COP 4610

Operating System Principles

Memory Management

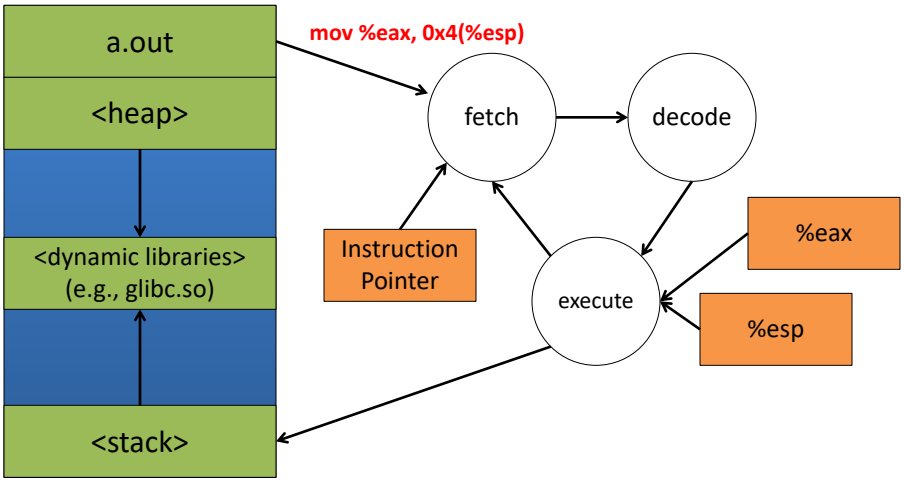
1

Overview

- Memory Access
- Address Binding
- Memory Protection
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

2

Memory Access



3

Program Execution & Memory

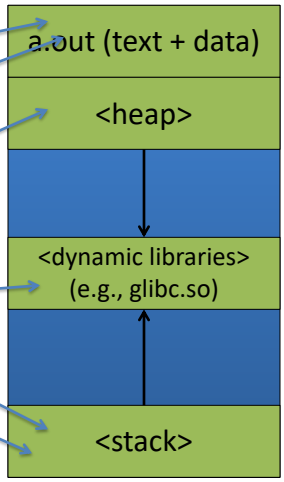
- Program (code & data) must be loaded into memory (from where?)
- Main memory & registers are the only storage the CPU can access
- Access to register: <= 1 clock cycle
- Access to memory: multiple cycles (“memory stall”)
- Cache sits between main memory & CPU
- Protection of memory needed

4

Variables & Storage Locations

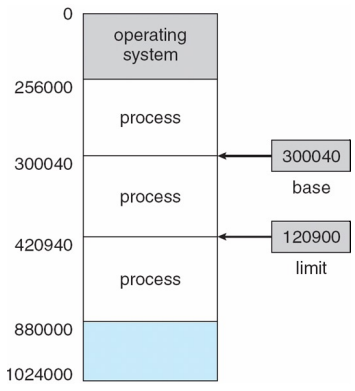
```
int global_variable;
const int constant_variable = 5;

int main(int argc, char *argv[]) {
   int *dynamic_variable = malloc(sizeof(int));
   int local_variable;
   printf("Hello world!\n");
   return 0;
}
```



5

Base and Limit Registers

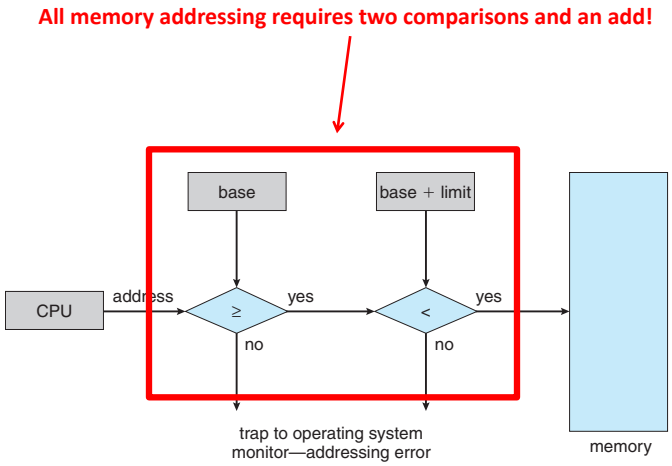


- **Base:** start address
- **Limit:** size

- CPU must check every memory access to be sure it is between base and limit for that user
- Processes cannot be moved!
- Processes cannot share memory!

