logical address space

Free Frames



OS maintains info of free frames (frame table)

Before allocation

After allocation

How to decide page / frame size?

- Calculating internal fragmentation
  - Page size = 2048 bytes
  - Process size = 72,766 bytes (35 * 2048 + 1086)
  - 35 pages + 1086 bytes
  - Internal fragmentation of 2048 - 1086 = 962 bytes
- How to decide frame or page size?
  - Worst case fragmentation = 1 frame – 1 byte
  - Average fragmentation = 1/2 of frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track

Optimum Page Size

- Assume that:
  - Average process size = S
  - Page table entry size = K bytes
  - Page size = P bytes
- Average internal fragmentation per process = P / 2
- Average number of pages per process = S / P (actually ceiling)
- Thus, total overhead V = KS / P + P / 2
- To find the value of P that minimizes overhead, set dV/dP = 0
  - $- K S / P^2 + 1/2 = 0$
  - Thus, $P = \sqrt{(2 S K)}$

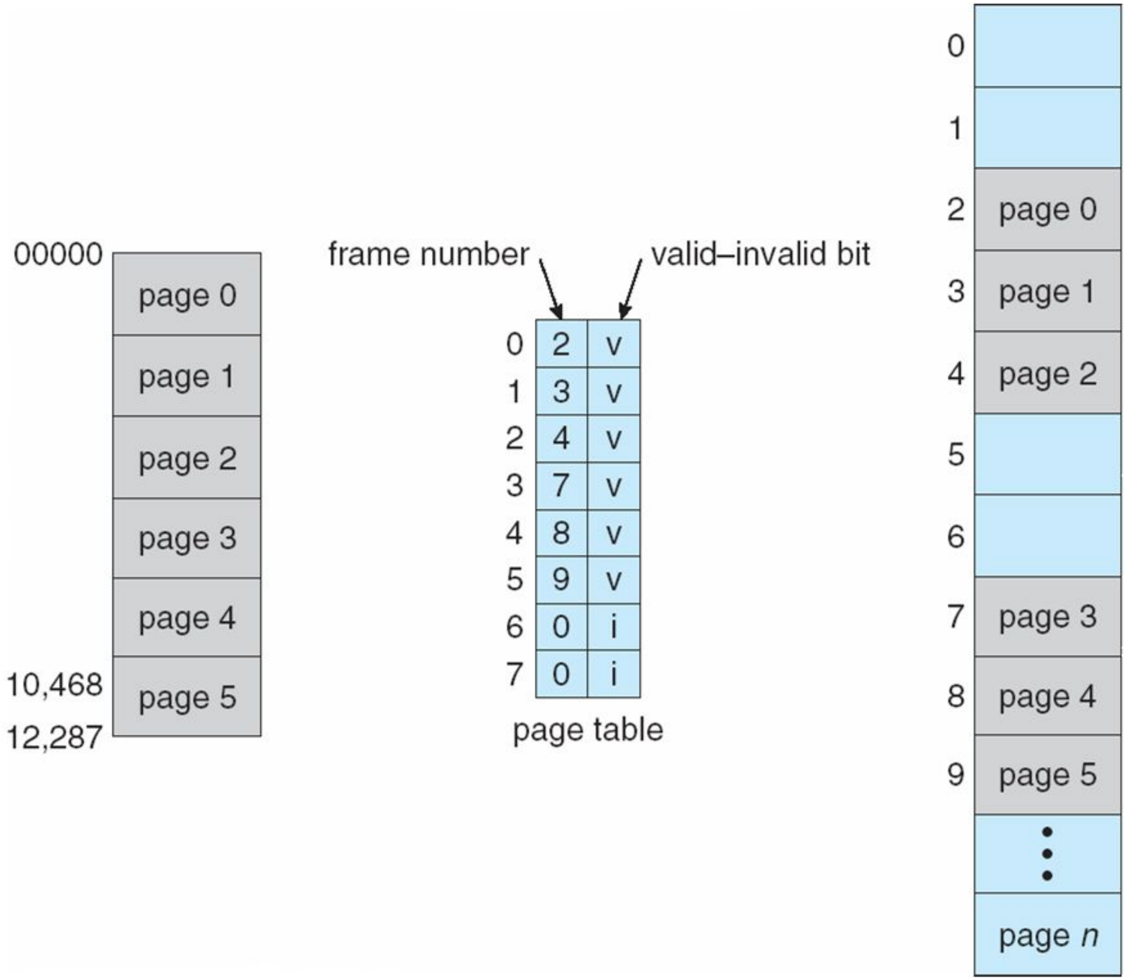
Small P implies less internal fragmentation

Large P implies smaller page table (lower overhead)

Memory Protection in Paging

- Memory protection implemented by associating protection bits with each page (these bits usually kept in the page table)
  - Indicates if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **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 (valid) page
  - “invalid” indicates that the page is not in the process’ logical address space
- Any violations result in a trap to the kernel

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



Advantages of Paging

- Clear separation between the programmer's view of memory and the actual physical memory

- Programmer's view of memory of his/her process
  - A single contiguous space, starting at (logical) address 0
  - Assume entire physical memory contains only this process

- Actual physical memory
  - Non-contiguous, scattered in pages across the physical memory
  - Occupies only a part of the physical memory

- Avoids external fragmentation, but internal fragmentation can still exist

Advantages of paging

- **Shared code**
  - One copy of read-only code shared among multiple processes (i.e., multiple processes running a text editor during a lab class)
  - Also useful for inter-process communication if sharing of read-write pages is allowed

- **Private code and data**
  - Each process might also keep its own copy of private code / data
  - The pages for the private code / data can appear anywhere in the logical address space

- **Dynamic increase of memory**
  - If a process needs more memory, can allocate an empty frame as an additional page

