Operating Systems 2023 Spring Term
Week 11

Dr. Emrah İnan (emrahinan@iyte.edu.tr)

Main Memory

May 18, 2023

Week 11: Sample Glossary

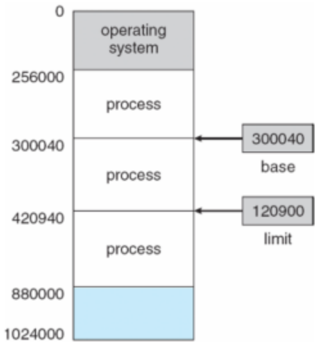
- **contiguous memory allocation:** A memory allocation method in which each process is contained in a single section of memory that is contiguous to the section containing the next process. (on Page 1244)
- **internal fragmentation:** Fragmentation that is internal to a partition. (on Page 1253)
- **compaction:** The shuffling of storage to consolidate used space and leave one or more large holes available for more efficient allocation. (on Page 1242)

Memory Management

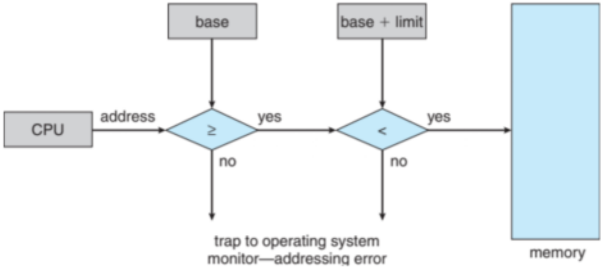
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or 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

Base and Limit Registers

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



Hardware Address Protection



Address Binding

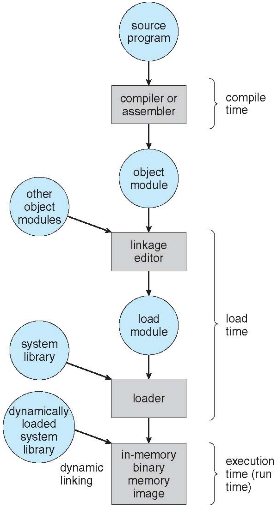
- Programs on disk, ready to be brought into memory to execute form an input queue
  Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  How can it not be?
- Further, addresses represented in different ways at different stages 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

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)

Multistep Processing of a User Program



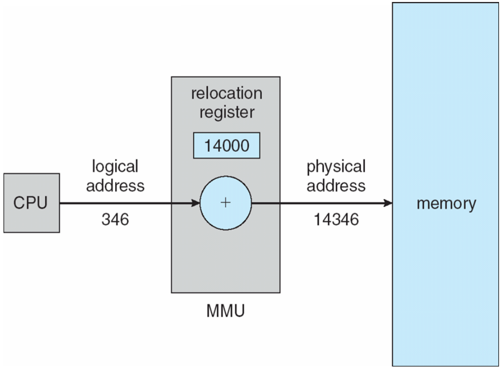
Logical vs. Physical Address Space

- 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

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
- To start, consider simple scheme where 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
  MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with logical addresses; it never sees the real physical addresses
  Execution-time binding occurs when reference is made to location in memory
  Logical address bound to physical addresses

Dynamic relocation using a relocation register



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

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

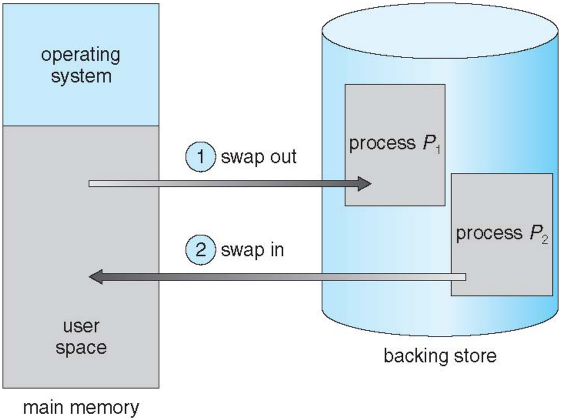
Swapping I

- 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 II

- 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 I

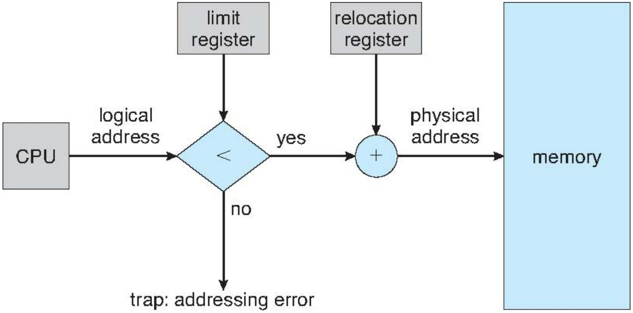
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  Swap out time of 2000 ms
  Plus swap in of same sized process
  Total context 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 including Swapping II

