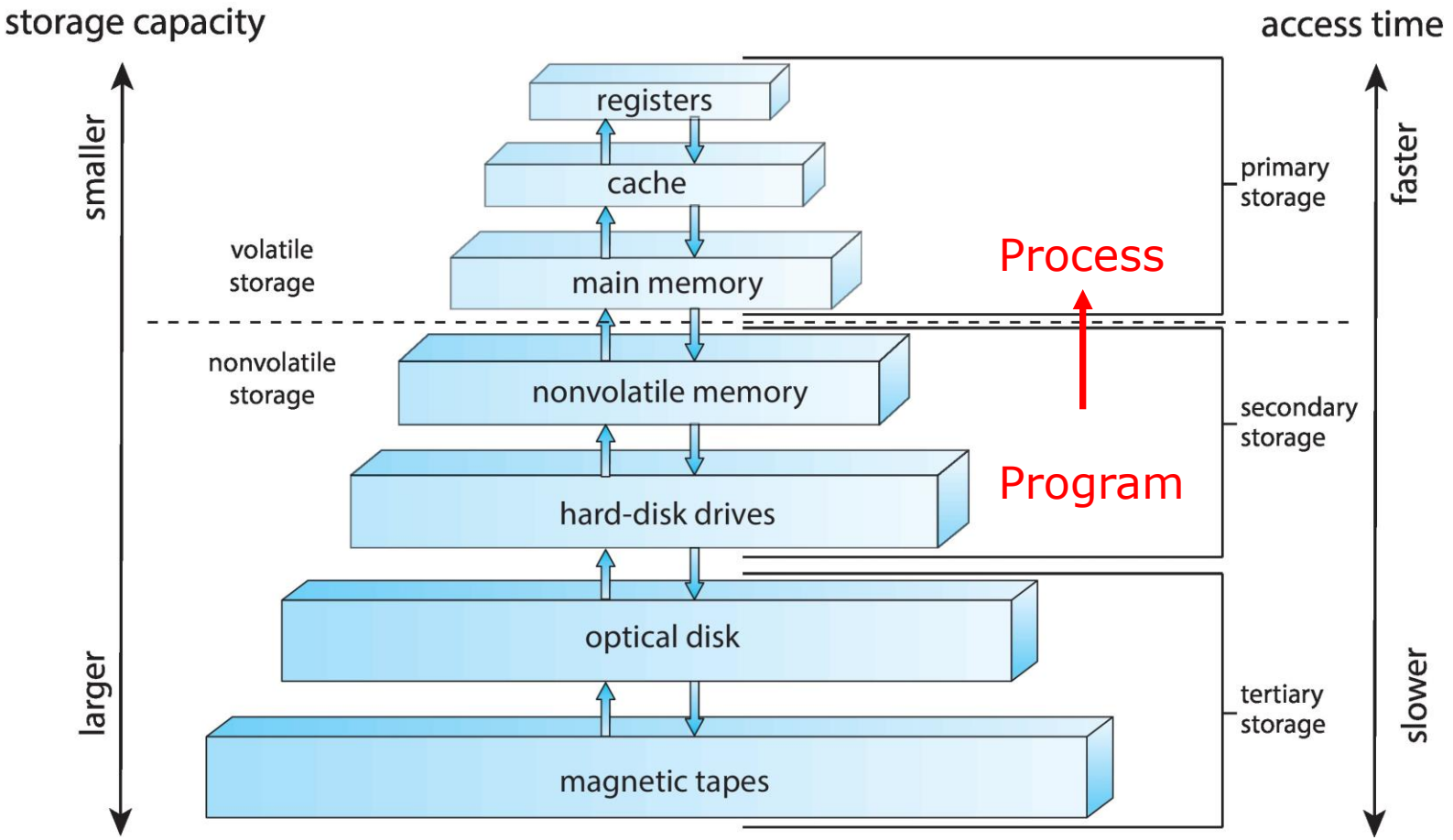
Chapter 9: Main Memory

Outline

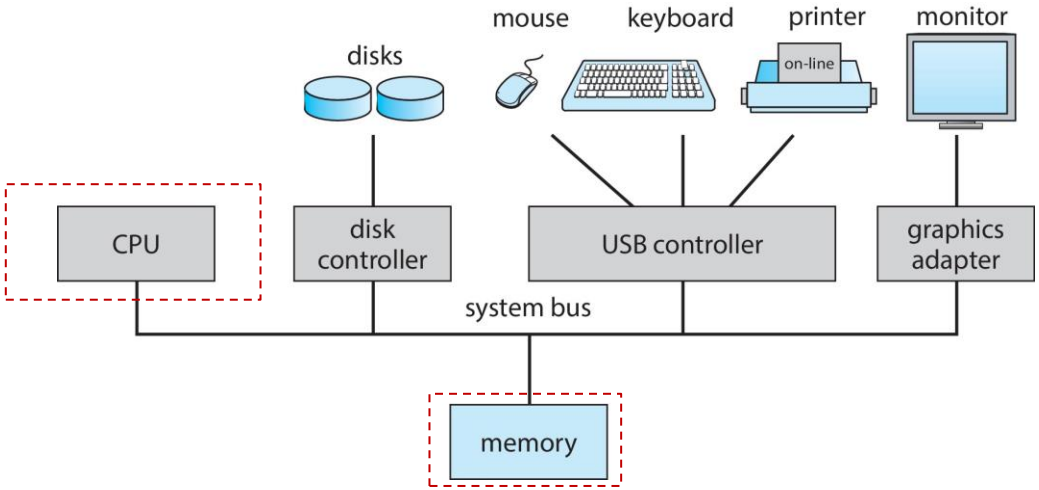
- Background
- Memory Allocation
  - Contiguous Memory Allocation
  - Frame
    - Page Table
    - Address Translation
  - Page Table Schema
    - Hierarchical Page Table
    - Hash Table
    - Inverted Table
  - Examples

Review: Storage Hierarchy



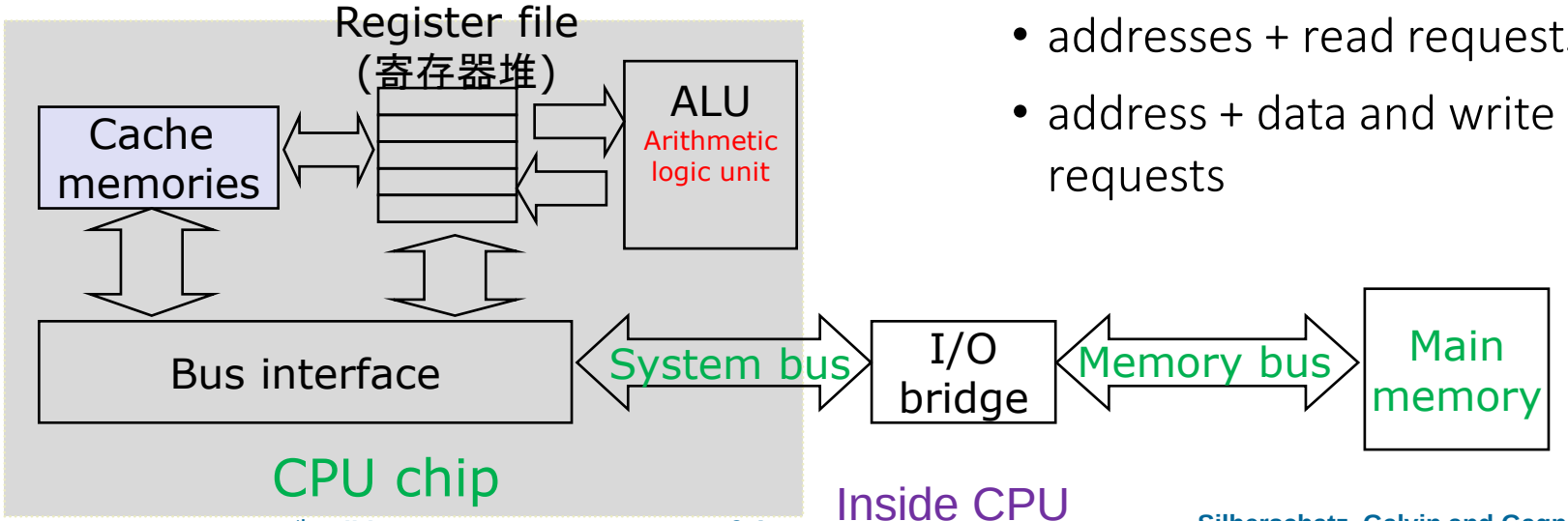
Then, how CPU accesses the process in the memory?

Relationship of CPU and Memory



A PC System Structure

- Main memory and registers are only storage that CPU can access directly
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a stall (delay)
- Memory unit only sees from CPU a stream of:
  - addresses + read requests, or
  - address + data and write requests



Inside CPU

Protection

CPU notifies memory where to read or write

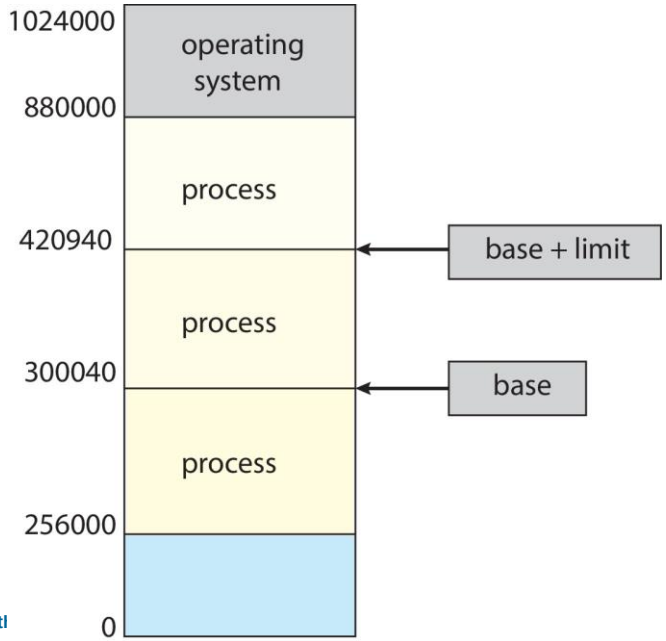
Is this “where” reliable?

Will the instructions go to other process’s land?