What a programmer wants from memory

- Memory addresses will start from 0
- Memory will be contiguous

First 3 requirements satisfied with paging

- I should be able to dynamically increase the memory
- I should have as much memory as I want

Need to satisfy requirements for large memory

In systems where multiple programs (processes) are in memory at the same time, EACH program has the above expectations

Memory Management: Topics

- Background
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table (and variants of the page table) – dealing with large Page Tables efficiently

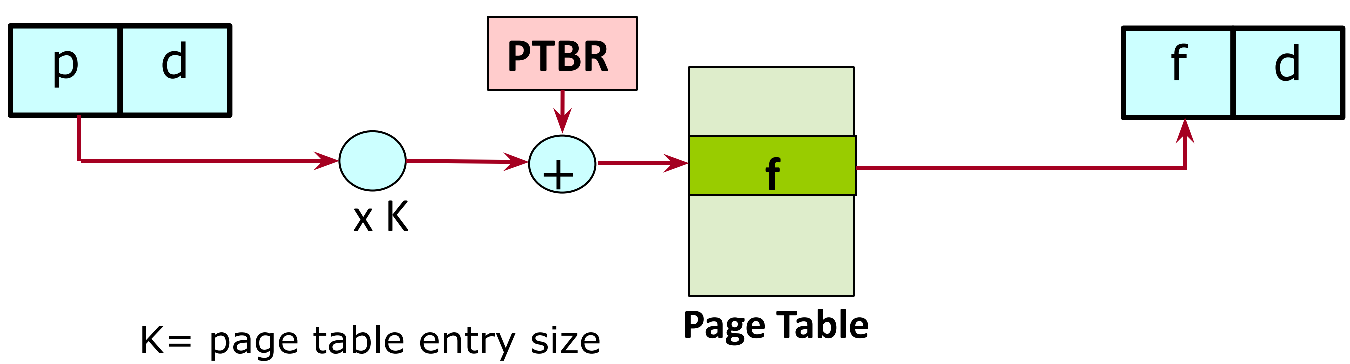
Implementation of Page Table

- Page Table has to be accessed for every memory access by a process
  - Implementation should allow fast access
- Where to store the Page Table?
- Best case – a set of dedicated registers
  - Only feasible if the page table is small
  - Not practical in most modern systems
- Practical case – Page Table stored in the memory

Implementation of Page Table

- In practice, page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table

- In this scheme every data/instruction access requires two memory accesses
  - One for the page table entry, and one for the actual data/instruction
  - Too expensive



Implementation of Page Table (Cont.)

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

- TLBs typically small (between 32 and 1,024 entries)
  - Contains a few of the (most frequently/recently accessed) page table entries
  - For a logical address generated by CPU, its page number is searched in TLB; If page number is found (TLB hit), its frame number is immediately available to access the data directly in memory

- On a TLB miss, a memory reference to the page table needed
  - 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

