Operating Systems - Main Memory

Mridul Sankar Barik

Jadavpur University

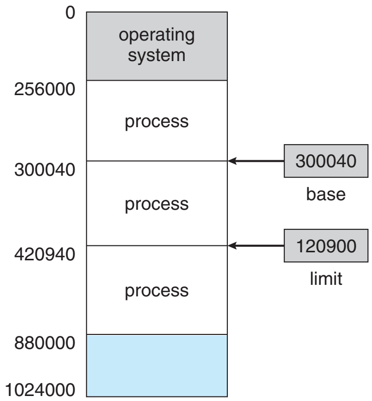
2024

Background

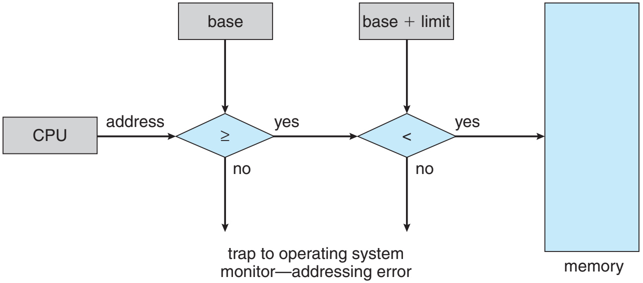
- Memory is large array of words or bytes each with its own address
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- CPU fetches instructions from memory according to the value of register Program Counter (PC)
- These instructions may cause additional loading from or storing to specific memory addresses
- Memory unit sees only a stream of memory addresses; it does not know
  - How they are generated
  - What they are for (instruction or data)
- 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



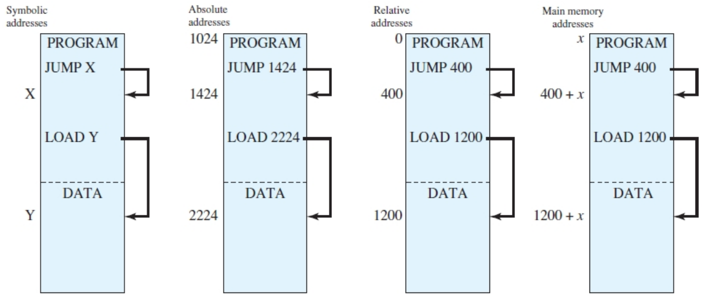
Hardware Address Protection



Address Binding

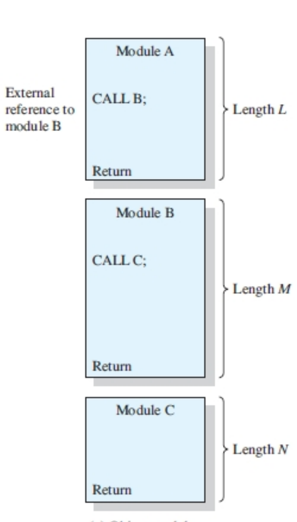
- Addresses are represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses bind to relocatable addresses, i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses, i.e. 74014
  - Each binding maps one address space to another
- Address binding 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
    - Example – MSDOS.COM programs
  - **Load time**: Compiler 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)

Absolute and Relocatable Load Module

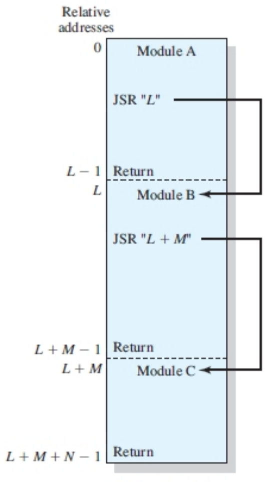


(a) Object module (b) Absolute load module (c) Relative load module (d) Relative load module loaded into main memory starting at location x

The Linking Function



(a) Object modules



(b) Load module

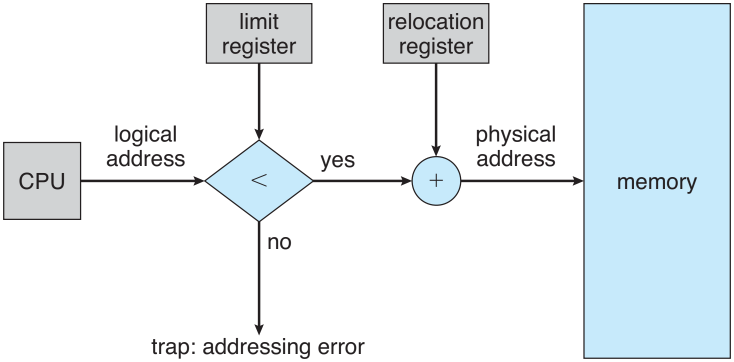
Logical vs. Physical Address Space

- 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
- Memory Management Unit
  - Hardware device that maps virtual to physical address
  - The user program deals with logical addresses; it never sees the real physical addresses
- **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

