Chapter 8: Memory Management

Contents

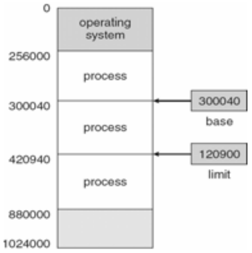
- Background
- Swapping
- Contiguous Memory Allocation
- Memory fragmentation
- Paging
- Structure of the page Table
- Segmentation
- Example: The Intel Pentium

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one cycle of the CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
  - To protect the OS from access by user processes and
  - To protect user processes from one another

Base and Limit Registers

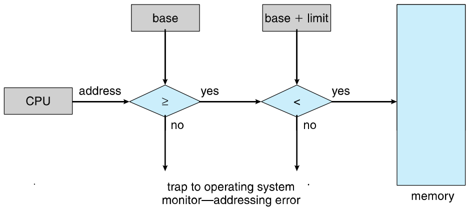
- We can provide protection by using two registers: a base and a limit
- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



In most schemes, the kernel occupies some fixed portion of main memory and the rest is shared by multiple processes

OS resides in low memory

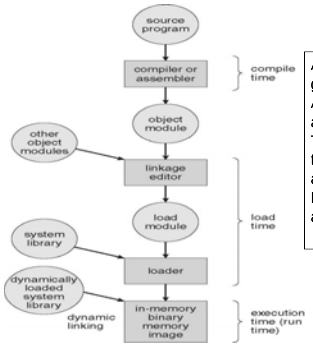
Hardware Address Protection with Base and Limit Registers



Any attempt by a program executing in user mode to access OS memory or other users’ memory results in trap to the OS

Address Binding

A user program will go through several steps (some which are optional) before being executed. The addresses may be represented in different ways during these steps



Addresses in the source program are generally symbolic
A compiler will bind these symbolic addresses to relocatable addresses
The linker or loader in turn will bind the relocatable addresses to absolute addresses
Each binding is a mapping from one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: The compiler knows where a process will reside in memory so generates *absolute code* (code that starts at a fixed address)
  - **Load time**: The compiler does not know where a process will reside so generates *relocatable code* (so its starting address is determined at load time)
  - **Execution time**: the process can move during its execution, so its starting address is determined at run time

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
- The set of all physical addresses corresponding to these logical addresses is a **Physical address space**

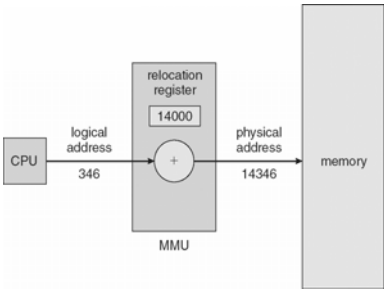
Memory-Management Unit (MMU)

- 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
- The run time mapping from logical to physical is done by hardware device called MMU.
- *Hiding* the mapping is one of the main aims of memory management schemes.
- Many methods possible, covered in the rest of this chapter

Dynamic relocation using a relocation register

- In MMU scheme, the value in the relocation register is *added* to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**

If the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.



Logical addresses (in the range 0 to *max) and physical addresses (in the range R+ 0 to R+ max for a base* Value R).

Dynamic loading

- It is necessary for the entire program and all data of a process to be in physical memory for the process to execute.
- The size of a process has thus been limited to the size of physical memory.
- To obtain better memory-space utilization, we can use dynamic binding
- With dynamic loading, a routine is not loaded until it is called. All routine are kept on disk in a relocatable load format
- Advantage of dynamic loading: Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines
- No special support from the operating system is required implemented through program design

Dynamic Linking

- Static linking
  - done at compile or link time
  - does not facilitate sharing of libraries
- Dynamic linking
  - linking is deferred until load or runtime to allow integration of libraries
- 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
- Unlike dynamic loading, dynamic linking generally requires help from OS
- Dynamic linking is particularly useful for libraries

Shared Libraries

- Prewritten code for commonly used functions is stored in libraries
- statically linked libraries
  - library code is linked at link time into an executable program
  - results in large executables with no sharing
- dynamically linked libraries
  - if libraries are linked at run time they can be shared between processes

