Part III
Memory Management

Memory Management

- The main purpose of a computer system is to execute programs.
- These programs, together with the data they access, must be at least partially in main memory during execution.

- To improve both the utilization of the CPU and the speed of its response to users, a general-purpose computer must keep several processes in main memory.
  - that is, we must share memory.

Next two chapters look into effective ways for managing shared memory

Chapter 9: Main Memory

Chapter 9: Memory Management

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

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques

Main Memory

Main memory refers to Physical Memory

- Memory that is internal to the computer.

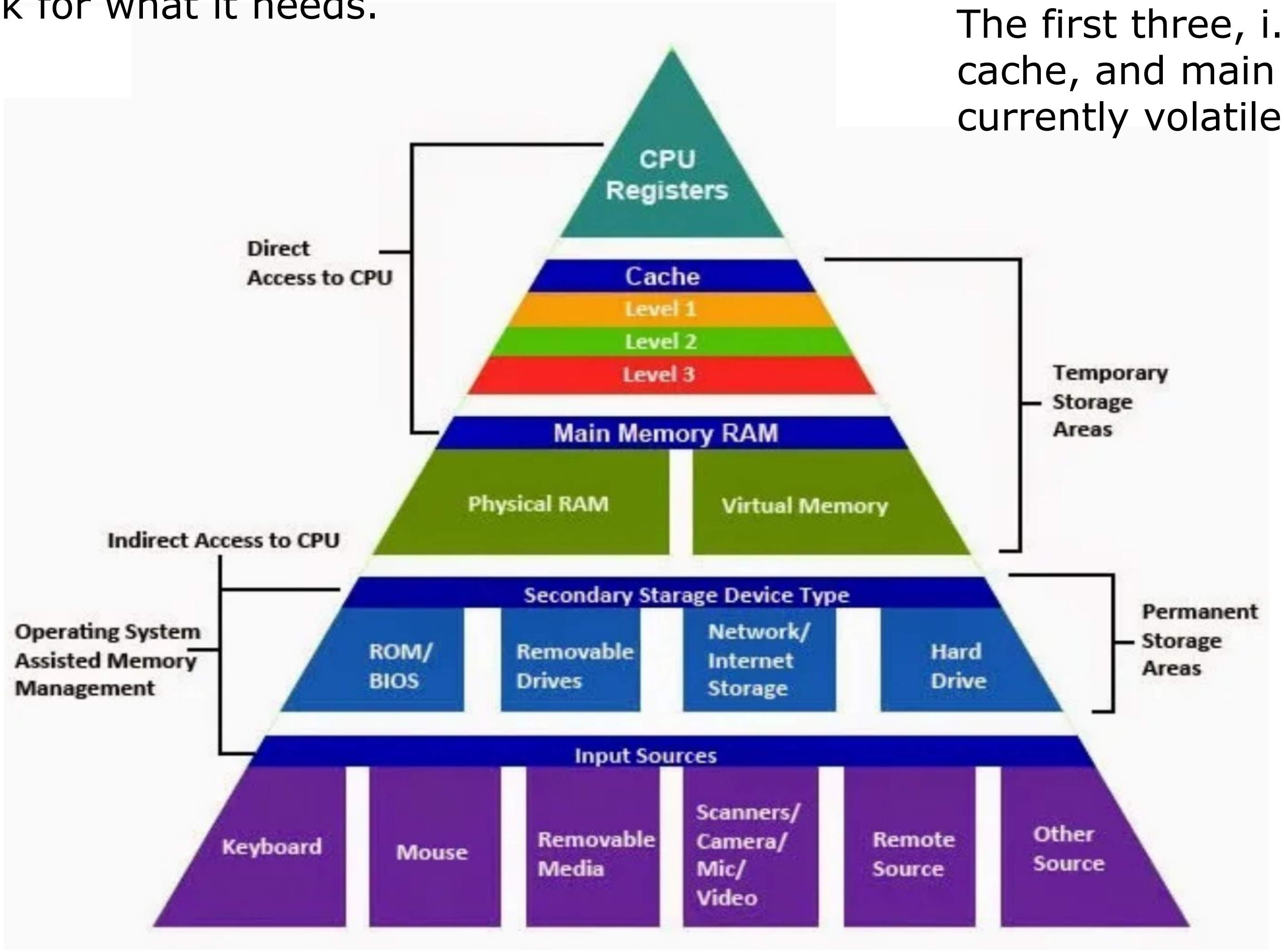
- Other terms used to mean Main Memory or Internal Memory include RAM and Primary Storage
  - The word RAM (Random-access Memory) is often associated with **volatile** types of memory (such as DRAM memory modules)
  - “Random” refers to the idea that any piece of data can be returned in a constant time. By contrast, magnetic or optical storage devices rely on the physical movement of the recording medium or a reading head.

- The amount of main memory on a computer is crucial because it
  - determines how many programs can be executed at one time and
  - how much data can be readily available to a program.

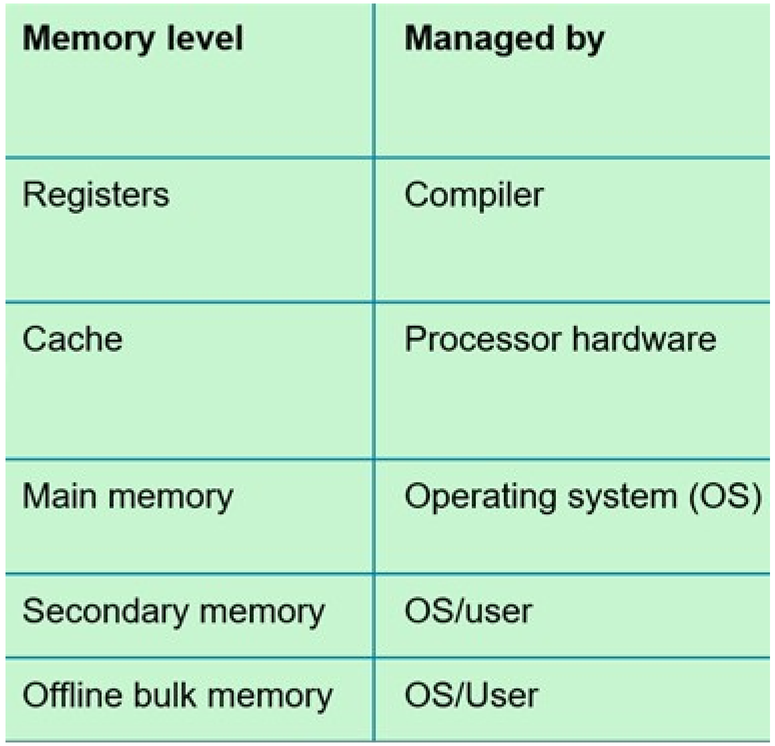
Hierarchy

Memory can be generalized into five hierarchies based upon intended use and speed. If what the processor needs isn't in one level, it moves on to the next to look for what it needs.

The first three, i.e., registers, cache, and main memory, are currently volatile memory.



Memory Hierarchy



Memory level	Managed by
Registers	Compiler
Cache	Processor hardware
Main memory	Operating system (OS)
Secondary memory	OS/user
Offline bulk memory	OS/User

- Registers
  - Typical access time: One clock cycle
- Cache
  - Typical access time: Between around 5 to 50 clock cycles

Memory Hierarchy

- Main Memory
  - Typical access time: Hundreds of clock cycles
  - Most commonly called RAM. Main memory is relatively fast and holds most of the data and instructions that are needed by currently running programs.
- Secondary Memory
  - Typical access time: Tens of thousands clock cycles for the fastest SSDs to millions for HDDs
  - Secondary memory is where data can be permanently stored, usually a hard disk drive (HDD) or solid state disk (SSD).
- Removable memory
  - Typical access time: Depends on the device or interface
  - Data that's intended to move around resides on removable memory. Examples include CDs and DVDs, and USB thumb disks. Modern interfaces like USB 3.2 and Thunderbolt can allow removable memory to be as fast as secondary memory.

How memory is accessed

Physical Memory Access

- Memory Bus

  Main memory is directly or indirectly connected to the CPU via a memory bus. It is actually two buses:

  - an address bus
  - and a data bus.

- The CPU first sends a number through an address bus, a number called memory address, that indicates the desired location of data. Then it reads or writes the data itself using the data bus.

A typical instruction-execution cycle, for example, first CPU fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.

Address Space

Memory can be thought of as one large array containing bytes each indexed by its own memory address (a unique identifier)

The range of addresses **assigned** to a running program is called an **address space**.

Logical Address Space:

- This is the area of contiguous virtual addresses available for executing instructions and storing data.
- Since a logical address does not physically exist, it is also known as a **virtual** address. This address is used as a reference by the CPU to access the actual physical memory location.
- The range of virtual addresses in an address space starts at zero and can extend to the highest address permitted by the operating system architecture.
- Most modern computers are *byte-addressable*. Each address identifies a single byte (eight bits) of storage

Physical & Logical Address Spaces

In computing, an Address Space defines a range of discrete addresses, each of which may correspond to:

1. **Physical Address Space**
2. **Virtual Address Space**, also called **Logical Address Space**

The Logical Address is generated by the CPU

Physical Address is Computed by MMU

The user program deals with logical addresses

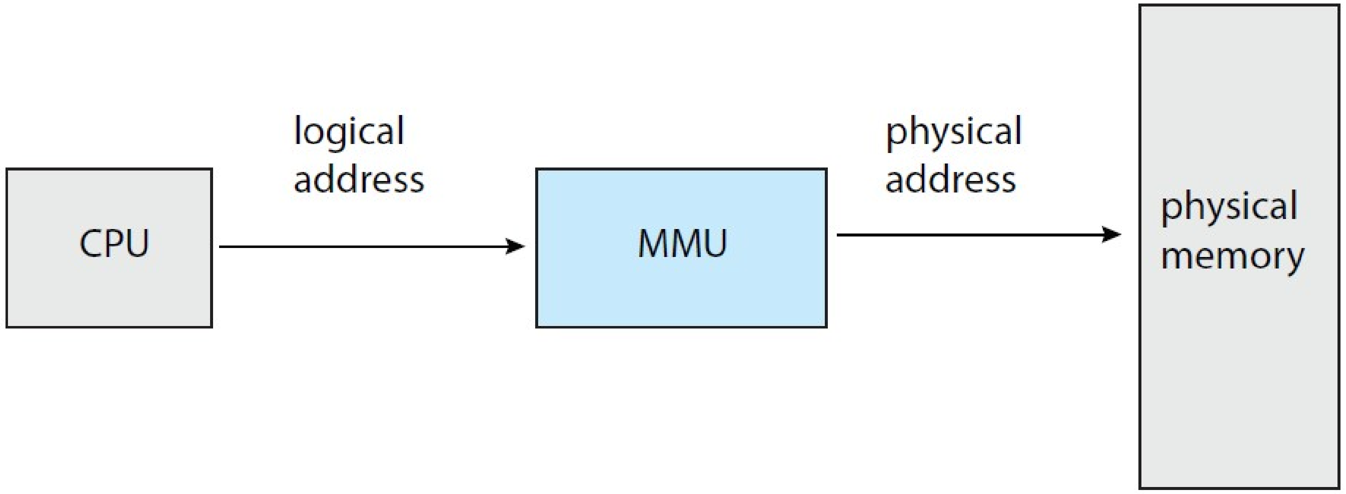
The address seen by the memory unit is the physical address

Logical & Physical Addresses

We now have two different types of addresses:

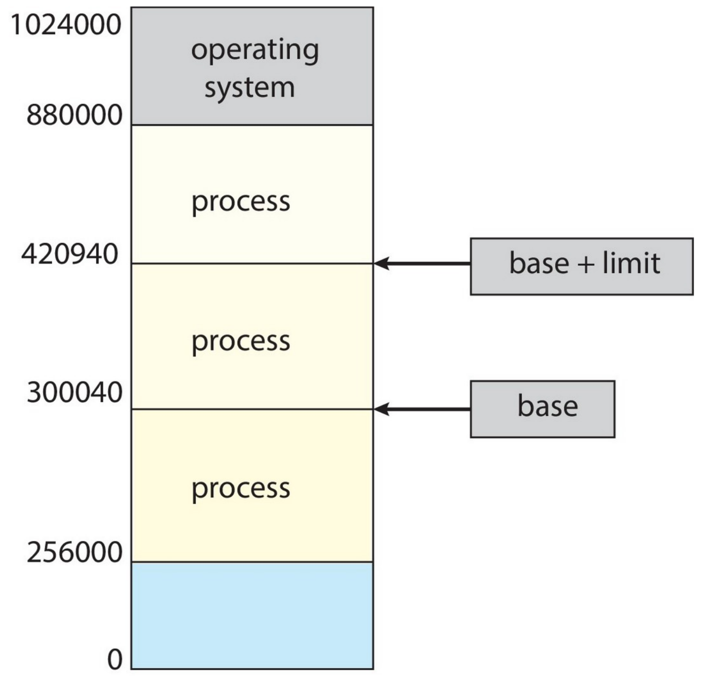
1. Logical addresses (in the range 0 to *max*) and
2. Physical addresses (in the range *R* + 0 to *R* + *max* for a base value *R*).

