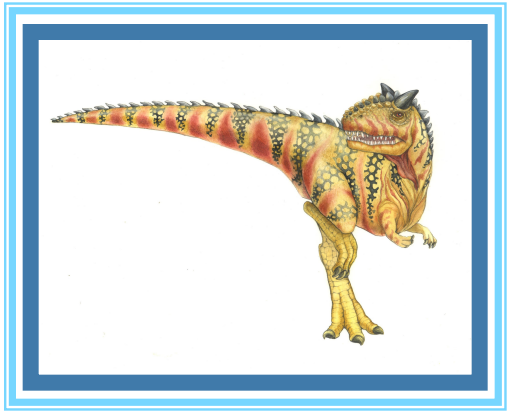
Chapter 9: Main Memory



Chapter 9: Memory Management

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

cpu is not able to run instructions, it would be too slow. You can only instructions when it comes from main memory. The program is brought from main memory and then it runs.

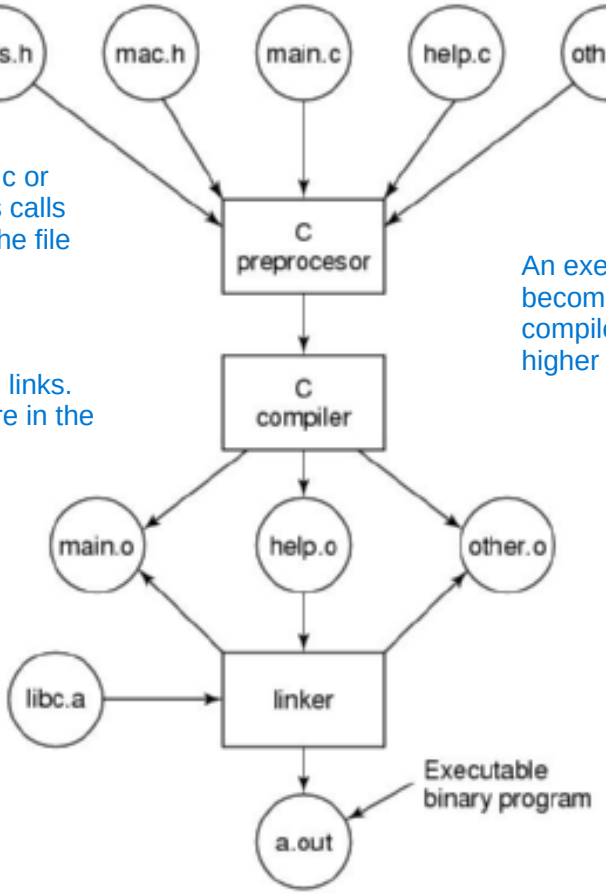
we have read adnd write for 2 reasons.

registers are fast, but we don't have many registers in the cpu but they are fast. CPU frequency is typically in order of giga hertz. Typically in order to read from main memory it will take a long time.

The Model of Run Time

In main.o we can have a call to main.c or c library. So every object file includes calls or references to data thar are not in the file hence there are memory addresses.

The linker is taking a notes of all this, and links. Putting pieces together. All instructions are in the executable.

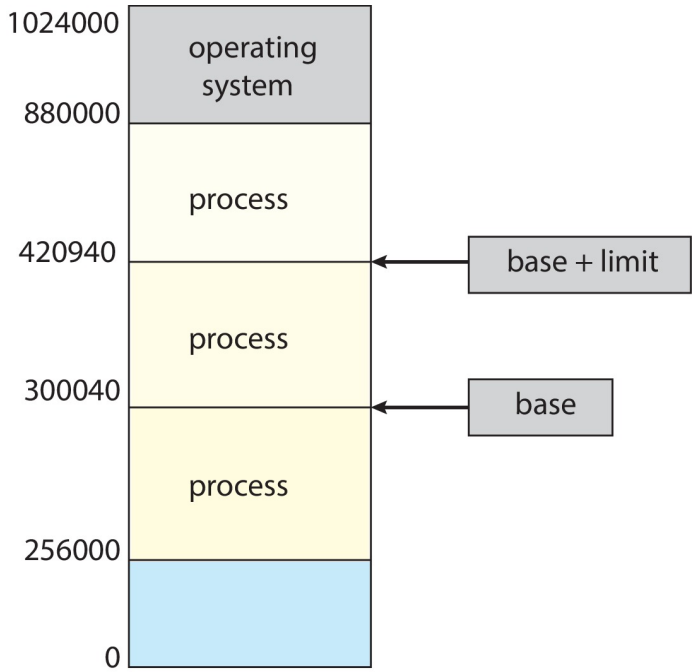


An executable running program, before becoming a.out has been processed by compiler which is essentially translating to higher language like to machine language/

Figure 1-30. The process of compiling C and header files to make an executable.

Protection

- Need to censure that a process can access only access those addresses in it address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



The OS here is placed at top but sometimes we can see its placed lower. But we see three processes assigned together in between. We need some constraints so that not all processes read and write together but have some limits.

Multiple Programs Without Memory Abstraction

We have two assembly level programs.
Every program has being written or compiled at length.
Both the gray and white program include instructions that have nothing to do with addresses. Jump24 because we want to go to mov instruction. jump 28 cuz of cmp instructions.

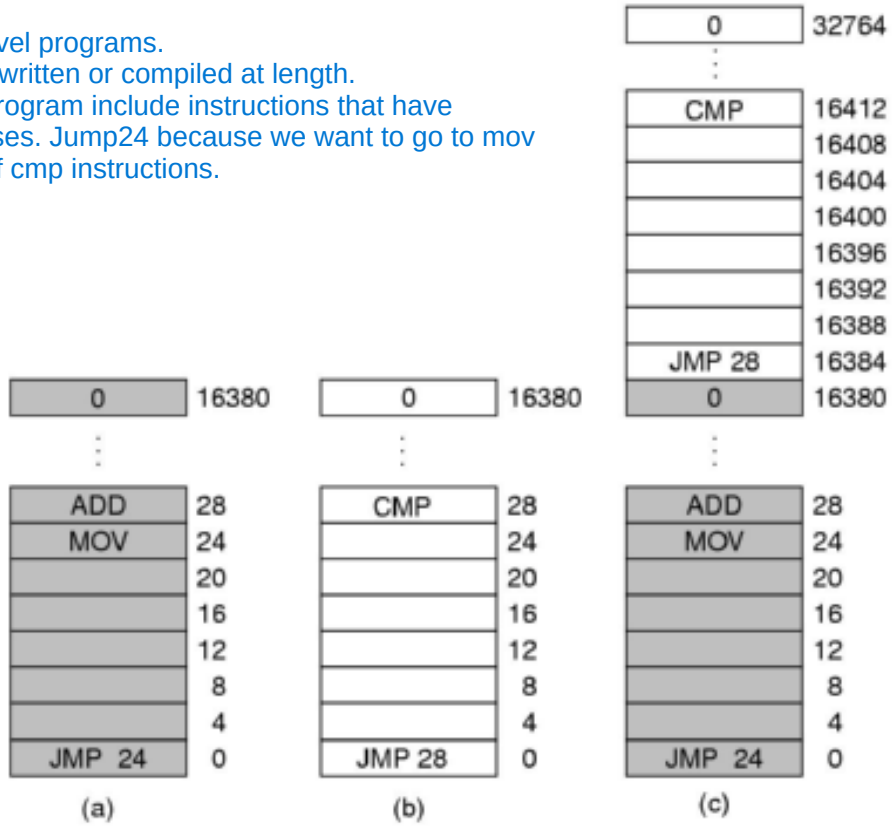
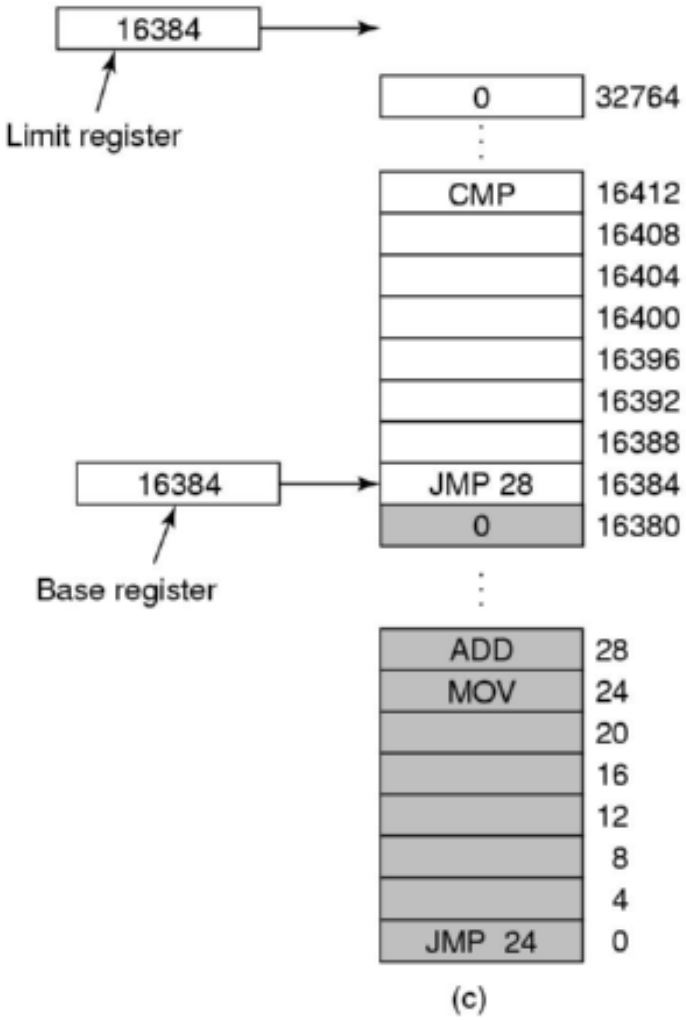


Figure 3-2. Illustration of the relocation problem.

Base and Limit Registers



Solution is dynamic relocation where in the software you still write jump28 but you will still keep base register. The jmp 28 will be added to 16384.

Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Address binding means when are we goin to define addresses of things like functions or data. Like names. The cpu wants to call things by mem addr and programmer wants names. When and how do we bind?

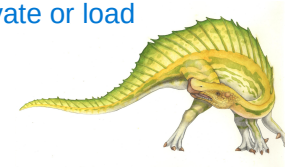
The 0 location is too crowded many programs want to stay at 0. Source codes have symbolic addresses. All programms indivisually have addresses that would be conflicting if in main memory. The linker or the loader will decide the real and final addresses. The absolute address is the final address. Can be decide by the linker when you run the executable when you know where it will be stored.