Contiguous Memory Allocation

- Main memory is usually divided into two partitions
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes held in high memory
- Each process is contained in a single contiguous section of memory
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
  - 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

Hardware Support for Relocation and Limit Registers



Multiple Partition Allocation

- Multiple-partition allocation
  - Degree of multiprogramming is limited by number of partitions
  - Variable-partition sizes used for efficiency
  - 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 are 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

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
- Reduce external fragmentation by
    1. 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
    2. Allow logical address space of a process to be non contiguous
        - Paging
        - Segmentation

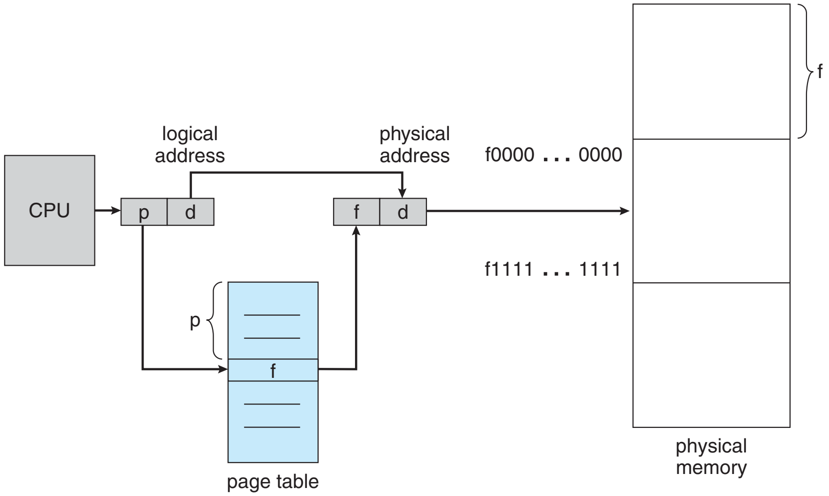
Paging

- 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, OS needs to find N free frames and load program
- Set up a page table to translate logical to physical addresses

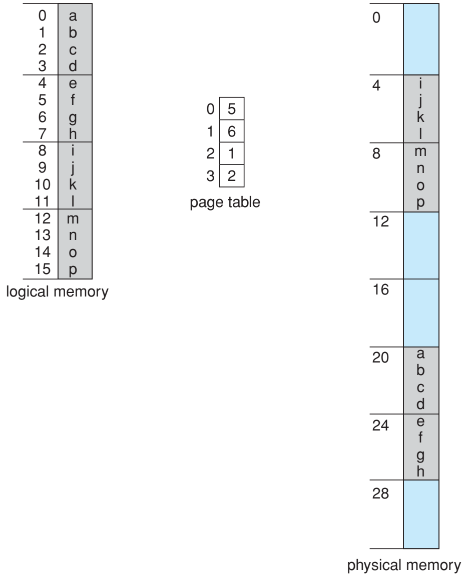
Address Translation Scheme

- 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

Paging Hardware



Paging Example



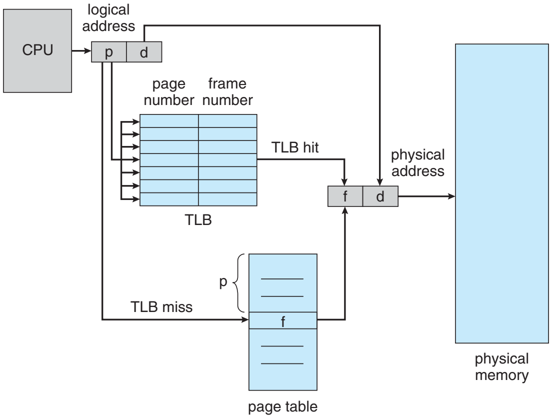
Paging

- 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
- Process view and physical memory now very different
- By implementation process can only access its own memory

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)
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, entry is loaded into the TLB for faster access next time
    - Replacement policies must be considered