- The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to *max.*
- However, these logical addresses must be mapped to physical addresses before they are used.



Base and Limit Registers

- A base and a limit register define a logical address space
- The base register holds
  - the smallest legal physical memory address
- The limit register
  - specifies the size of the range



Example:

The base register holds 300040

The limit register is 120900,

then the program can legally access all addresses from 300040 through 420939 (inclusive).

Logical Address Space for this process is addresses 300040 to 420939

Protection

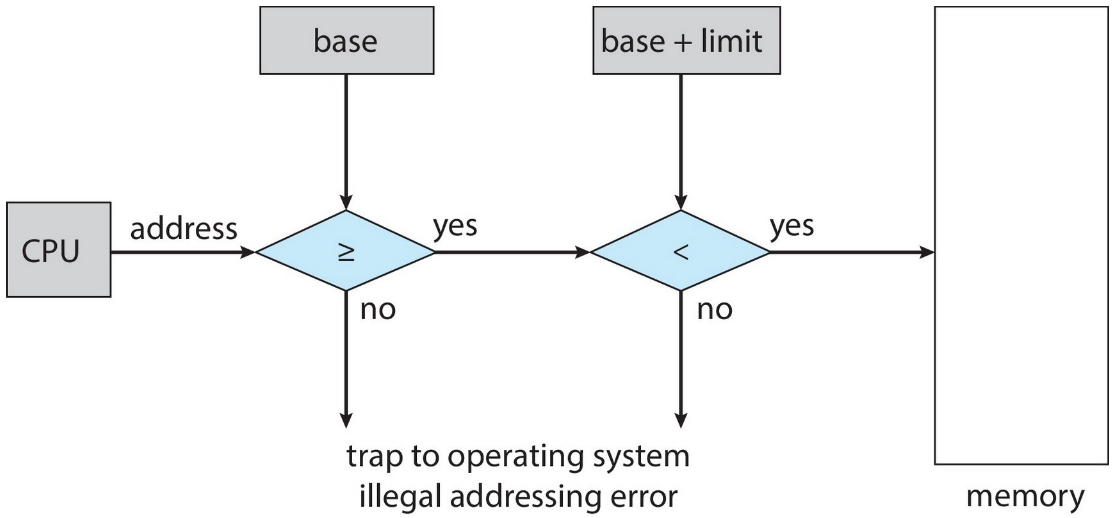
- Each process has a separate memory space
  - Need to ensure that a process can access only those addresses in its own address space
  - Must protect the OS from access by user processes
  - Any attempt by a program executing in user mode to access operating system memory, or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error

This protection must be provided by the hardware because the OS doesn't usually intervene between the CPU and its memory accesses

- We can provide this protection by using a pair of **base** and **limit registers** to define the logical address space of a process

Hardware Address Protection

- 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 loading the base and limit registers are privileged
  - The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction, executing in kernel mode
  - The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory

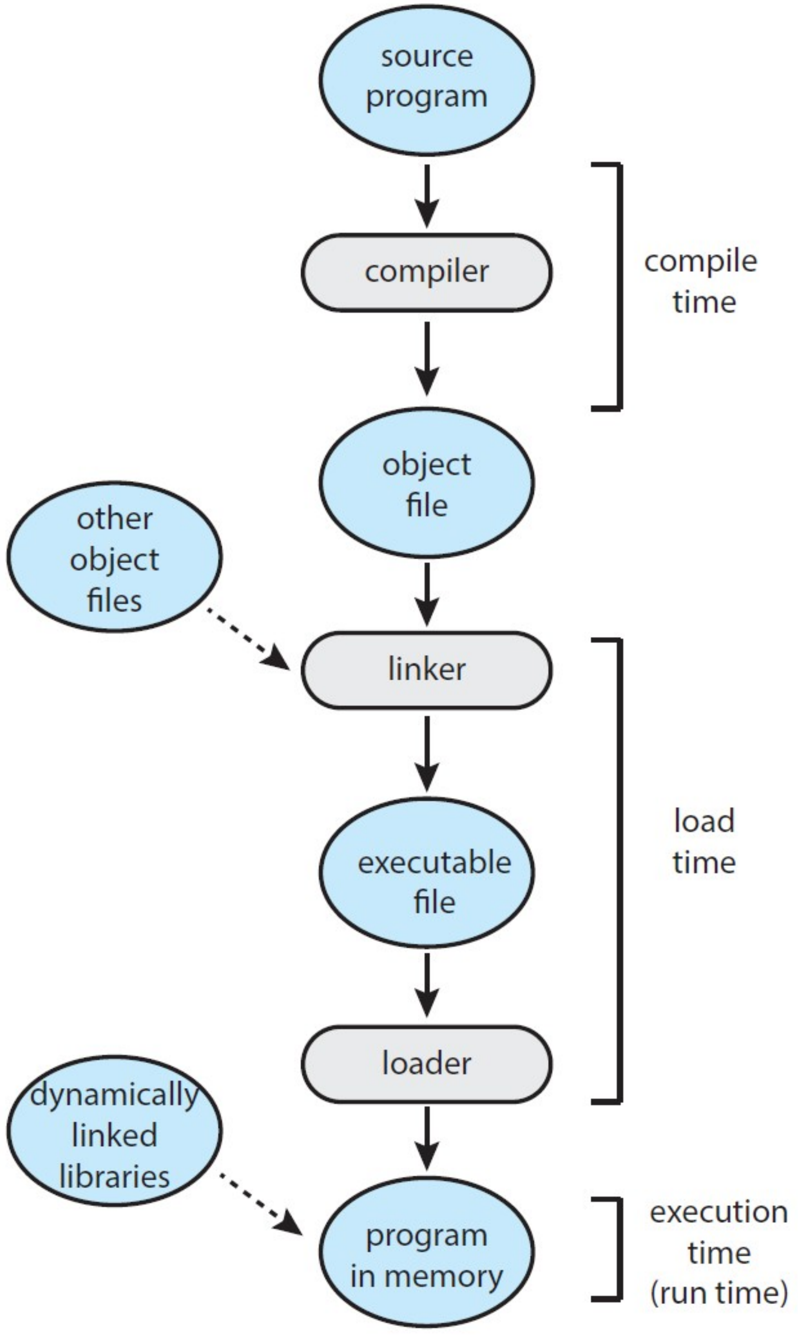
Address Binding

Address binding **maps one address space to another**

Usually, a program resides on a disk as a binary executable file. To run, the program must be brought into memory and placed within the context of a process where it becomes eligible for execution on an available CPU.

- As the process executes, it accesses instructions and data from memory.
- **Logical and physical addresses are the same** in compile-time and load-time address-binding schemes
- **Logical (virtual) and physical addresses differ** in execution-time address-binding scheme

Multistep Processing of a User Program



Linker intakes the object codes generated by the assembler and combine them to generate the executable module.

Loader:- It loads the executable code into main memory; program and data stack are created, register gets initialized.

Binding of Instructions and Data to Memory

The binding of instructions and data to memory addresses can be done at any step along the way:

Compile time binding:

- If you know at compile time where the process will reside in memory, then **absolute code** can be generated. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

- **Load time binding:** Binding is delayed until load time.
  - Binds the **relocatable addresses** to absolute addresses
- **Execution time**: Binding delayed until run time, if the process can be moved during its execution, from one memory segment to another.
  - Need hardware support for address maps (e.g., base and limit registers)

Memory management unit (MMU)

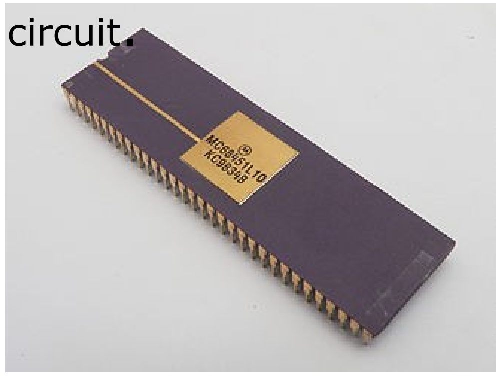
MMU is a small hardware device between CPU and Physical Memory (RAM) for recalculating the actual memory address.

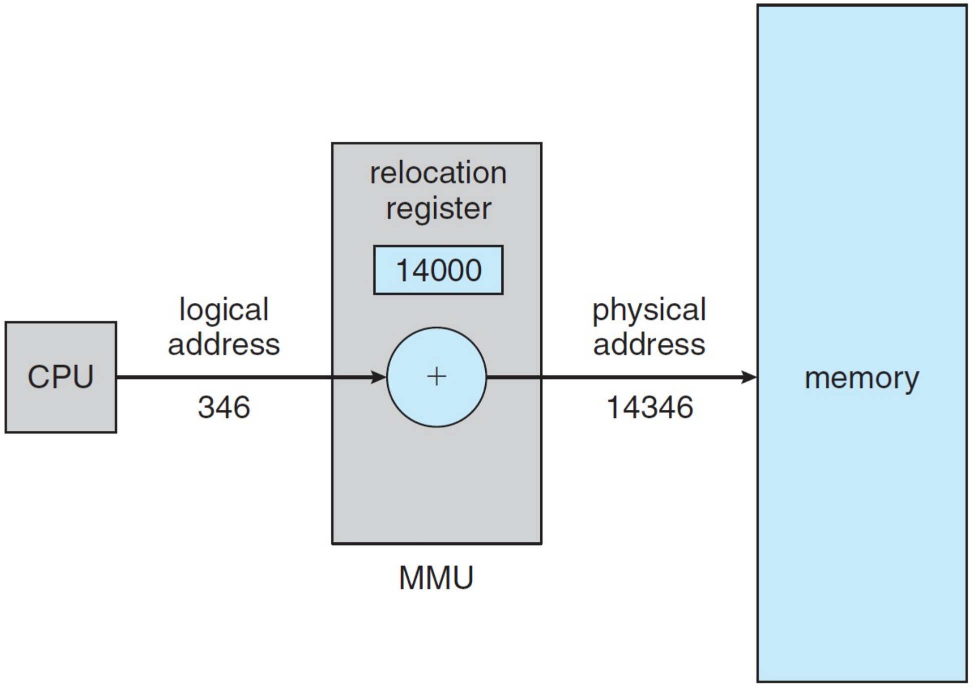
MMU maps virtual to physical address at run time

All memory references passed through it

- Primarily performs the translation of virtual (logical) memory addresses to physical addresses.