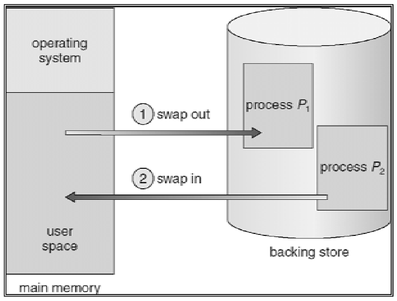
Swapping and Virtual Memory

- Sometimes there is not enough main memory to hold all the currently active processes.
- So excess processes must be kept on disk and brought into run dynamically
- Two approachs
  - Swapping:
    - Bring in each process entirely, running it for a while and then put it back on the disk
  - Virtual memory:
    - Brings each process partially, not entirely

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **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
- System maintains a **ready queue** of ready-to-run processes which have memory images on the backing store

Schematic View of Swapping



Swapping

- Context-switch time in swapping is fairly high
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Assume that hard disk with a transfer rate of 50MB/sec.
- A transfer of 100 MB process to or from main memory will take:
  - 100/ 50= 2 s = 2000 ms
- Assuming an average latency (delay) of 8 milliseconds
- Total swap time:

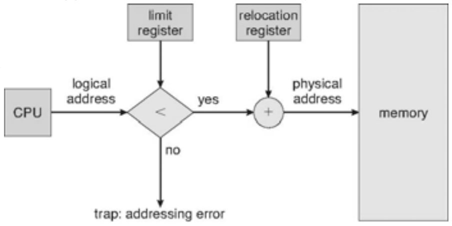
  = transfer out + transfer in + (2 * latency)
  = 2000 + 2000 + (2 * 8)
  = 4016 ms
- System with 4GB main memory, 1 GB for OS and maximum size of user process is 3GB. However many user processes may be less than 100MB, A 100MB process could be swapped out in 2 seconds compared to 60 seconds required to swapping 3 GB process
- Must be careful not to swap out a process that is waiting for I/O.
- Better memory management solution
  - Require more swapping time and provide less execution time

Contiguous Memory Allocation

- Main memory usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- In contiguous memory allocation, each process is contained in a single contiguous section of memory

Hardware Support for Relocation and Limit 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* by adding the value in the relocation register
  - This mapped address is sent to memory



Memory Allocation

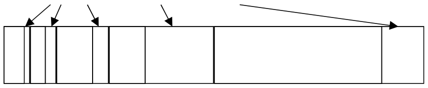
- Fixed-size partition: Divide memory into a fixed no. of partitions, and allocate a process to each.
- Partitions can be different *fixed* sizes.

16	16	32	64	128

continued

Memory Allocation

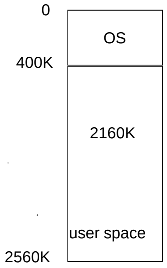
- Processes are allocated to the smallest available partition.
- *Internal fragmentation* will occur:



Internal Fragmentation - allocated memory may be slightly larger than requested memory and not being used.

Variable-size Partitions

- In variable-size partition scheme OS keeps track of:
  - a) allocated partitions b) free partitions (hole)

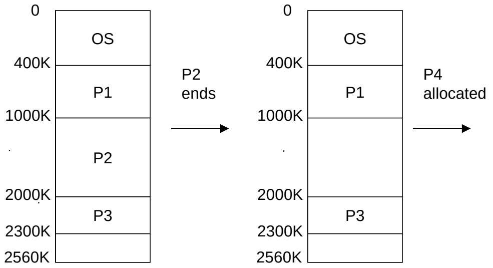


Job Queue

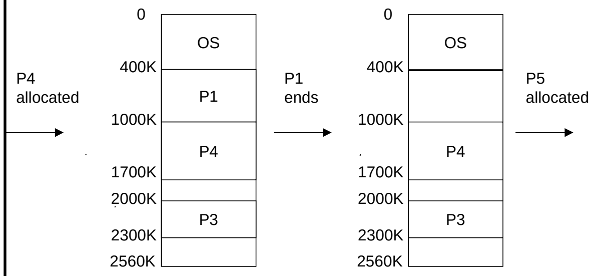
process	memory	Burst time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15

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

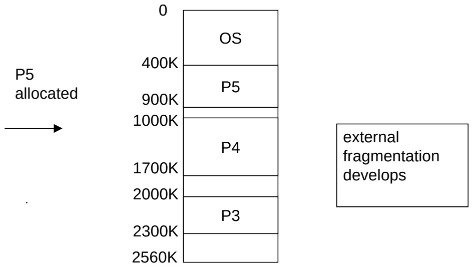
Variable-size Partitions



Variable-size Partitions

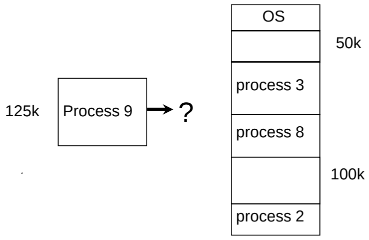


Variable-size Partitions



External Fragmentation

- External Fragmentation - total memory space exists to satisfy request but it is not contiguous