Protection is needed

An old but simple method

- a pair of base register and limit register for each process to address of the process

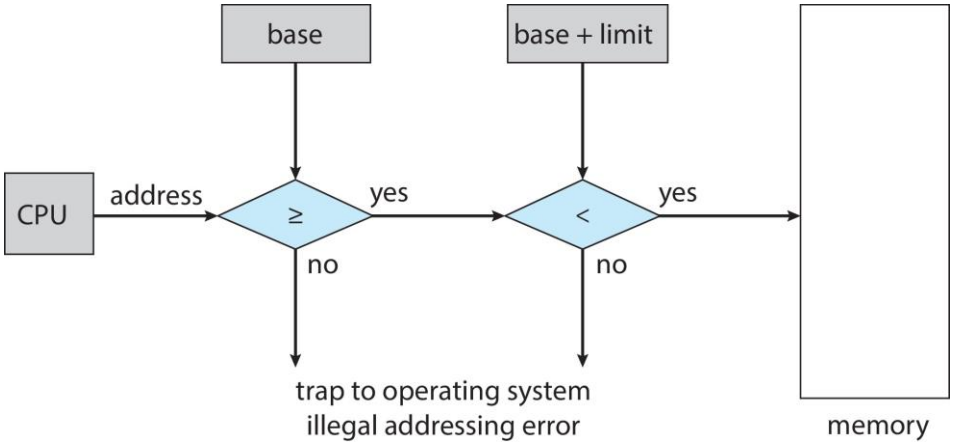


Hardware Address Protection

Two registers are needed

- Base register
- Limit register

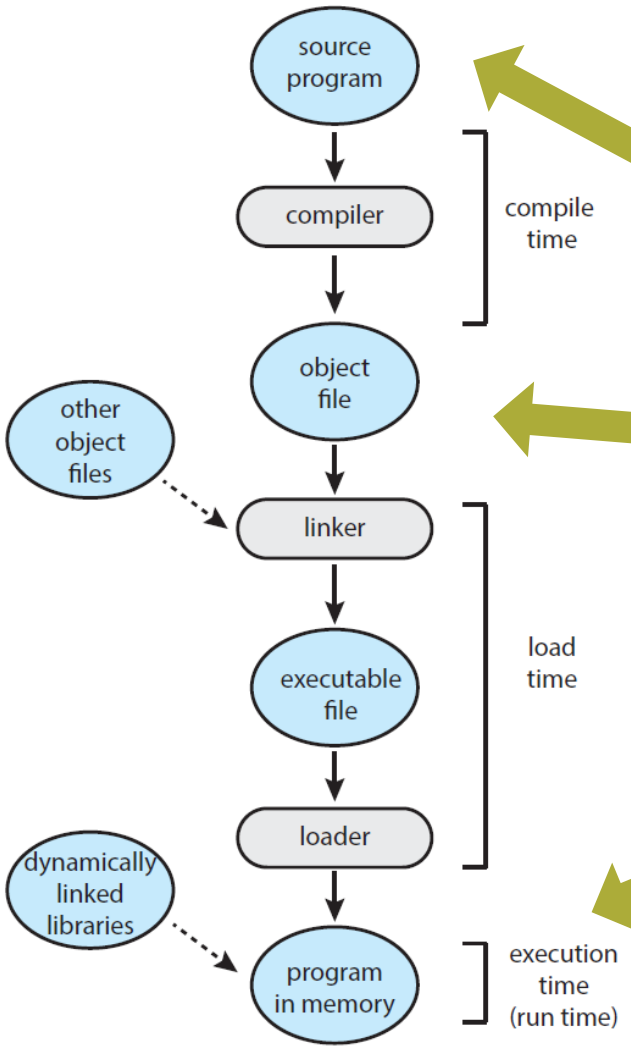
CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



The instructions to load base and limit registers are privileged

- No process can spy on another one by changing its own base register to be the same as other's.

Address Binding in A Typical Multistep Processing of a User Program



How are process addresses determined in memory?

Addresses are represented in different ways at different stages of a program's life

- Source code addresses usually symbolic
  - i.e. int count; &count
- Compiled code addresses bind to relocatable addresses (addresses relative to the start of the code).
  - i.e. "14 bytes from beginning of this code module"
- Linker or loader will bind relocatable addresses to absolute addresses (addresses relative to the lowest address 0000 in memory).
  - i.e. 74014

Each binding maps one address space to another address space

Linking: Dynamic vs Static

Static linking –system libraries and program code are combined by the loader into the binary image

- Every program includes library: wastes memory

Dynamic linking –linking is postponed until execution/run time

- Operating system checks if routine (子程序) is in process memory address, If not in address space, add to address space
- Small piece of code, stub (存根), is used to locate the appropriate memory -resident library routine
  - Stub replaces itself with the address of the routine and executes the routine
- Particularly useful for libraries which are used by many processes
  - Keep only one copy of the library in memory which is shared by all the processes.
  - known as shared libraries (.dll on Windows and .so on Linux, and .a is Linux static library)

Binding of Instructions and Data to Memory

Types of systems determine when the binding to memory address happens

Compile time: Absolute address is used in code

- Example: embedded systems with no kernel.
- Disadvantage: code must be re-compiled if starting location in memory changes

Load time: Relocatable code is generated if memory location is not known at compile time

- Example: very old multiprogramming systems
- Program's code has a relocation table of relative addresses of all the things in the code
- The real memory address is calculated according to the given location.
- Disadvantage: the address in memory is fixed after the program is loaded into memory.

Binding of Instructions and Data to Memory

Types of systems determine when the binding to memory address happens

Execution time: Binding is delayed until run time.

- Example : most computers today
- A process can be moved during its execution from one memory segment to another
- Need hardware support for address maps
  - e.g., MMU

This lecture talks about the binding in execution time.

Logical vs. Physical Address Space

Memory management is to

bind a logical address space to a separate physical address space

- Logical address
  - generated by the Program/CPU
  - also referred to as virtual address
  - all the addresses used by the CPU
  - Addressed in bytes
  - Size of space depends on the CPU bits
- Physical address
  - all the addresses used by the memory hardware
  - Addressed in bytes
  - Size of space depends on real size of physical memory

Thinking:
If a computer has 32-bit CPU and memory address is 16-bit, how big is the logical address space and the physical memory address?

Logical vs. Physical Address Space

Logical and physical addresses are the same in
- compile-time address binding, and
- load-time address-binding schemes

logical (virtual) and physical addresses differ in
- execution-time address-binding scheme

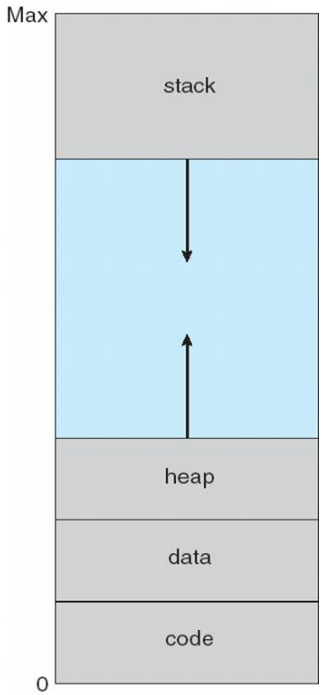
This lecture talks about the binding of logical and physical addresses at execution time.

Memory-Management Unit (MMU)

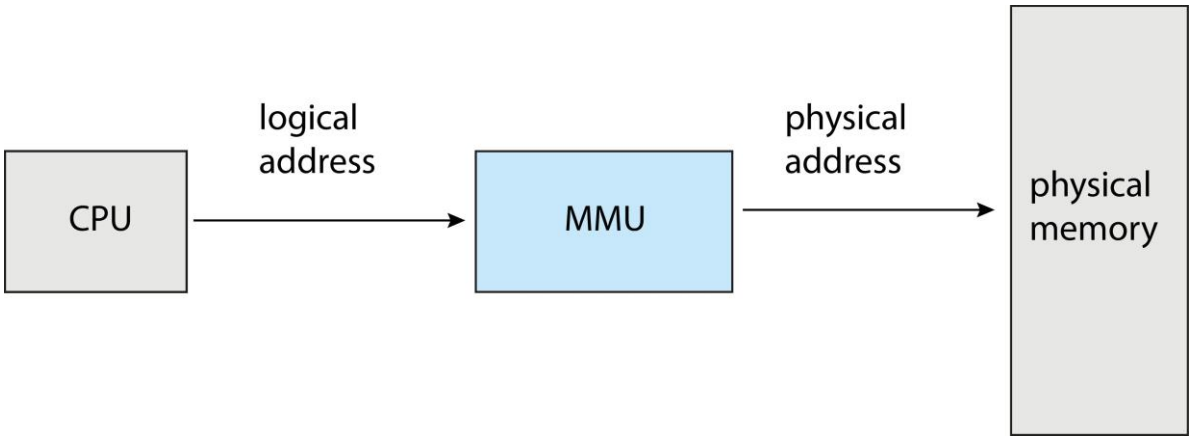
MMU

Memory Management Unit

Hardware device that at run time maps (translates) logical (virtual) address to physical address



Logical address space

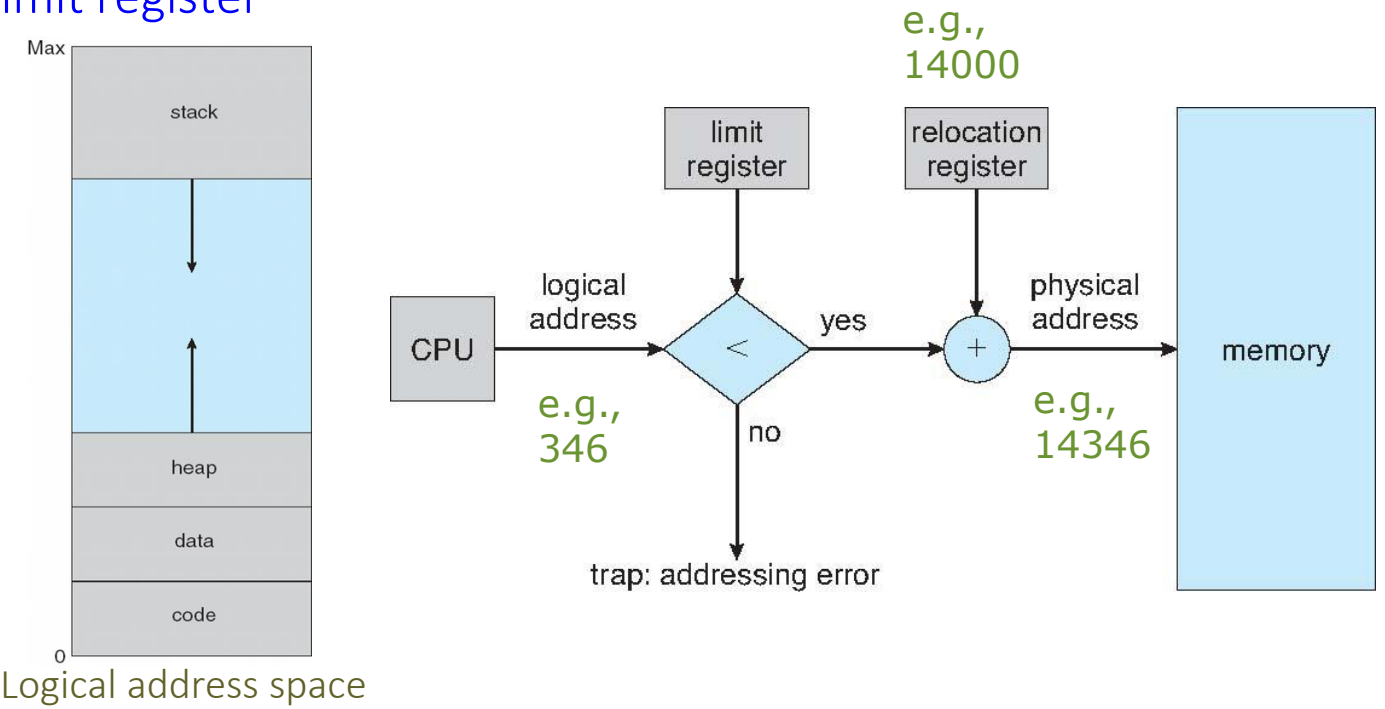


Memory-Management Unit (Cont.)