Implemented as part of the central processing unit (CPU), but it also can be in the form of a separate integrated circuit.



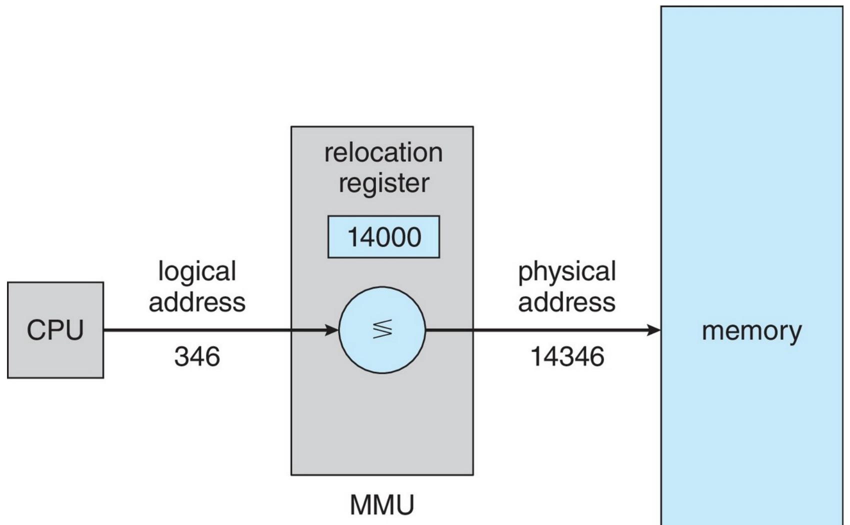


Memory-Management Unit (Cont.)

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- 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
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic Relocation

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



Dynamic Loading

- The entire program does need to be in memory to execute
- Routine is not loaded 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

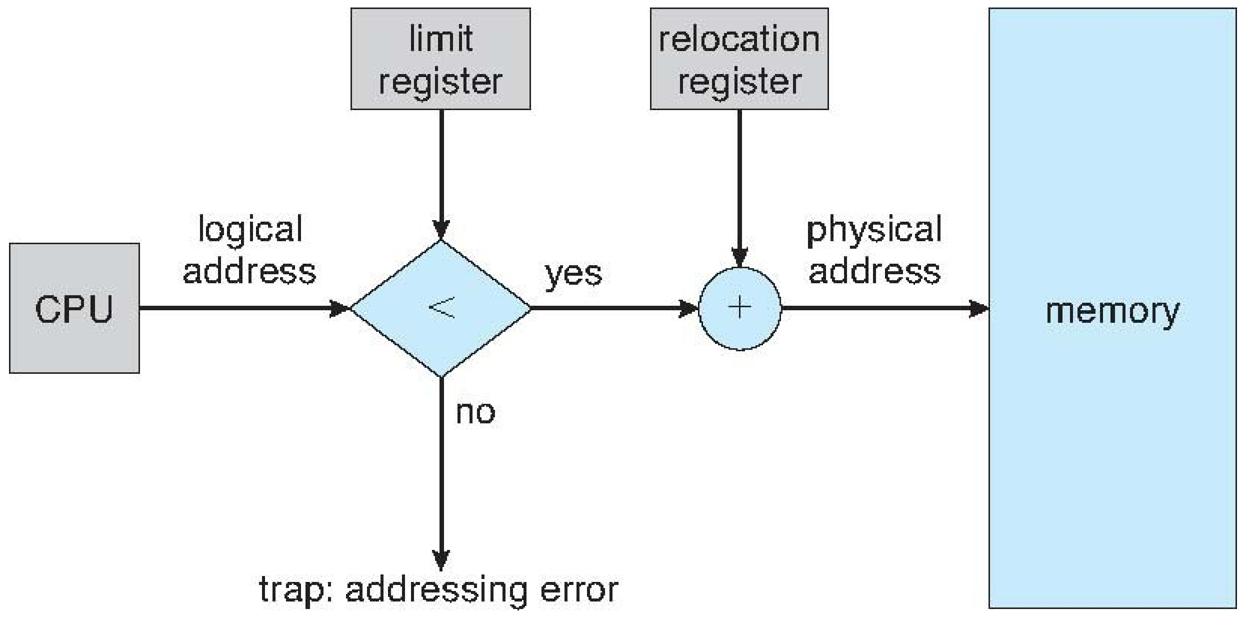
Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- 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
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

Relocation registers

- 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*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers

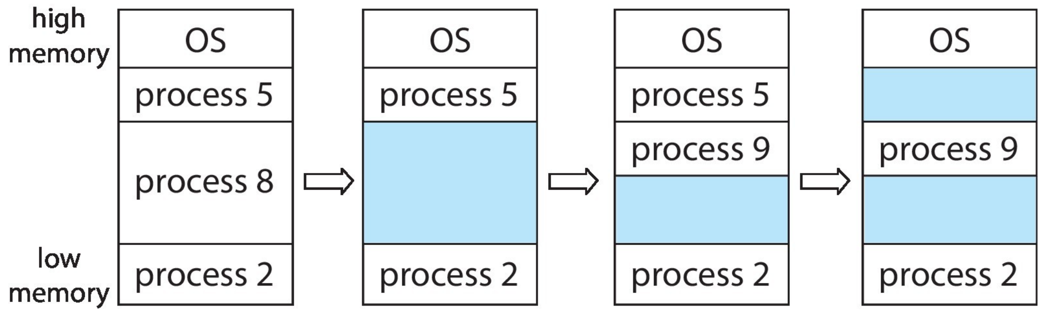


Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector. User processes then held in high memory
  - Many operating systems (including Linux and Windows) place the operating system in high memory
  - Each process contained in single contiguous section of memory

Variable Partition

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

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

Fragmentation

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

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

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

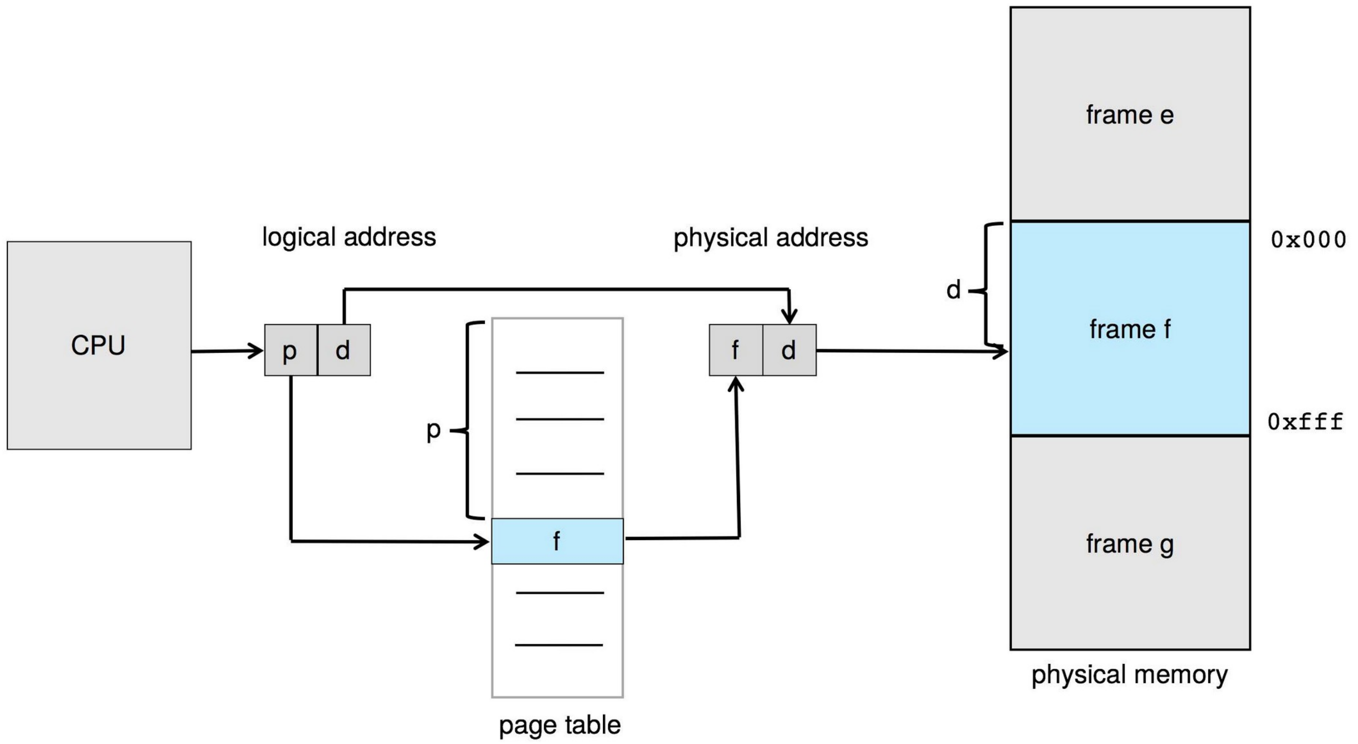
Internal fragmentation

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

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
- Divide physical memory into fixed-sized blocks called **frames**
  - Size of a page is power of 2, between 4 KB and 1GB per page
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). Backing store likewise divided into fixed-sized blocks that are the same size of the 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
- Still have Internal fragmentation

Paging Hardware



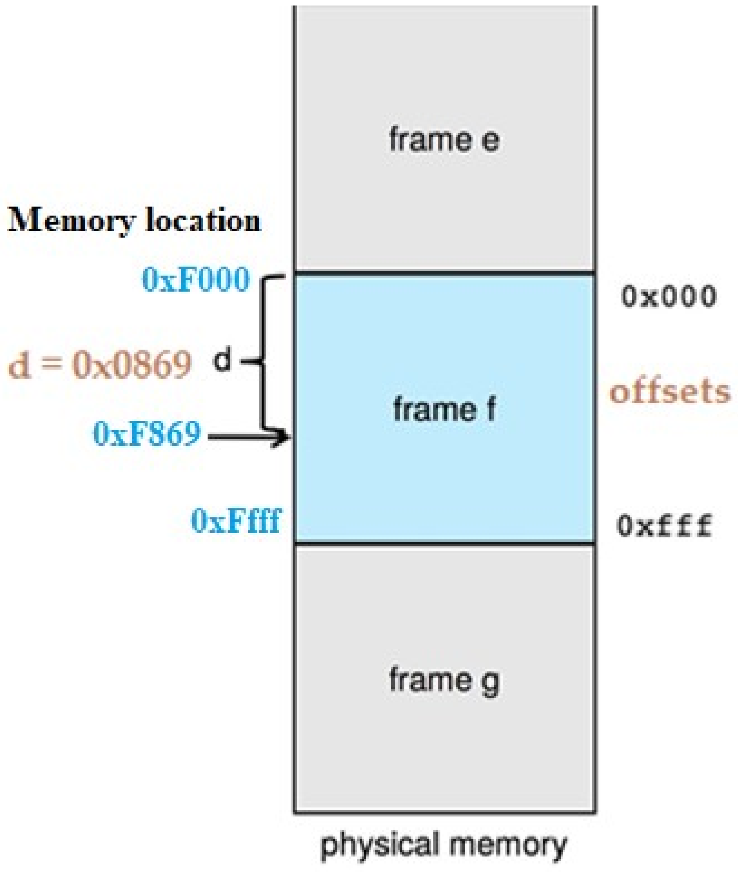
Page Number & Offset

- Paging is implemented by breaking up an address into a page and offset number.
- Page size like the frame size is defined by the architecture
- Every Address generated by CPU is divided into:

page number	page offset
p	d

  - **Page number** (***p***) – used as an index into a per-process **page table**
    - The page table contains the base address of each frame in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit
    - Offset is the location in the frame being referenced.