Dynamic Storage-Allocation Problem

How to satisfy a request of process 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

Both the first-fit and best fit strategies suffer from external fragmentation

First fit analysis reveals that given *N* allocated blocks, another 0.5 *N* blocks lost to fragmentation. This property is called **50-percent rule**

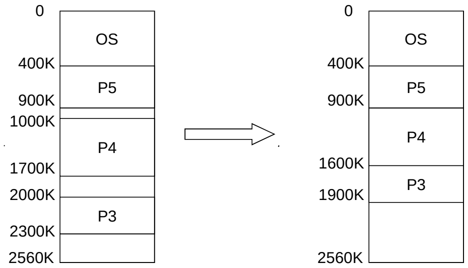
First-fit Best-fit and Worst-fit Example

- Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?
  - First-fit: 212K is put in 500K partition 417K is put in 600K partition 112K is put in 288K partition (new partition 288K = 500K - 212K) 426K must wait
  - Best-fit: 212K is put in 300K partition 417K is put in 500K partition 112K is put in 200K partition 426K is put in 600K partition
  - Worst-fit: 212K is put in 600K partition 417K is put in 500K partition 112K is put in 388K partition 426K must wait In this example, best-fit turns out to be the best.

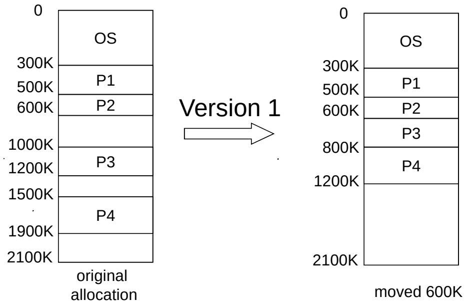
Compaction

- One solution to the external fragmentation is 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
- Another possible solution to the external fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available
- Two complementary techniques achieve this solution is
  - Paging and segmentation

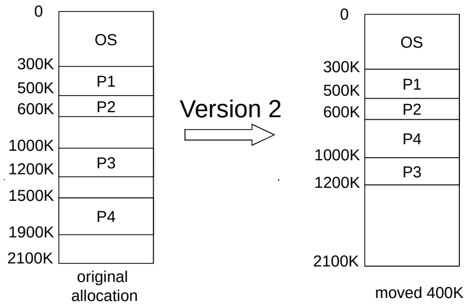
Compaction



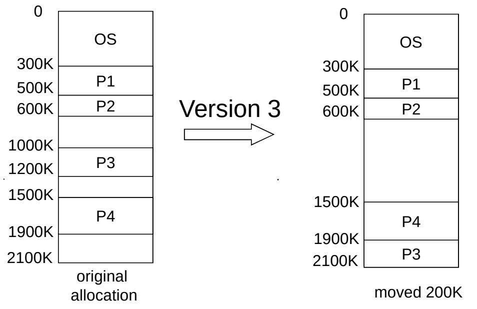
Compaction: Different Strategies



Compaction: Different Strategies



Compaction: Different Strategies



Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous
  - Avoids external fragmentation
  - Avoids the need for compaction
  - May still have internal fragmentation
- Divide logical memory into blocks of same size called pages
  - Backing store is also split into pages of the same size
- Divide physical memory into fixed-sized blocks called frames
  - the frames may be located anywhere in memory
- To run a program of size *N pages, need to find N free frames and load program*

Address Translation Scheme

- Each logical address has two parts:
  - *Page number (p)* - index to page table
    - *page table* contains the mapping from a page number to the base address of its corresponding frame
  - *Page offset (d)* - offset into page/frame
- The size of a page is typically a power of 2:
  - 512 (2^9) -- 8192 (2^{13}) bytes
- This makes it easy to split a machine address into page number and offset parts.
- For example, assume:
  - the memory size is 2^m bytes
  - a page size is 2^n bytes (n < m)
- The logical address format becomes:

page number	page offset
p	*d*
m - n	*n*

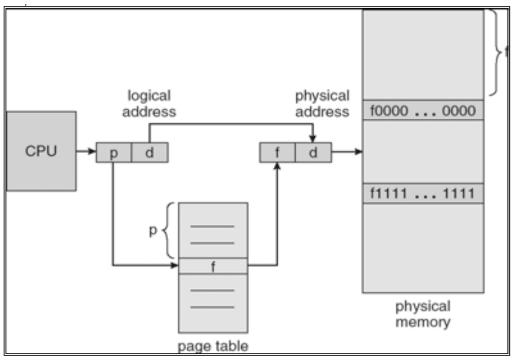
Example

- Assume:
  - Memory size is 32 bytes (2^5)
  - Page size: 4 bytes (2^2)
  - Therefore, there can be 8 pages (2^3)
- Logical address format:

page number n	page offset d
3	2

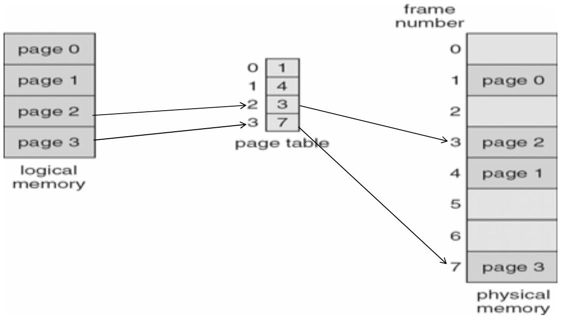
continued

Paging Hardware

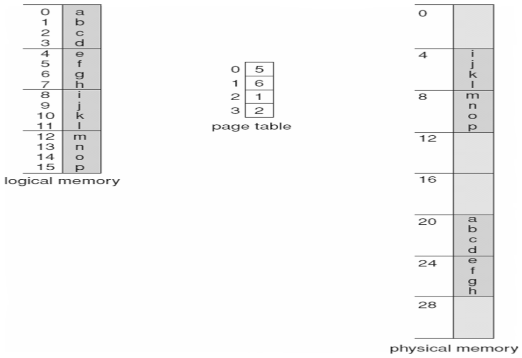


Paging Model of Logical and Physical Memory

- Page size 4 bytes
- Memory size 32 bytes (8 pages)



Paging Example for 32-byte memory with 4-byte pages



Paging Example

- **Example setup:**
- n=2 and m=4 (*p and d are each 2-bits)*
- Memory size 32-bytes and
- Page size 4-bytes (8 pages can fit in physical memory space)
- **Example 1:**
- Logical address 10 is in page 2 at offset 2
- According to the page table, page 2 is located in frame 1
- Physical memory address is: (Frame # * Page size) + Offset
- Physical memory address for logical address 10 is : (1 * 4 bytes) + 2 byte offset = 6

Another paging example

- **Example 2**:
- Logical address 4 is in page 1 at offset 0
- According to the page table, page 1 is located in frame 6
- Physical memory address is: (Frame # * Page size) + Offset
- Physical memory address for logical address 4 is : (6 * 4 bytes) + 0 byte offset = 24

- **Example 3:**
- Logical address 7 is in page 1 at offset 3
- According to the page table, page 1 is located in frame 6
- Physical memory address is: (Frame # * Page size) + Offset
- Physical memory address for logical address 7 is : (6 * 4 bytes) + 3 byte offset = 27

Using the Page Table

Logical Address	*Physical Address*
0	(5*4) + 0 = 20
3	(5*4) + 3 = 23
5	(6*4) + 1 = 25
14	(2*4) + 2 = 10

Features of Paging

- No external fragmentation
  - any free frame can be allocated to a process that needs it

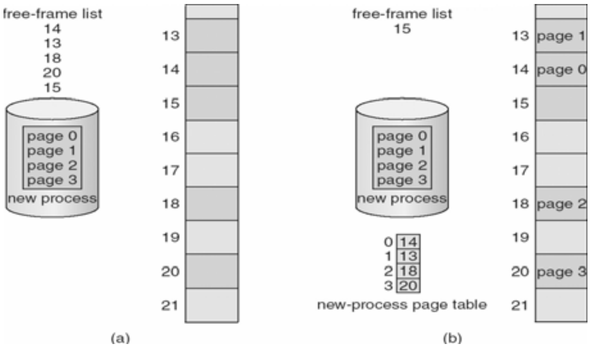
- Internal fragmentation can occur
  - If the memory requirements of a process do not happen to coincide with page boundaries
  - For example, if page size is 2048 bytes, a process of 72766 bytes will need 35 pages plus 1086 bytes. It will allocated 36 frames, resulting in internal fragmentation of 2048-1086 = 962 bytes

- There is a clear separation between logical memory (the user's view) and physical memory (the OS/hardware view).