6

Hardware Address Protection



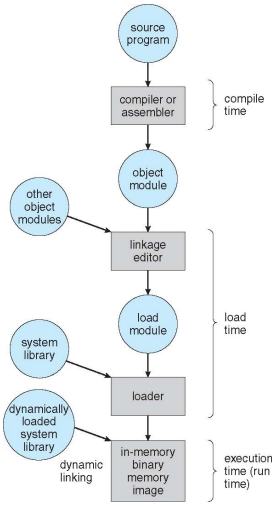
7

Address Binding

- **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)

8

Address Binding



9

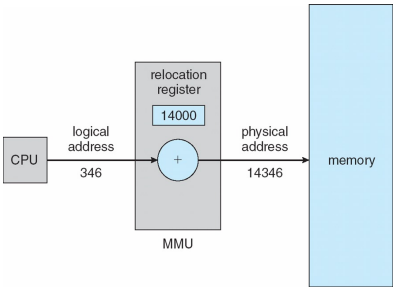
Logical & Physical Address

- **Physical address** – address seen by the memory unit
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- Compile-time & Load-time: same!
- Execution-time: differ!
- **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

10

Memory Management Unit - MMU

- Hardware device that at run time maps virtual to physical address
- Simplest MMU: add value in relocation register to logical address before accessing memory



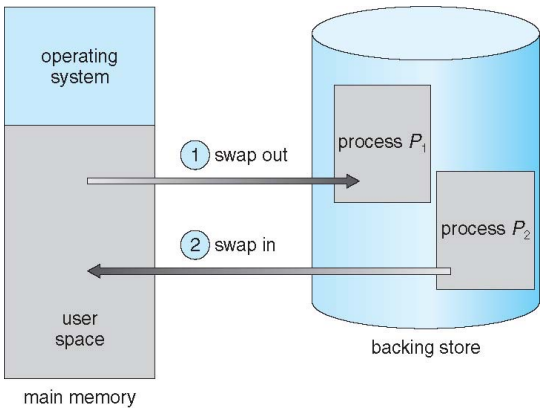
Dynamic Linking

- **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**

Swapping

- Process temporarily moved to backing store.
- **Backing store:** disk space containing process images.
- **Roll-out roll-in:** type of swapping where lower-priority process gets swapped out to make room for higher-priority process.
- Swapping typically very costly!
  - Typically disabled; starts when used memory goes above certain threshold and disabled again when it falls below threshold
  - Impacts context switch time

Swapping

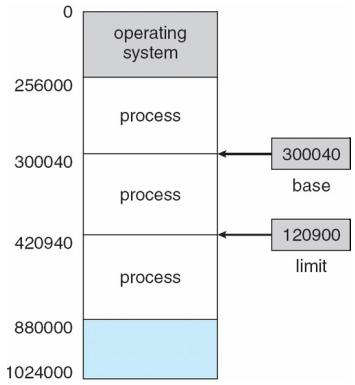


Swapping

- Constraints:
  - Don't swap memory with pending I/O (or always transfer I/O to kernel space and then user space – double buffering)
- Mobile:
  - Not typically supported; small amount of flash memory; large delays for writing/reading to/from flash
  - iOS asks apps to give up memory (read-only data thrown out and restored from flash when needed; iOS can force termination if needed)
  - Android terminates apps if low on memory (write application state to flash for quick restart)

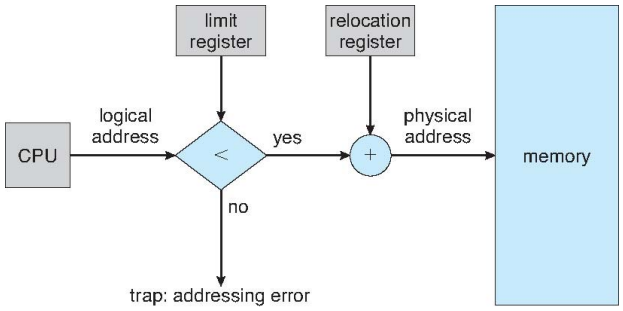
Contiguous Allocation

- OS & processes share memory
- Each process in single contiguous section of memory