Offset



When storing and retrieving data, an offset is used to determine the location in which the data is stored.

An **offset** usually denotes the number of address locations added to a base address in order to get to a specific absolute address

For example, a 12-bit offset embedded within certain instructions provided a range of between 0 and 4095 (4096 = 2^{12}) bytes

Assume we want to refer to memory location 0xf867. One way this can be accomplished is by first defining a page with beginning address 0xf000, and then defining an offset of 0x0867.

Dec to hex:
$15_{10} = f_{16}$
$16_{10} = 10_{16}$

$4095_{10} = 4096 - 1 = 1000_{16} - 1 = fff_{16}$

$fff_{16} = 1111\ 1111\ 1111_2$

$4096_{10} = 1000_{16}$

Convert 4096 to hexadecimal:
4096/16 = 256, remainder is 0
256/16 = 16, remainder is 0
16/16 = 1, remainder is 0
1

PAGE SIZE ON LINUX SYSTEMS

On a Linux system, the page size varies according to architecture.

There are several ways of obtaining the page size.

One approach is to use the system call

getpagesize()

Example:

```
#include <unistd.h>
int sz = getpagesize();
```

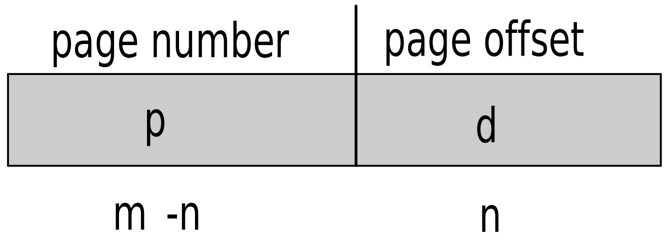
Another strategy is to enter the following command on the command line:

```
getconf PAGESIZE
```

Each of these techniques returns the page size as a number of bytes.

Why are page sizes always powers of 2?

- The selection of a power of 2 makes the translation of a logical address into a page number and page offset particularly easy
- For given logical address space 2^m and page size 2^n
  - If the size of the logical address space is 2^m, and a **page size** is **2^n** bytes, then the high order *m-n* bits of the logical address designate the page number, and the ***n* low-order bits** designate the **page offset**. Thus the logical address is as follows:



Because **each bit position represents a power of 2**, splitting an address between bits results in a page size that is a power of 2.

Translation of Logical Address to Page Number and Page Offset

	page		offset	
	0	0	0	0
page 0	0	0	0	1
	0	0	1	0
	0	0	1	1
	0	1	0	0
page 1	0	1	0	1
	0	1	1	0
	0	1	1	1
	1	0	0	0
page 2	1	0	0	1
	1	0	1	0
	1	0	1	1
	1	1	0	0
page 3	1	1	0	1
	1	1	1	0
	1	1	1	1

Example shows 16 logical addresses.

- $16 = 2^4$
- 2^4 addresses can be stored using 4 bits
- 4 bits represent the 16 logical addresses from 0000 to 1111

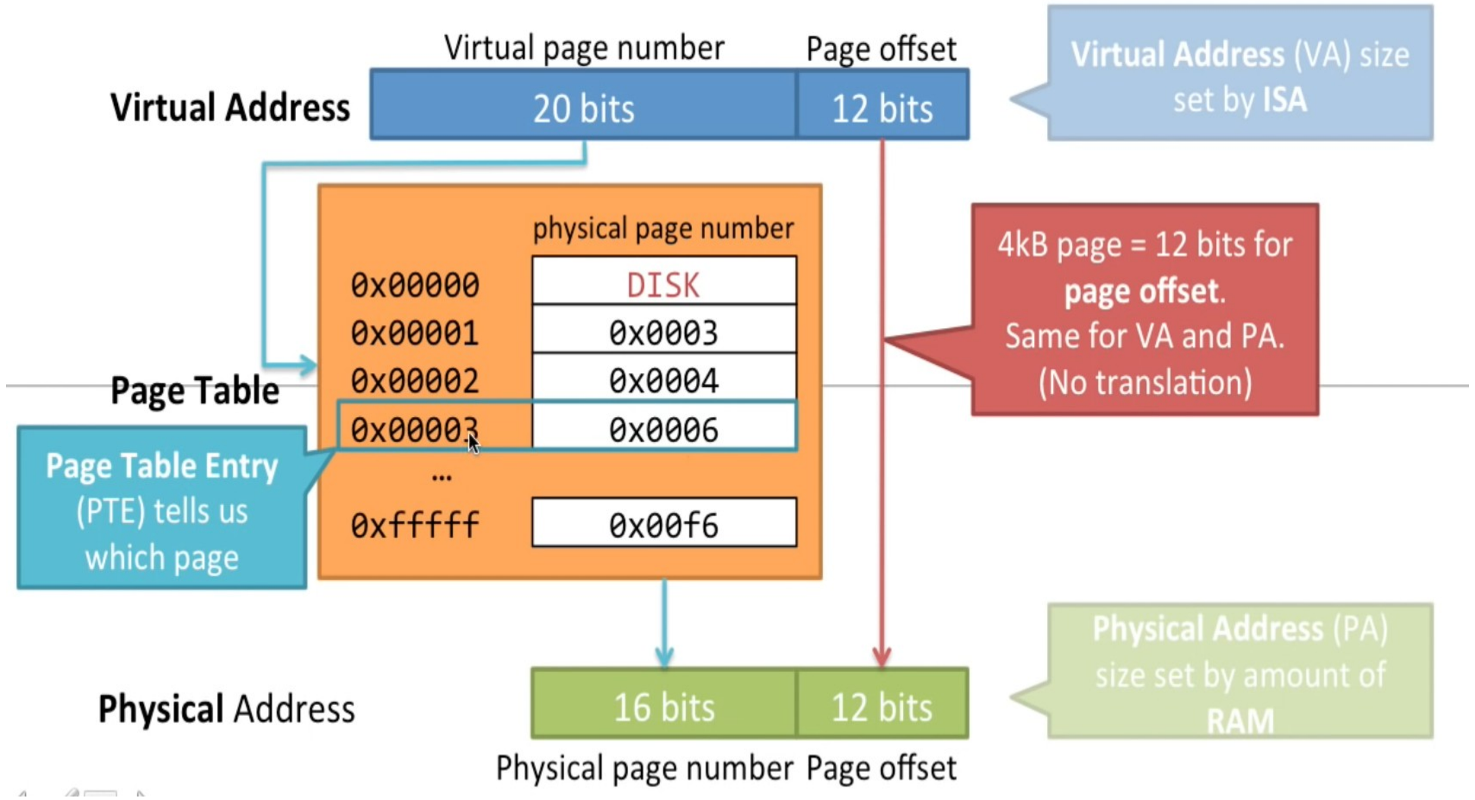
- The higher order 2 bits represent page numbers
  - $2^2 = 4$ pages
  - 2^2 pages from 00 to 11

- The low order 2 bits represent page offsets (i.e., page size)
  - $2^2 = 4$ addresses in each page

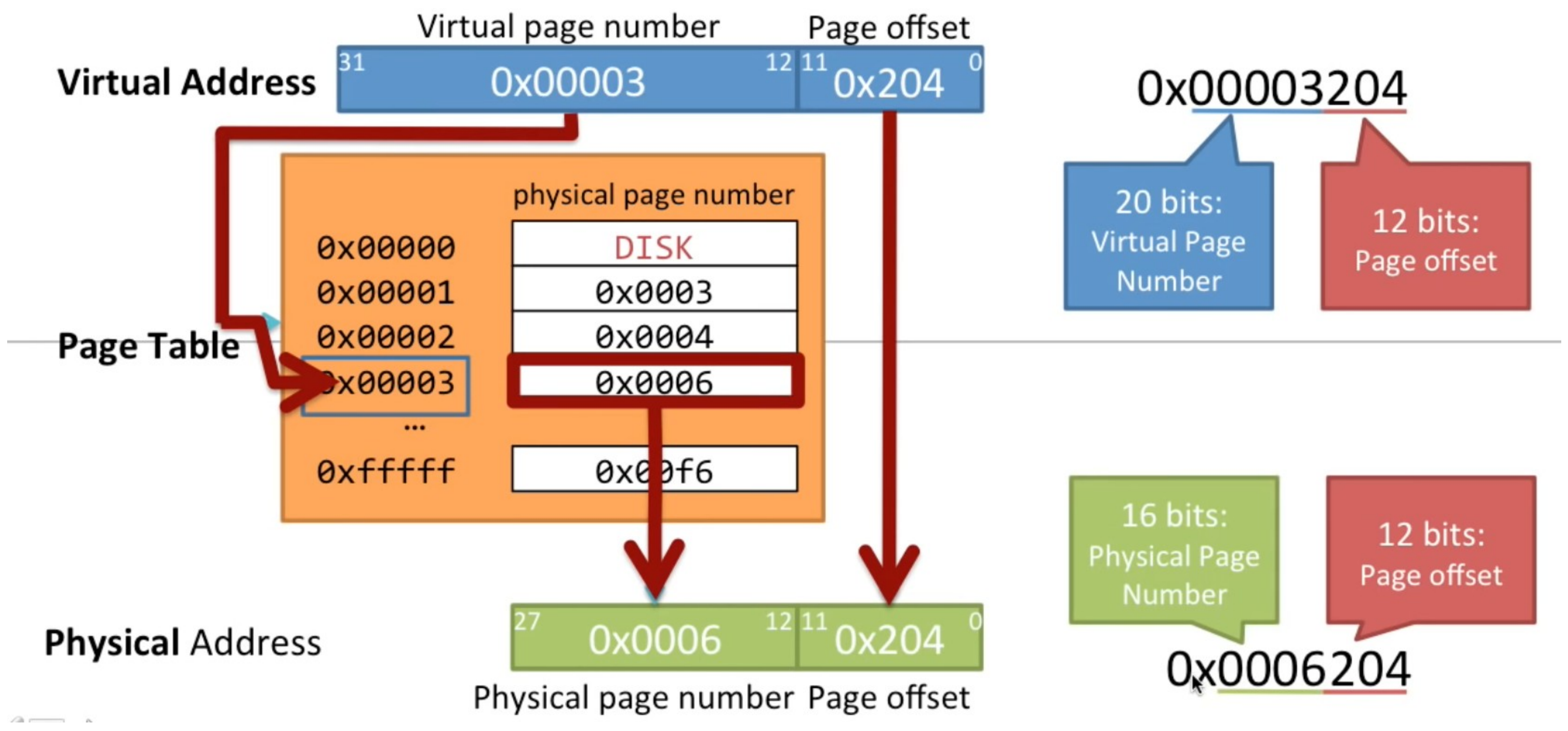
Power of 2 simplifies the translation:
Logical addresses = 16 = 2^m *m* bits
Page size = 4 = 2^n *n* bits
Number of pages = Total addresses / Page size
= $2^m / 2^n = 2^{m-n}$
m bits for addresses, n bits for offsets,
m-n bits for pages