Free Frames

- Since the operating system is responsible for managing memory, it must be aware of the allocation details of the physical memory
  - which frames are allocated
  - which frames are available
  - how many frames there are in total
  - This information is generally kept in a data structure called a frame table

Free Frames

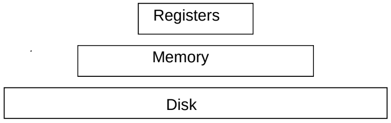


Before allocation

After allocation

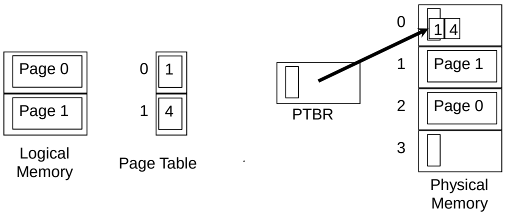
Implementation of Page Table

- If Page table is small then it is kept in registers
- Fast!
- Expensive!



Implementation of Page Table

- If Page table is large then it is kept in main memory
- ***Page Table Base Register*** **(PTBR) points to the page table in memory**



In this scheme every data/instruction access requires two memory accesses

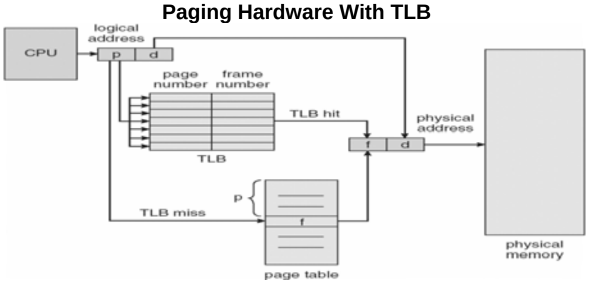
-One for the page table and one for the data / instruction

-Solution? *Associative Registers*

Implementation of Page Table

- The two memory access problem can be solved by the use of a special fast lookup hardware cache called **translation look-aside buffers (TLBs) or Associative registers**
- TLBs are special-purpose hardware to improve access times for paging
- TLBs acts as cache for page table entries
- Only few page table entries are kept in the associative registers
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry
  - ASIDs uniquely identifies each process to provide address-space protection for that process

Paging Hardware With TLB



Page number of logical address generated by CPU presented to TLB. If page number is found, its frame number is available and is used to access memory
If page number is not in the TLB (known as TLB miss), a memory reference to the page table must be made, When the frame number is obtained, we can use it to access memory. In addition we can add page number and frame number to TLB, so that we can access it quickly in the next reference
If TLB is already full, the OS must select one for replacement (LRU or Random)
Some TLBs allow certain entries to be wired down (cannot be removed)
Typically TLB entries for kernel code are wired down

Associative Register Performance

- *Hit Ratio* - percentage of times that a page number is found in associative registers
  Effective access time (EAT) =
  hit ratio *x* hit time + miss ratio *x* miss time
  hit time = reg time + mem time
  miss time = reg time + mem time * 2

Performance

- Assume:
  - memory access takes 100 nsec
  - TLB access takes 20 nsec
- A 80% hit rate:
  - effective access time is
    = (0.8 * 120) + (0.2 * 220)
    = 140 nsec
  - *40% slowdown* in the memory access time
- A 98% hit rate:
  - effective access time is
    = (0.98 * 120) + (0.02 * 220)
    = 122 nsec
  - *22% slowdown* in the memory access time

TLB +
memory access for page table &
frame number +
memory access

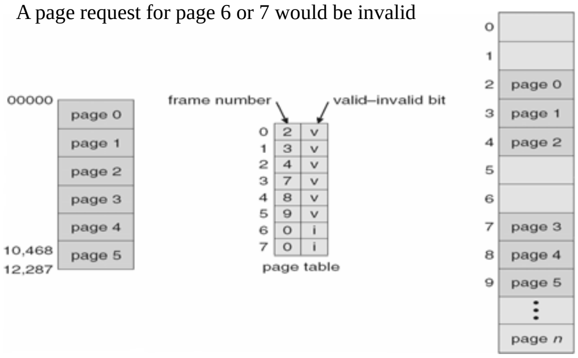
continued

Memory Protection

- Memory protection implemented by associating **protection bit** with each frame
- **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
- Any violations result in a trap to the kernel