A simple mapping scheme

relocation register

limit register



Logical addresses of all processes start at 0000

Two processes can use the same logical addresses, these addresses will be mapped to different places in the physical memory

The kernel changes the relocation and limit register during a context switch

Dynamic Loading

Should the entire program be loaded in to the memory to start the execution?

- No!
- Code is loaded from the hard disk into memory only when it is needed, on demand
- Dynamic loading
  - The loading of a process routine when it is called rather than when the process is started.

Why

- Better memory-space utilization
- Unused routine is never loaded
- Some large amounts of code handle infrequently occurring cases

How

- All routines are kept on disk in relocatable load format
- Implemented through program design by programmers, not OS
- OS can help by providing libraries to implement dynamic loading

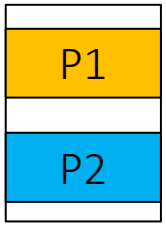
Memory Allocation

Memory can be shared by many processes, including kernel and user processes

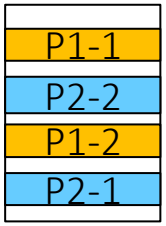
- It should be allocated efficiently
- Resident kernel usually is held in low memory
- User processes is held in high memory

Memory for each process can be

- Contiguous (early method)



- Not contiguous (modern method)



Continuous Partition

Memory is divided into multiple partitions

- Size of each partition is fixed
  - Multiple-partition allocation
  - Disadvantages:
    - Degree of multiprogramming is limited by number of partitions
    - Memory is wasted if a process does not fully use its partition
- Size of each partition is variable
  - Variable-partition
  - Propositional to a given process' size

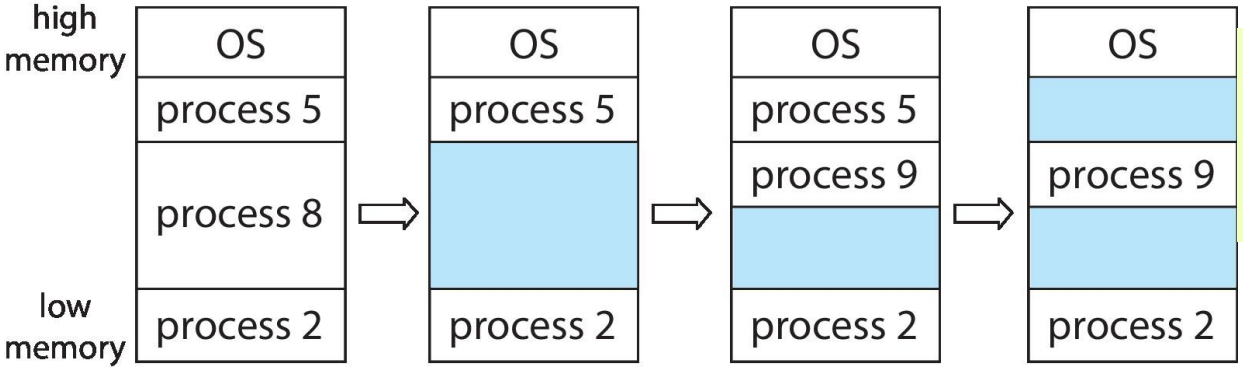
Continuous Partition: Variable Partition

Kernel maintains information about

- allocated partitions
- free partitions (hole)

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
- When a process exits, its partition is freed, adjacent free partitions are combined (coalesced/merged)



Thinking:
Memory after process 9 is released?

Continuous Partition: Variable Partition

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

Three allocation algorithms

- First-fit: Allocate the *first* hole that is big enough
- Best-fit: Allocate the *smallest* hole that is big enough;
- Worst-fit: Allocate the *largest* hole

Comparison of three algorithms

- Speed
  - First-fit > best-fit and worst-fit
    - Both best-fit and worst-fit need to search the whole list
- Memory usage
  - Best-fit > first-fit > worst-fit (theoretically)
  - First-fit > best-fit (case studies)
    - Reason: best-fit produces many small holes that can never be used

Fragmentation

External Fragmentation

Total memory space exists to satisfy a request, but it is not contiguous

Small holes might be wasted because the holes are too small to fit any process.

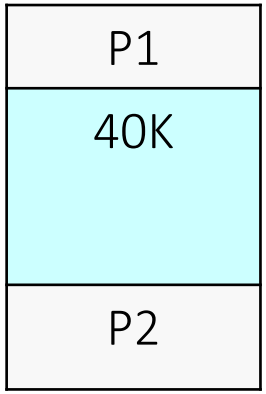
Internal Fragmentation

Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Any memory which is allocated to a process but that the process did not ask for is wasted.

P3: 31K
20+20>31,
but not contiguous

External fragmentation



P3: 31K
2^5K=32K will be allocated,
but 1k is not used

Internal fragmentation

Fragmentation

First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation

- 50-percent rule
  - the memory lost to fragmentation is about 50% the size of the allocated memory
- 1/3 of RAM may be unusable

Fragmentation

Solutions to fragmentation problems

External fragmentation

- Compaction
  - Shuffle memory contents to place all free memory together in one large block
  - possible only if relocation is dynamic, and is done at execution time
  - Pay attention to I/O problem while there is I/O operations on the affected memory
- Use non-contiguous memory allocation

Internal fragmentation

- No way
- Memory is allocated in the block of 2^n bytes, rather than byte by byte.

Non-continuous Partition

Physical address space of a process can be noncontiguous

Process is allocated physical memory whenever the total free memory blocks are enough

- Avoids external fragmentation

Frame

- Divide physical memory into fixed-sized blocks
- Each block is called a frame
- The size of each frame is power of 2, between 512 bytes and 16 Mbytes
- Kernel keeps track of all free frames

Thinking:

If RAM's address uses 16 bits, each frame is 4K, how many frames are in the memory?

Non-continuous Partition

0000	Frame 0	0000 0000 0000
0001	Frame 1	1111 1111 1111 0000 0000 0000
0010	Frame 2	1111 1111 1111 0000 0000 0000 1111 1111 1111
1111	Frame 15	0000 0000 0000 1111 1111 1111

16 bits: 2^{16} , 4K: $4*2^{10} = 2^{12}$

Total frames: $2^{16}/ 2^{12} = 2^{4}$

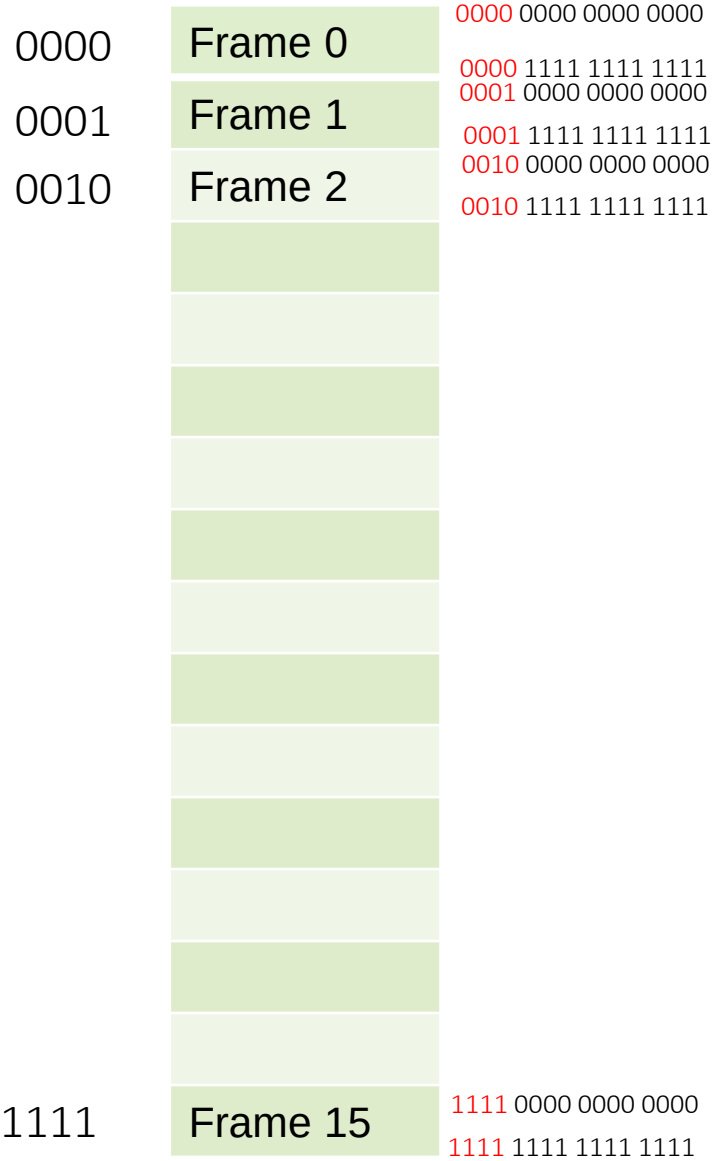
Frame number

Offset in each frame

Thinking:

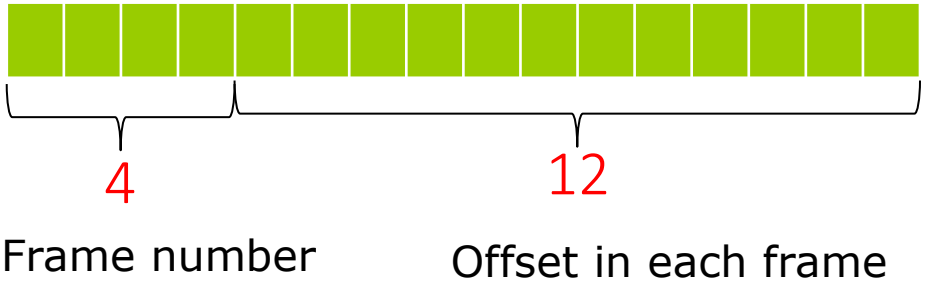
If RAM's address uses 16 bits, each frame is 4K, how many frames are in the memory?

