Soongsil Univ.
http://design.ssu.ac.kr

Digital System Design Lab.

Ch. 09 Main Memory

Chanho Lee
Soongsil University

1

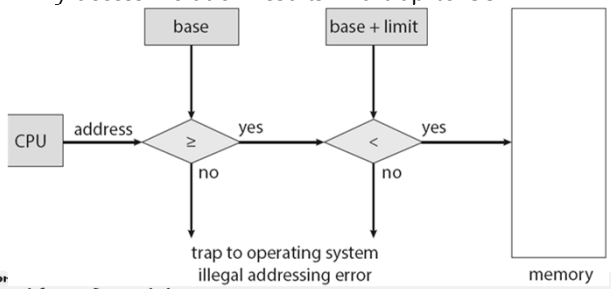
Objectives

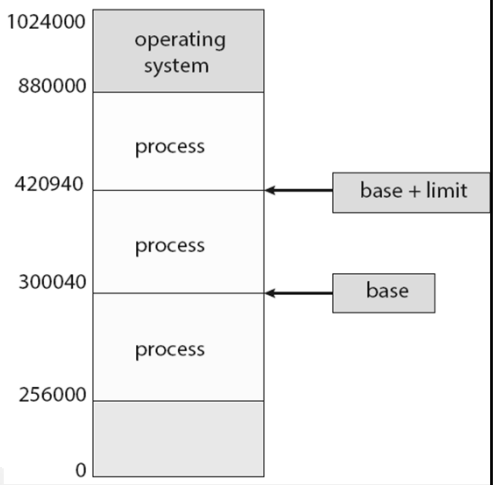
- Memory management unit (MMU)
  - ►Logical and a physical address
  - ►Translating addresses.
- Contiguous memory allocation
  - ►First-, best-, and worst-fit strategies
- Internal and external fragmentation.
- Paging system
  - ►Translation look-aside buffer (TLB).
  - ►hierarchical paging, hashed paging, and inverted page tables.
  - ►address translation for IA-32, x86-64, and ARMv8 architectures.

2

9.1 Background

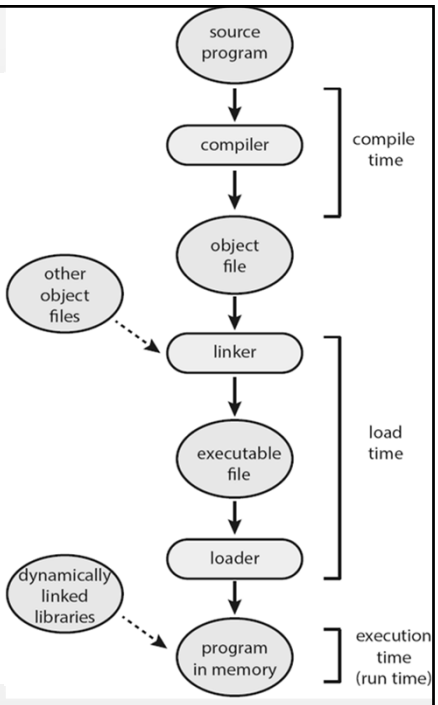
- Basic Hardware
  - ►CPU can access directly main memory and the registers only: data holder
  - ►Main memory: many cycles for access → processor stall → cache
    - DDR5 DRAM latency: 13.75 ~ 18.18ns: 80 ~ 106 cycles @5.8GHz, CL: 22@3.2GHz ~ 46@6.4GHz
  - ►Protection OS from a user process and user processes from another user during memory access
    - Separate per-process memory space
    - Hardware implementation: (one possible implementation) base register and limit register → controlled by OS
    - Any access violation results in a trap to OS