Contiguous Allocation

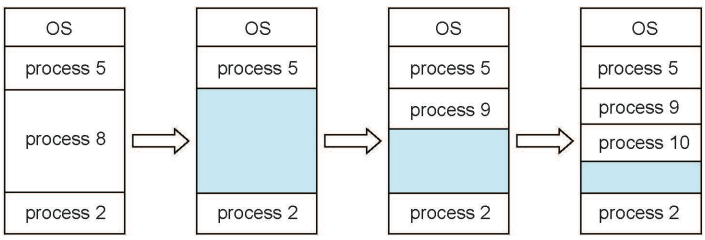
- **Base (relocation) register** contains value of smallest physical address
- **Limit register** contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address **dynamically**



17

Multiple/Variable-Partition Allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency
- **Hole**: block of available memory
- New process: allocate memory from a hole large enough to accommodate it
- Terminating process: free partition (adjacent free partitions combined)
- Operating system maintains information about: **allocated partitions** and **free partitions** (holes)



18

Dynamic Storage-Allocation Problem

- **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

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

Fragmentation

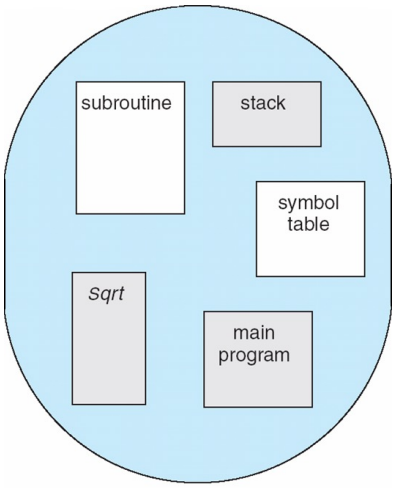
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O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

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of **segments**; logical units such as:

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

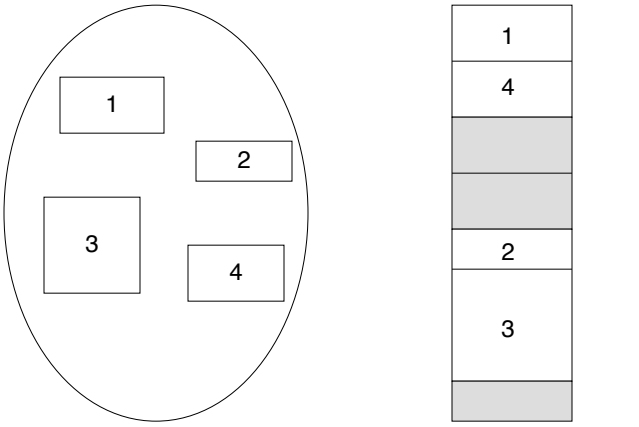
User's View of Program



23

23

Logical View of Segmentation



24

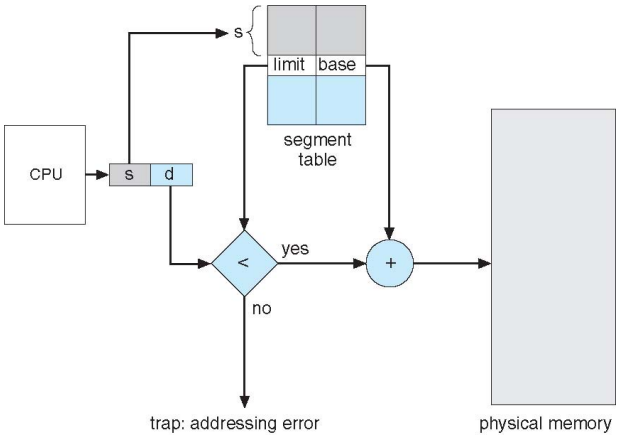
Segmentation Architecture

- Logical address consists of a two tuple:
  <segment-number, offset>,
- **Segment table**:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Architecture