Linker produces an executable.
we don't download a source. if the executables already knows the addr

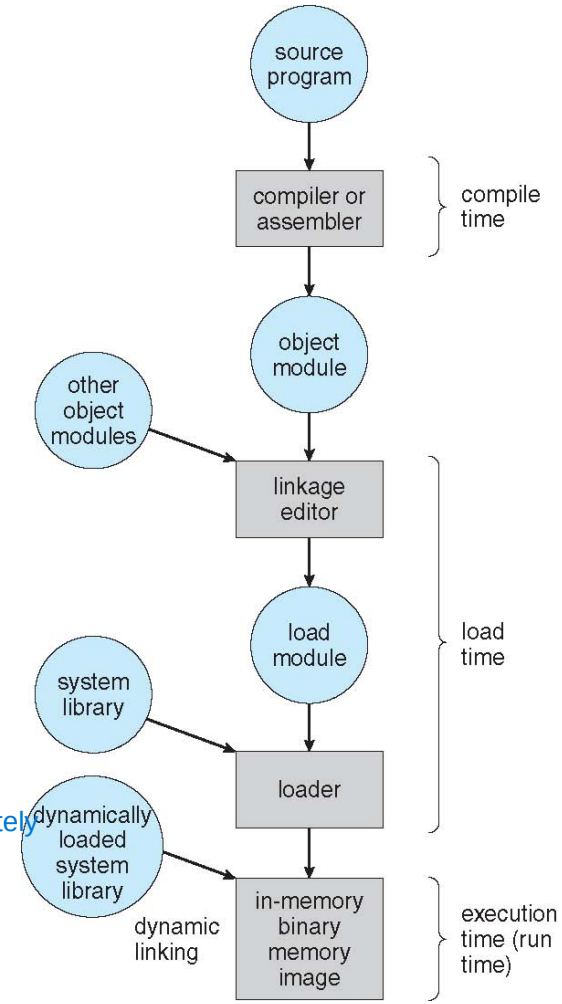
Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **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)

Binding of instruction to memory can happen at compile time. Sometimes compilation and links are put together and called build. The executable knows the address. The load time mean the loader receives the request, start an instance of executable. Which means we should be able to do this.
at execution time: when we load we are immediately going to load. Load time and execution time is kinda similar. So what's the difference between 2, basically here we mean execution not of the full program but part of the program. Suppose your program can be seen as a set of func, at some point of time you are running inside a functions execution time means it could be that we want to just activate or load just part of the program or just to bind the part of the program that we are going to use.



Multistep Processing of a User Program



system library is going to be loaded when you double clicked.

dynamic is not going to be loaded immediately

The loaders module in loading in the main memory.

Logical vs. Physical Address Space

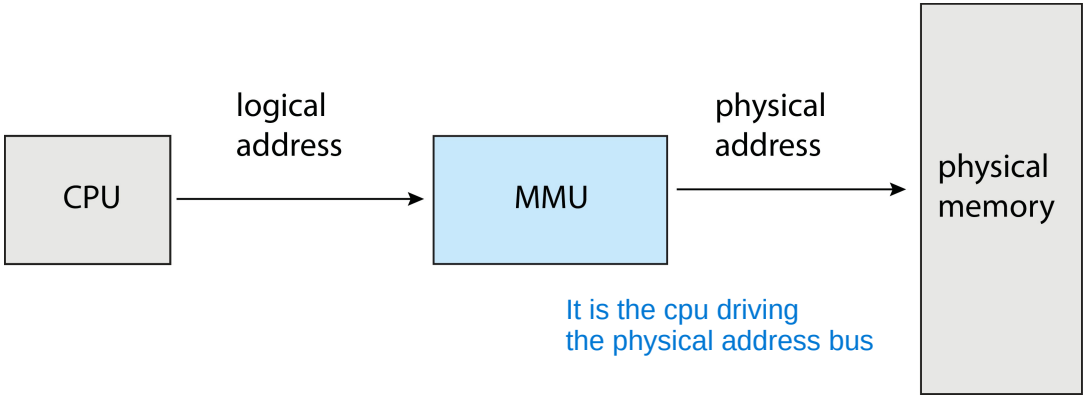
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical 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
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Logical address will start from 0. are the ones that can be written in the software, in machine code part of the system. for the software

physical are used in the address bus and go in the main memory to we must change something. the difference from logic and physical is addition.

Memory-Management Unit (MMU)

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



- Many methods possible, covered in the rest of this chapter

When a given register is asking loads from the memory, the memory register are going to be loaded with the physical address.

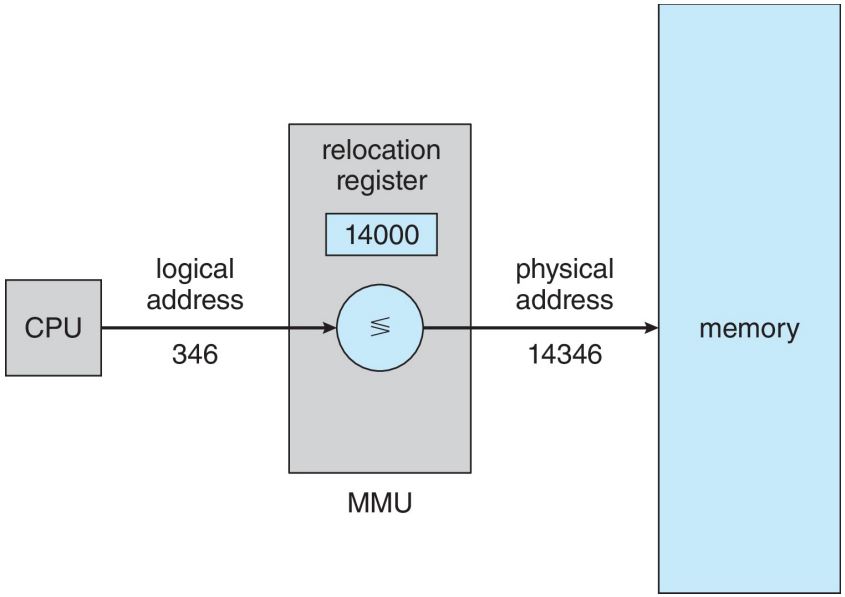
the mmu is basically a plus and comparator, you have logicall add and the relocation in .....

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

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



Dynamic Loading

- The entire program does need to be in memory to excute
- 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
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

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

- See also:

  Linking & Loading

  CS-3013 Operating Systems - Hugh C. Lauer

  WPI

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
  - Each process contained in single contiguous section of memory

In memory you should have both os and processes.

the simplest solution

Contiguous allocation is a method used in computer science and operating systems for allocating disk space to files. In this method, files are stored on disk as contiguous blocks, meaning that all blocks belonging to a file are stored together in consecutive sectors on the disk. This contrasts with non-contiguous allocation methods like linked allocation or indexed allocation, where files are stored in fragmented or scattered locations on the disk.

Contiguous Allocation (Cont.)

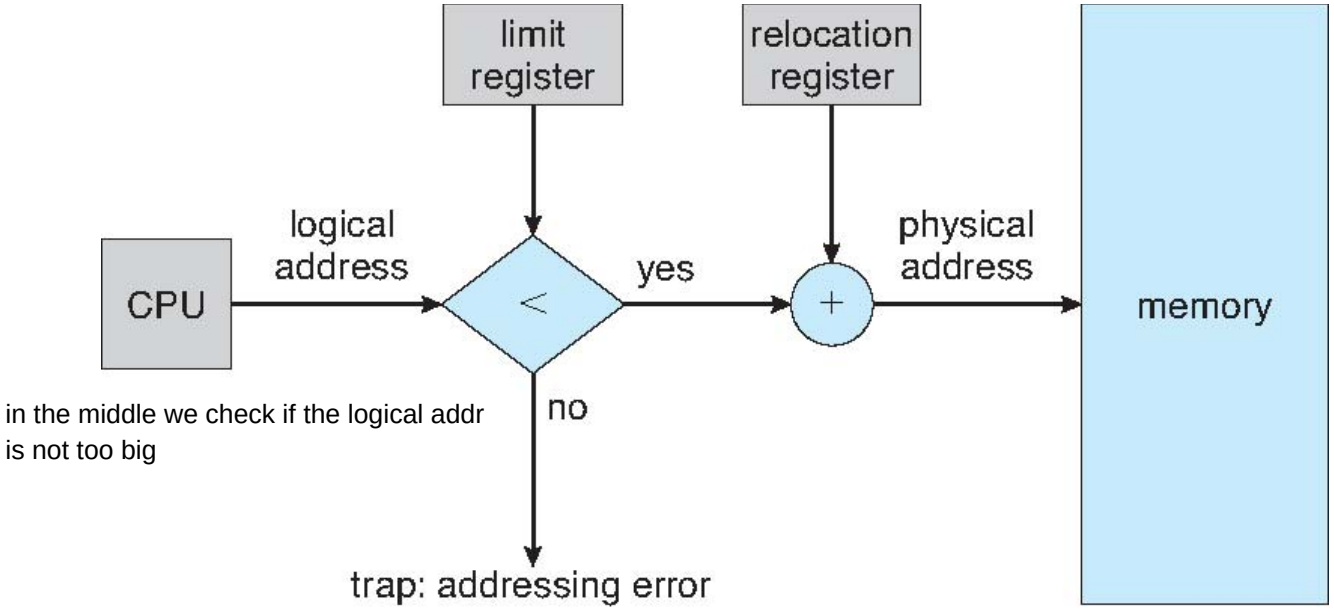
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data

The registers are key in order to provide relocation.

  - 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

We can also think about moving a process from one place to another its technically possible! By changing the base register to be added to logical addressess

Hardware Support for Relocation and Limit Registers

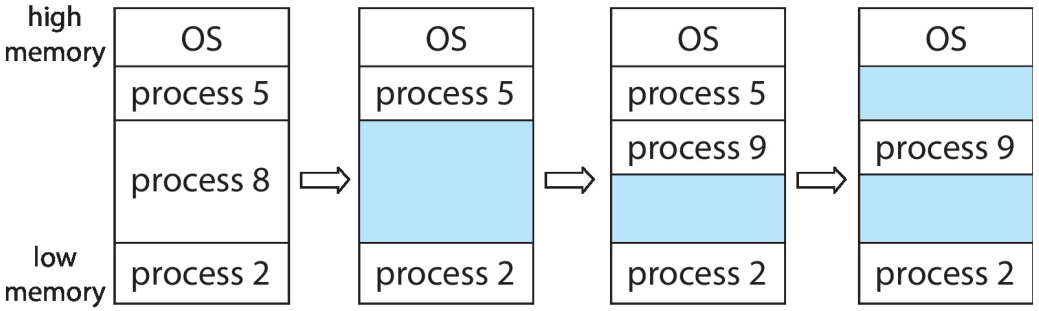


in the middle we check if the logical addr is not too big

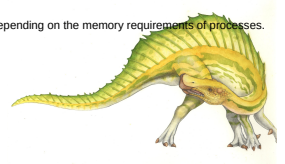
Variable Partition

- 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)

The consequence of variable, when a p ends, there is free memory. You need another p to start which is small like p9. When you have random alternation of processes starting and ending and each is taking random variable size. Similar situtaion in dynamic memory allocation where each time you allocate the size can be random.