Non-continuous Partition



16 bits: 2^{16} , 4K: $4*2^{10} = 2^{12}$

Total frames: $2^{16}/ 2^{12} = 2^4$



Thinking 1:

If a datum is in the address

0100 0000 0000 0001?

Which frame is it in? where is it in the frame?

Thinking 2:

If a process's size is 8K + 1, how many frames are allocated? Is there internal fragmentation?

Non-continuous Partition

How a logical address maps to a physical address?

Pages

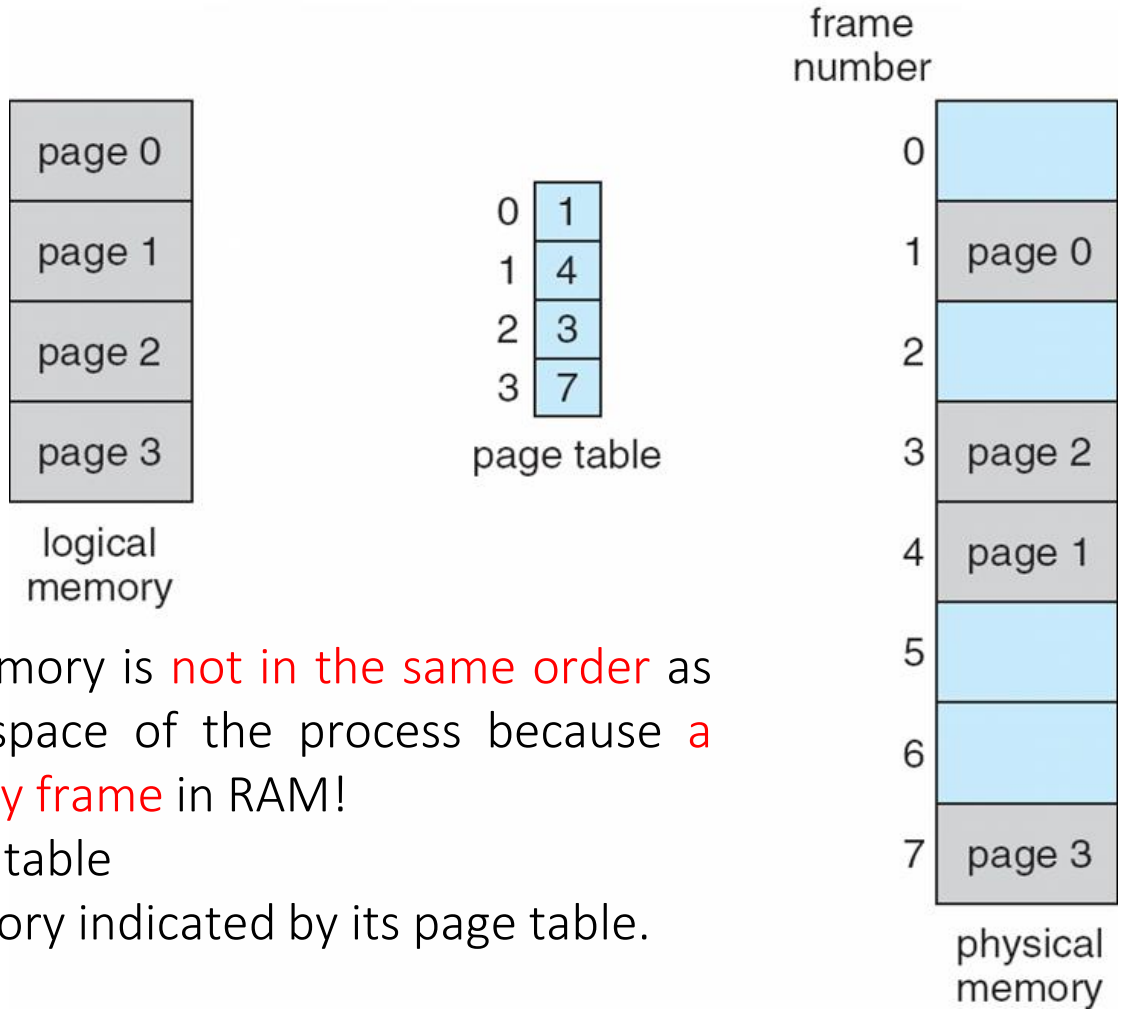
- Divide logical memory into blocks of same size called pages
- Page size = frame size
- To run a program of size *N* pages, need to find *N* free frames and load program
- Frames can be anywhere(non-contiguous) in RAM!

A page table is needed to translate logical to physical addresses

Thinking:

If logical space uses 32 bits, each frame is 4K, how many pages are there in logical space? $2^{32}/ 2^{12}= 2^{20}$

Address Translation Scheme: Page Table



The data in physical memory is not in the same order as in the logical address space of the process because a page can be stored in any frame in RAM!

Each process has a page table

Process access the memory indicated by its page table.

Page table is

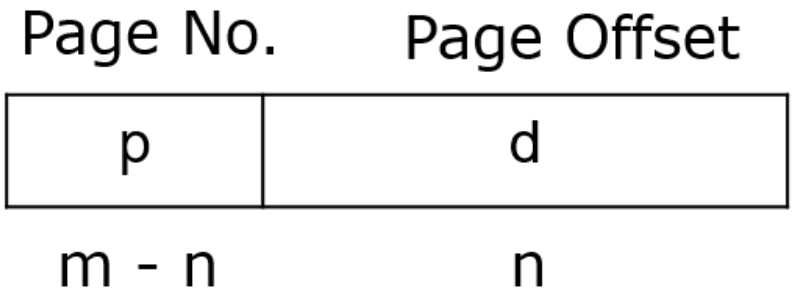
- part of the process's PCB.
- invisible to the process itself.

Address Translation Scheme: Page Table

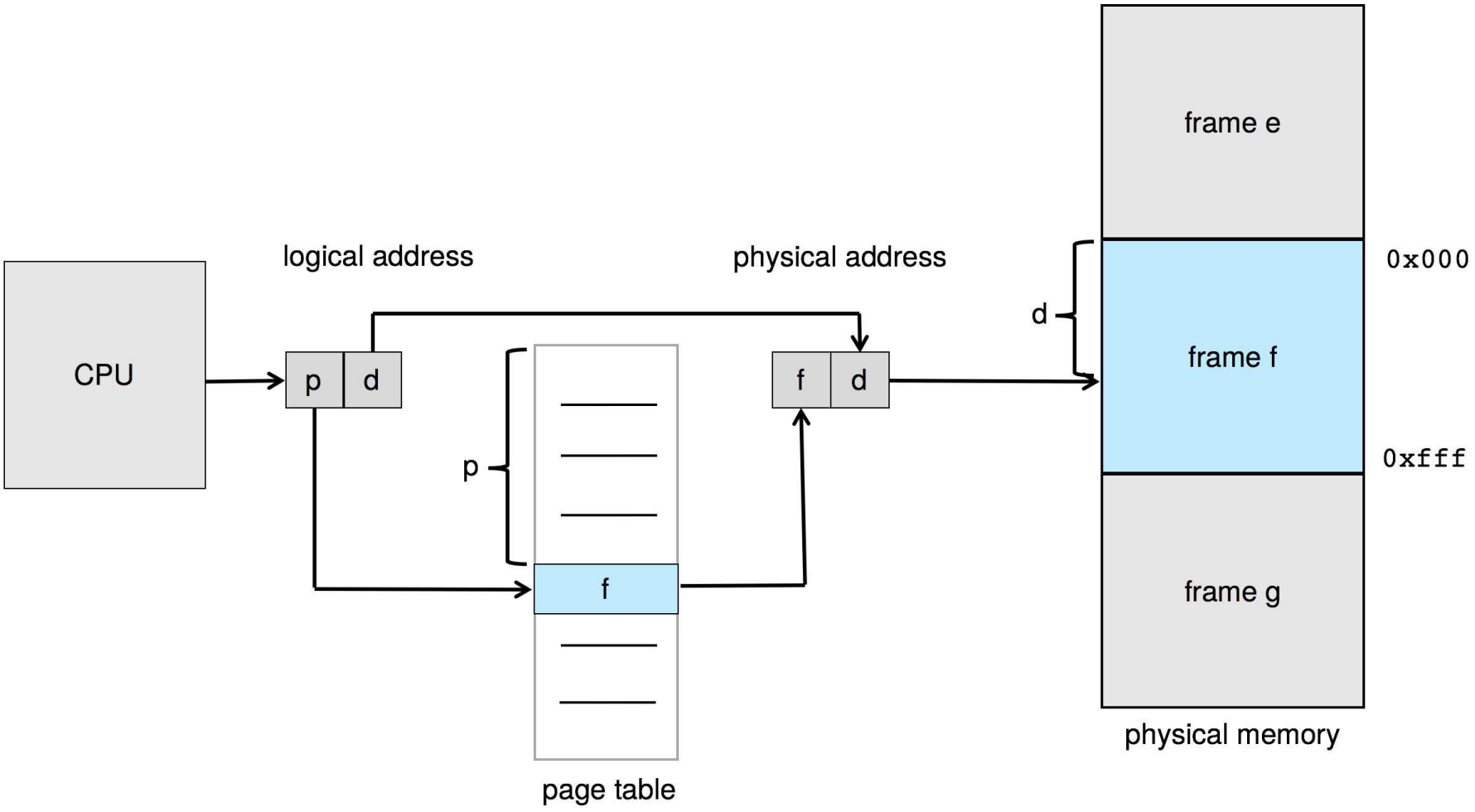
Given a logical address space 2^m and page (frame) size 2^n, there are $2^m/2^n = 2^{m-n}$ pages in a process's logical address space

Logical address generated by CPU (m bits) is divided into

- Page number (p) – used as an index into a page table which contains base address (frame number) of each page in physical memory
- Page offset (d) – combined with base address (frame number) to define the physical memory address that is sent to the memory unit



Paging Hardware



Page number *p* to frame number *f* translation

An Paging Example

Logical address: 4 bits (16 bytes)

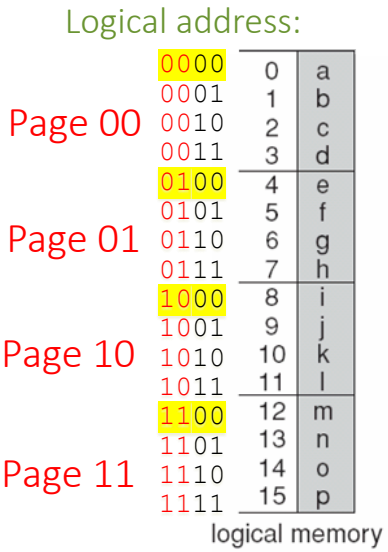
Each page: 4 bytes (2 bits)