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



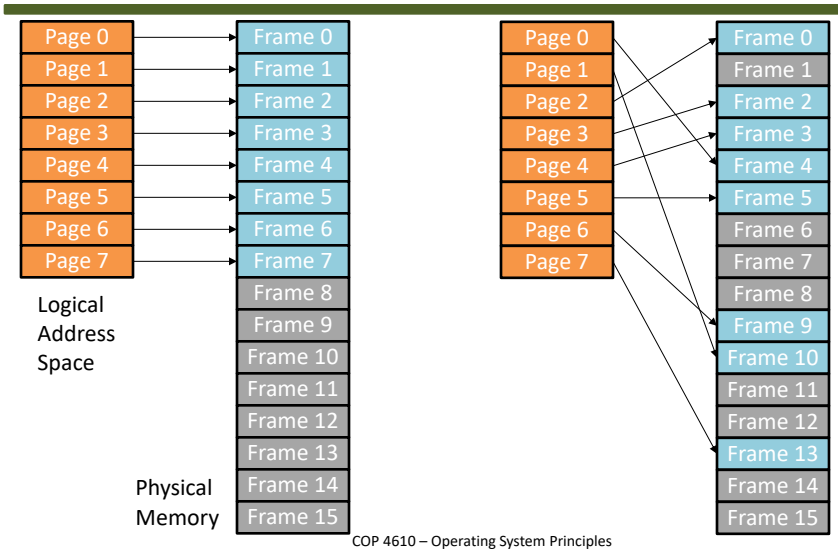
27

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

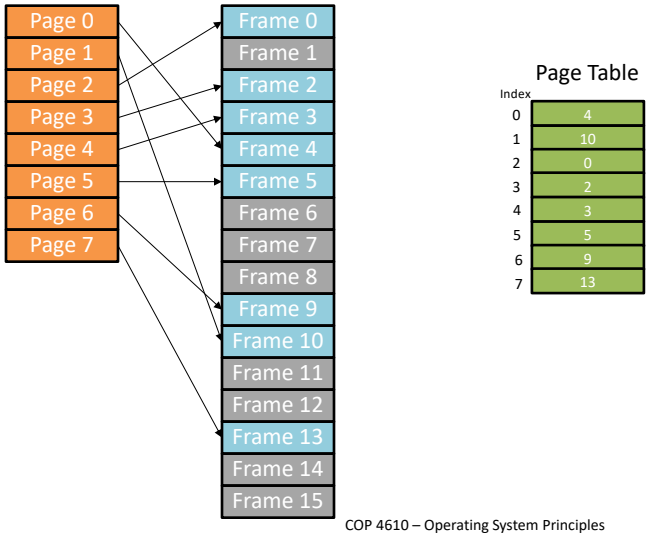
28

Paging



29

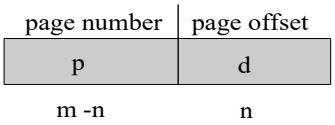
Paging



30

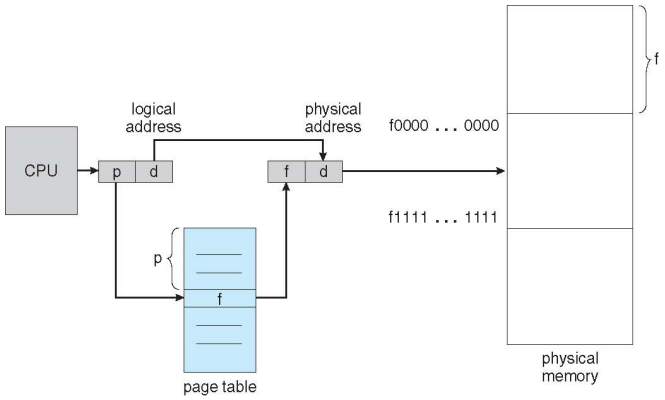
Address Translation Scheme

- 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

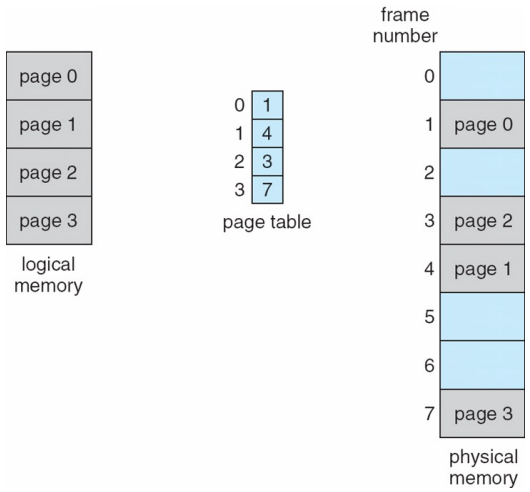


  - For given logical address space 2^m and page size 2^n