Variable partition refers to a memory management scheme used in operating systems to allocate memory to processes dynamically. In this scheme, the available memory is divided into variable-sized partitions, and each partition can accommodate a single process. The size of these partitions can vary depending on the memory requirements of processes.



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

First-fit:

This algorithm allocates memory by selecting the first available hole (block of memory) that is large enough to accommodate the process.

It is relatively fast since it stops searching for a suitable hole as soon as it finds one that fits the process's size.

However, it may lead to fragmentation, as it might allocate a larger hole than necessary, leaving behind smaller unusable holes.

Best-fit:

The best-fit algorithm allocates memory by selecting the smallest hole that is large enough to accommodate the process.

This strategy requires searching through the entire list of free holes to find the smallest suitable one, unless the list is ordered by size.

Fragmentation

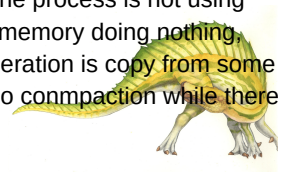
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals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

As a result of contiguous and variable size, we require fragmentation.

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

compaction happens when you defraction of your disk. Imagine an appartment that is full, and you wanna renovate The best solution same as disk, take another disk or apartment put everything there and do the job. So take another disk (ram memory) and then come back here. Well we don't have the free extra disk so we use different chambers of try finding space via defragmentation. defragmentation is expensive because you don't have enough space to move to another part so we have to shift, which is not an easy job. There is additional problem, IO problem. When time sharing and concurrency was introduced one of the motivation was the following: lets consider single cpu for now. 1 cpu, could be useed by one p at a time, a sequenc of p no conc, if the process was not doing I/O. While the IO is happening the cpu is idle, when waiting for input, or data from network the cpu is idle. Waiting for IO means no competition for process. Concurrency is not only motivated by needing many conc task also by the fact that the cpu can do it. cuz the process is not using the cpu all the time. In general you have processes doing IO. What is the process is doing while a IO ongoing, the p is in wait state in memory doing nothing, the cpu is now dedicated to another process and somebody is taking care of IO, a hardware or another software is doing IO. An IO operation is copy from some memory location (RAM) to disc, or disc to memory(output). Something is copying data from meory to disc or vice versa. Conc if you do conmpaction while there is io going basically you destroying data because you are moving data while it needs it during io.



Paging

Paging means no variable size

if all processes had the same size then good, in the end not only non variable but also noncontiguous

physical memory is called frams and logical memory is pages

- 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 is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program

page size = frame size

- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages

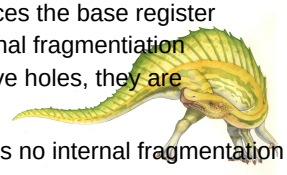
logical is contigous and physical is not

- Still have Internal fragmentation

The page table is a data structure that replaces the base register and is more expensive. We don't have external fragmentiation because we only to allocate pages. If we have holes, they are always a multiple of page.

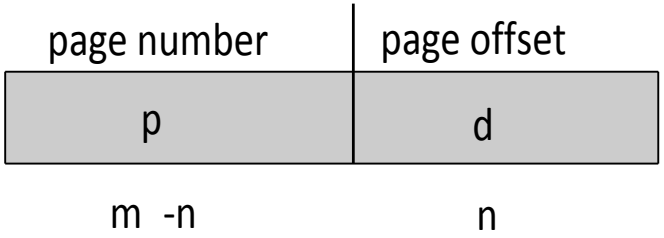
But we still have internal fragmentation, we have a process whose size is the remainder of the last page is called internal fragmentation. If you need to write a song in a notebook, the first few pages will be only written the left over is called internal fragmentation.

In contiguous fragmentation there is no internal fragmentation



Address Translation Scheme

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

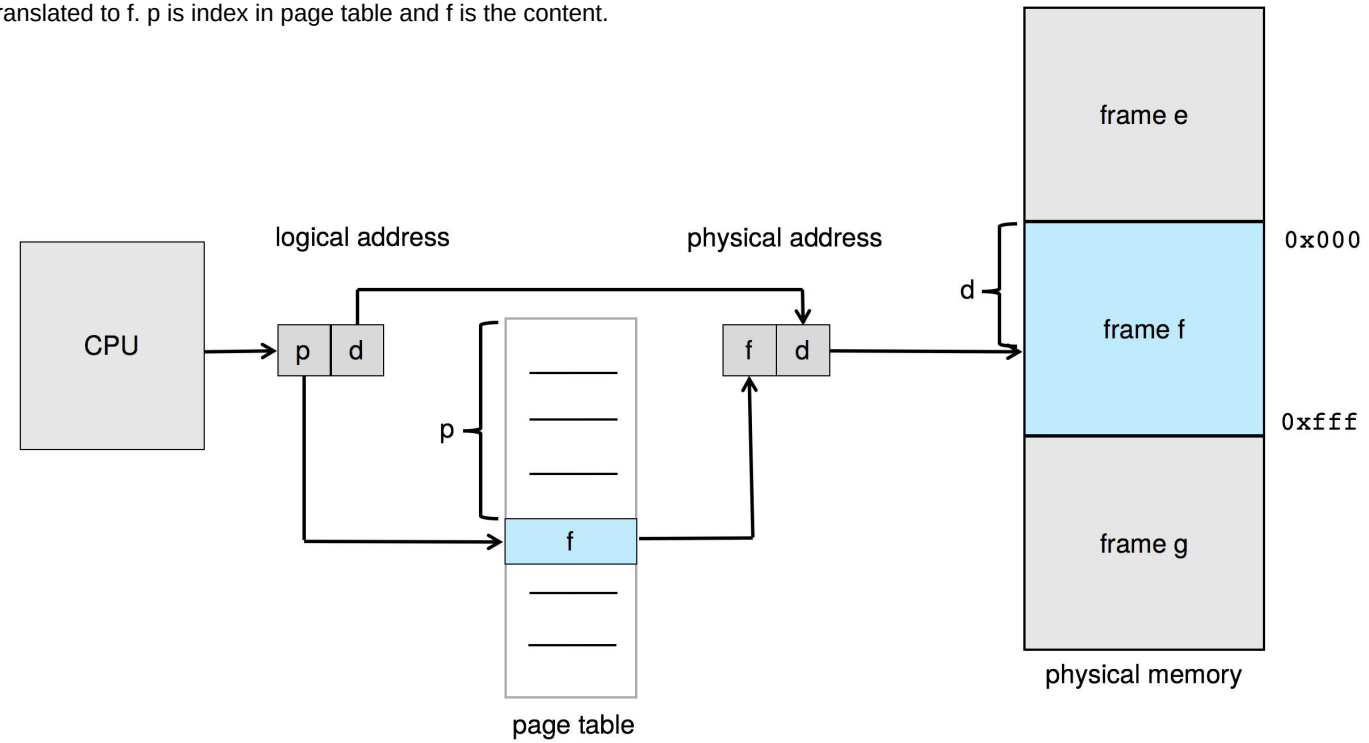


  - For given logical address space 2^m and page size 2^n

How do we translate.

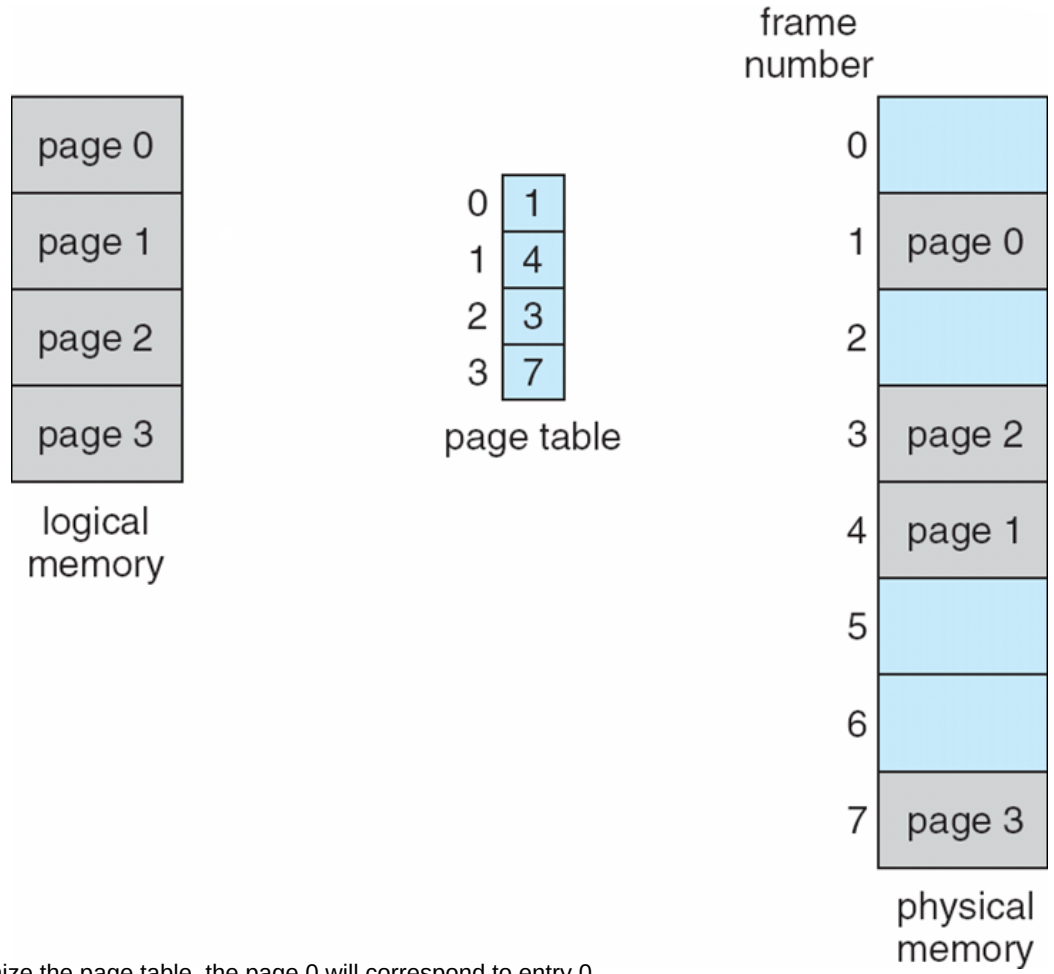
Paging Hardware

translation is done this way where logical address will be split, the displacement in page will directly go to displacement in frame, byte 100 in page will be byte 100 in frame. cuz frame is exactly taking one page. only thing changes is relocation of page.
p is translated to f. p is index in page table and f is the content.