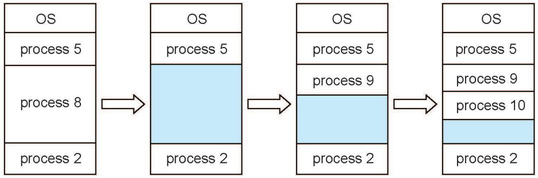
Concurrency arises in:

- 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

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation



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

Dynamic Storage-Allocation Problem

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

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

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

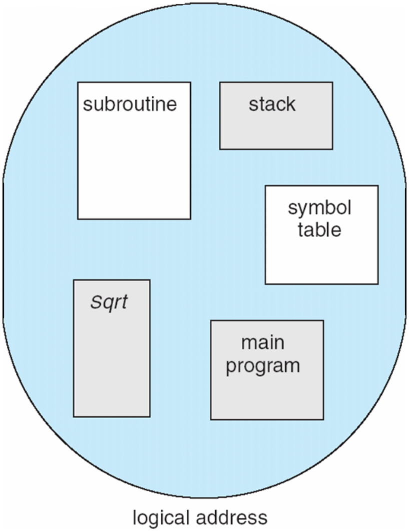
Fragmentation

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

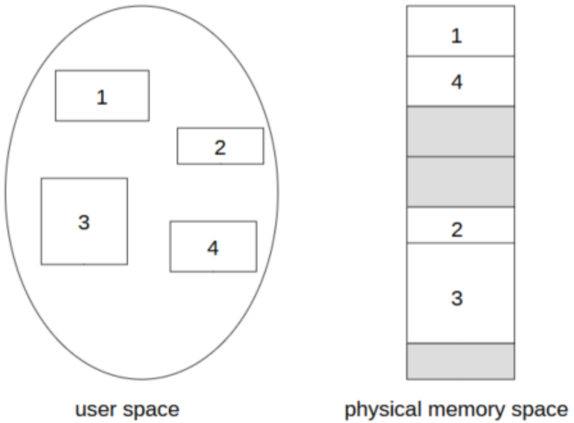
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  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



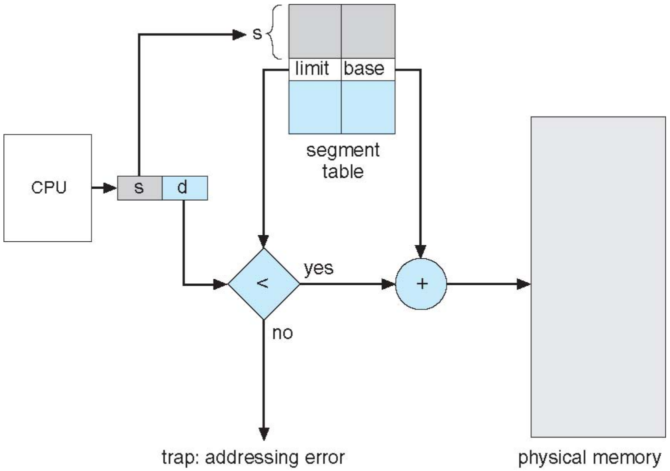
Segmentation Architecture I

- Logical address consists of a two tuple:
  <segment-number, offset>,
- Segment table – maps two-dimensional physical addresses; each table entry has:
  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 II

- 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



Paging

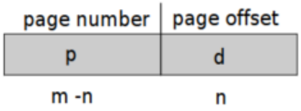
- Memory management discussed thus far has required the physical address space of a process to be contiguous. We now introduce paging, a memory management scheme that permits a process's physical address space to be non-contiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation.
- 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

Address Translation Scheme

Address generated by CPU is divided into:

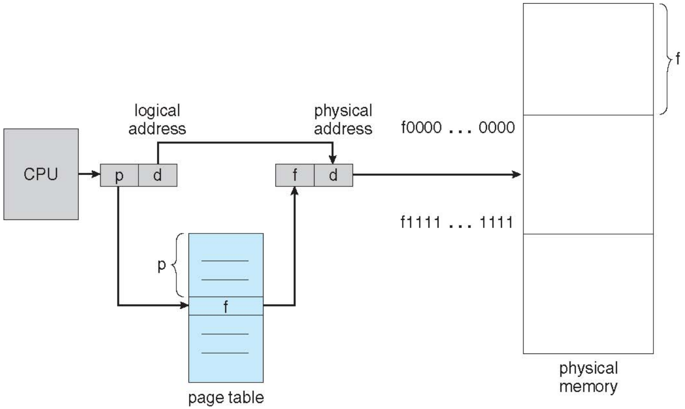
Page number (*p*) – used as an index into a **page table** which contains base address of each page in physical memory

Page offset (*d*) – combined with base address to define the physical memory address that is sent to the memory unit



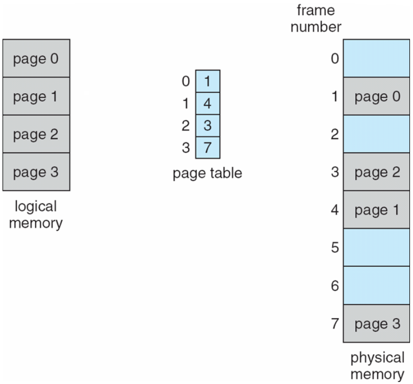
For given logical address space 2^m and page size 2^n

Paging Hardware



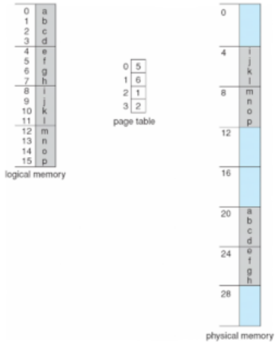
The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address.

Paging Model of Logical and Physical Memory



Suppose that page / frame size (PS) -> 100 and the logical address (LA) is 250
Page number (PN) = LA / PS
Offset = LA % PS -> Offset = 250 % 100 = 50; PN = 250 / 100 = 2
Physical address (the page number 2 refers to the frame number 3 in the page table) -> 3 * 100 + 50 = 350

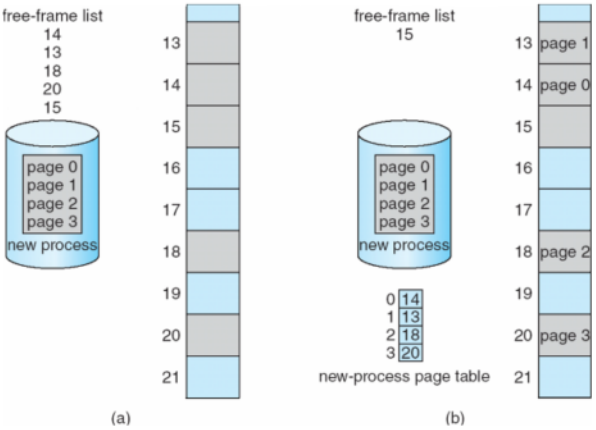
Paging Example



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

Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]

Free Frames



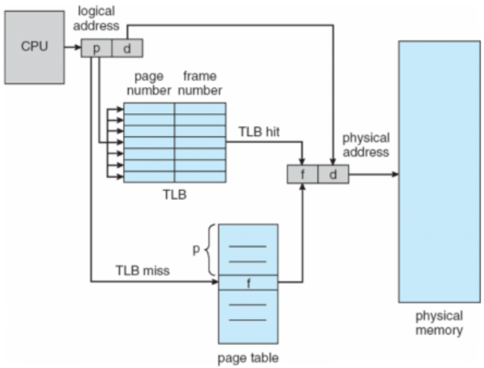
Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory
- Page-table base register (PTBR) points to the page table
- Page-table length register (PTLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