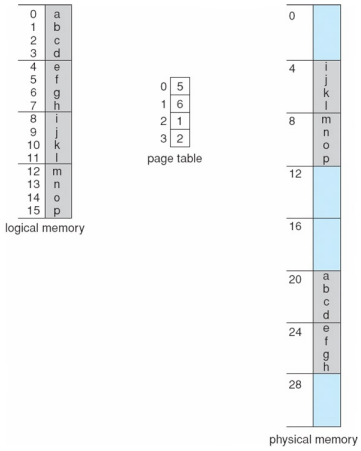
Paging Hardware



Paging

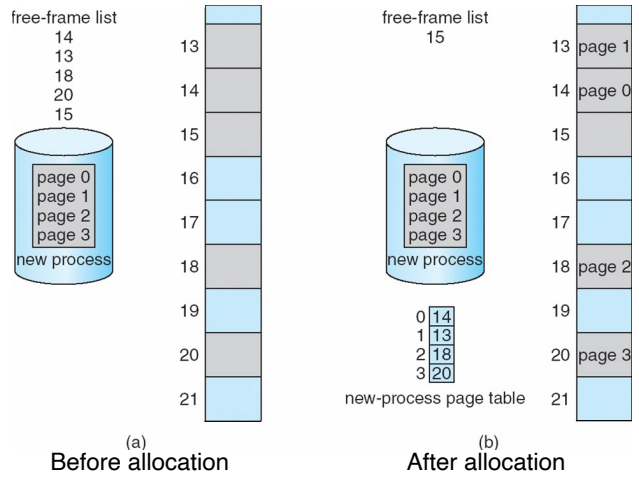


Paging Example



n=2 and *m*=5 32-byte memory and 4-byte pages

Free Frames



Page Table Implementation

- 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 **associative memory** or **translation look-aside buffers** (**TLBs**)

Associative Memory

- Associative memory – parallel search

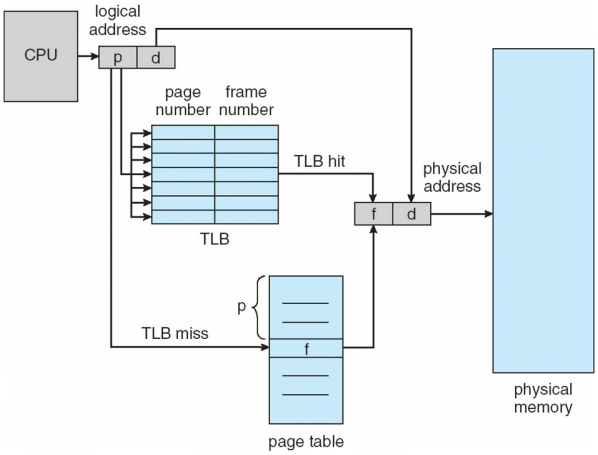
Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Page Table Implementation