Where is the page table? It is not in the cpu, the mmu is in cpu but the page table is not in the cpu cuz it doesn"t fit. too big. One page table of pages of 1 kb that's. imagine 1 mil entries in table.too big. The page table is in memory, it has a consequence. Suppose the cpu asks for fetch, read or write opp, in order to read you have to translate from p to f and read the page table. You need one memory opp you need to do extra mem opp to translate. The page table is the data structure of a kernel. The translation is done through a kernel in the structure. You need direct access. You need to calculate from p where to go. the allocation of kernel data structure from that of processes. For one r/w you need extra r/w. This is a slow down. A very dramatic slow down. So, you don't have fragmentation anymore but you have a slow down.

Paging Model of Logical and Physical Memory



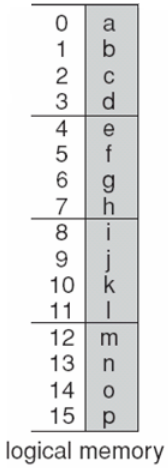
How do we organize the page table, the page 0 will correspond to entry 0.

Physical memory is a set of frames, and each set can be empty or contain one page. No continuity and even the order can change. page 1 is in 4, page2 in 3. so nocontinguous allocation.
The allocation task is done page by page.

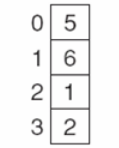
each register for each page is too much so we can't use registers. cuz pages are too many. not possible to have an array of pages. So we basically we use a page table that is an array thats used to map pages to frames. where is this array saved? in kernel memory itself.

Paging Example

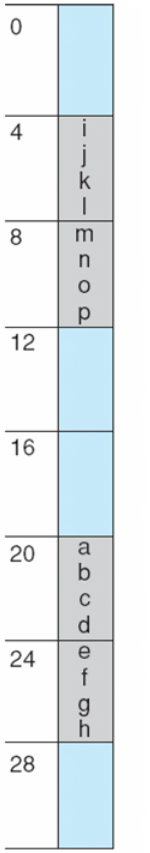
- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



logical memory



page table



physical memory

suppose you have a logical mem of 16 addr, and they are on 4 bits. from 0 to 15. suppose that the 16 addr are devided in 4 pages of 4 bytes for them. 0,1,2, and 3 the binary number on 4 bits start with 4 zeroes. 4,5,6,7 all top most bits are 01. 8,9,10,11 are ....
If you take the top 2 bits and group them which can be seen as the index of a page.
You can say you have four bits, the first 2 bits are the page numbers and the least significant bits are the offset of the page.

letter k is in page 2 with offset 2. L in page 2 with offset 3

for better understanding solve this on your own on paper. Need to visualize with bits.!!!!!!!!!

Paging -- Calculating internal fragmentation

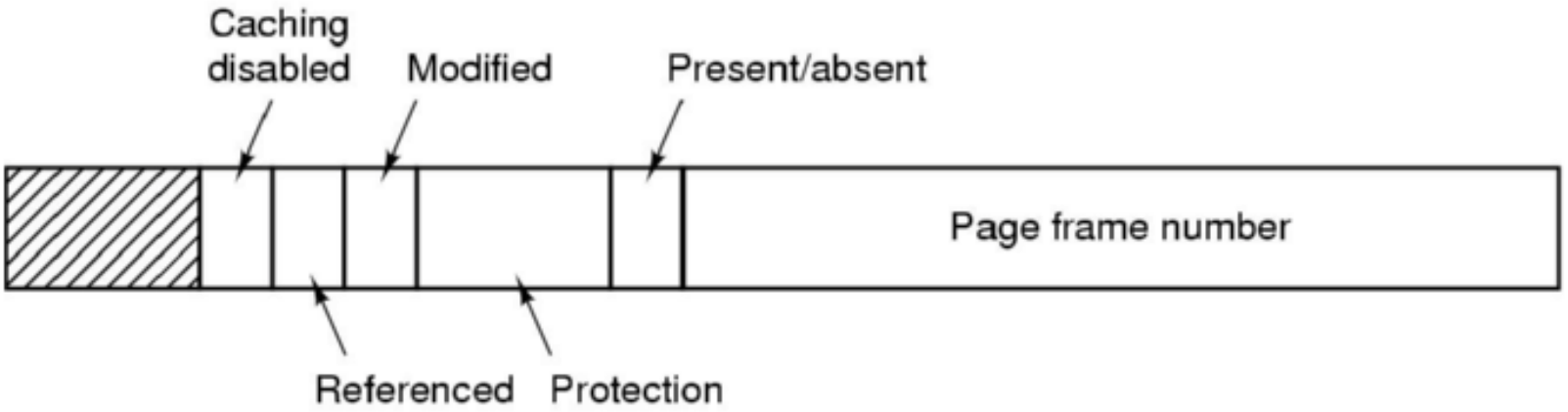
- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

In order to fit this process we need 35 fully used page and last one partially used (1086 bytes. the rest 962 bytes that is left is internal fragmentation.

Internal fragmentation is the unused part, don't confuse with the used part.

In order to minimize fragmentation, small pages is better. Fragmentation is asking for small pages, page table is asking for larger pages. Since, every page in a process needs one entry. So in the end the page table would prefer larger pages, worst fragmentation. These are two criteria which have to be given priority to one accordingly. The size of processes is increasing over the years. Larger pages are going to be considered.

Structure of Page Table Entry



Since pages are allowed to have one entry for each page. Frame number is a number that could be in the range of millions. Typically entry in an array of page tables should be in bytes. Page table entries of 32 bits (imagine), lets say frame number is happy with 24-26 bits. we can use the extra bits to add extra info associated to those pages. Instead of having a protection for all we can protection for each. You can spcify differenct characteristic for each page. You can have pages that are cached and pages that are not cached. Page table is implemented the following way:

Figure 3-11. A typical page table entry.

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**).

Reading in the page table is done through a dedicated hardware. That hardware is expoiting two register. Plus a piece in the cpu a very small table, which is working as a cache for the page table. Slow down a factor of 2 when you cant use this cache.

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

If the tlb is dedicated to a single process

You can have a tlb shared among diff processes, but in order to do that you need to atleast in each of the line an identifier of the process. You can have addresss, page number 10 for process and page number 10 for process 2. You need to be able to detect which one is which. For this you need address-space identifier (ASIDS). which means entry of tlb is not only page num or frame num but also asids.

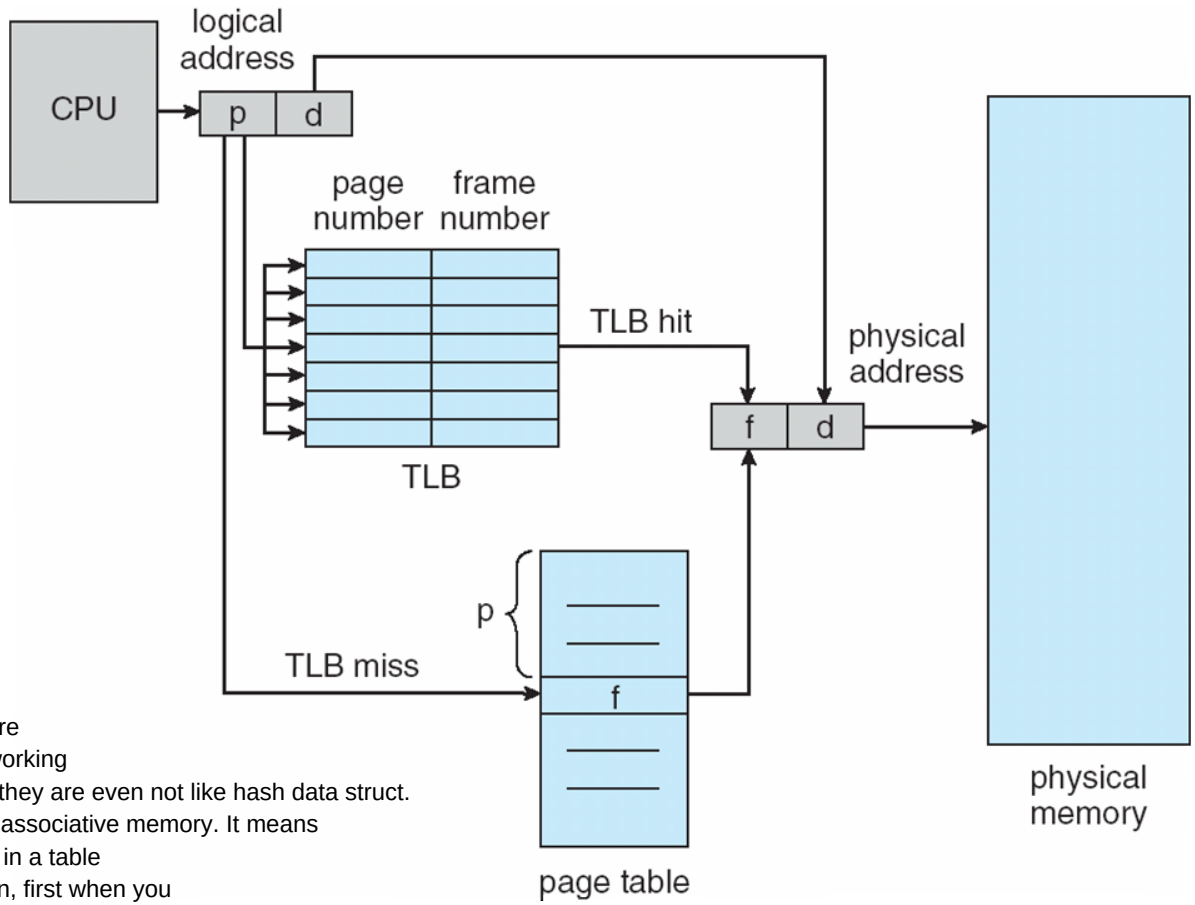
Translation Lookaside Buffers

Valid bit here just means in the tlb you can have entries valid or not valid. Suppose you have a tlb of 100 numbers. You have two possibilities. When you start with an empty tlb all 0s. Then you will have 1s. It just means usable or not (valid). Modified means that page written has been modified with respect to the copy on the disc cuz sooner or later it will be need to written, so needs to be save again.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging.

Paging Hardware With TLB



highly probable addresses are
in the cache. The tlb is not working
p is the index and f content, they are even not like hash data struct.
Here we have the so called, associative memory. It means
when you search something in a table
you typically have 2 condition, first when you
know the location and know where it is,, 2nd
to search you do several comparisons. Associative mem
are the ones where you simply tell the memory, you don't
tell the memory the addr but .....
It is basically the only row that is corresponding to p. if p corresponds to one of the left entries on the table you will get the right f.