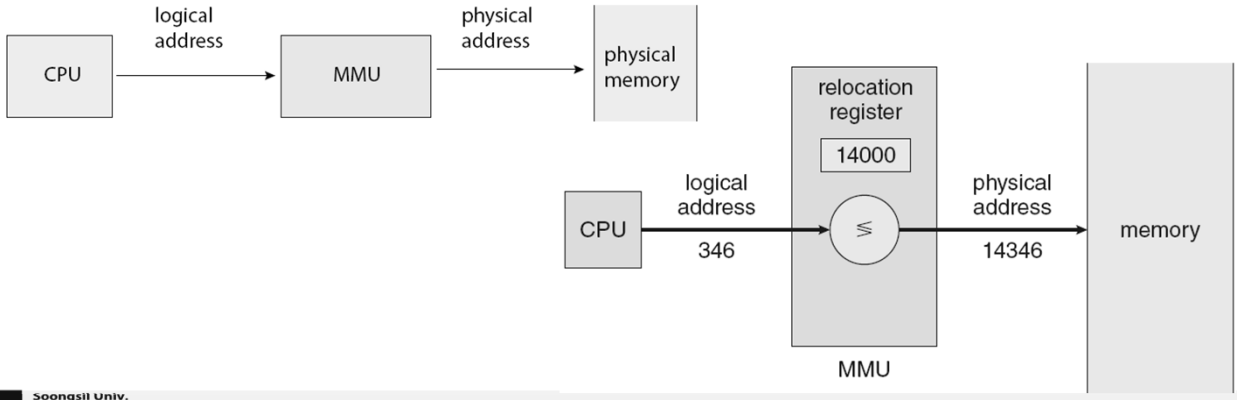
9.1 Background

- Address Binding
  - ►User source program → executable → process
  - ►Compile time: binds symbolic addresses to relocatable ones
    - Memory location known → absolute code (with linker)
    - Change of address → recompile the code.
  - ►Load time: binds relocatable addresses to absolute ones
    - Memory location known → absolute code after linker
    - Memory location unknown → final binding at load time.
    - Change of address → reload the code
  - ►Execution time.
    - Process relocation during execution → binding at run time.
    - Special hardware (Section 9.1.3).
    - Most operating systems



9.1 Background

- Logical Versus Physical Address Space
  - ►Logical address: address generated by the CPU
  - ►Physical address: address seen by the memory unit
  - ►Logical address is referred as virtual address when logical ≠ physical address
  - ►Memory-management unit (MMU): run-time mapping from virtual to physical addresses
    - Simple scheme using a relocation register

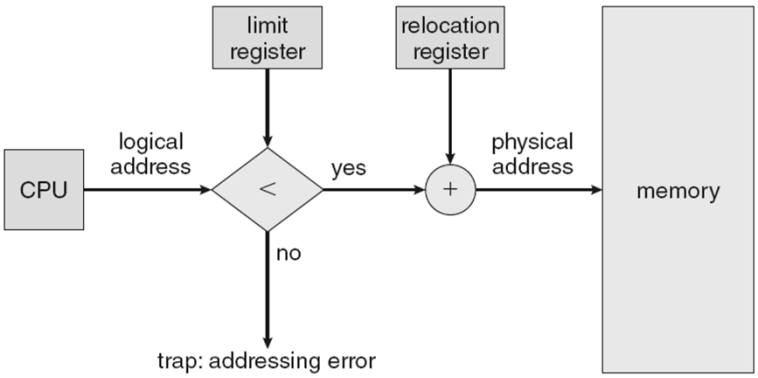


9.1 Background

- Dynamic Loading
  - ►Objective: to obtain better memory-space utilization
  - ►The main program is loaded and is executed
    - a routine is loaded when it is called
    - useful when large amounts of code are needed to handle infrequent cases, such as error routines
      → loaded program portion may be much smaller although the total size may be large
  - ►OS may help the programmer by providing library routines to implement dynamic loading
- Dynamic Linking and Shared Libraries
  - ►Dynamically linked libraries (DLLs): system libraries linked when the user programs are run
    - Cf) static linking: system libraries are combined by the loader into the binary program image
  - ►Also known as shared libraries: used extensively in Windows and Linux
    - shared among multiple processes
  - ►references a routine in a dynamic library → loader locates the DLL → loading if not in memory
  - ►Version information included for library updates: libraries with corresponding version are used
  - ►Managed by OS