- 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

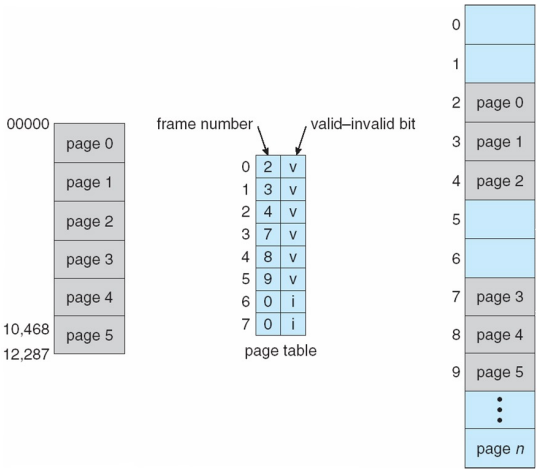
Paging Hardware with TLB



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

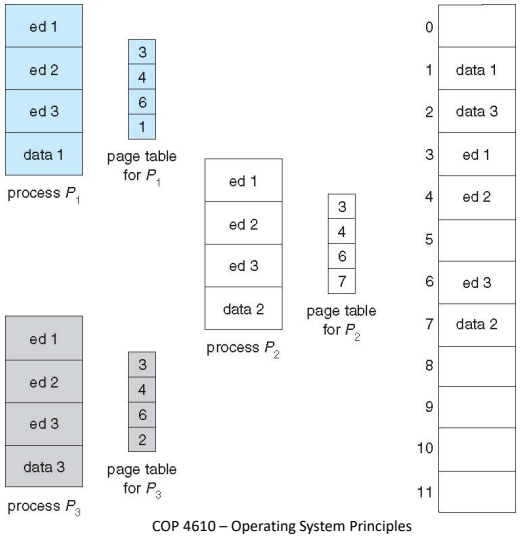
Valid/Invalid Bit



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



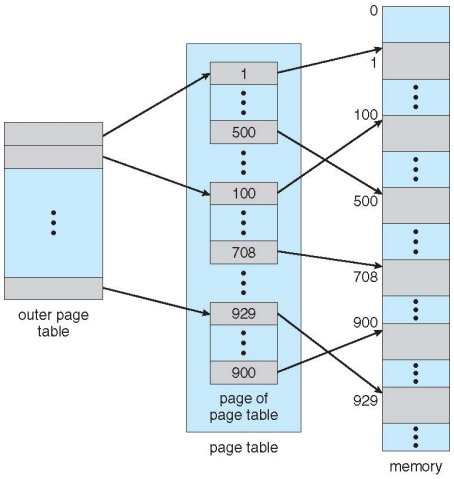
Structure of Page Table

- 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 -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

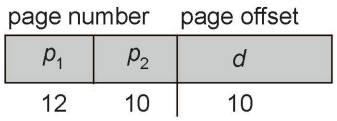
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



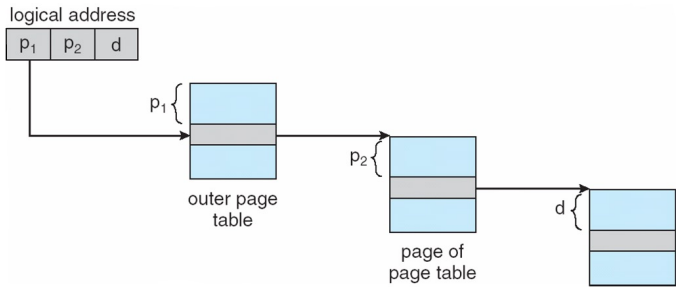
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



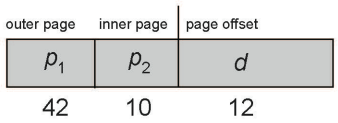
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address Translation Scheme



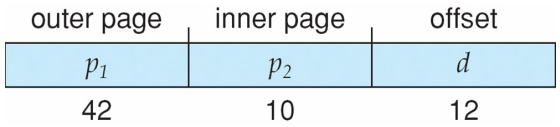
64-bit Logical Address Space

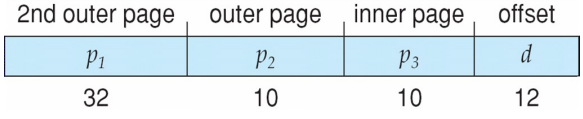
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - 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

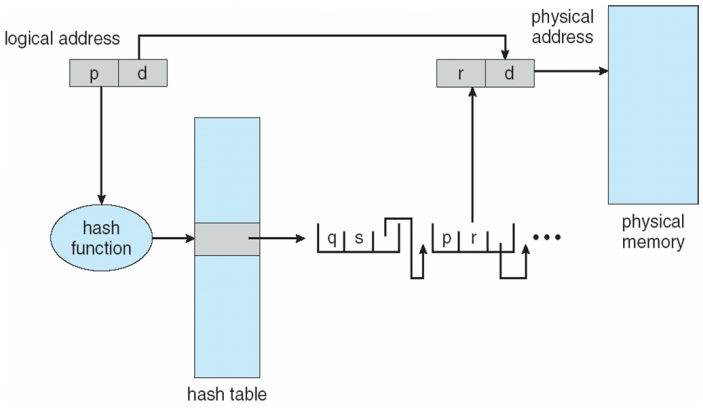




Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains:
  - the virtual page number
  - the value of the mapped page frame
  - 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