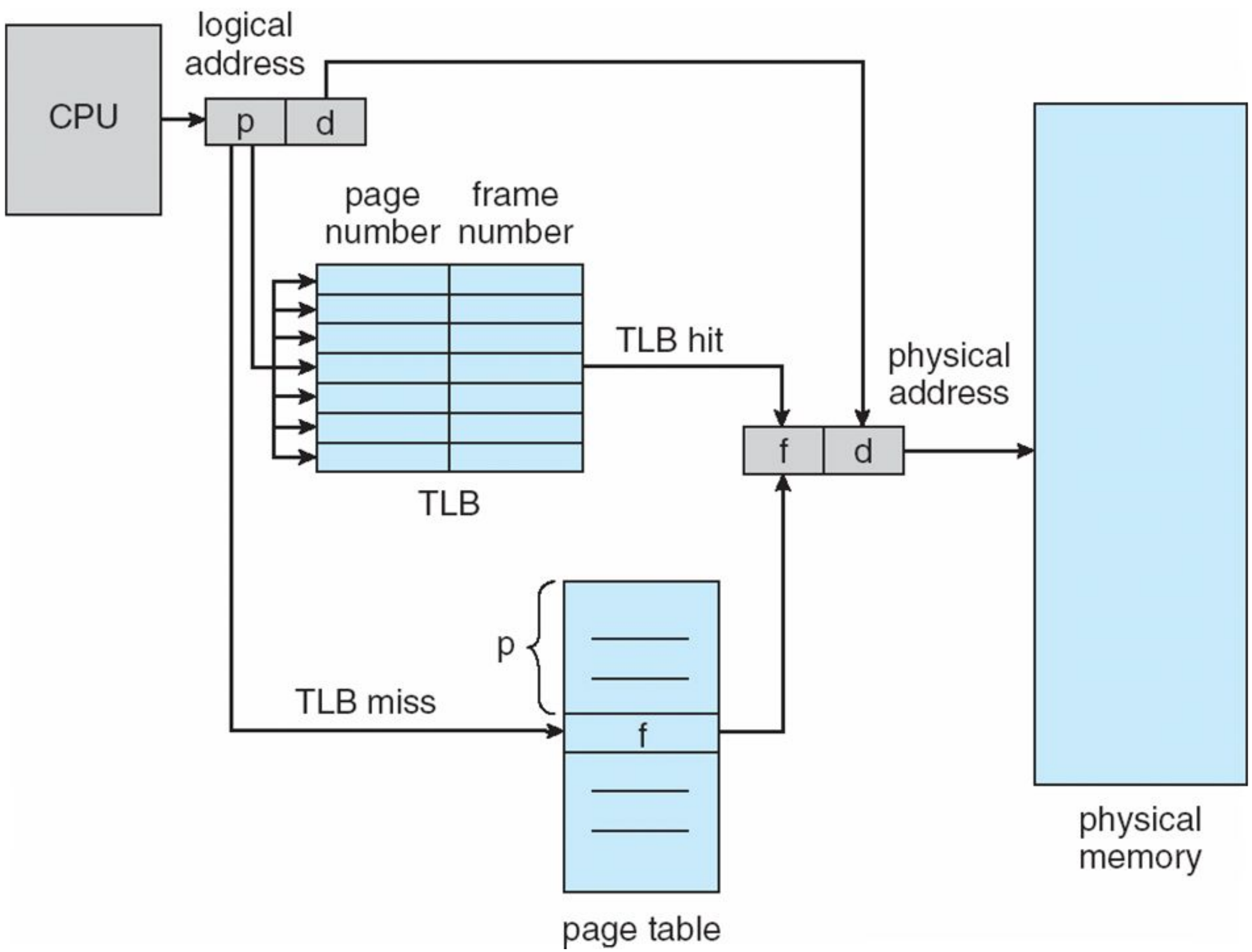
Associative Memory

- TLB is associative memory – allows parallel search (item to be searched is compared with all keys simultaneously)

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

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α (% of times that a page number is found in TLB)

- **Effective Access Time (EAT)**

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

  Even larger for multi-level page tables

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

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: 2^{32} byte memory locations
  - Page size of 4 KB (4x 2^{10} byte = 2^{12} byte)
  - Page table would have ~1 million entries (2^{32} / 2^{12})
  - If each page table entry is 4 bytes

    * 4 MB (2^{20}x4 byte) of physical address space / memory for page table alone – equivalent to 1024 pages of 4KB each

    * Don't want to allocate that contiguously in main memory

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: 2^{32} byte memory locations
  - Page size of 4 KB (4x 2^{10} byte = 2^{12} byte)
  - Page table would have ~1 million entries (2^{32} / 2^{12})
  - If each page table entry is 4 bytes

    * 4 MB (2^{20}x4 byte) of physical address space / memory for page table alone – equivalent to 1024 pages of 4KB each

    * Don't want to allocate that contiguously in main memory

We moved to paging because we wanted to avoid allocating large blocks of contiguous memory

But we are back to the same problem – as logical address space increases, page tables need large memory blocks

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: 2^{32} byte memory locations
  - Page size of 4 KB (4x 2^{10} byte = 2^{12} byte)
  - Page table would have ~1 million entries (2^{32} / 2^{12})
  - If each page table entry is 4 bytes

    * 4 MB (2^{20}x4 byte) of physical address space / memory for page table alone – equivalent to 1024 pages of 4KB each

    * Don’t want to allocate that contiguously in main memory

- **Solutions**
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

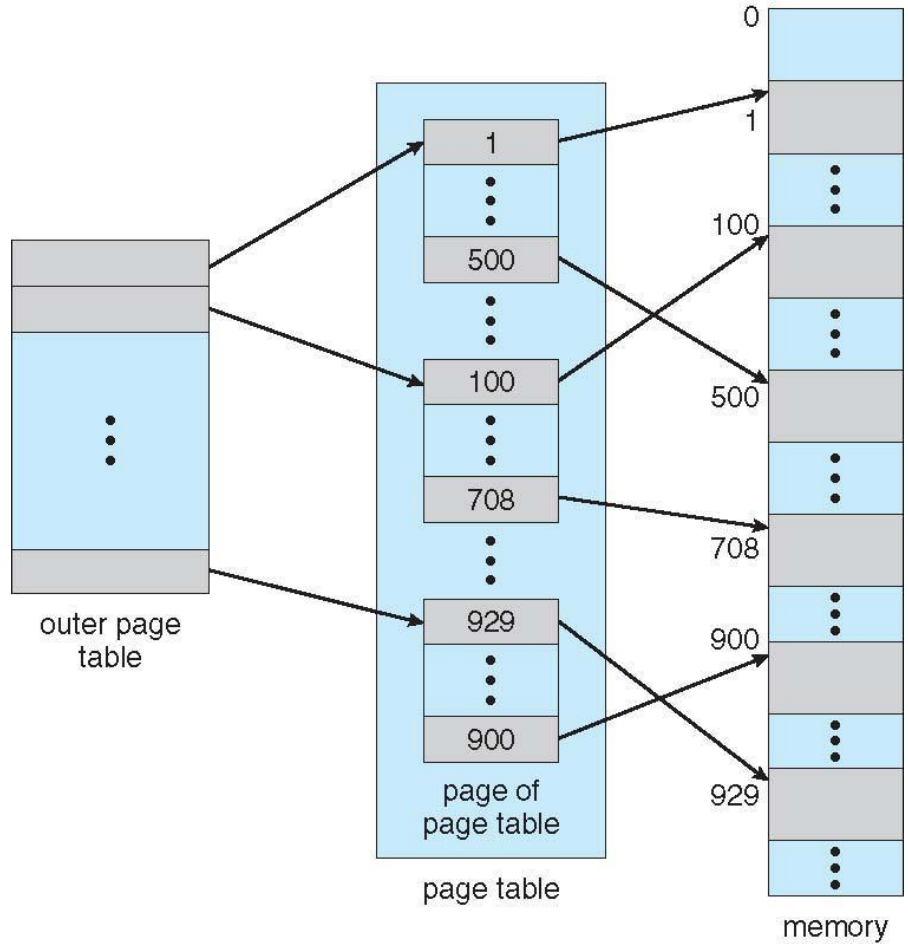
Hierarchical Page Tables

- 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**

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 (POPT: pages of page table)**