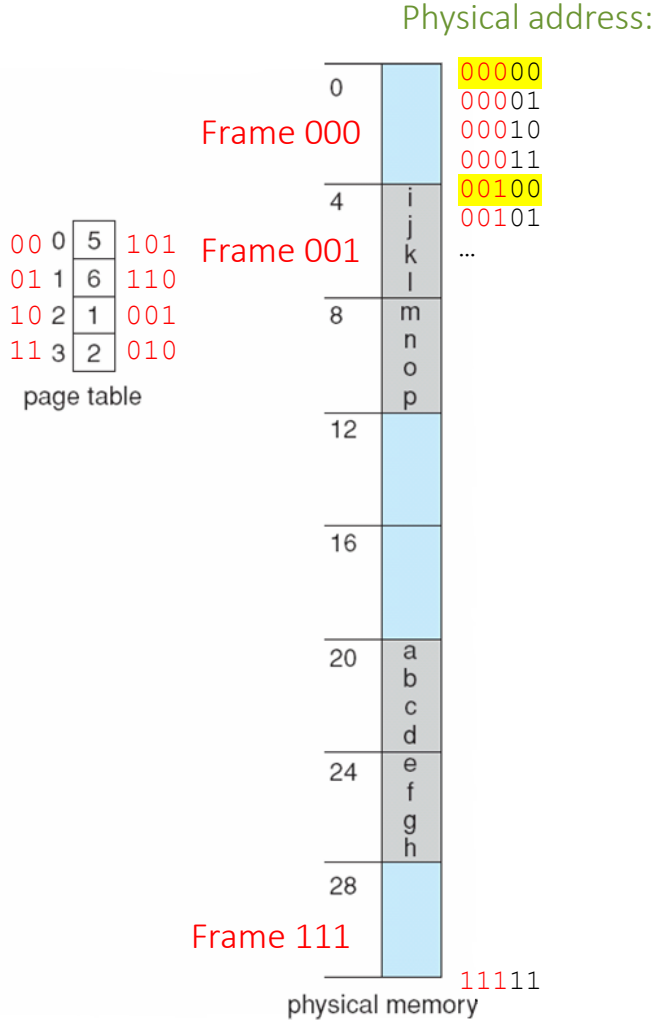
m = 4 and n = 2

Page No.	Page Offset
p	d
m - n	n

Physical address: 5 bits (32 bytes)

Each frame: 4 bytes (2 bits)





Thinking:

Given a logical address is 1011, what is its physical address?

Paging -- Calculating internal fragmentation

Suppose a process:

page size = 2,048 bytes

process size = 72,766 bytes

Thinking 1:

How many frames are needed to store this process?

Thinking 2:

What is the loss of memory due to the internal fragmentation?

Thinking 3:

What is the worst case in the internal fragmentation?

Thinking 4:

Is it a good idea to reduce the size of page to minimize the internal fragmentation?

Page Size

Fact: page sizes growing over time

- Some systems support two page sizes – 8 KB and 4 MB
- X86-64: **4 KB** (common), **2 MB** (“huge” for servers), **1GB** (“large”)

Free Frame Allocation to A New Process

Before allocation

After allocation

Implementation of Page Table

Page table is

part of the process's PCB

kept inside the kernel's own memory

- Two registers are in the MMU
  - Page-table base register (PTBR) points to the start position of page table in memory
  - Page-table length register (PTLR) indicates length of the page table
- These two registers are used by the MMU to find the page table in main memory.

Thinking:

How many memory accesses are needed for one data access?

Implementation of Page Table

Problem

- Two memory accesses
  - one for the page table (logical address -> physical address)
    one for the data / instruction (physical address -> data)
- Too slow!

Solution

- TLB(Translation Look-aside Buffer)
  - also called associative memory
  - hardware cache inside the MMU.

TLB

TLB table

Page #	Frame #

Given a logical page number *p*

If *p* is in TLB (called TLB hit)

- get corresponding frame number from it

If *p* is not in the TLB table (called TLB miss)

- get frame number from page table in memory
- then update the TLB to contain this page # and frame #
  - Reason
    - » if that page was just used by the process, it is probably going to be used again soon.

TLB

TLB is

typically small

- 64 to 1,024 entries

Shared by multiple processes

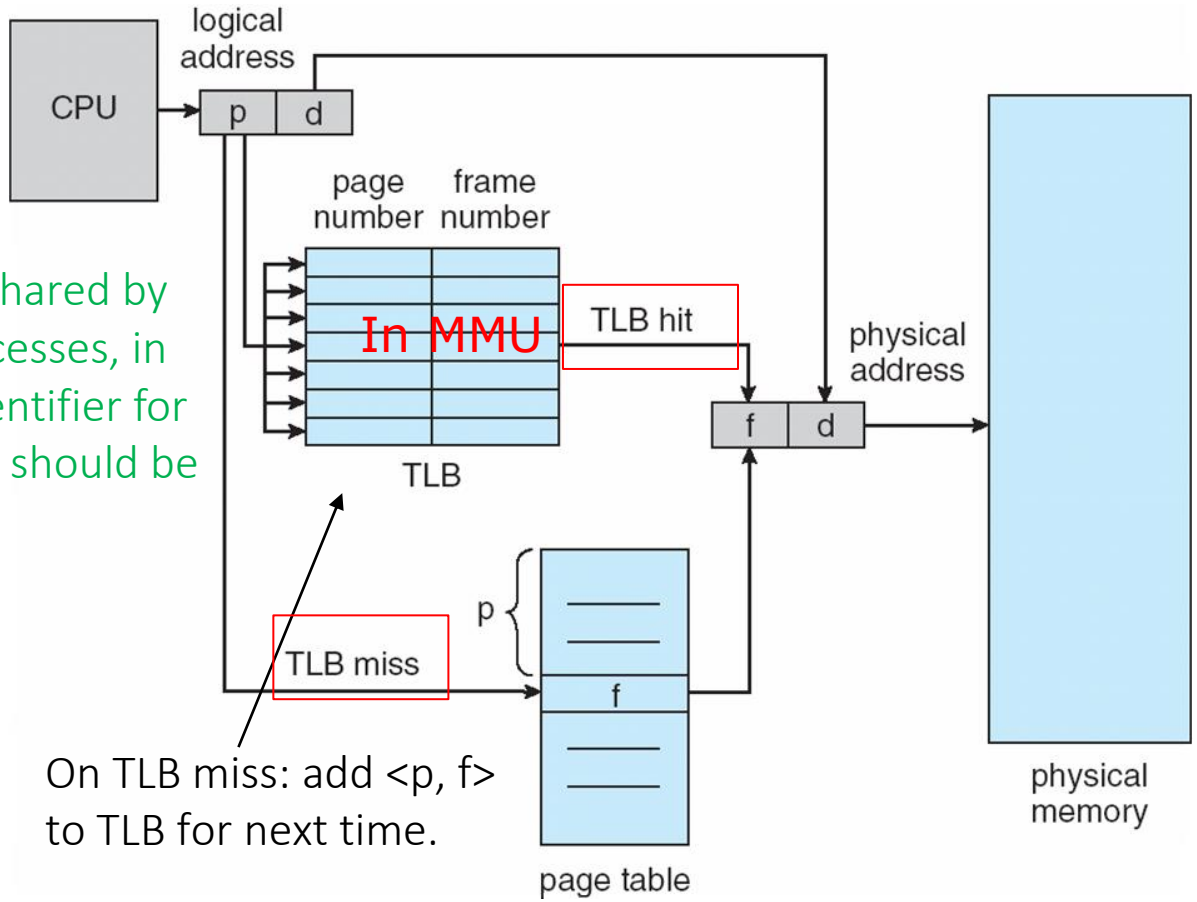
- Store address-space identifiers (ASIDs) in each TLB entry to uniquely identify each process to provide address-space protection for each process
- Reduce the fresh time at every context switch between processes

How is a TLB entry replaced by a new page-frame pair?

Replacement policies must be considered

Some entries can be wired down (never be replaced) for permanent fast access (for key kernel code)

Paging Hardware With TLB



TLB: can be shared by multiple processes, in that case, identifier for each process should be added.

Page table: own by each process

Effective Access Time: An Example

Hit ratio α is

percentage of times that a page number is found in the TLB

Effective memory access time

EAT = α x (c+m)+(1- α) x (c+2m)

c: TLB access time, m: memory access time

An example

α: 80%, c: 10 nanoseconds, m: 90 nanoseconds

- EAT = 0.80 x (10+90) + 0.20 x (10+90 x2) = 118 ns
- 18% slowdown in access time

If α: 99%

EAT = 0.99 x (10+90) + 0.01 x (10+90 x2) = 100.9ns

only 0.9% slowdown in access time

Shared Pages Example

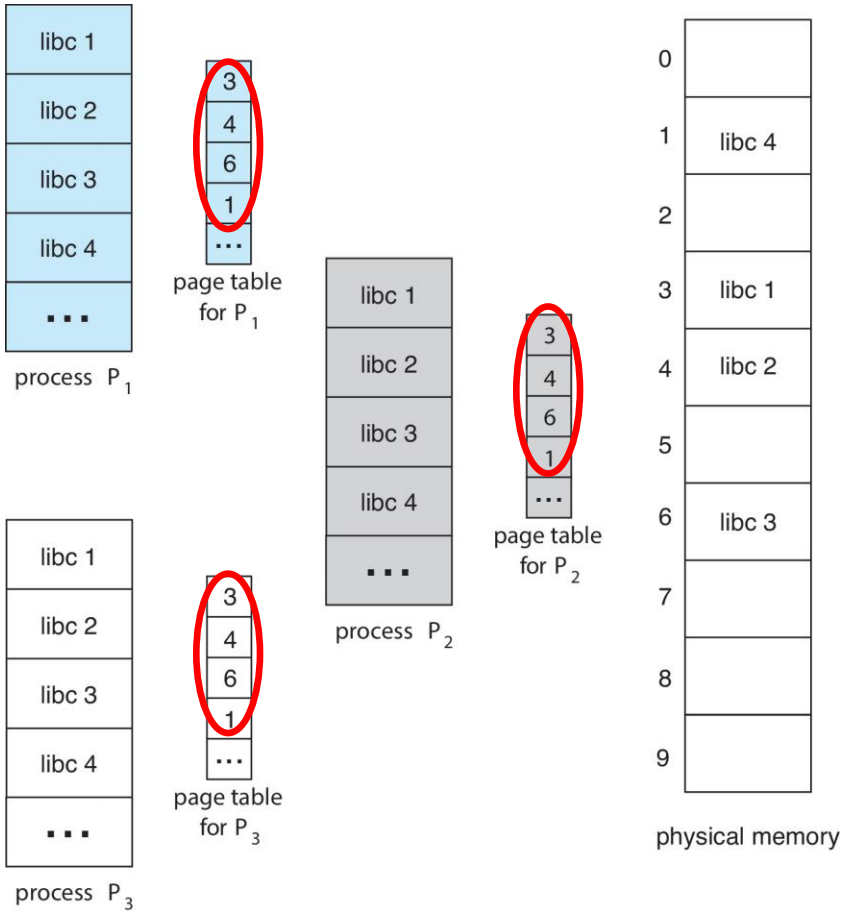
Some code (included libraries) can be shared by multiple processes

E.g., the code of the printf function is shared by all the processes executing C programs, so there is only one copy of the code of printf in physical memory.