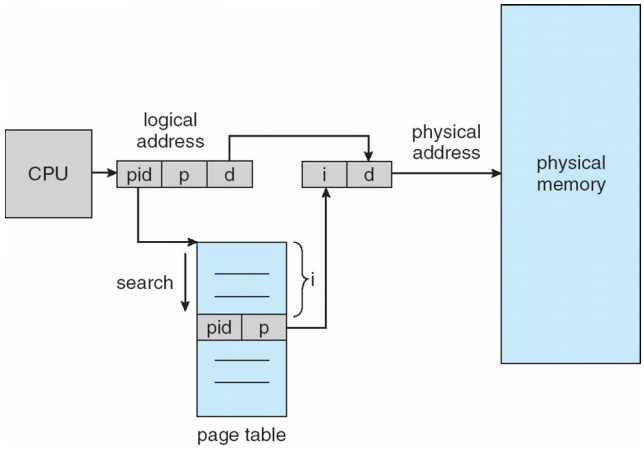
Hashed Page Tables



Inverted Page Tables

- 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

Inverted Page Tables



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)

Intel 32-bit & 64-bit

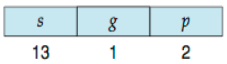
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

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 8K 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**))

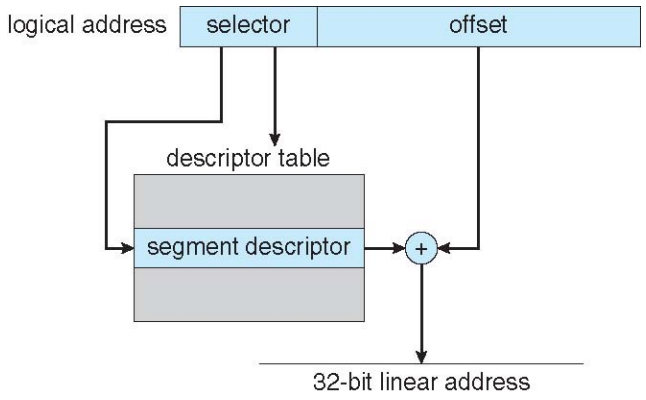
Intel IA-32 Architecture

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

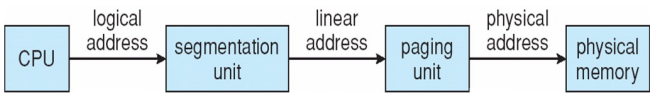


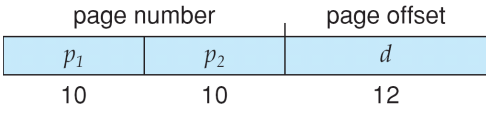
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Intel IA-32 Segmentation

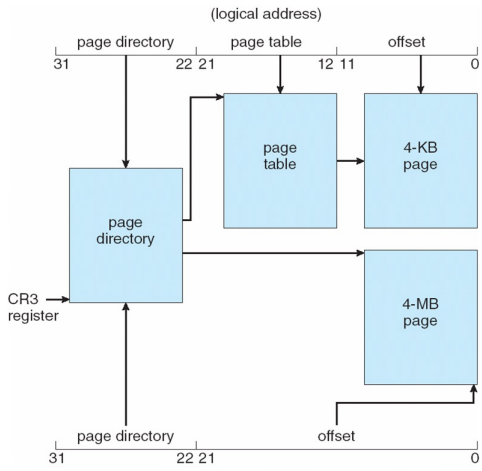


Logical to Physical in IA-32



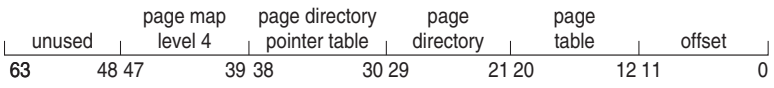


Intel IA-32 Paging



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



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