Logical to Physical memory translation

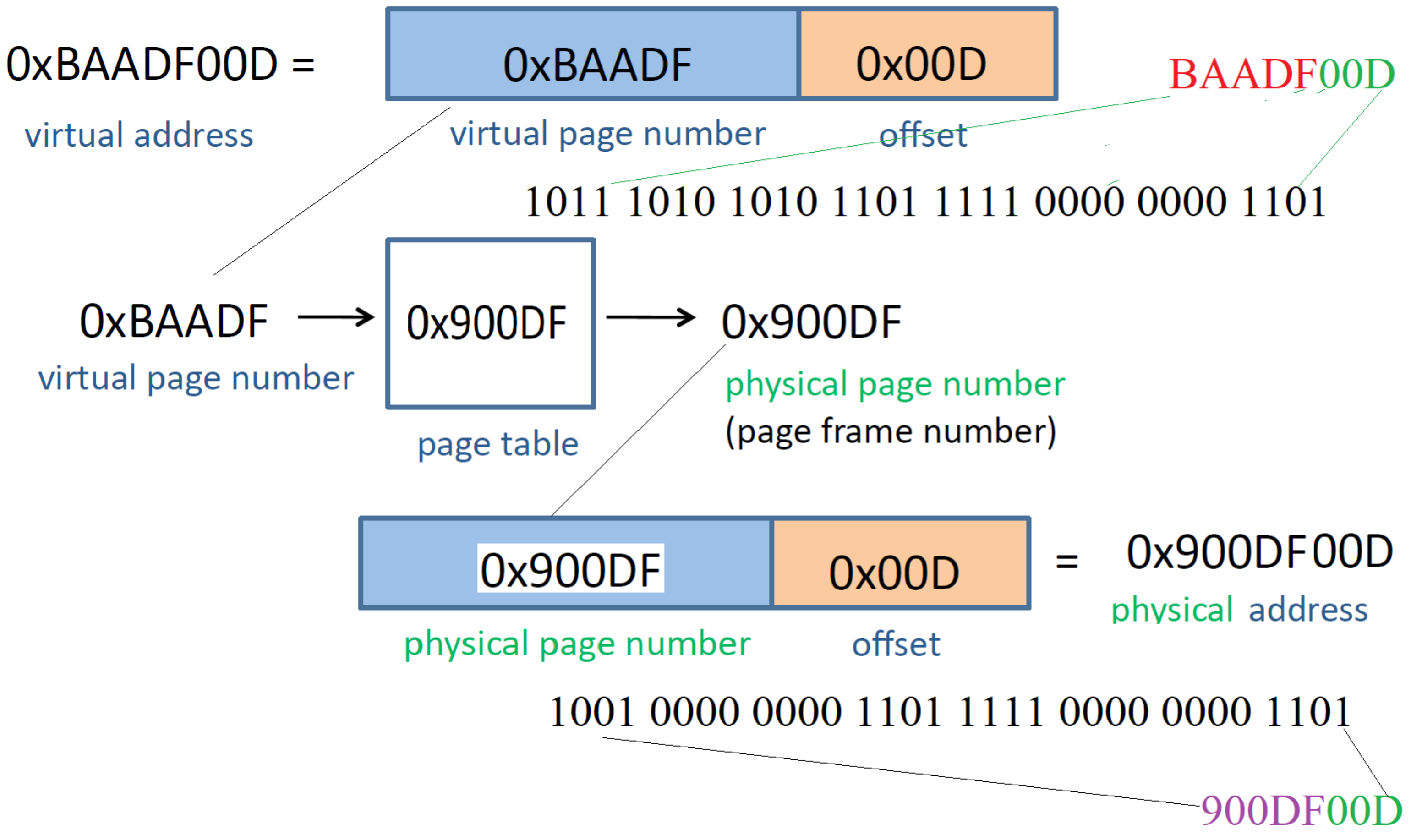
- Virtual address is set by the **computer architecture** (ISA, the Instruction Set Architecture)
- Page size also is determined by the **architecture**. Page size is a power of 2, varying between 4 KB and 1 GB (= $1024^3 = 2^{30}$) bytes per page, depending on the computer architecture.



Logical to Physical memory translation



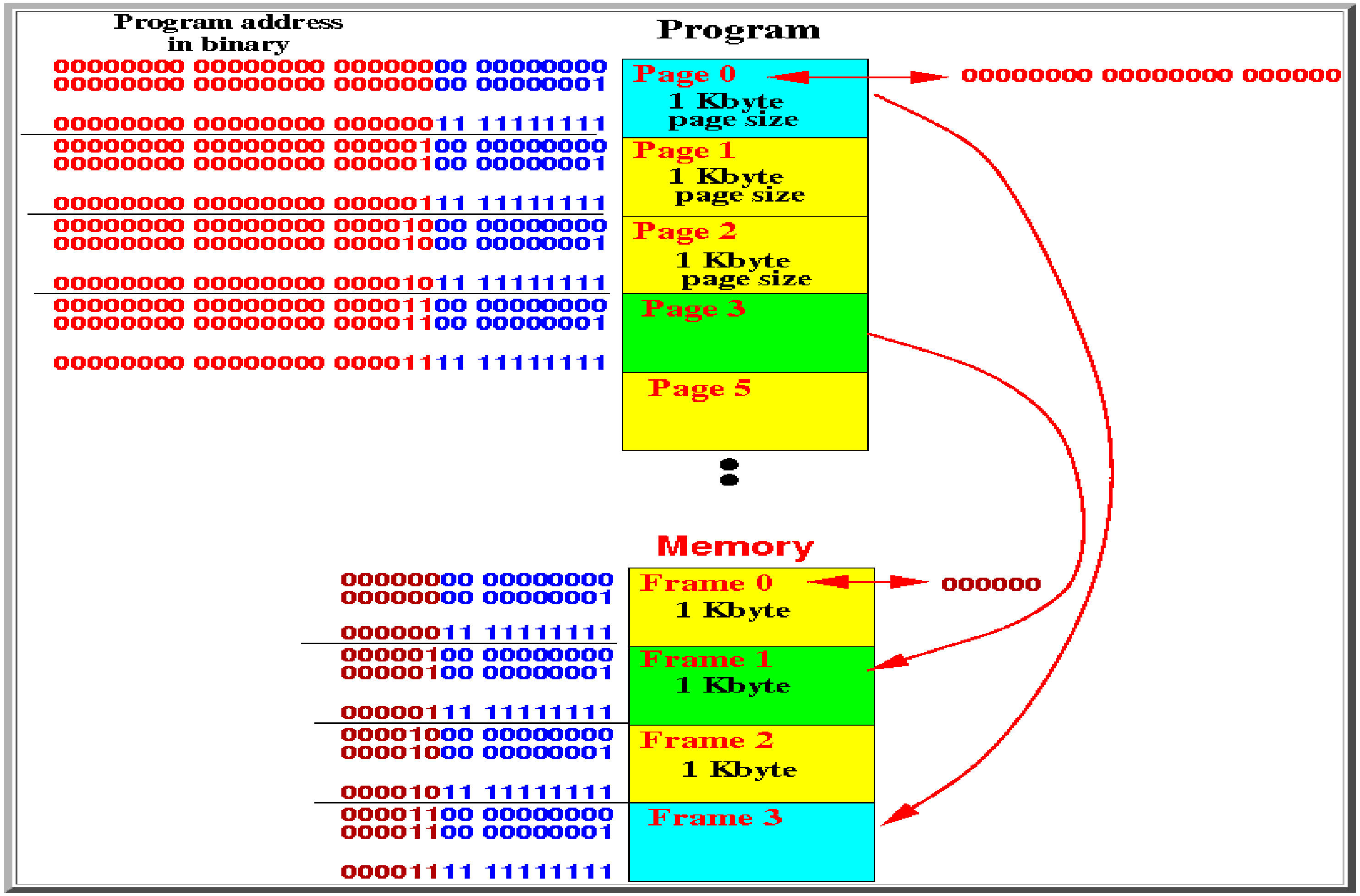
Logical to Physical memory translation



- **Suppose**:

  - **Page 0** of the **program** is **currently** stored in **frame 3** in the **memory**
  - **Page 3** of the **program** is **currently** stored in **frame 1** in the **memory**

The following **diagram** shows the **address mapping** in **detail**:



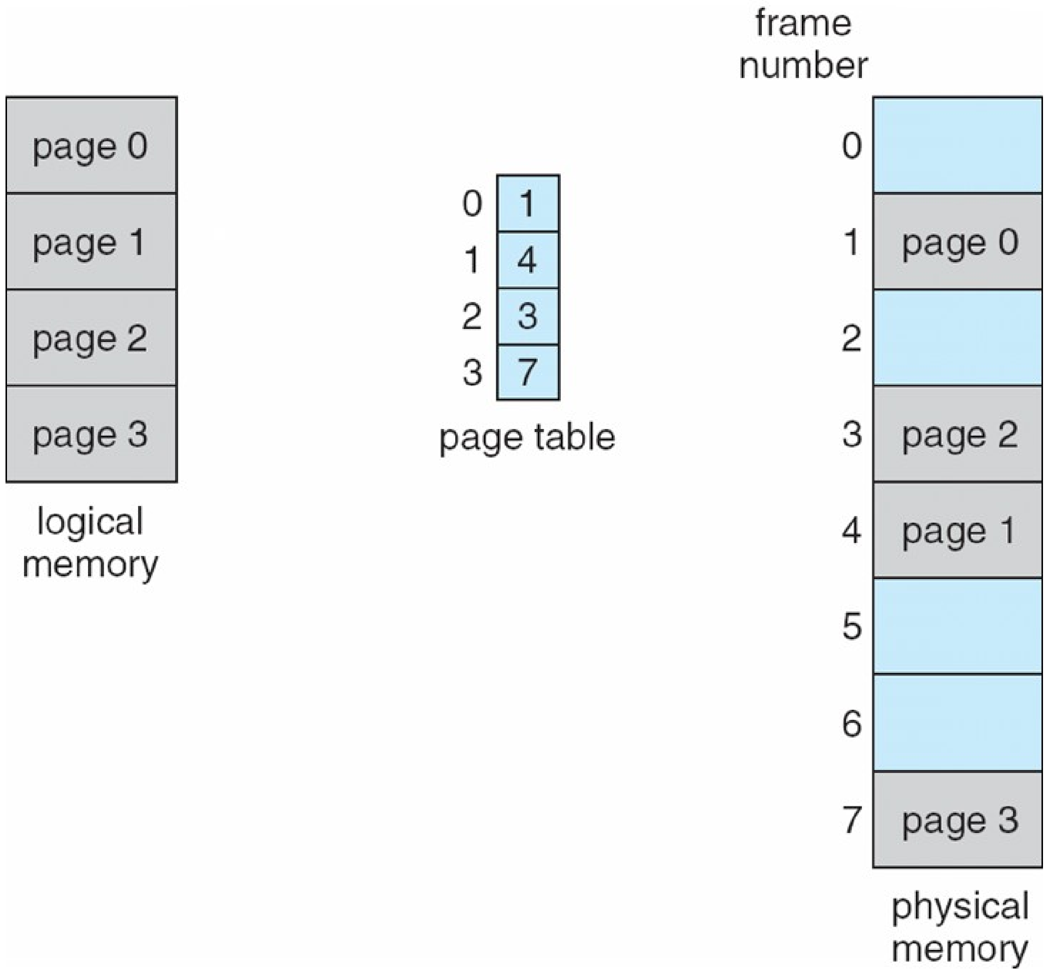
2^{32} = 4 Gigabyte (4,294,967,296 bytes)

2^{16} = 64 KB (65,536 bytes)

Same **sample** of **program address** to **memory address** mapping values:

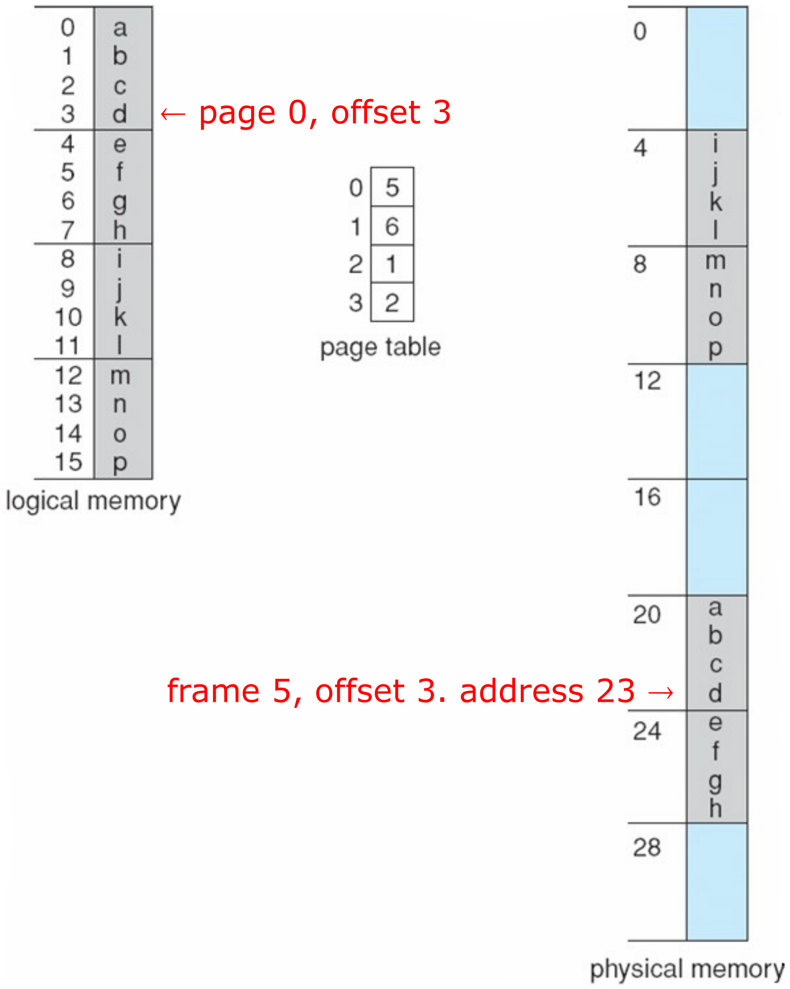
Program Address	---------->	Memory address
00000000 00000000 00000000 00000000		00001100 00000000
00000000 00000000 00000011 11111111		00001111 11111111
00000000 00000000 00001100 00000000		00000100 00000000
00000000 00000000 00001111 11111111		00000111 11111111

Paging Model of Logical and Physical Memory



Paging Example

- Logical address space = $2^m = 2^4 = 16$ where m is the total number of bits in logical address
- Page size = $2^n = 4$ bytes where n is the number of bits in a page
  n low-order bits designate the page offset



Physical memory of $2^5 = 32$ bytes
Number of frames = 32 bytes / 4 bytes in each frame
= 8 frames

Logical memory mapping to physical memory:

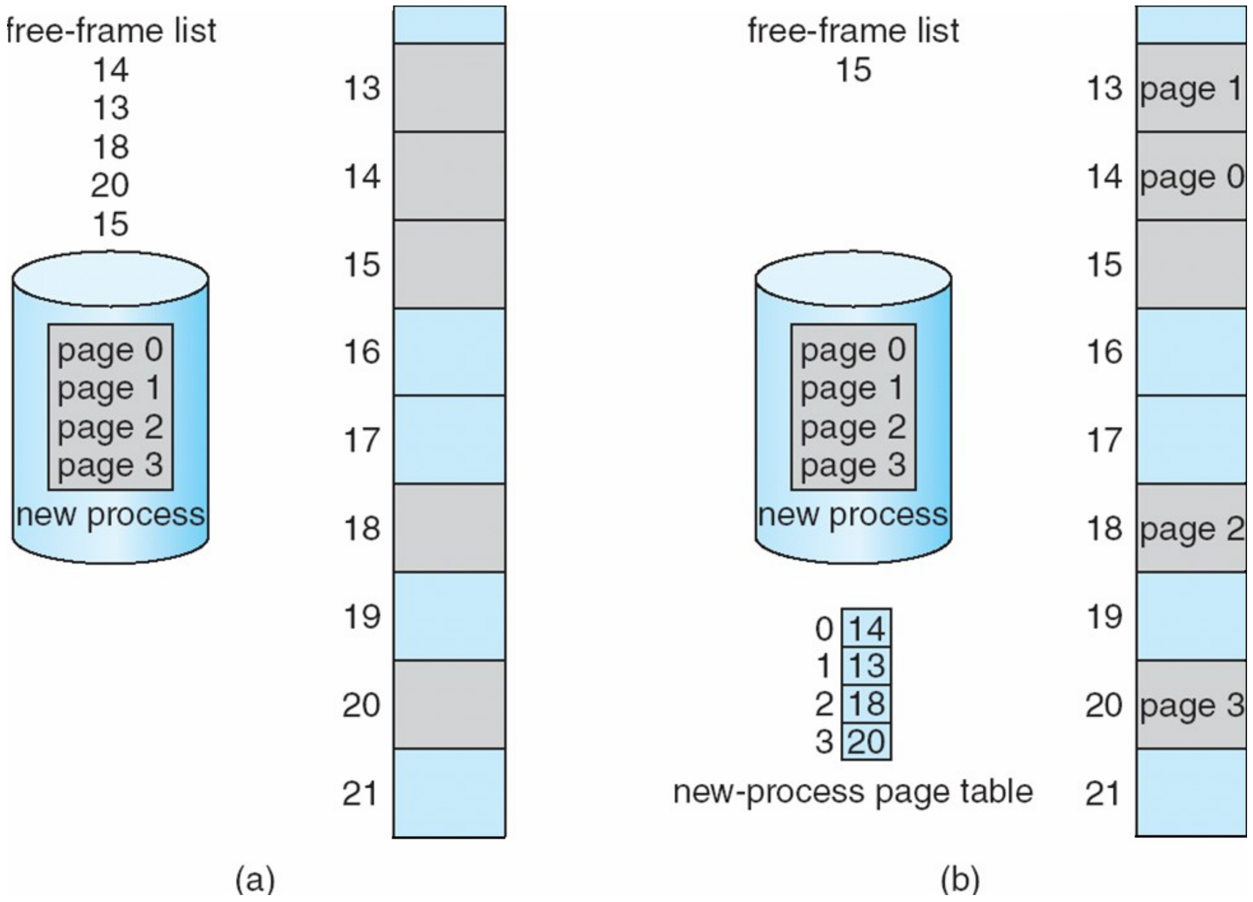
1. Logical address 0 is page 0, offset 0. Indexing page table finds page 0 in frame 5.
   Thus, logical address 0 maps to physical address 20 i.e., $(5 \times 4) + 0$
2. Logical address 3 (page 0, offset 3) maps to physical address 23 i.e., $(5 \times 4) + 3$
3. Logical address 4 is page 1, offset 0;
   page 1 is mapped to frame 6
   Thus, logical address 4 maps to physical address 24 i.e., $(6\times 4) + 0$

Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes

  72766 / 2048 = 35 72766 mod 2048 = 1086
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames



Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) 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
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

Translation Look-Aside Buffer

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

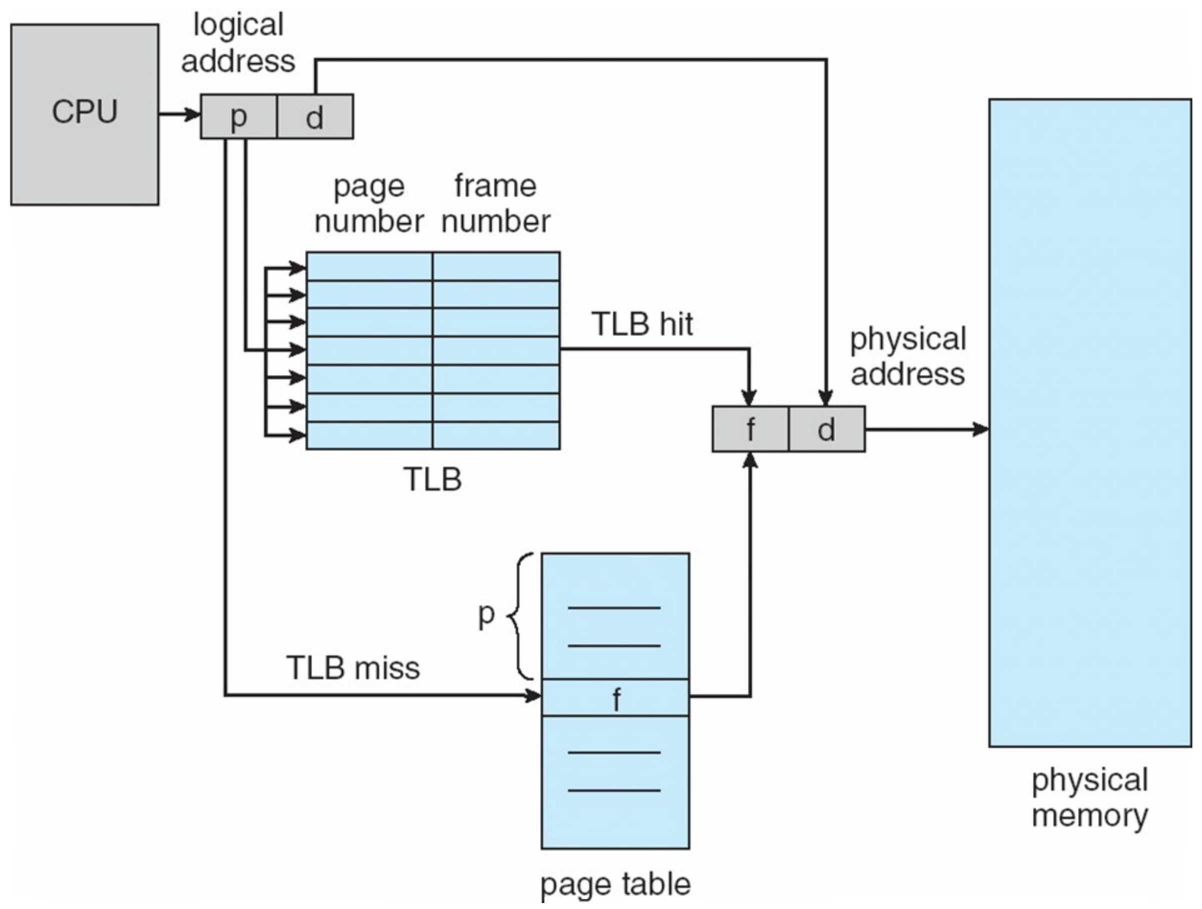
Hardware

- Associative memory – parallel search

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

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

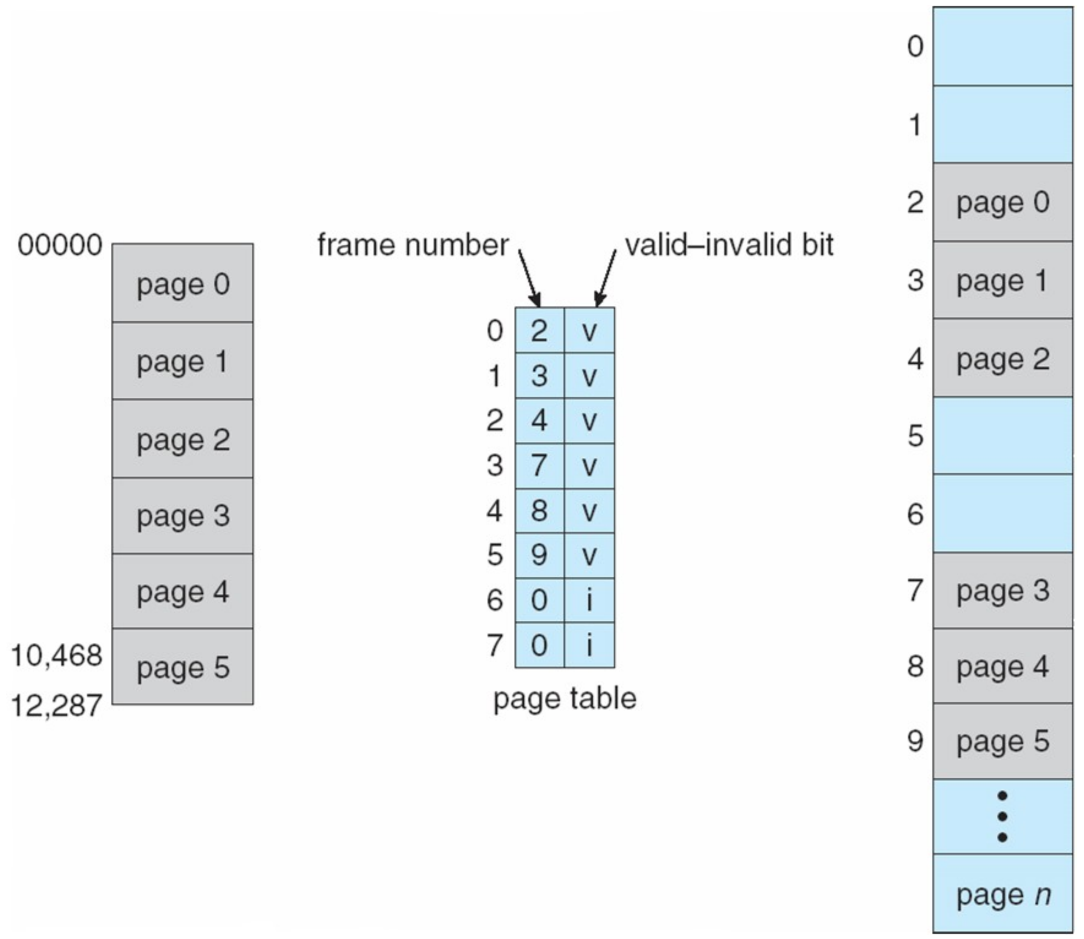
  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate 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 page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