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**)

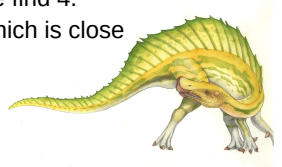
  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  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.

what is the effect of tlb? It is a cache. What is the final result EAT. You have a certain hit ratio suppose 80% means every 5 you have find 4. The EAT is equal to = the prob of hit (80%) x 10 nanoseconds + the prob miss (20%) + 20 nanoseconds. If you want a performace which is close to the original performance you need to have prob close to 1. or high 90s.



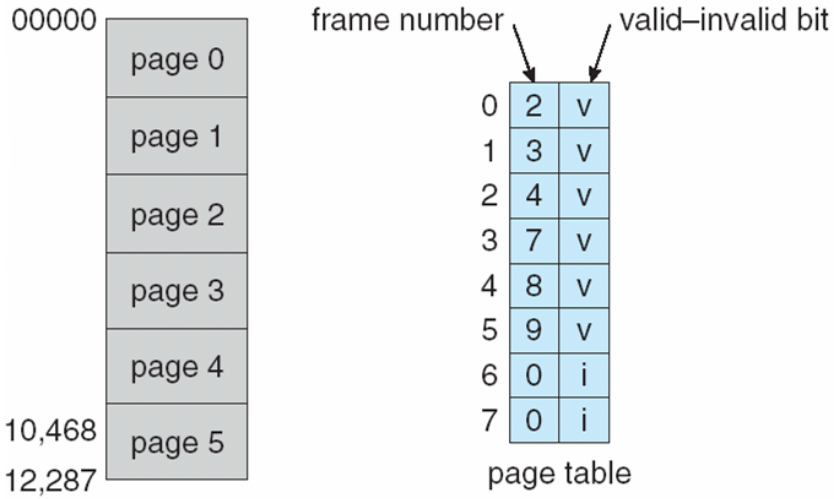
Memory Protection

- 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

Both the page table and tlb can provide protection in terms of specifying page if read only writable or execute. Execute is diff from reading, execute means only fetch and you cannot copy to somewhere else. The executable code is not readable but can with fetch. Valid is another way where you could use in the page table some validity bit where you can say each bit is not valid. For now a page is not valid when a page does not exist. Suppose your page table of 8 entries but your process only has 6 entries. You have only one possibility to protect valid address. An invalid pointer is an invalid address.

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

one way to protect page 6 is not table is to say yes page table has only 6 entries. The non valid pages could also be in the middle in the future that the addr space could have things on top or bottom of stack and empty in the middle. You could have page table with 100 entries and invalid ones are not only 101..100+ but it could be in the middle.

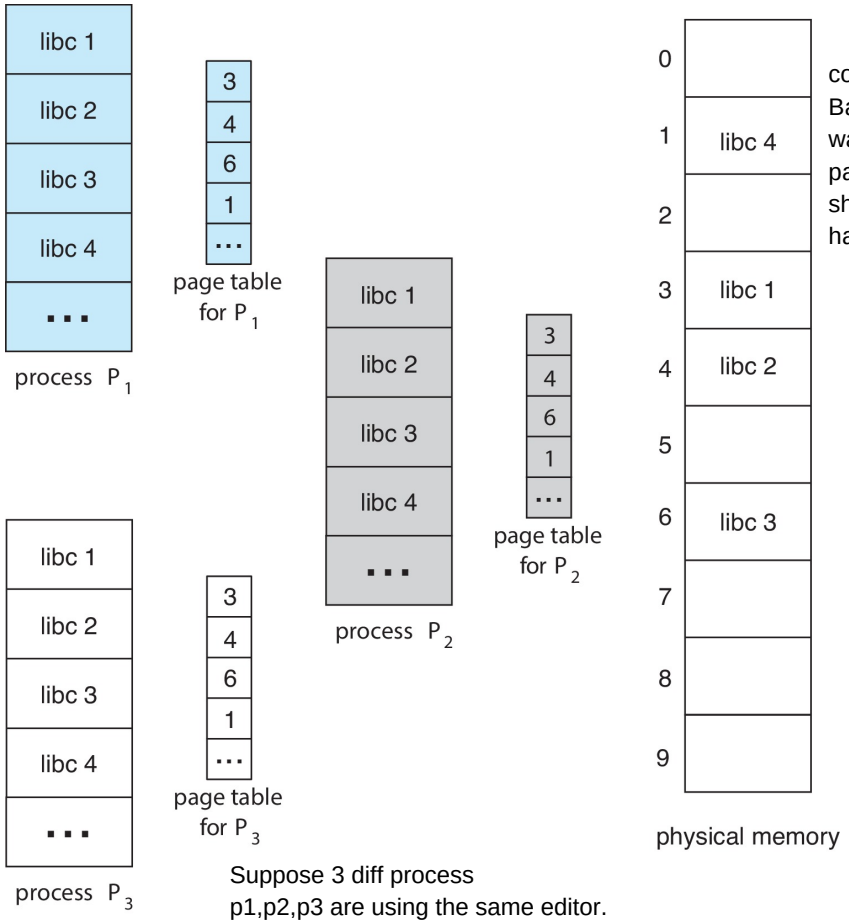


Shared Pages

- **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 are what we have been asking for shared libraries.

Shared Pages Example



cont of text below:
Basically you are sharing the editor, this is the way pages can be mapped to share frames. each page table will contain frame that can be shared. shouldn't contain data for one the process they don't have global variables.

Suppose 3 diff process
p1,p2,p3 are using the same editor.
lets say 3 editing word or ppt or notepad.
so process p1 has a copy virtually of libc1 ,
libc2, libc3 where libc3 here is the editor. If the program was statically linked and loaded it will be a copy in my executable. So logically i see 4 pages libc1,2,3,4.
Suppose same thing is done my process 1 or 2 or 3.

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

How can we implement page table? They are a kernel data structure and continuous. You don't read page tables by fetch or r/w. Before being used implicitly, they need to be loaded and need to be prepared.

The kernel when creating a process is creating the page table.

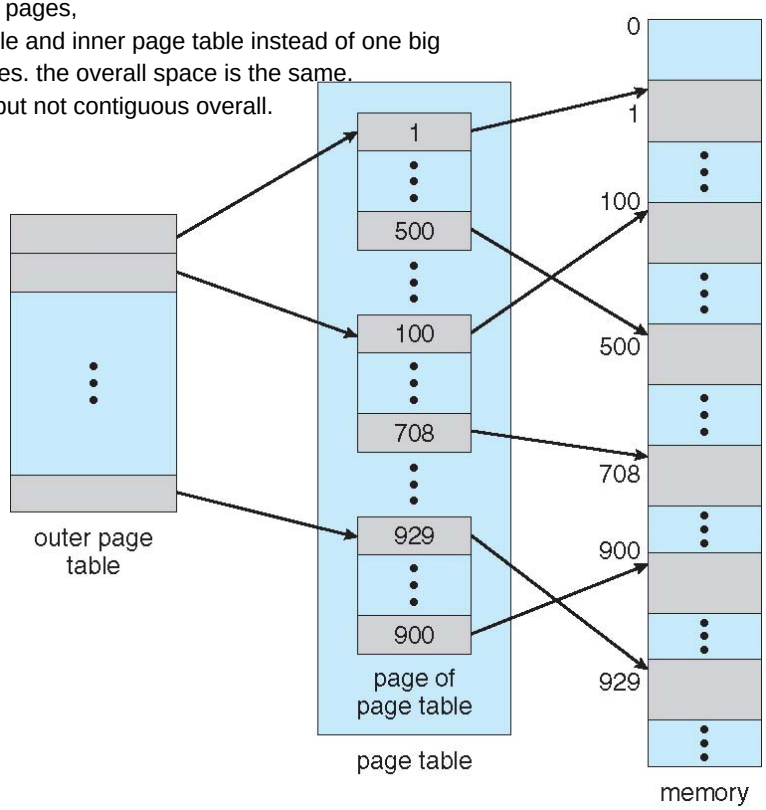
Cpu is doing load instruction.

so each entry in page table, in 4 bytes we fit page nm, protection, validity and something else.

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

Suppose 4megs is too much we split into 4 smaller pages,
same as you do with directories. An outer page table and inner page table instead of one big contiguous page table we split into many page tables. the overall space is the same.
each second level page table is contiguous inside but not contiguous overall.

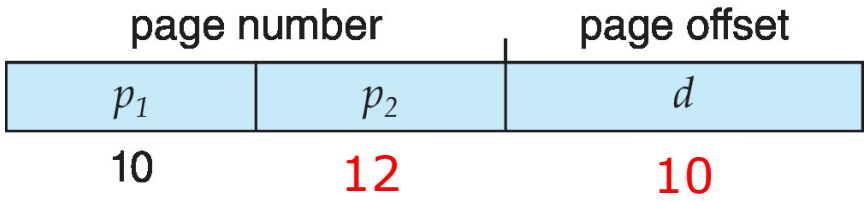


Two-Level Paging Example

- 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 10-bit page number
  - a 12-bit page offset

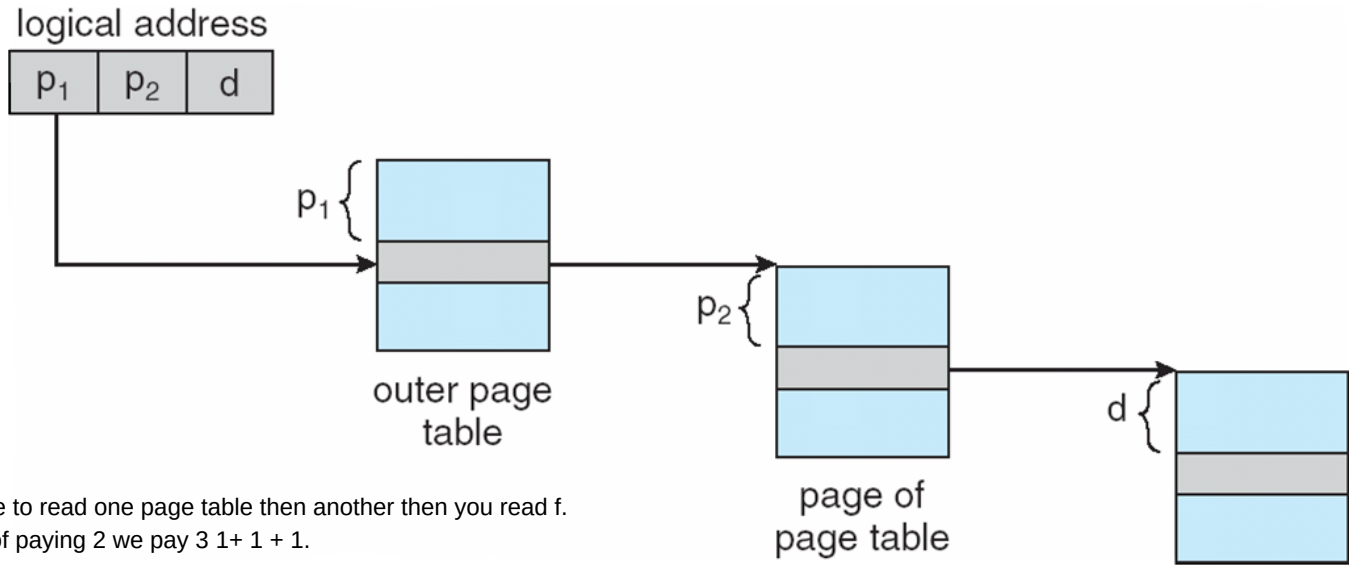
Suppose you have 22 bit of page numbers, we further split to 10 and 12. we have outer page table driver by 10 bits. outer page table can have upto.