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

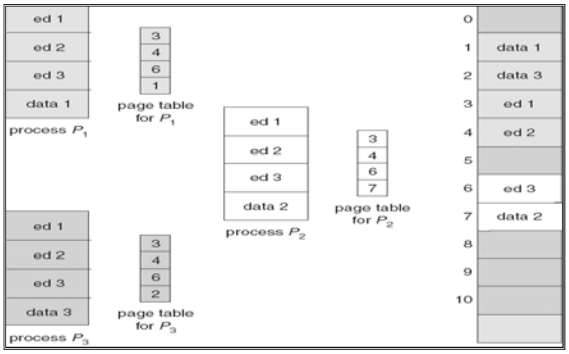
Pages 0 through 5 access valid frames in physical memory
A page request for page 6 or 7 would be invalid



Shared Pages

- Ordinary page tables allow sharing of common code
  - Useful for reusing *reentrant code* (read only code) (i.e. code that does not modify itself)
- Shared code must appear in same location in the logical address space of all processes
- Example
  - Three users of a text editor (150KB, split into three pages of size 50 KB) and their data (50KB each, one page each)

Editor Usage



Memory Savings

- Total physical memory usage (*with* sharing):
  = 150 + (3 * 50) = 300 KB
- Total physical memory usage (*without* sharing):
  = 3 * (150 + 50) = 600 KB

Problems

- Sharing relies on being able to map several pages (logical memory) to a single frame (physical memory).

Multilevel Paging

- In modern computer systems, the logical address space for a process is very large:
  - 2^{32}, 2^{64} bytes
  - the page table becomes **too** large
- Assume:
  - a 32 bit logical address space
  - page size is 4 KB(2^{12})
- Logical address format:

page number n	page offset d
20	12

- The page table must store 2^{20} addresses (~ 1 million), each of size 32 bits (4 bytes)
  - need 4 MB physical address space for page table
  - too big
- *Solution*: use a *two-level paging scheme* to make the page tables smaller.

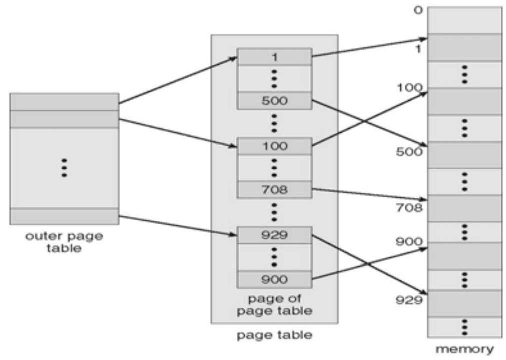
Two-level Paging Scheme

- Instead of using huge page tables, the page table itself can be paged
  - divide the page table into two levels
- The page number is further divided into a 10-bit page number and 10-bit page offset
- Thus logical address format is:

page number		page offset
p1	p2	d
10	10	12

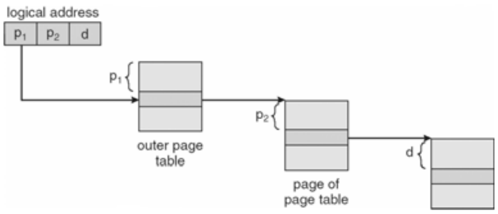
- p1 = index into the *outer page table*
- p2 = index into the page obtained from the outer page table
- The outer page table (and other page tables) will contain 2^{10} addresses (~1000) , each of size 32-bits (4 bytes)
  - page table size = 4KB

Two-Level Page-Table Scheme



Address-Translation Scheme

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



Three-level Paging Scheme

- For a 64-bit logical address space, a two level paging scheme is not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If we use two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Outer page table has 2^{42} entries or 2^{44} bytes
  - Logical Address would look like

outer page	inner page	page offset
p_1	p_2	d
42	10	12

  - One solution is to add a 2nd outer page table, giving us a three-level paging scheme

2nd outer page	outer page	inner page	offset
p1	p2	p3	d
32	10	10	12

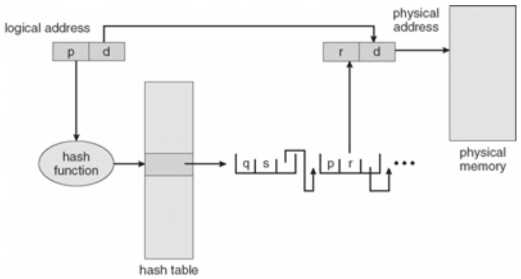
- The *second outer page table* (the new p1) still 2^{34} bytes in size
  - go to a four-level paging scheme!