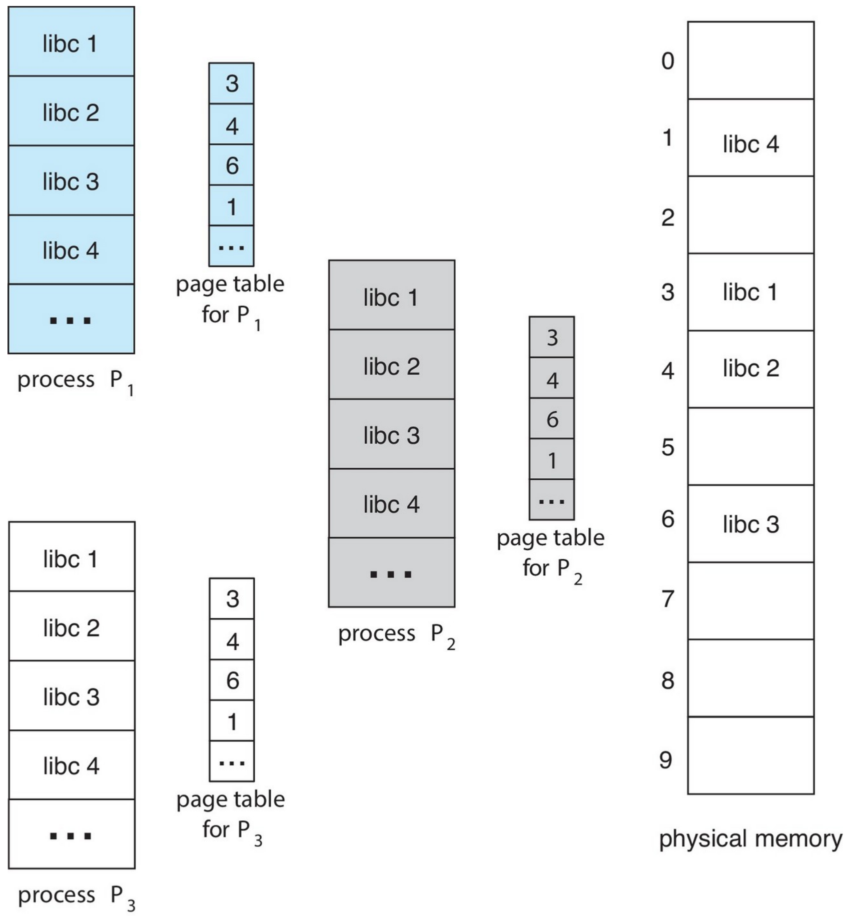
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **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

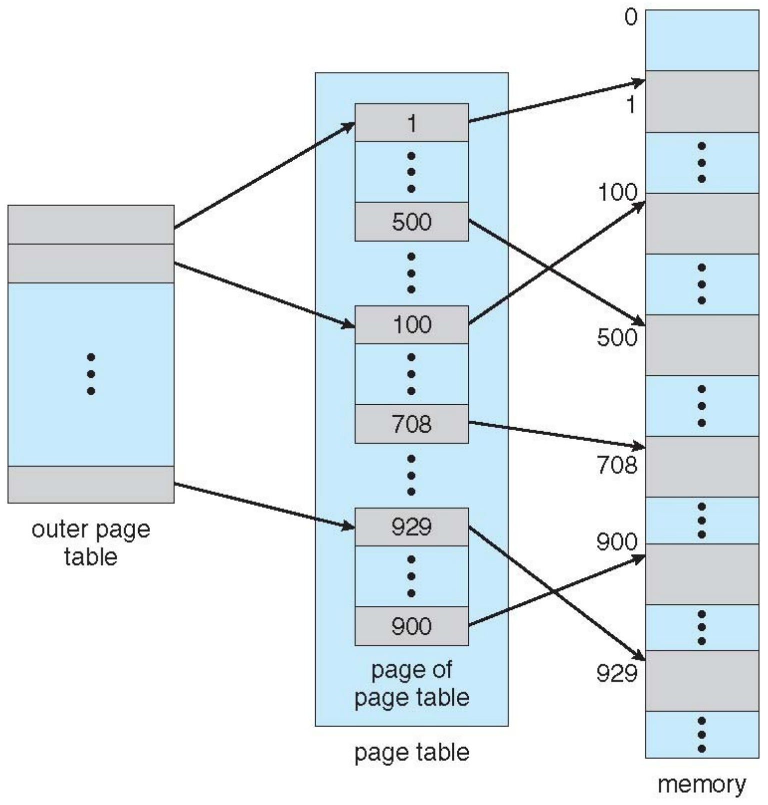
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes ℗ each process 4 MB of physical address space for the page table alone
    - Don’t want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables

Hierarchical Paging

- Most modern computer systems support a large logical address space(2^{32} to 2^{64}).
- In such an environment, the page table itself becomes excessively large.
- For example, consider a system with a 32-bit logical address space.
- If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$).
- Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- Clearly, we would not want to allocate the page table contiguously in main memory.
- One simple solution to this problem is to divide the page table into smaller pieces

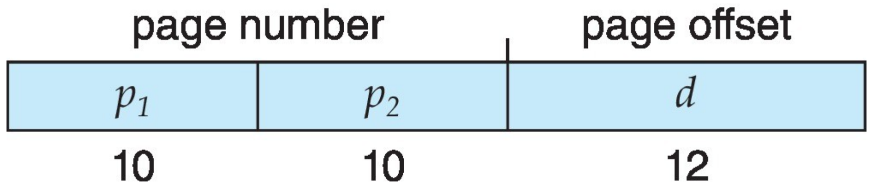
Hierarchical Page Tables

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



Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

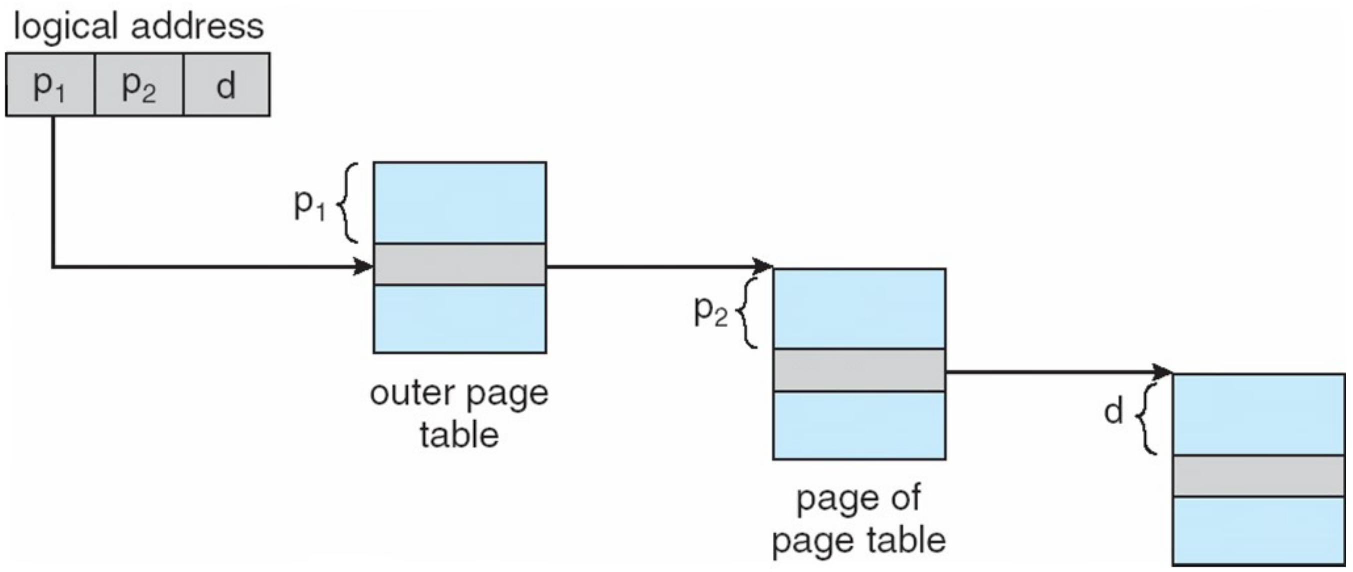


where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

- Because address translation works from outer page table inward, this scheme is known as **forward-mapped page table**

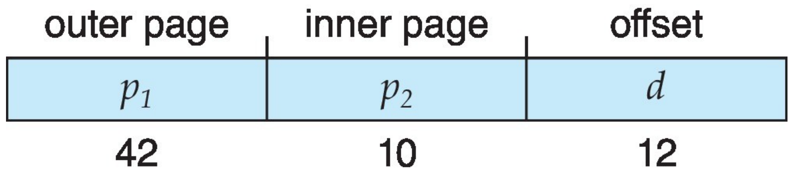
Address-Translation Scheme

Address translation for a two-level 32-bit paging architecture



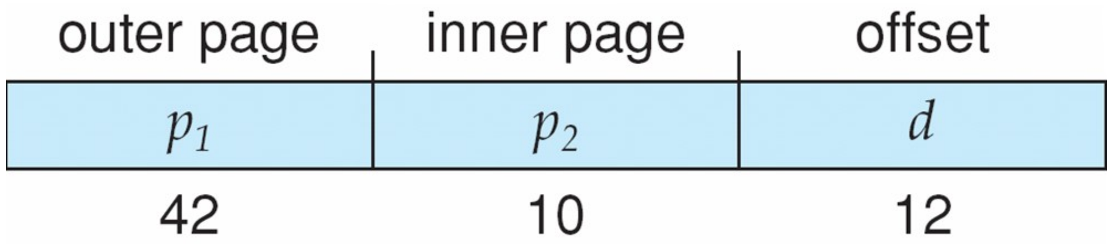
64-bit Logical Address Space

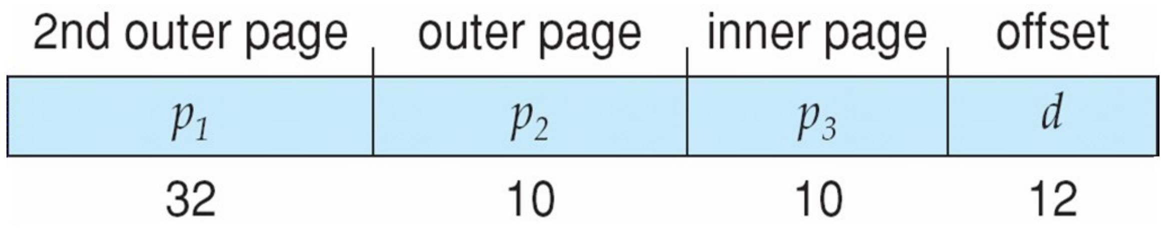
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12}) — $2^{64}/ 2^{12} = 2^{52}$
  - Then page table has 2^{52} entries — i.e., 64 – 12 = 52
  - If we use a two-level paging scheme, inner page tables can be one page long, or contain 2^{10} 4-byte entries — $2^{10} \times 2^2 = 2^{12}$
  - Address would look like — $2^2 =$ 4 bytes per entry (i.e., 4 bytes in each page)