Paging Hardware With TLB

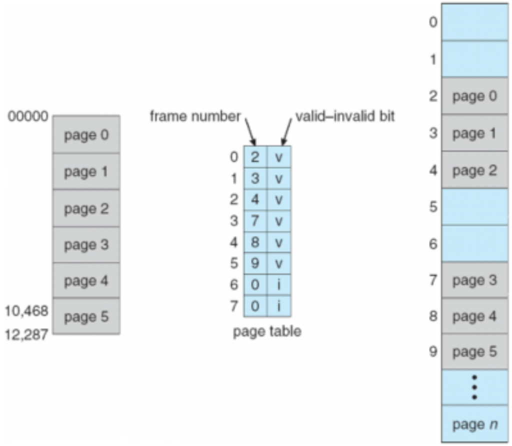


The percentage of times that the page number of interest is found in the TLB is called the hit ratio. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the 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 (v) or Invalid (i) Bit In A Page Table

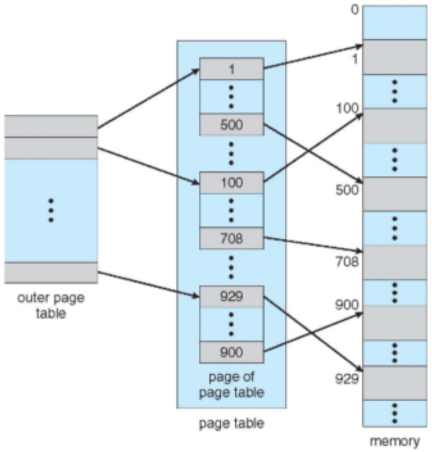


Structure of the Page Table

- 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 -> 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 Paging Example

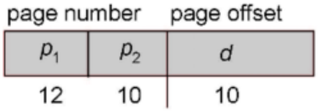
A logical address (on 32-bit machine with 1K page size) is divided into:

- 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

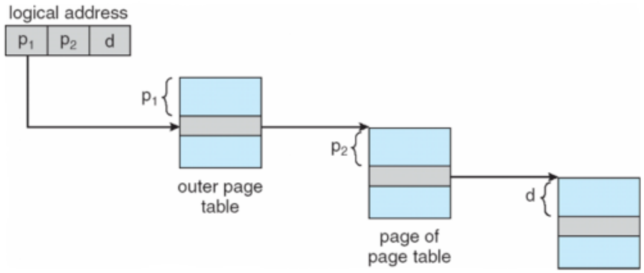
Thus, a logical address is as follows:



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**

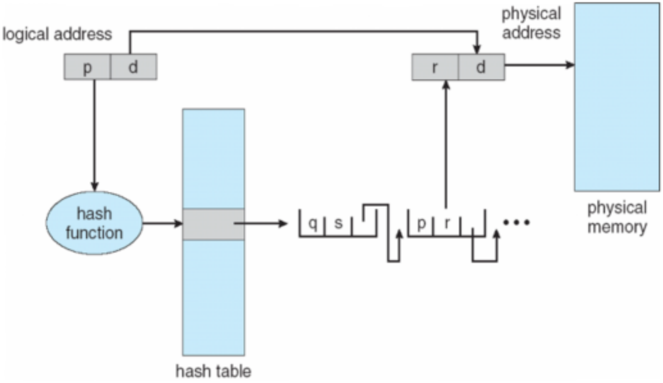
Address-Translation Scheme



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 (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)

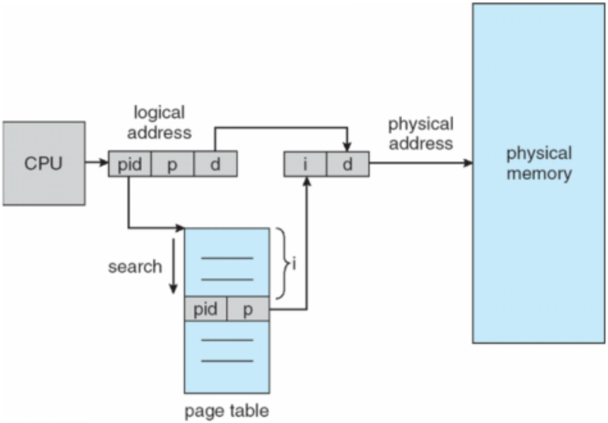
Hashed Page Table



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


CPU
logical address
pid
p
d
physical address
i
d
physical memory
search
i
pid
p
page table