Two-Level Page-Table Scheme



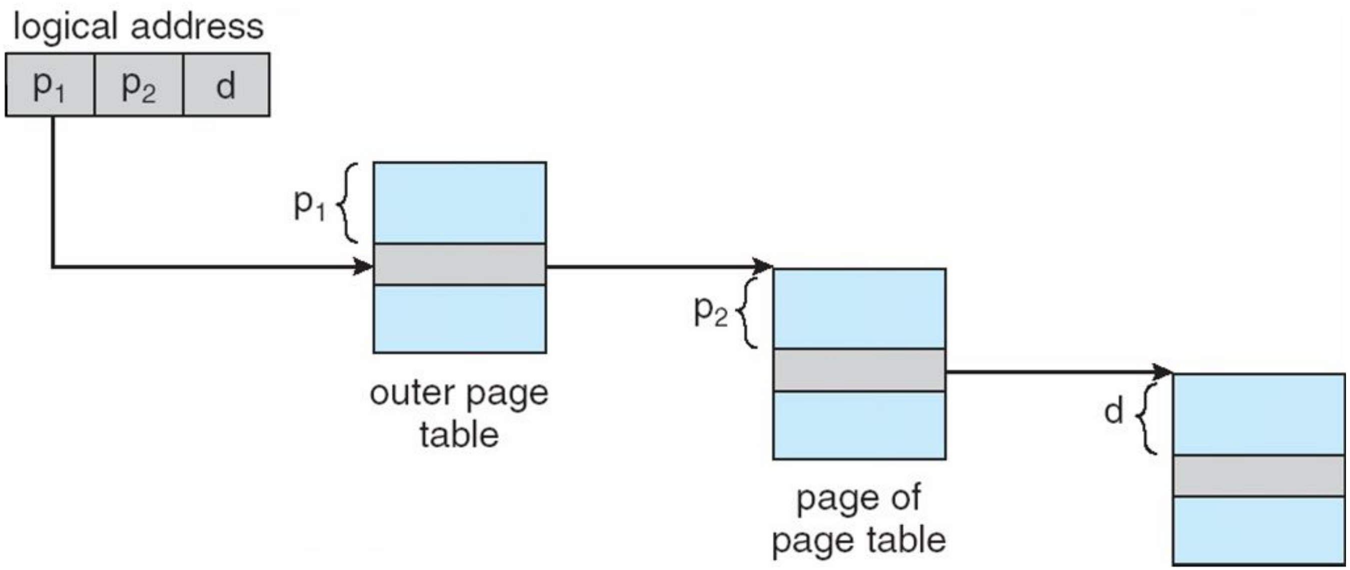
Two-Level Paging Example

- 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:

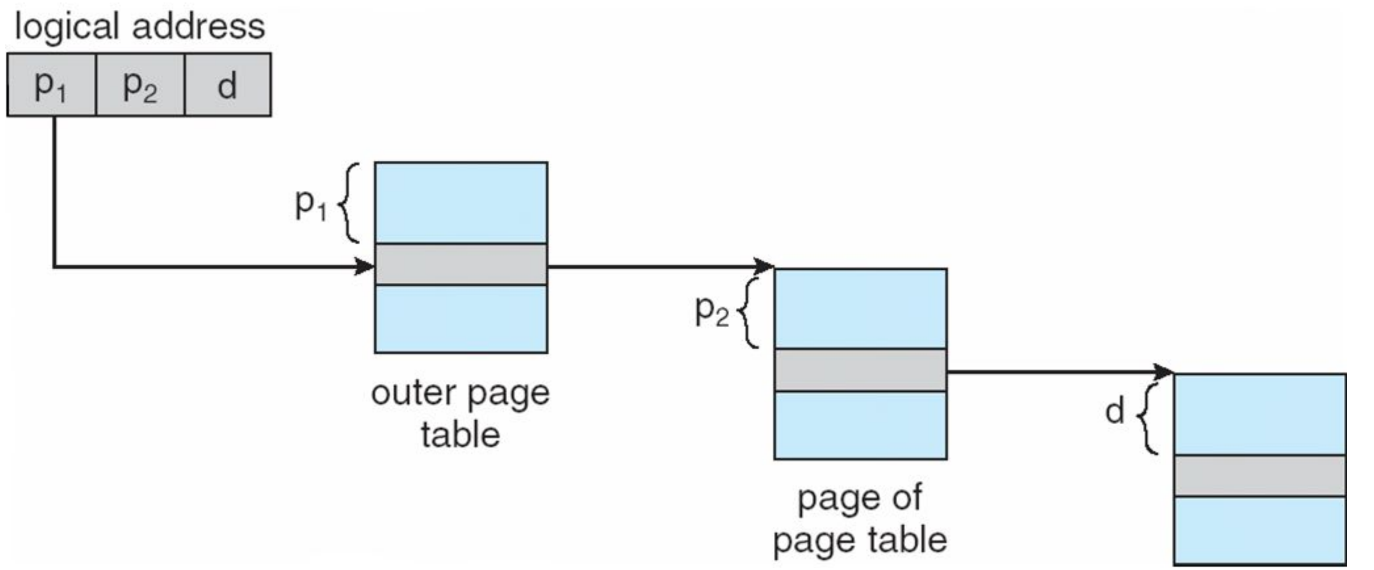
page number		page offset
p_1	p_2	d
12	10	10