- 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
- Known as **forward-mapped page table**

Address-Translation Scheme

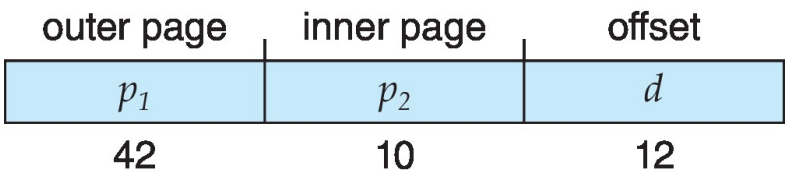


You have to read one page table then another then you read f.
instead of paying 2 we pay 3 1+ 1 + 1.

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 two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

addresses going from 32 to 64 were done.4



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} 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

outer page	inner page	offset
p_1	p_2	d
42	10	12

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

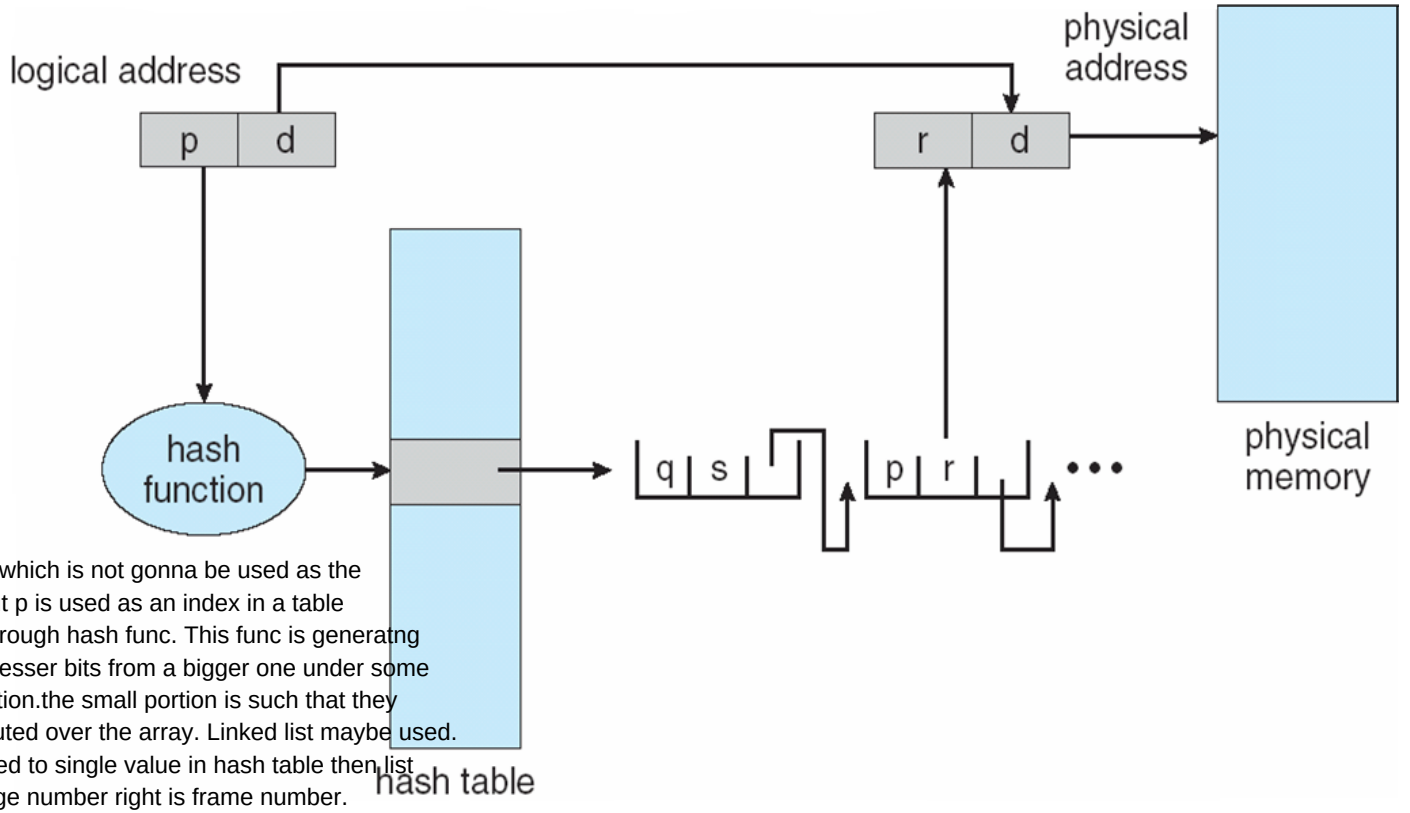
Hashed Page Tables

Hashing is considered in software a way to logically compact a big direct access array in a smaller array but we need a func to convert a big index to small index. The potential addr are very big but we know we know all are not gonna be used.

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

Hashed Page Table

Review this topic (hash tables) and dictionaries in python.



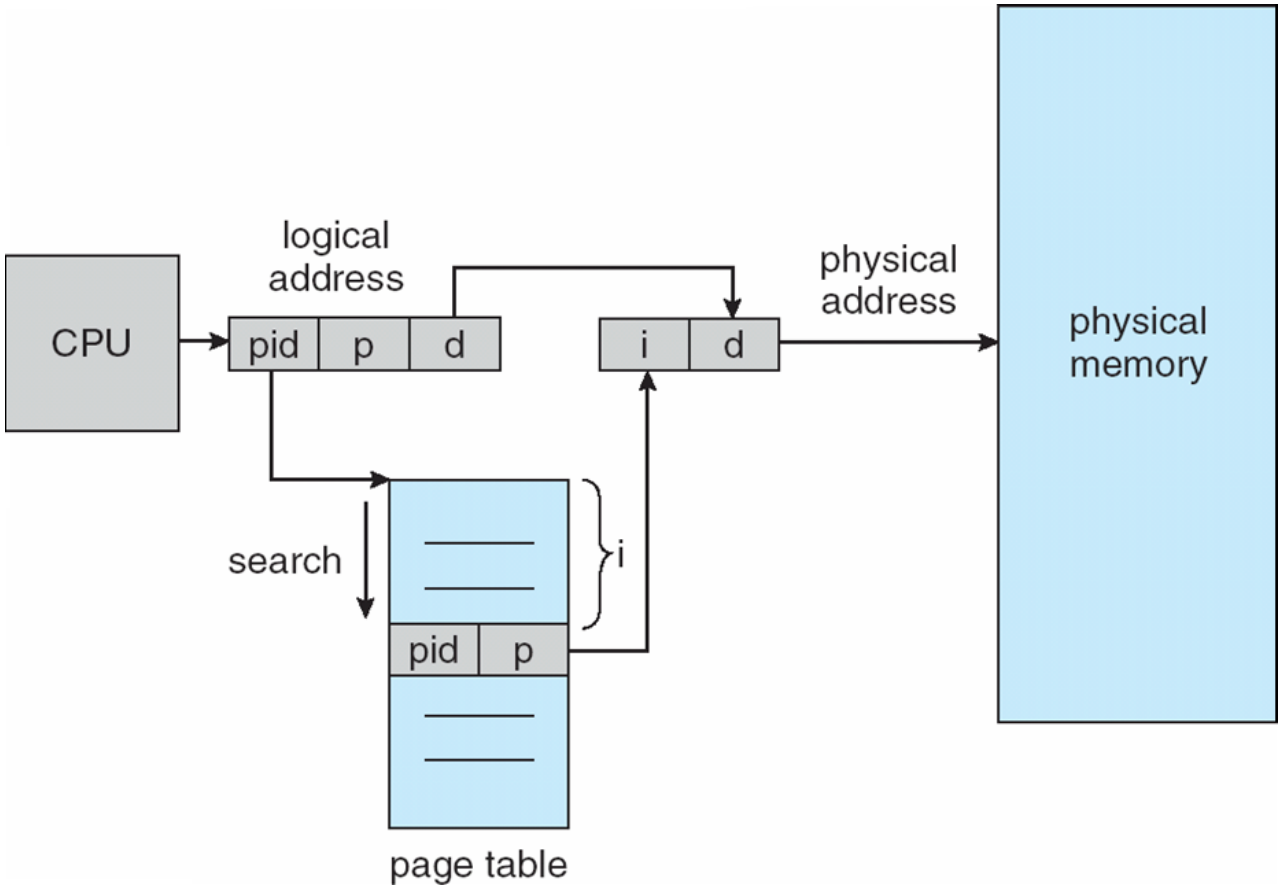
Chaining, given p which is not gonna be used as the index in a table but p is used as an index in a table after translation through hash func. This func is generatng smaller num with lesser bits from a bigger one under some statistical assumption.the small portion is such that they will be well distributed over the array. Linked list maybe used. Two values mapped to single value in hash table then list is used. Left is page number right is frame number.

In the list you will have translation of page num 100 and 200. so when you translate you need to traverse this list and find the right p.