Paging Hardware With TLB



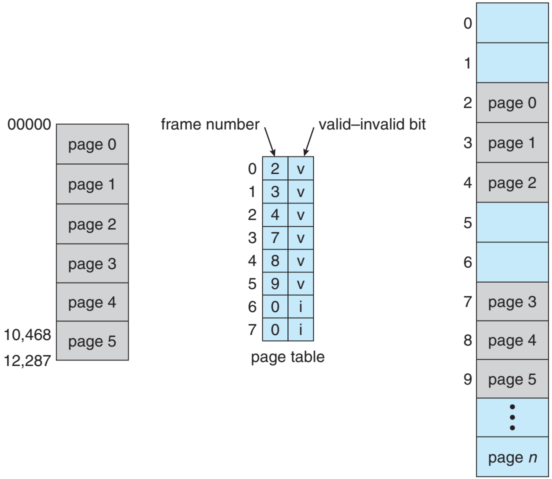
Effective Access Time

- Associative Lookup $= \epsilon$ time unit
  - Can be $<$ 10% of memory access time
- Hit ratio $= \alpha$
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider $\alpha = 80\%$, $\epsilon = 20$ ns for TLB search, $m_a = 100$ ns for memory access
- Effective Access Time (EAT)
  - $EAT = (m_a + \epsilon)\alpha + (2m_a + \epsilon)(1–\alpha)$
- $EAT = 0.80 \times (100 + 20) + 0.20 \times (200 + 20) = 96 + 44 = 140$ ns
- Consider more realistic hit ratio $\Rightarrow \alpha = 99\%$, $\epsilon = 20$ ns for TLB search, $m_a = 100$ ns for memory access
  - $EAT = 0.99 \times (100 + 20) + 0.01 \times (200 + 20) = 118.8 + 2.2 = 121$ ns

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
- Any violations result in a trap to the kernel

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



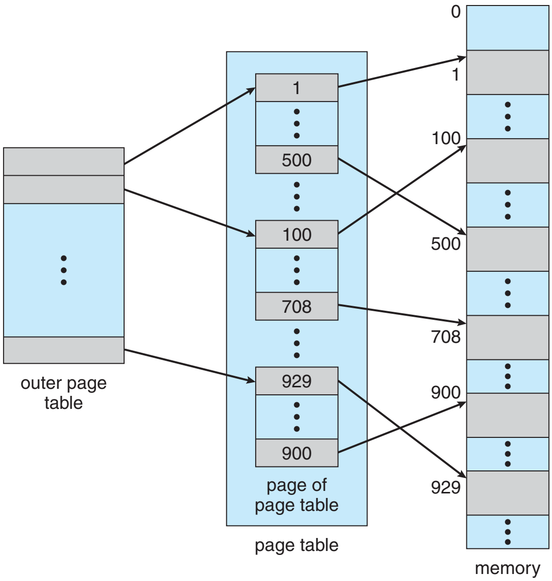
Structure of the Page Table

- 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 ($2^{32}/2^{12} = 2^{20}$) entries
    - If each entry is 4 bytes $\rightarrow$ 4 MB of physical address space / memory for page table alone
        - That amount of memory may not be available in contiguous areas 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

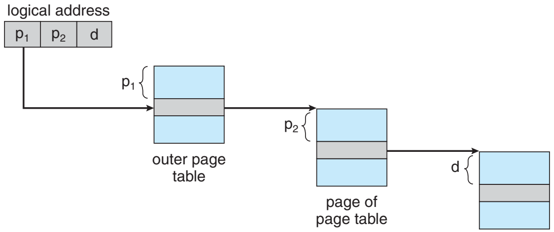
Two-Level Page-Table Scheme



Two-Level Paging Example

- A logical address (on 32-bit machine with 2K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset

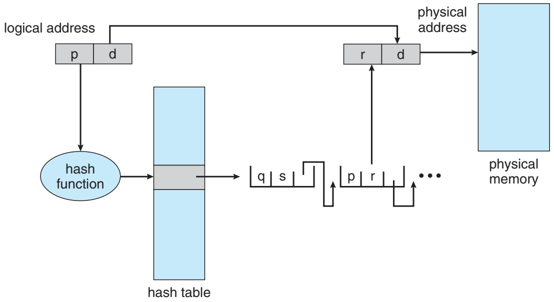
Address-Translation Scheme



Hashed Page Tables I

- 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

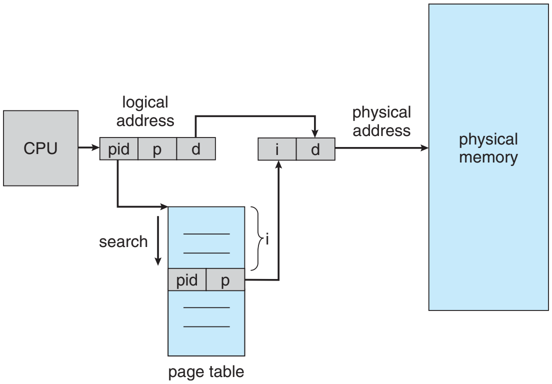
Hashed Page Tables II



Inverted Page Table I

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical frames
- One entry for each real frame 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

Inverted Page Table II


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