- 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

Address-Translation Scheme



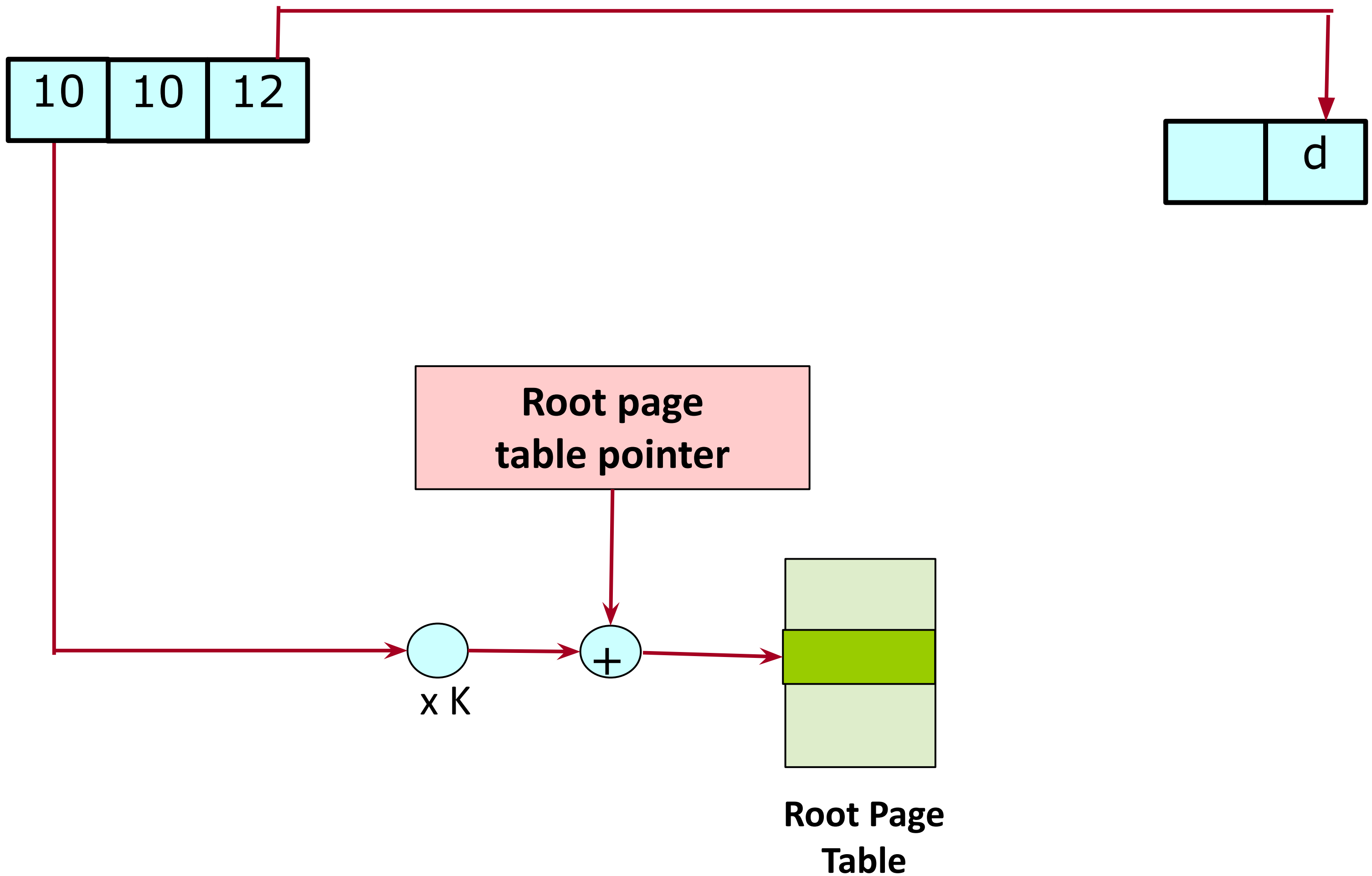
Address-Translation Scheme



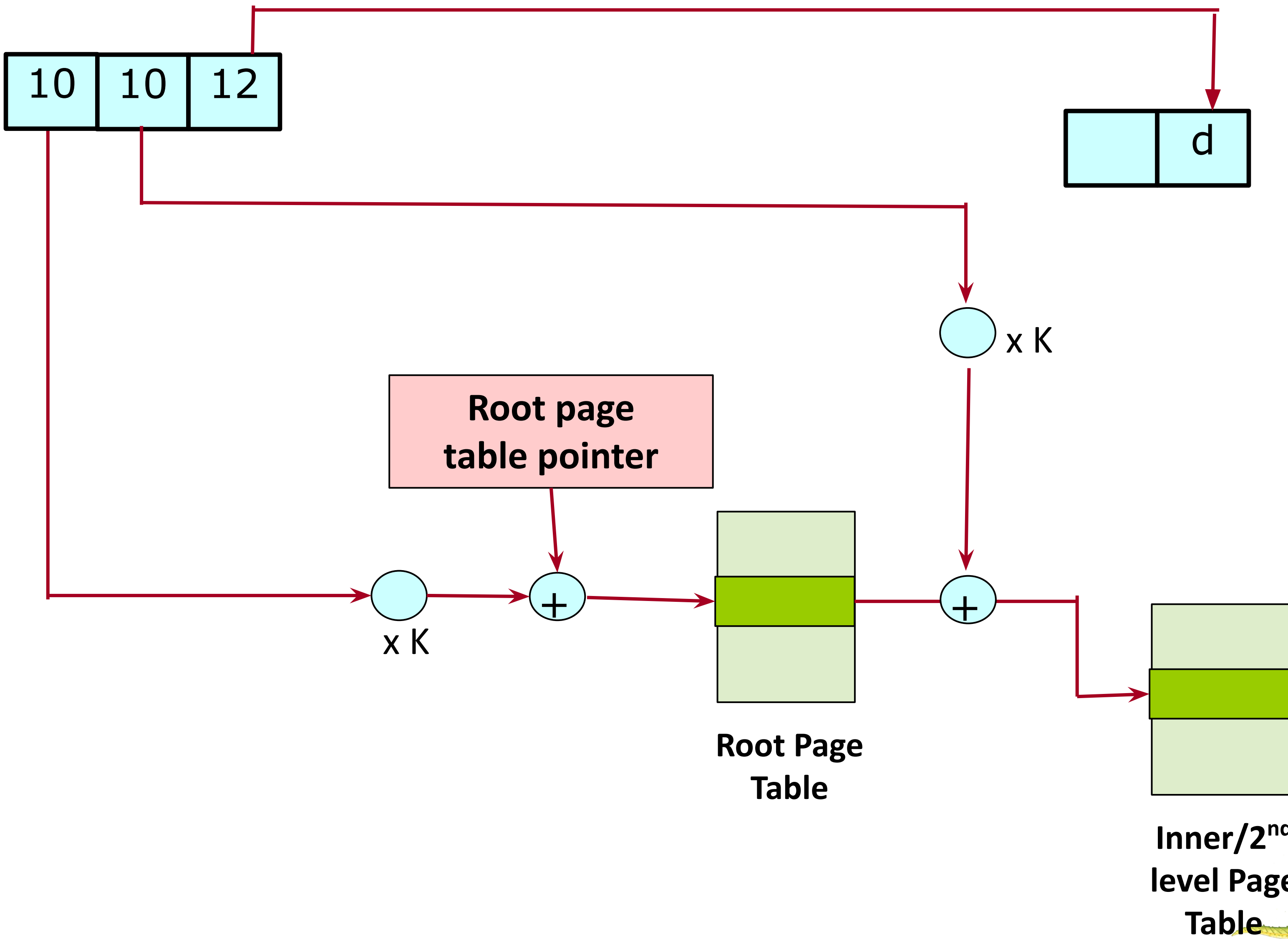
Cost: Three memory Accesses
(previously two)