Inverted Page Table

- 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
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Inverted Page Tables

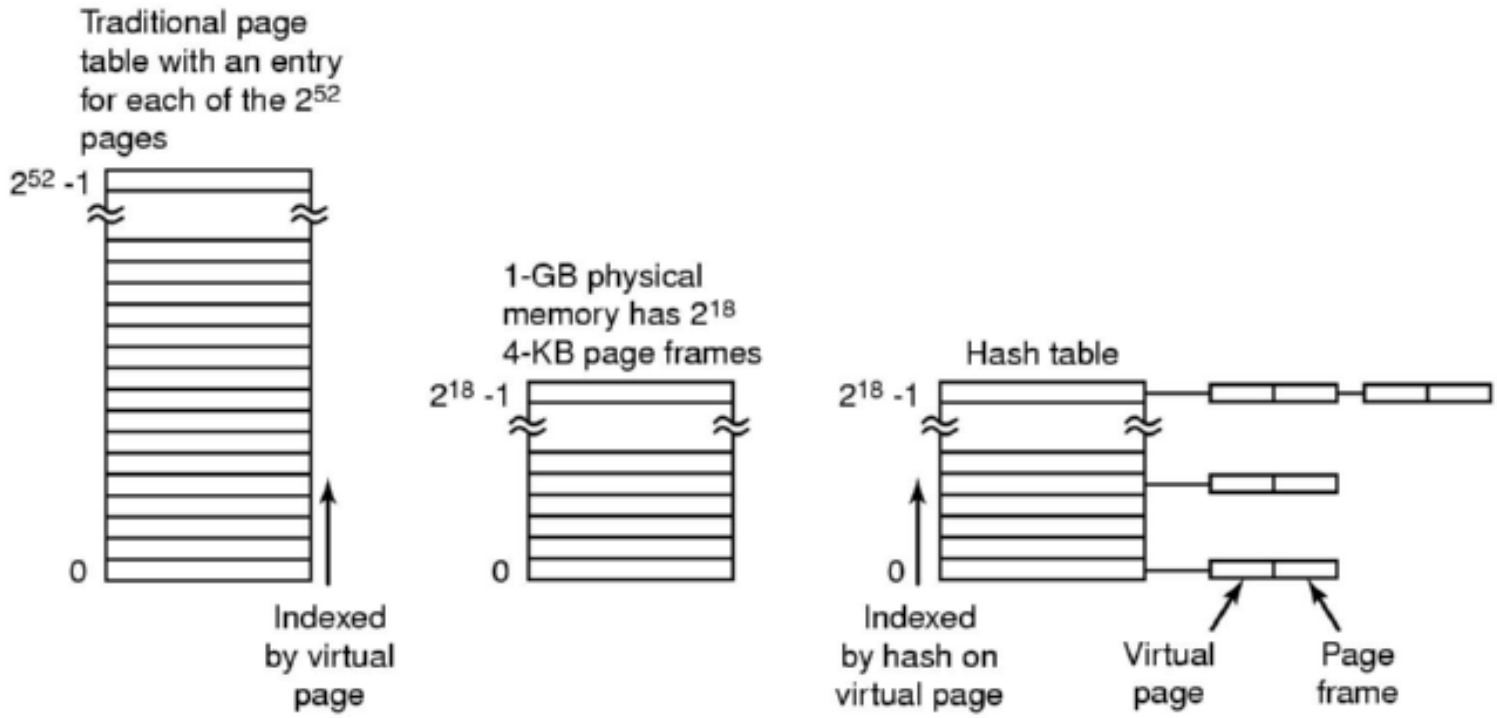


Figure 3-14. Comparison of a traditional page table with an inverted page table.

Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

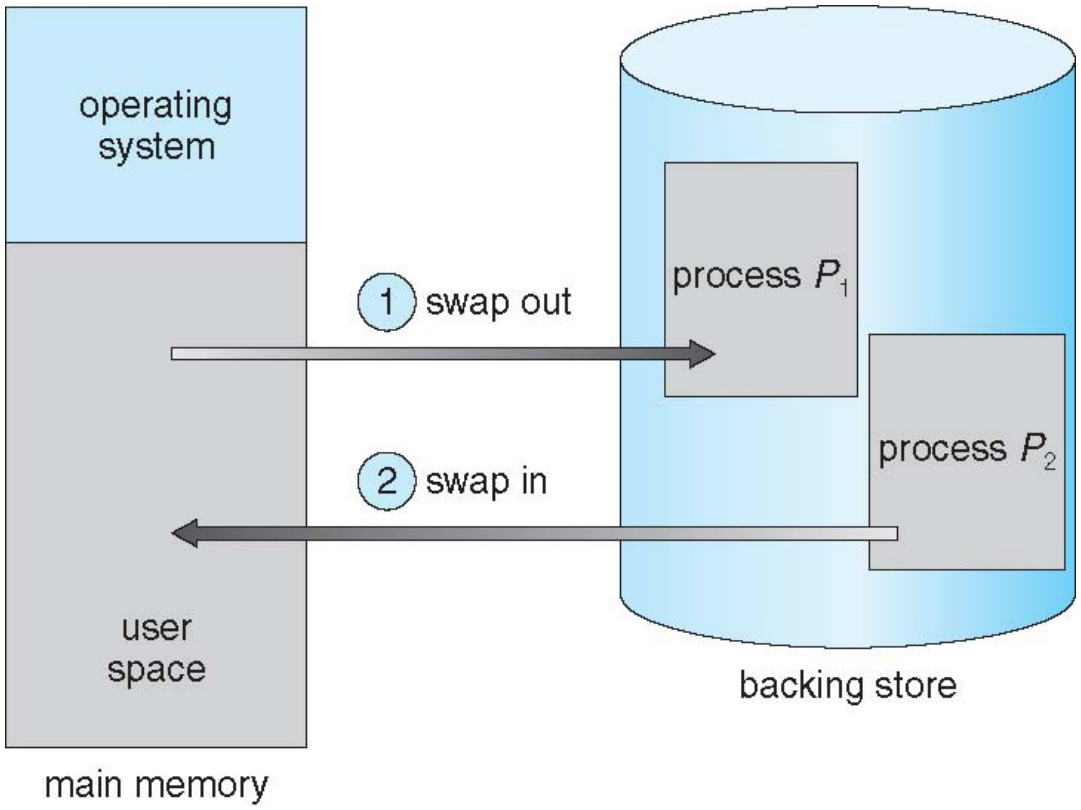
Swapping

- 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()`

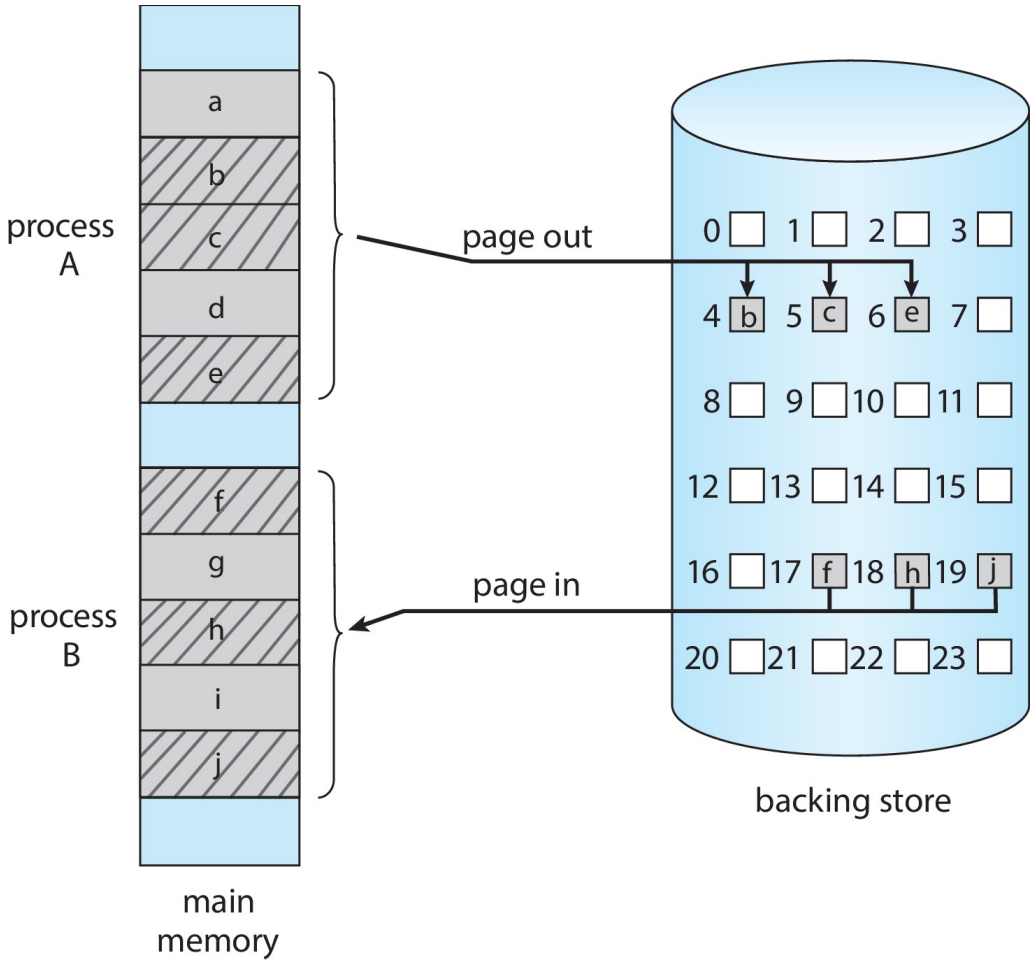
Context Switch Time and Swapping (Cont.)

- 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



Example: The Intel 32 and 64-bit Architectures

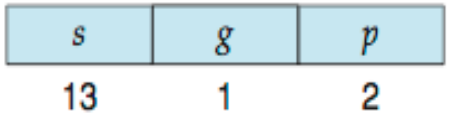
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))

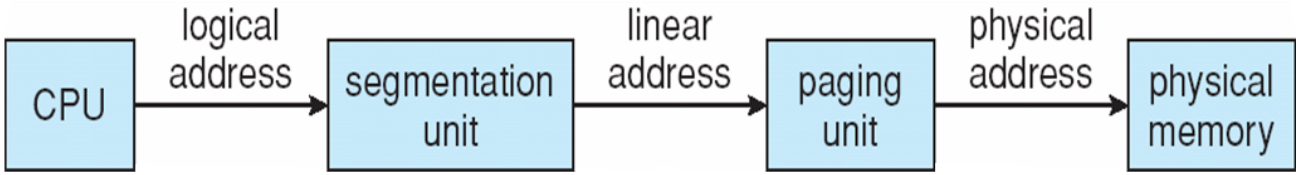
Example: The Intel IA-32 Architecture (Cont.)