Shared code

one copy of read-only code is shared among processes.

can stay at logical locations in the logical address spaces of the different processes,



Process p_1 , p_2 and p_3 share libc 1, 2, 3, 4

Structure of the Page Table

Page No.	Page Offset
p	d
m - n	n

Problem

Memory structures for paging can get huge

For example

- logical address space: 32-bit (m=32), page size: 4 KB (n=12)
- page table length: $2^{m-n} = 2^{20}$ (1M)
- if each entry is 4 bytes → 4 MB of physical address space
- need a single hole of free memory which is big enough!
- More: 4M is only for one process.

Solutions

Hierarchical Paging

Hashed Page Tables

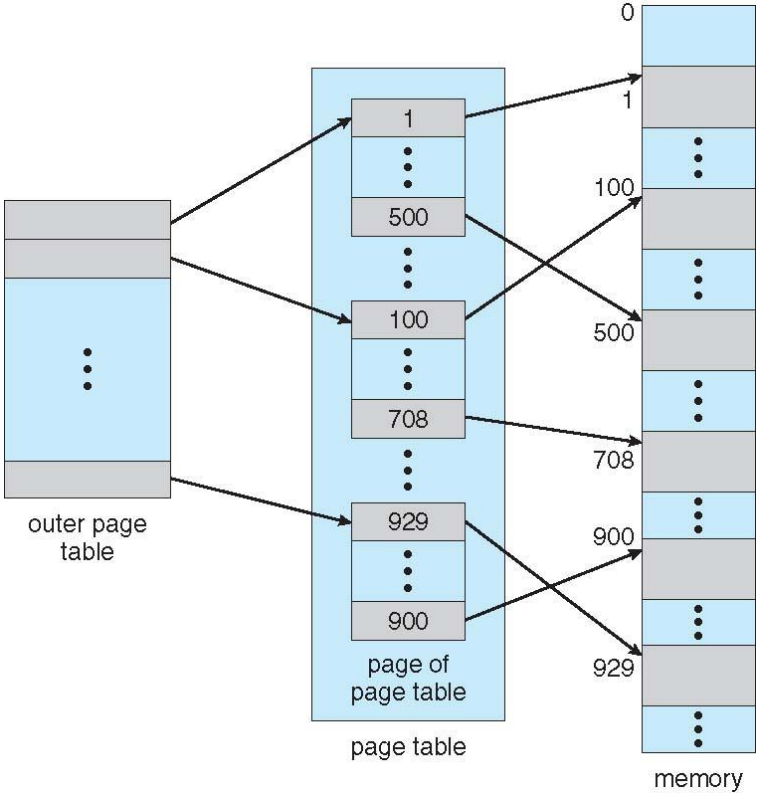
Inverted Page Tables

Hierarchical Page Tables

Split one (big) page table into multiple (small) page tables

two-level page table

- Outer page table
  - Each entry indicates the position of small page table
- Inner (small) page tables
  - Each small page table (also called page of page table) has the page size
  - Different small page tables can be put in different places
  - Only the part of the page table that the process requires right now needs to be in memory!



Two-Level Paging Example

logical address space: 32-bit (m=32), page size: 4 KB (n=12)

Originally

- page table length: $2^{m-n} = 2^{20}$

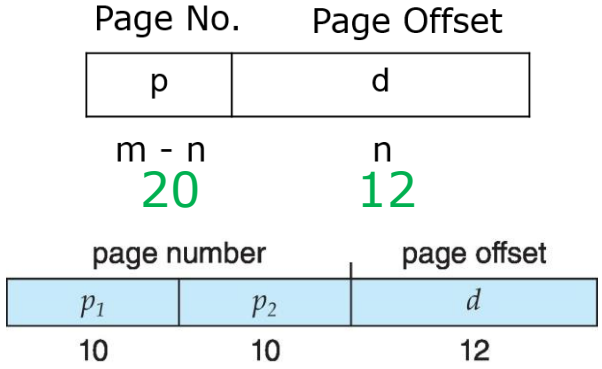
Two levels

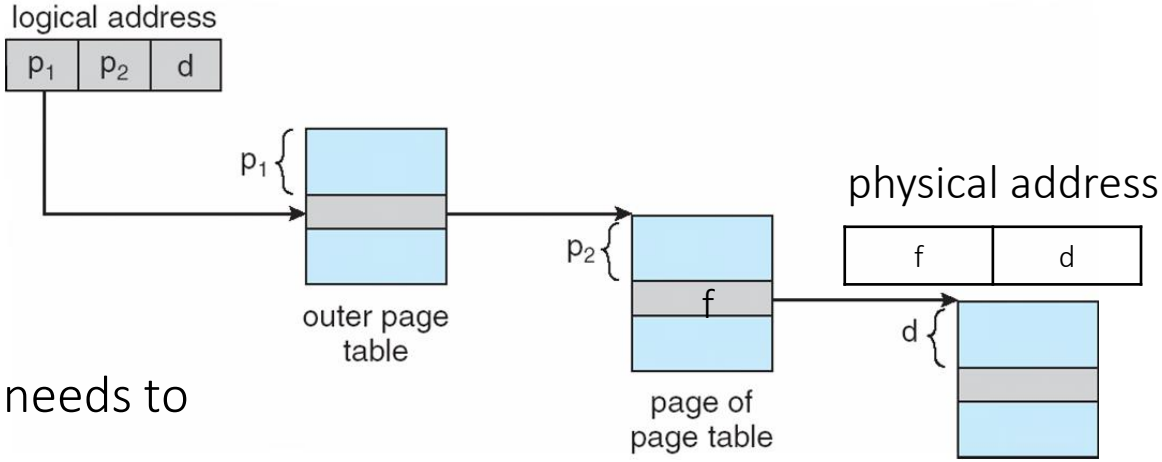
p_1 : a 10-bit page number (1KB)

- an index into the outer page table

p_2 : a 10-bit page offset (1KB)

- the displacement within the page of the inner page table





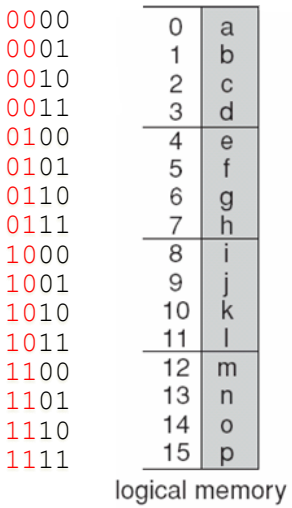
- Advantage: save memory
  - only part of the page table needs to be in memory
- Disadvantage : three memory access