continued

Hashed Page Tables

- Hashed page table is used if the address space > 32 bits
- The page number is hashed into a hash table to get virtual page number (hash value)
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions)
- Each element contains 3 fields
  - (1) the virtual page number
  - (2) the value of the mapped page frame
  - (3) a pointer to the next element in the linked list
- Virtual page numbers are compared with filed 1 for a match
  - If a match is found, the corresponding page frame (field 2) is extracted to get physical address
  - If there is no match , subsequent entries in the linked list are searched for a matching virtual page number

Hashed Page Tables



Clustered Page Tables

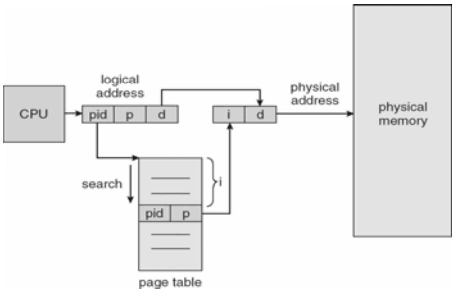
- **Clustered page tables is a** variation of hashed page tables that is proposed for 64-bit addresses spaces
- Similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

Inverted Page Table

- One of the drawbacks of ordinary page tables is that each page table may consists of millions of entries
- To solve this problem, we can use inverted page table
- An *inverted page table* has one entry for each frame, which says which PID (process ID) and page are using it (currently)
  - reduces the amount of physical memory required to store each page table, but increases time needed to search the table when a page reference occurs
- A logical address is represented by:
  < process-id, page-number, offset >

Inverted Page Table

When a memory reference occurs, part of logical address (<**process-id, page-number**>) is presented to memory subsystem.
The inverted page table is then searched for a match, if match is found at entry **i** (say), then the physical address <**i, offset**> is generated. If no match, then an illegal access has been attempted

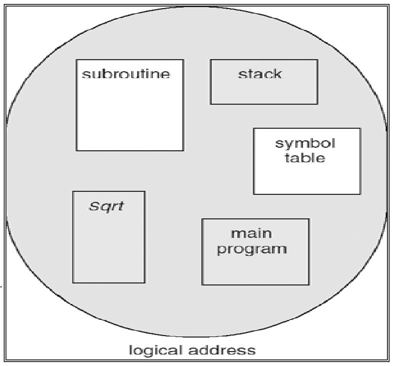


Segmentation

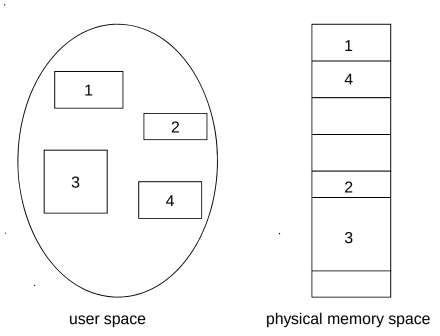
- Memory-management scheme that supports user view of memory
- A program is a collection of segments.
- A segment is a logical unit such as:

  main program,
  procedure,
  function,
  method,
  object,
  local variables, global variables,
  common block,
  stack,
  symbol table, arrays

User's View of a Program



Logical View of Segmentation

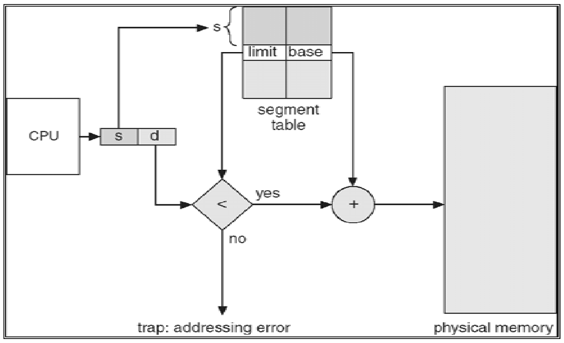


Segmentation Architecture

- Logical address consists of a two tuple:
  <segment-number (s), offset (d)>,
- Segment table maps two-dimensional user defined address into one-dimensional physical address
- Each segment table entry has:
  - **Segment base** – contains the starting physical address where the segments reside in memory
  - **Segment limit** – specifies the length of the segment