Implementation of 2-level paging

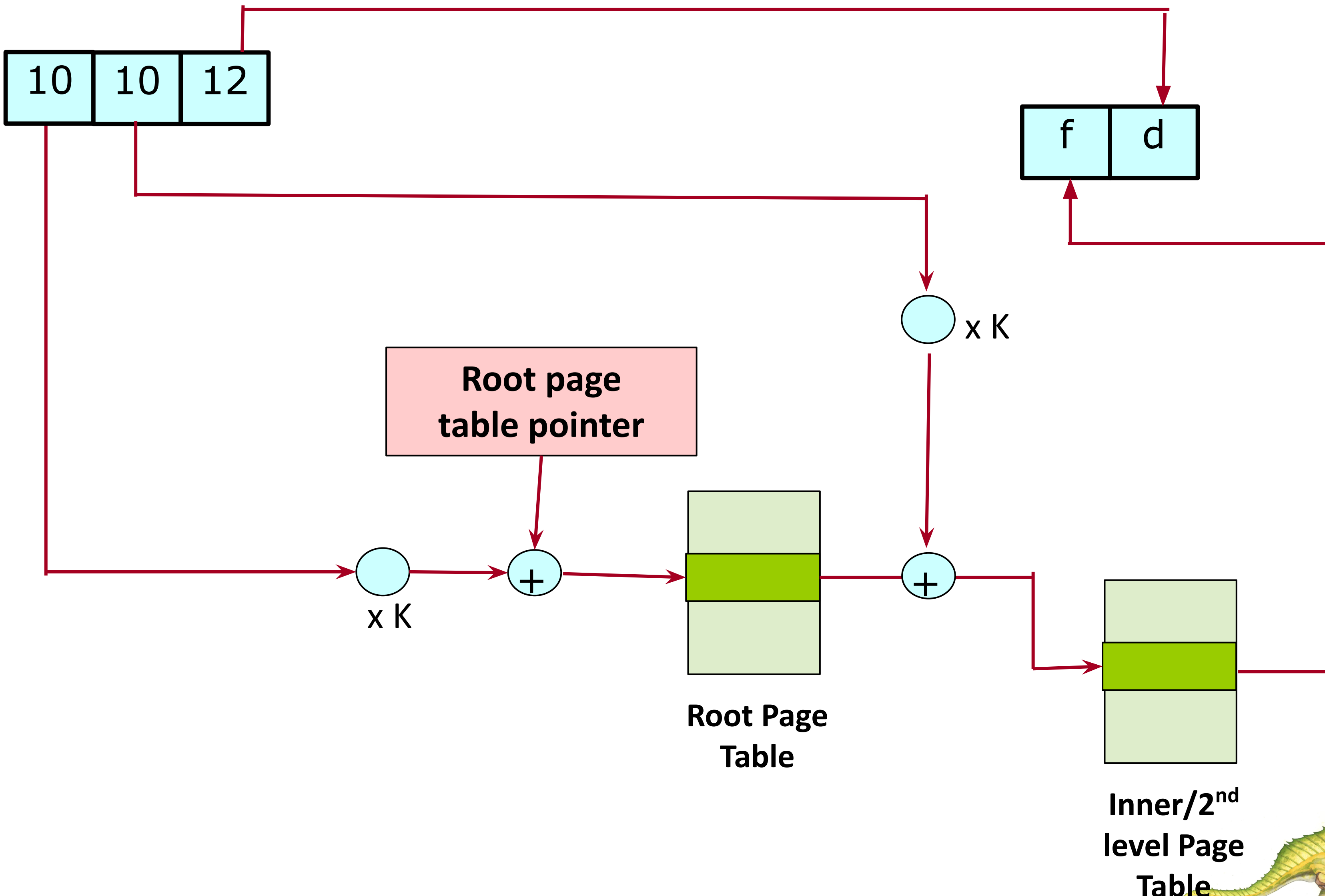
Implementation of 2-level paging



Implementation of 2-level paging



Implementation of 2-level paging



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 2-level scheme, inner/2nd level 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 = **2^{14} GB**

    Won't work!