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

Three-level Paging Scheme





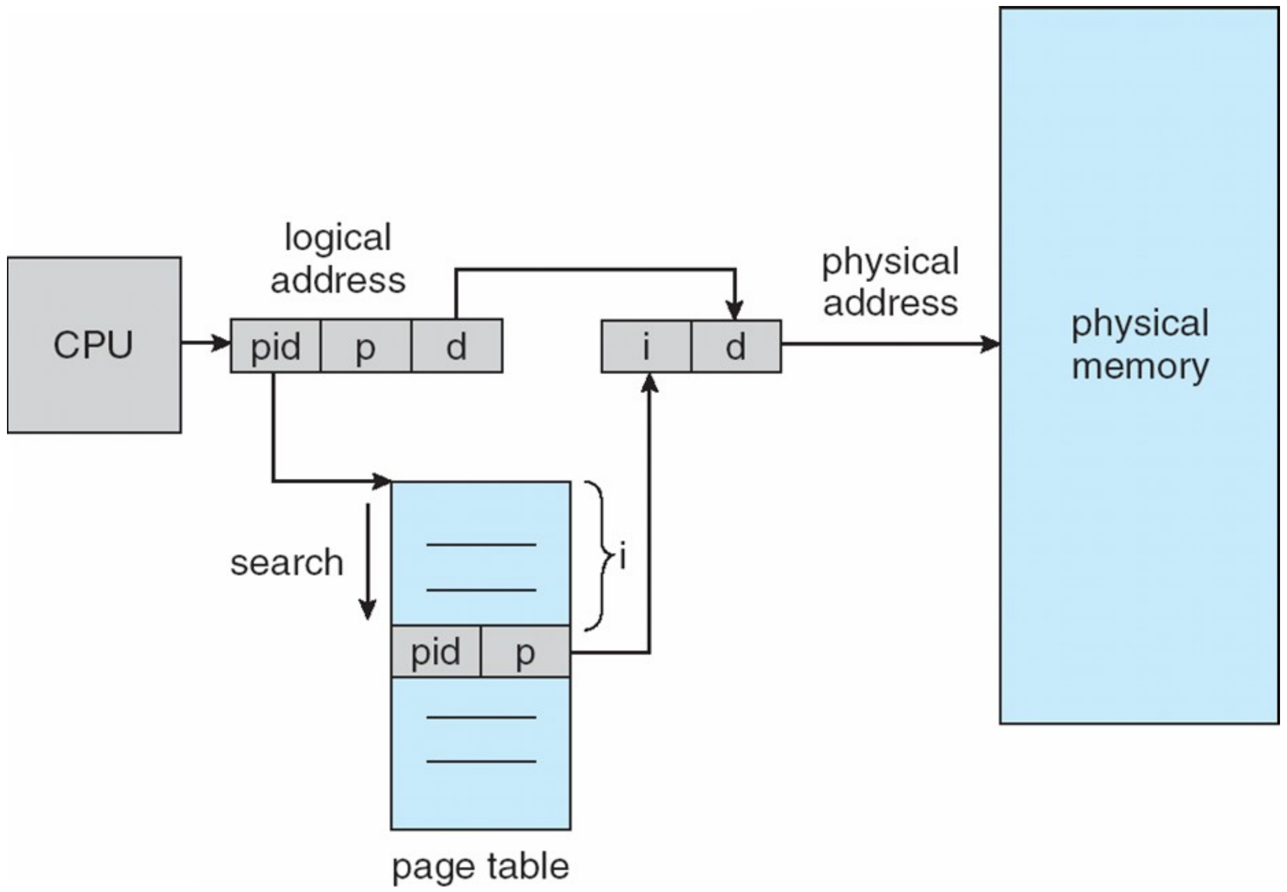
Inverted Page Table

Inverted Page Table is the one single global page table maintained by OS for all processes.

- Number of entries is equal to the number of frames in main memory.
- Frames are the indices and the information saved inside the block will be Process ID and page number.

- 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
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

Inverted Page Table Architecture



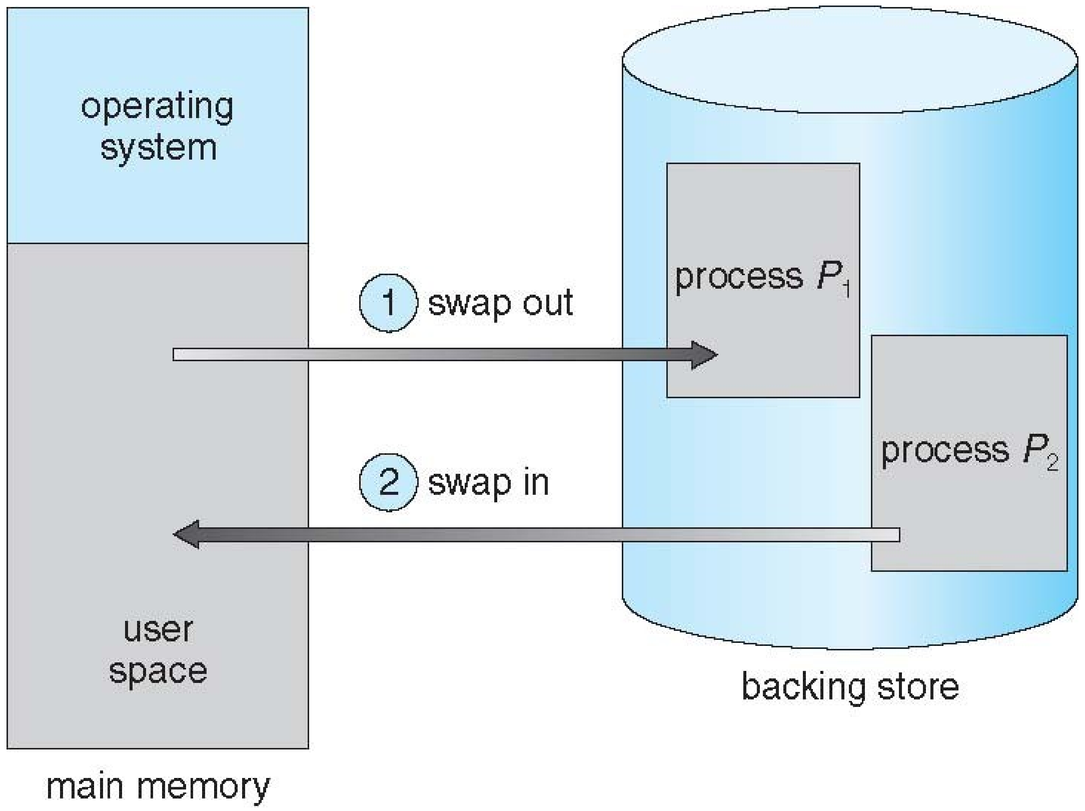
Swapping

- 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
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., 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



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

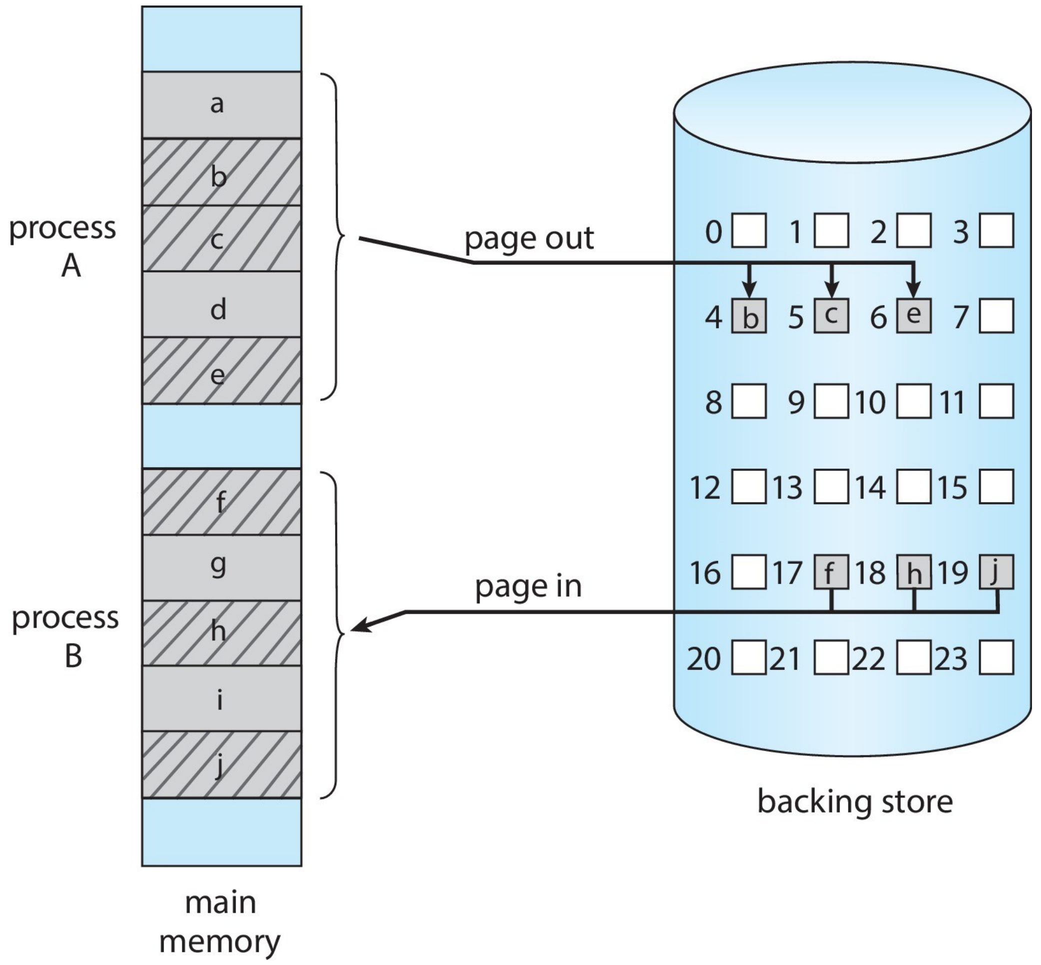
Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can’t swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

Swapping with Paging



End of Chapter 9

Memory

Abbr.	Prefix name	Decimal size	Size in thousands	Binary	Address size
K	kilo-	10^3	1,000	$1{,}024 = 2^{10}$	10
M	mega-	10^6	$1{,}000^2$	$1{,}024^2 = 2^{20}$	20
G	giga-	10^9	$1{,}000^3$	$1{,}024^3 = 2^{30}$	30
T	tera-	10^{12}	$1{,}000^4$	$1{,}024^4 = 2^{40}$	40
P	peta-	10^{15}	$1{,}000^5$	$1{,}024^5 = 2^{50}$	50
E	exa-	10^{18}	$1{,}000^6$	$1{,}024^6 = 2^{60}$	60