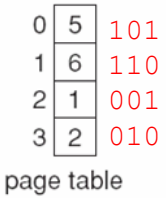
Address translation process

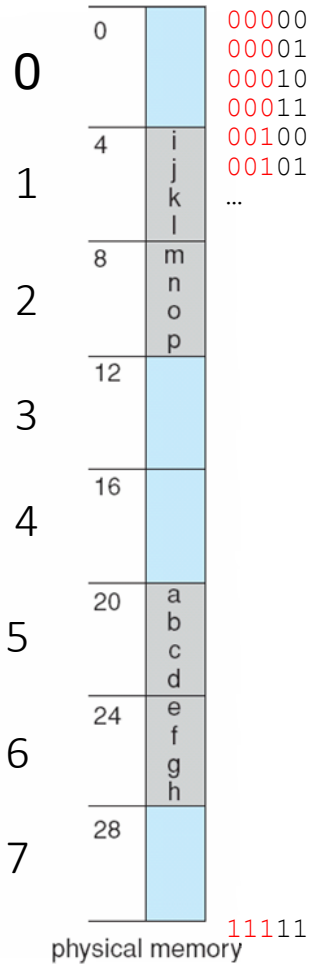
Paging Example

m=4, n = 2

2	2
00	00
01	01
10	10
11	11







p_1	p_2	d
1	1	2

0 Table 0 address

1 Table 1 address

Page table 0

0	5	101
1	6	110

Page table 1

0	1	001
1	2	010

Outer table

0 0

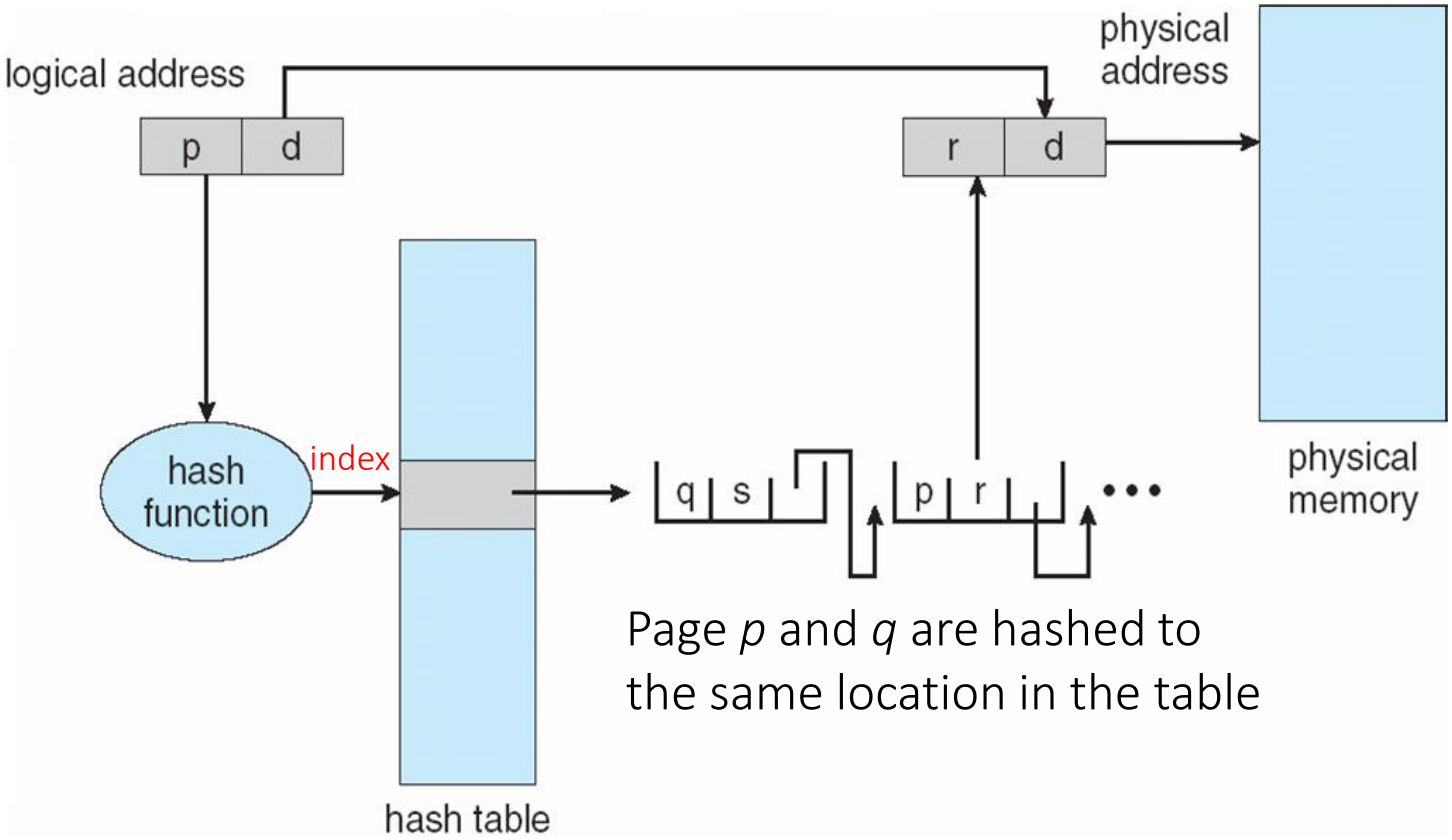
1 1

Separated page tables

Thinking:

Given a logical address 1011, how to find its physical address?

Hashed Page Tables

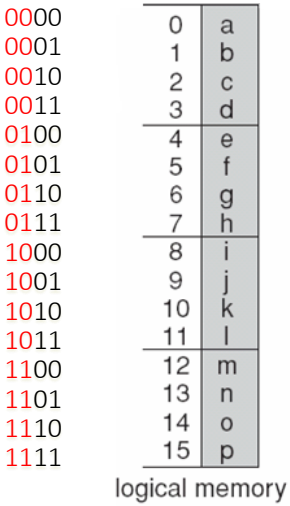


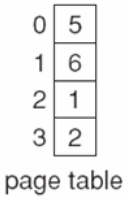
- An entry in the hash table contains a chain of elements hashing to the same location
- Each element contains
  1. the virtual page number (e.g., *q*)
  2. the value of the mapped page frame (e.g., *s*)
  3. a pointer to the next element.
- This method is commonly used for the address with more than 32 bits

Paging Example

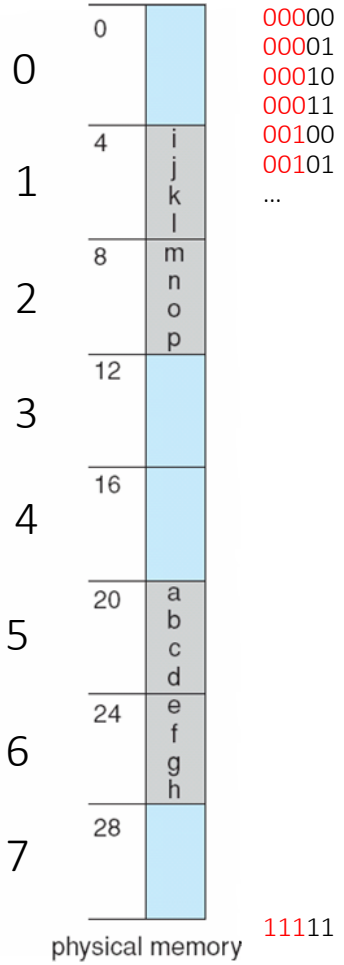
m=4, n = 2

2	2
00	00
01	01
10	10
11	11

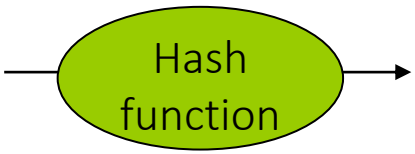




8 Frames

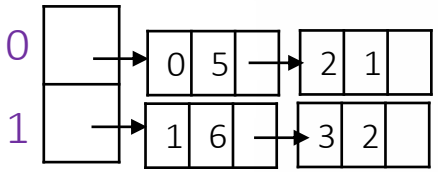


0 00
1 01
2 10
3 11



0, 2 -> 0

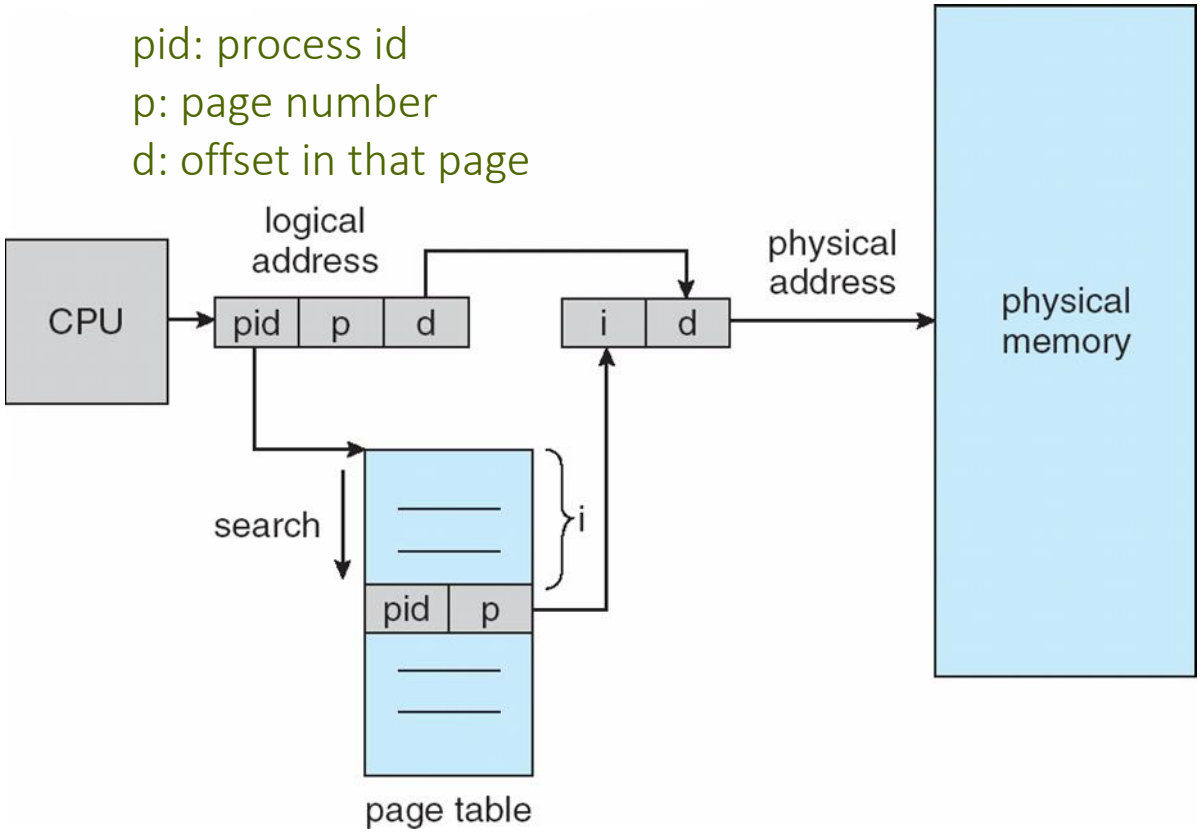
1, 3 -> 1



Thinking:

Given a logical address 1011, how to find its physical address?

Inverted Page Table



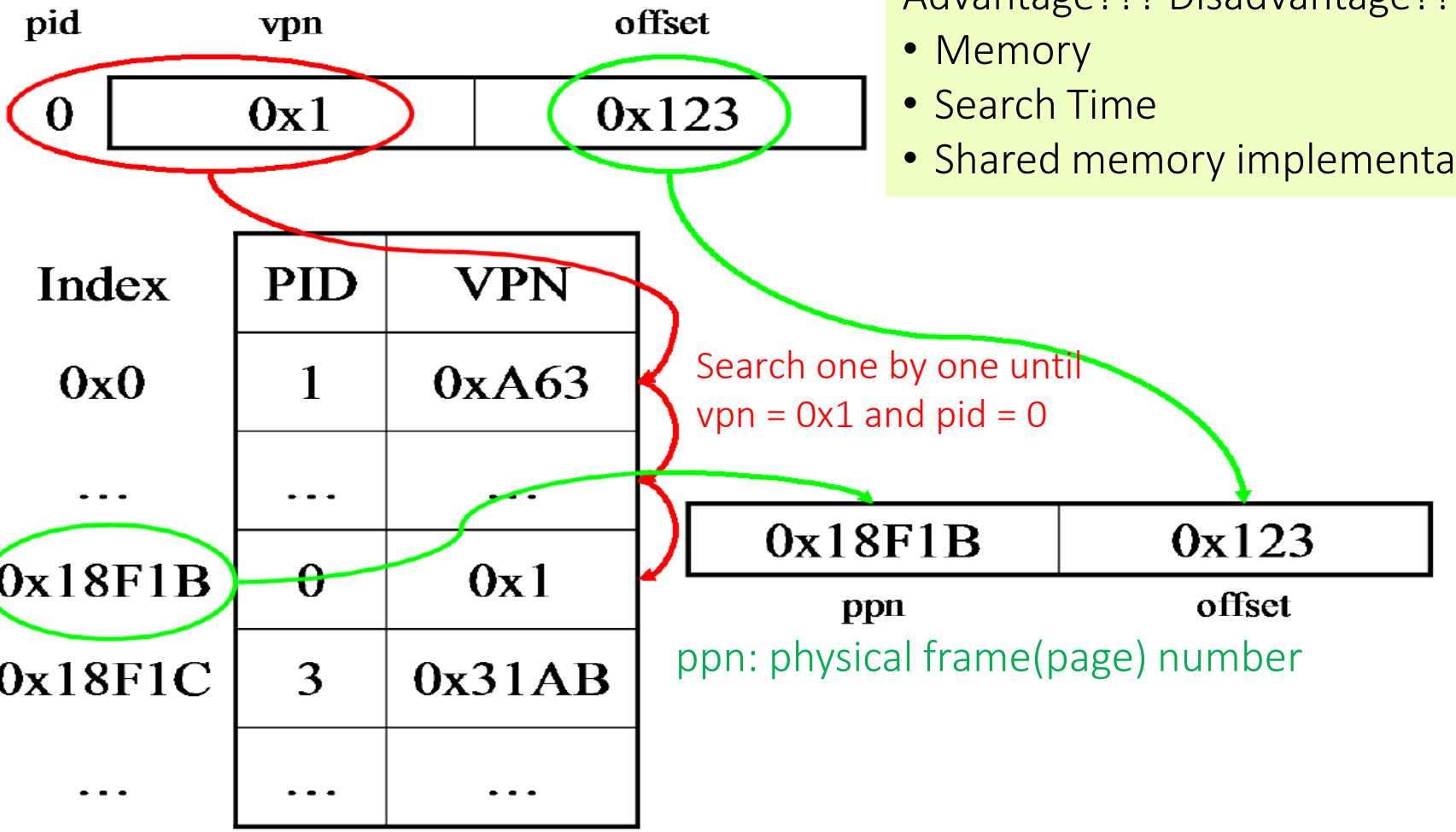
The length of page table is the same as the size of physical memory

All processes share one page table

The page table is actually a reflection of layout of the physical memory frames

Inverted Page Table Example

vpn: virtual page number



ppn: physical frame(page) number

Thinking:

Advantage??? Disadvantage???