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

- One solution is to add a 2nd outer page table

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

- One solution is to add a 2nd outer page table

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

- One solution is to add a 2nd outer page table

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

- But in this example the 2nd outer page table is still 2^{34} bytes in size
  - Even 16 GB is too huge to store contiguously
  - Also 4 memory access to get to one physical memory location

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

- One solution is to add a 2nd outer page table

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

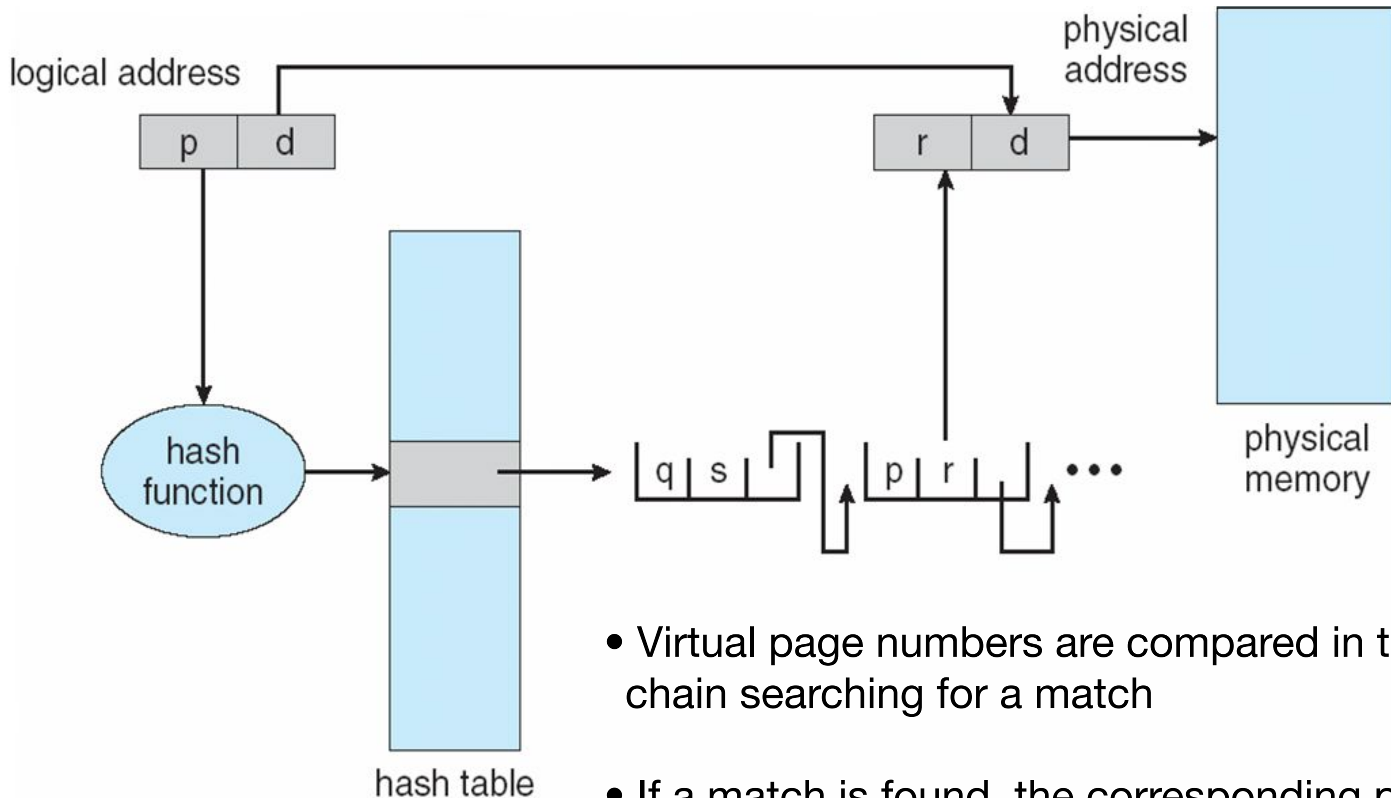
- But in this example the 2nd outer page table is still 2^{34} bytes in size
  - * Even 16 GB is too huge to store contiguously
  - * Also 4 memory access to get to one physical memory location

Multi-level paging schemes do not scale well
Need alternate mechanisms

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 physical page frame
  - (3) a pointer to the next element