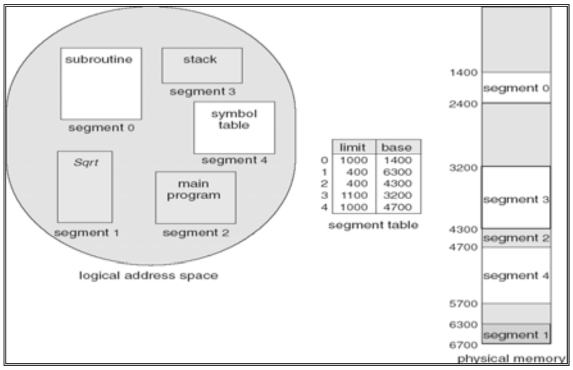
Segmentation Hardware

The offset **d** of the logical address must be between **0** and **segment limit**.
If not we **trap** to the OS, otherwise **offset** is added to **segment base** to get **physical address**



Example of Segmentation

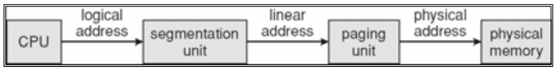
A reference to byte **53** of **segment 2** is mapped onto location **4300+53 = 4353**
A reference to byte **1222** of **segment 0** would result in a **trap** to the OS



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- In Pentium CPU generates logical address
  - 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



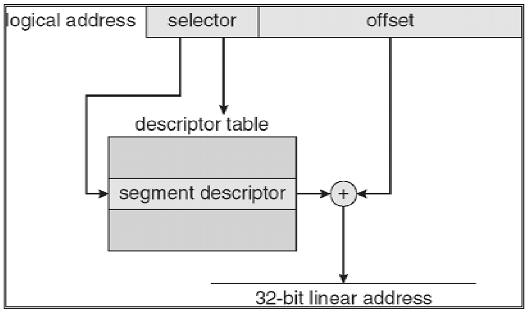
Pentium Segmentation

- Pentium allows a segment can be as large as 4 GB
  - Up to 16 K segments per process
  - Logical address space is 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**))
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit number
- CPU generates logical address is a pair (selector, offset)
  - Selector is a 16-bit number in which (s is segment number, g indicates whether the segment is in GDT or LDT, and p deals with protection)

s	*g*	*p*
13	1	2

  - Selector given to segmentation unit
    - Which produces linear addresses

Intel Pentium Segmentation



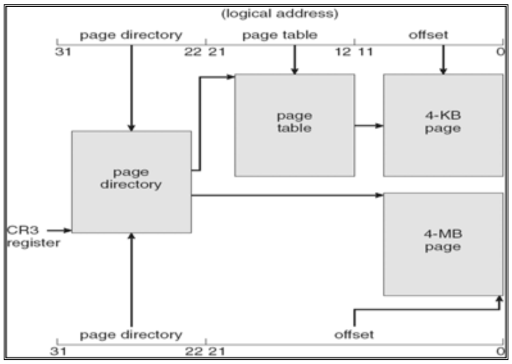
Pentium paging

- Pentium allows a page size of either 4 KB or 4 MB
- For 4KB pages, Pentium uses a two-level paging scheme in which division of the 32-bit linear address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

- 10 high-order bits reference an entry in outermost page table (page directory)
- One entry in page directory is page size flag which if set indicates that size of page frame is 4 MB
- If this flag is set, the page directory points to 4 MB page frame bypassing the inner page table and the 22 low-order bits linear address refer to the offset in the 4-MB page frame

Pentium paging architecture

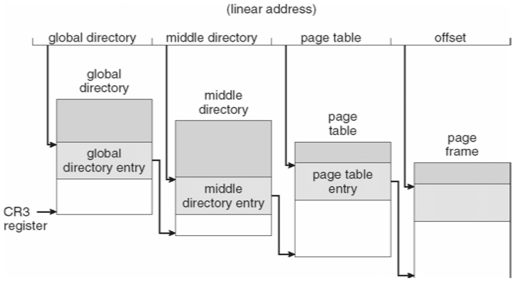


Linux on Pentium systems

- Linux uses only 6 segments (kernel code, kernel data, user code, user data, task-state segment (TSS), default LDT segment)
- Linux only uses two of four possible modes – kernel and user
- Uses a three-level paging strategy that works well for 32-bit and 64-bit systems
- Linear address broken into four parts:

global directory	middle directory	page table	offset

Three-level paging in Linux


(linear address)
global directory
middle directory
page table
offset
global directory
global directory entry
middle directory
middle directory entry
page table
page table entry
page frame
CR3 register