- Memory
- Search Time
- Shared memory implementation

Example: The Intel 32 and 64-bit Architectures

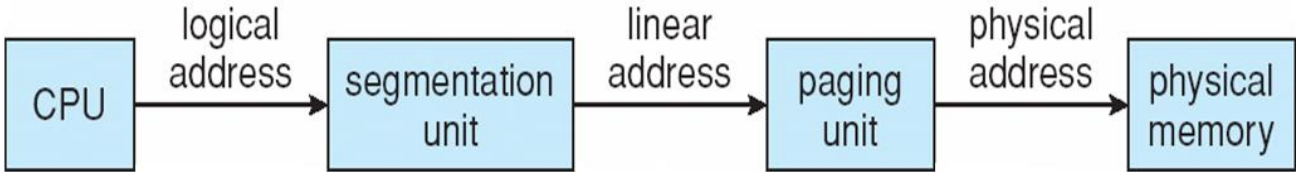
- Dominant industry chips
  - IA-32 architecture (x86 or i386) (CPUs are 32-bit)
  - IA-64 architecture (x86-64 or amd64) (CPUs are 64-bit)
  - Many variations in the chips

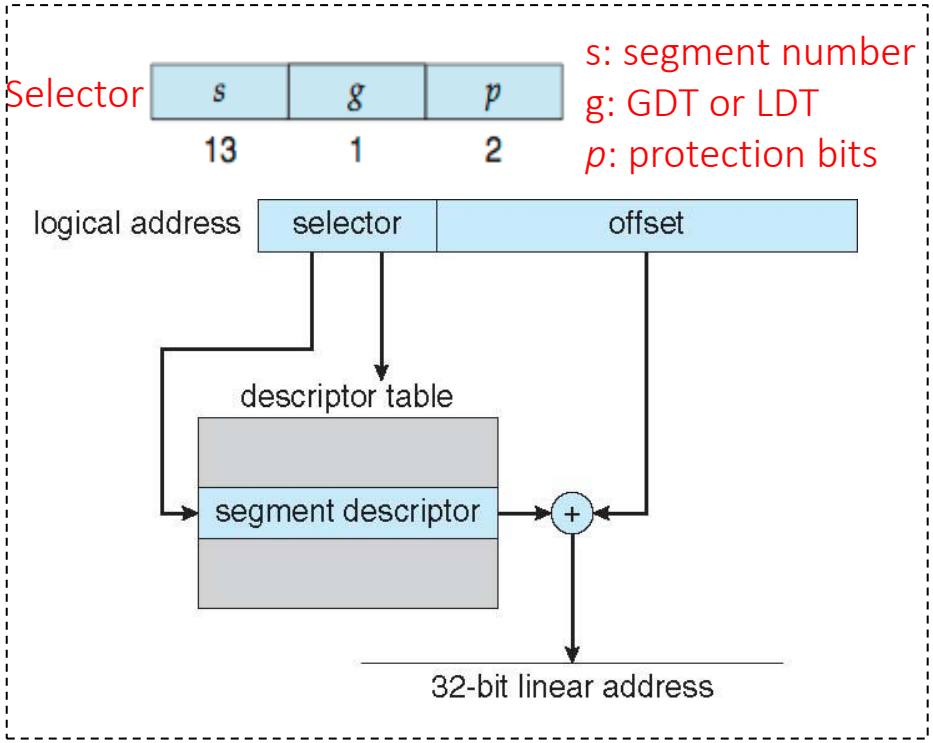
Example: The Intel IA-32 Architecture

Supports both segmentation (pieces of partition) and segmentation with paging

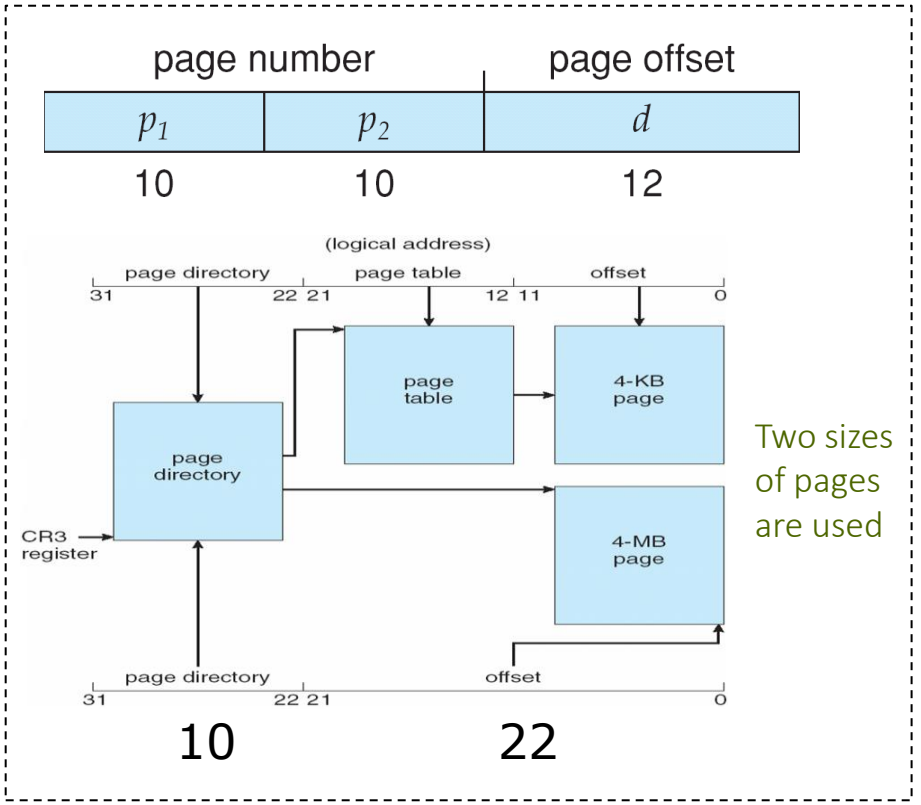
- Each process has up to 16 K segments
  - First partition has up to 8 K segments (private to process, kept in local descriptor table (LDT))
  - Second partition has also up to 8K segments (shared among all processes, kept in global descriptor table (GDT))
- Each segment can be 4 GB

Logical to Physical Address Translation in IA-32





Segmentation unit



Paging unit

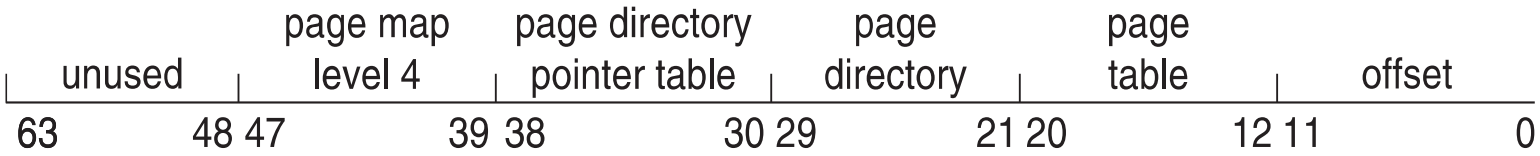
Example: Intel x86-64

64 bits is enormous (> 16 exabytes)

In practice only implement 48 bit addressing

- Page sizes of 4 KB, 2 MB, 1 GB
- Four levels of paging hierarchy

Can also use PAE (page address extension) so virtual addresses are 48 bits but support 52 bits physical addresses (4096 terabytes)



Example: ARM Architecture

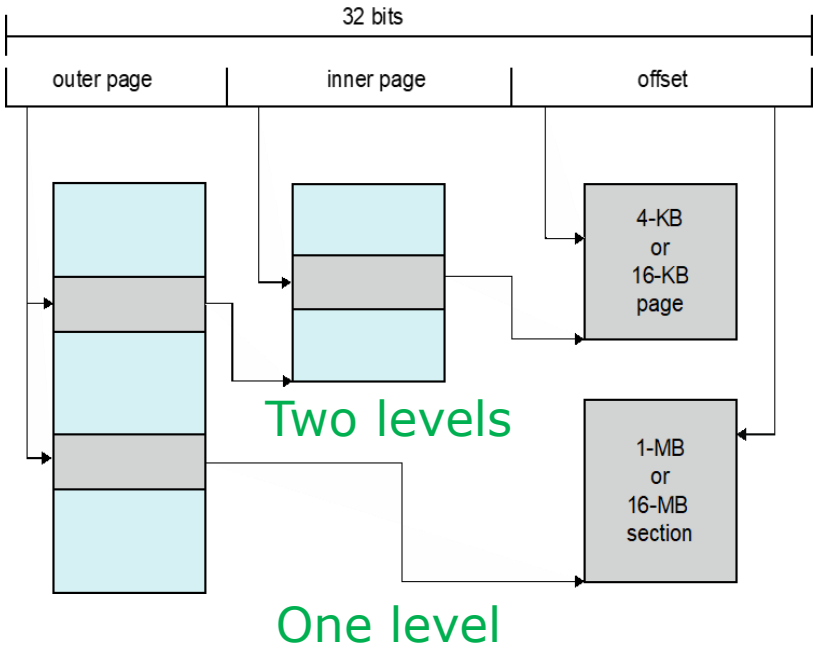
Dominant mobile platform chip (Apple iOS and Google Android devices for example)

ARM

Modern, energy efficient, 32-bit CPU

Two page sizes: 4 KB or 16 KB pages (two levels)

1 MB or 16 MB pages (one level)



One-level paging for sections, two-level for smaller pages

Two levels of TLBs

inner level has two micro TLBs (one data, one instruction)

outer is single main TLB

inner is checked first, on miss, outers are checked, and on miss again, page table is accessed

64 bit CPUs now exist too

End of Chapter 9