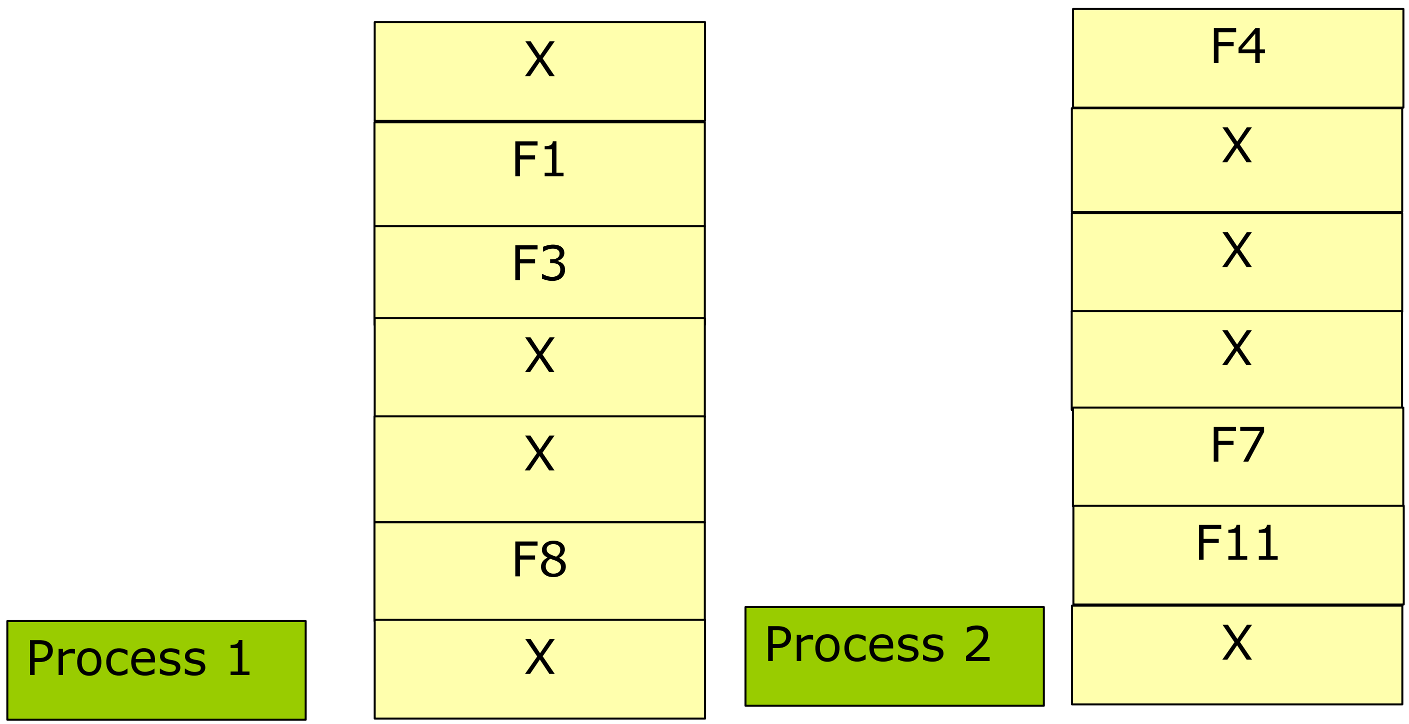
Hashed Page Table



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

Inverted Page Table: Background

- Till now what we have discussed:
  - Each process has a page table that keeps track of all logical pages of this process

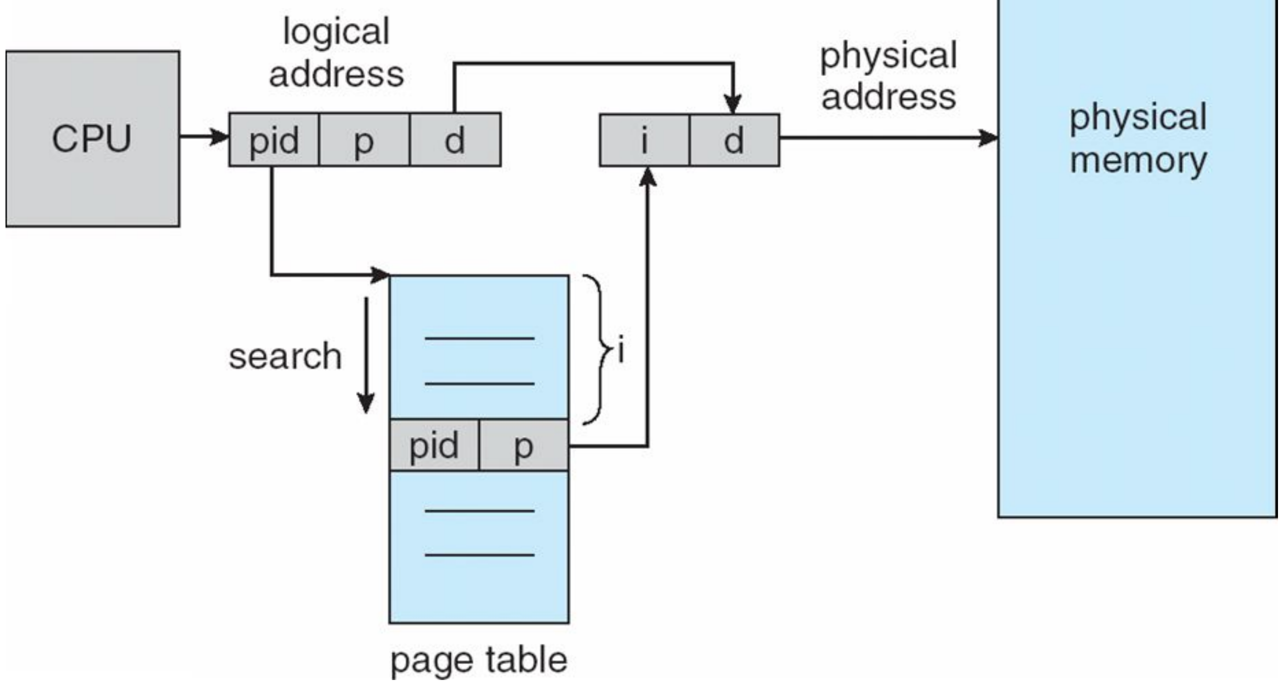


Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages (frames)
- Inverted page table: One entry for each real page (frame) of memory
  - Entry = the virtual address of the page stored in that real memory location + information of the process that owns the page

Inverted Page Table Architecture

A logical address is a triple:
<process-id, page-number, offset>



Inverted Page Table

- Decreases memory needed to store each page table, but increases time needed to search the table for a page reference
  - Searching uses the CPU-generated virtual address, but the inverted page table is sorted by the physical address

- Can make searching physical address relatively fast:
  - Use hash table (as described earlier)
  - TLB can accelerate access further (even before consulting hash table)

Summary

- Contiguous allocation with Relocation register
- Paging – avoid external fragmentation, process sees contiguous memory space, but physically non-contiguous

- Where to store the Page Table? In the memory
  - Cost – multiple memory accesses for each data access
  - Optimization for faster access – use TLB (associative cache)

- Structure of the Page Table in memory
  - Page Table can be too large for contiguous memory allocation
  - Hierarchical multi-level Page Table
    - Store pages of page table (POPT)
    - Does not scale well as logical address space becomes larger
  - Hashed Page Table
  - Inverted Page Table

What a programmer wants from memory

- Memory addresses will start from 0
- Memory will be contiguous
- I should be able to dynamically increase the memory
- I should have as much memory as I want

First 3 requirements satisfied with paging. Also, we now have ways of allowing huge memory to processes.

I may even want more memory than the physical mem size. I do not want to be limited to the physical memory size on a particular system.

Virtual memory – to be discussed next