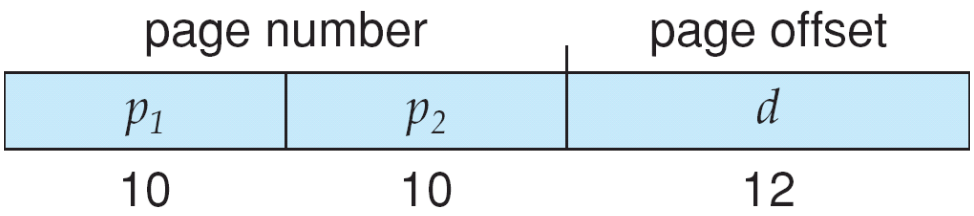
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



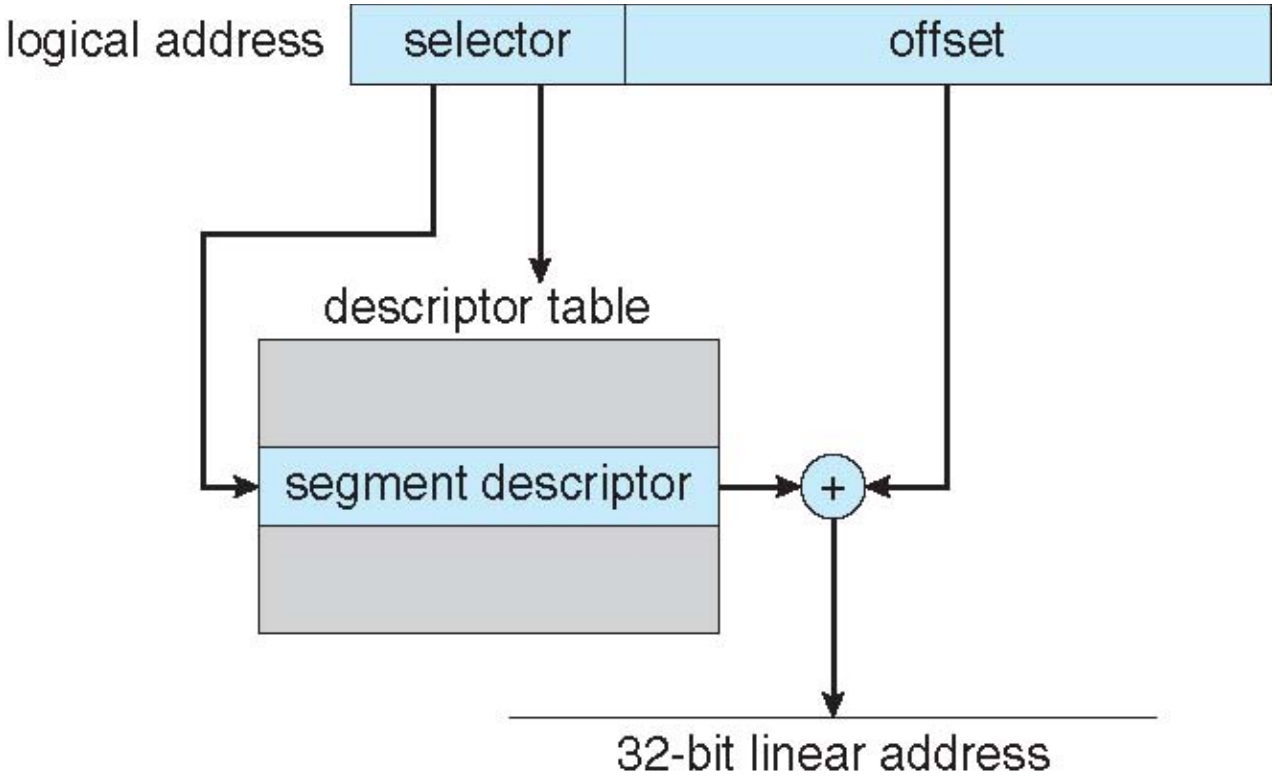
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32

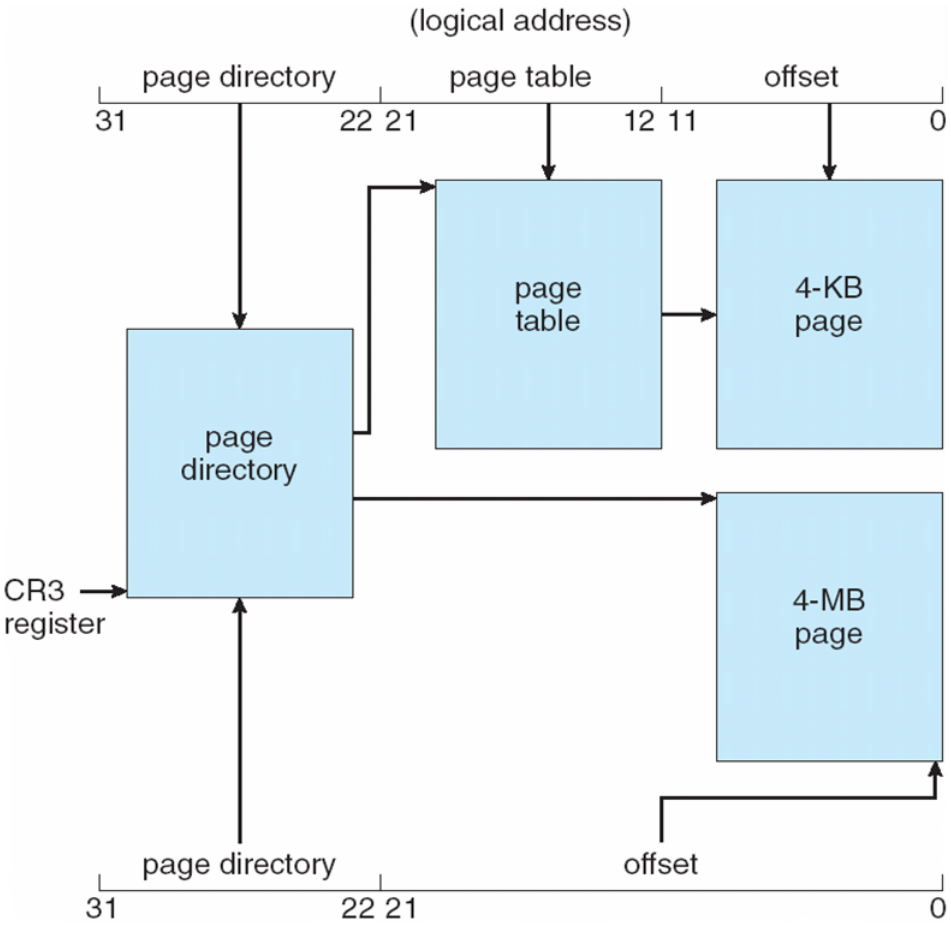




Intel IA-32 Segmentation

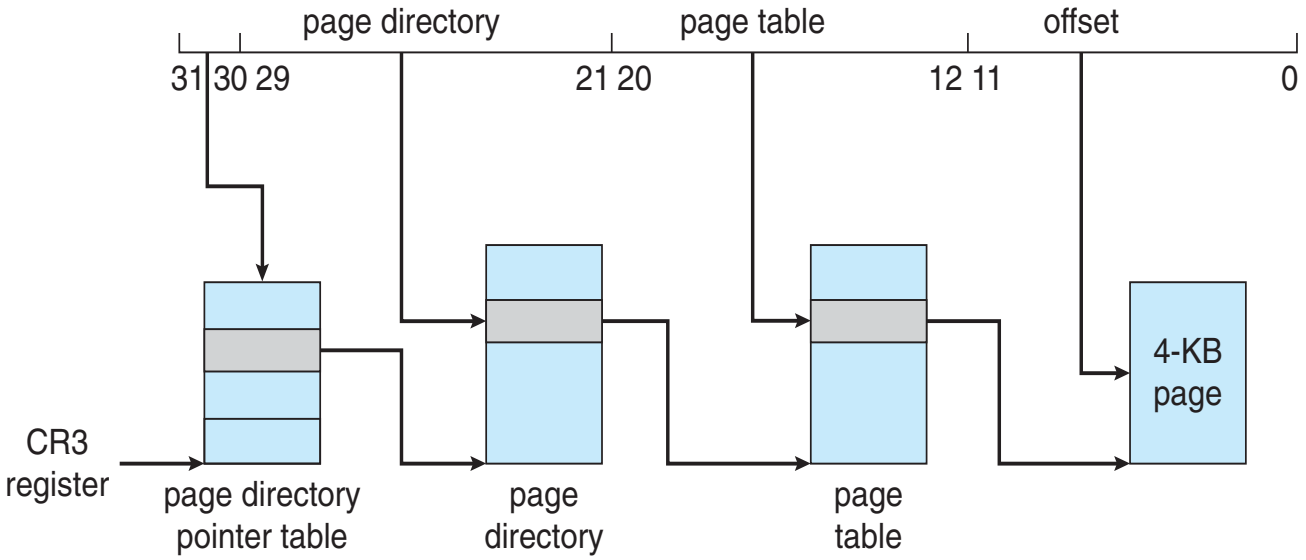


Intel IA-32 Paging Architecture



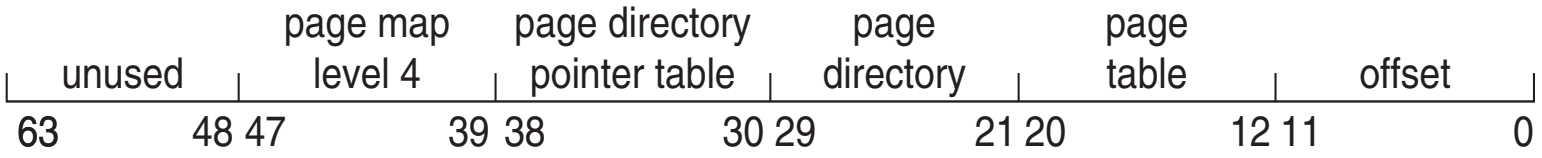
Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



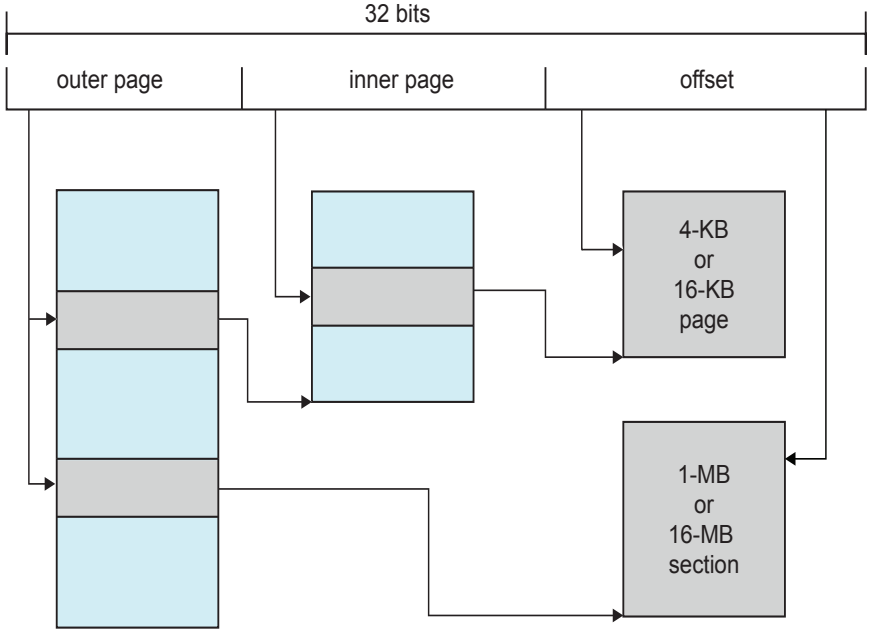
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 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 so virtual addresses are 48 bits and physical addresses are 52 bits



Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



End of Chapter 9