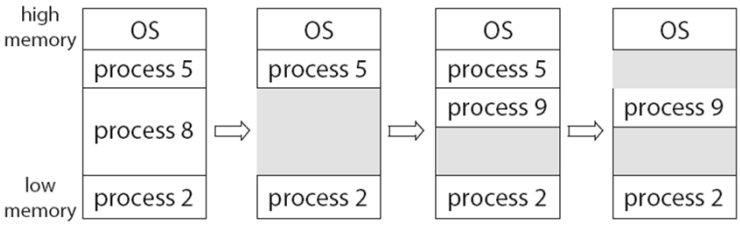
9.2 Contiguous Memory Allocation

- Many OS (including Linux and Windows) place the OS in high memory addresses
- Memory Protection
  - ►Relocation and limit register
  - ►Flexibility in OS's size: ex. loading and unloading device drivers



9.2 Contiguous Memory Allocation

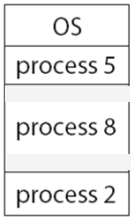
- Memory Allocation: Variable partition scheme
  - ►Each partition contains exactly one process
  - ►keeps a table for holes and occupied partition



  - ►Not sufficient memory → reject and error message, or into a wait queue
  - ►Dynamic storage allocation problem: how to allocate a process into a hole
    - First fit: allocate the first hole that is big enough. faster
    - Best fit: allocate the smallest hole that is big enough.
    - Worst fit: allocate the largest hole. Worst in terms of storage utilization

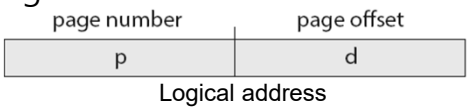
9.2 Contiguous Memory Allocation

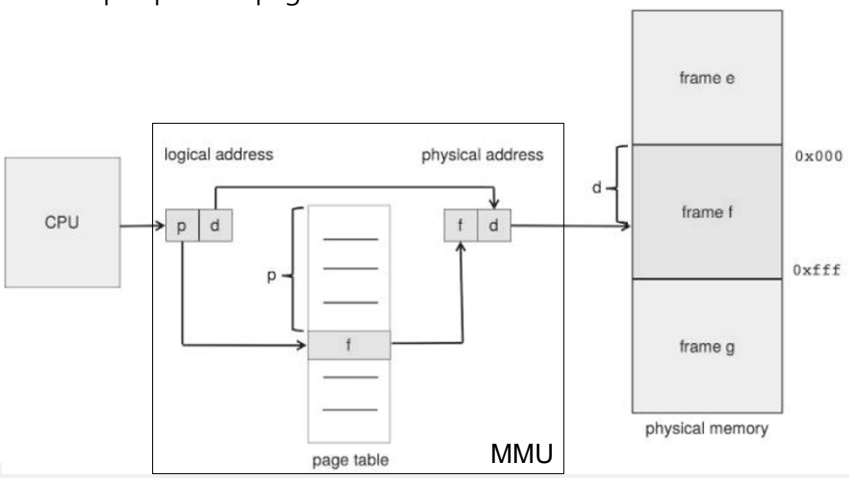
- Fragmentation
  - ►External fragmentation: many small holes (variable partition)
    - Worst case: holes between every two processes
  - ►50-percent rule: N allocated blocks and 0.5N fragmented blocks, statistically.
  - ►The overhead to keep track of small holes → fixed partition size
  - ►Internal fragmentation: unused memory internal to a partition
  - ►Compaction
    - place all free memory together in one large block
    - possible only if relocation is dynamic and is done at execution time
    - expensive



9.3 Paging

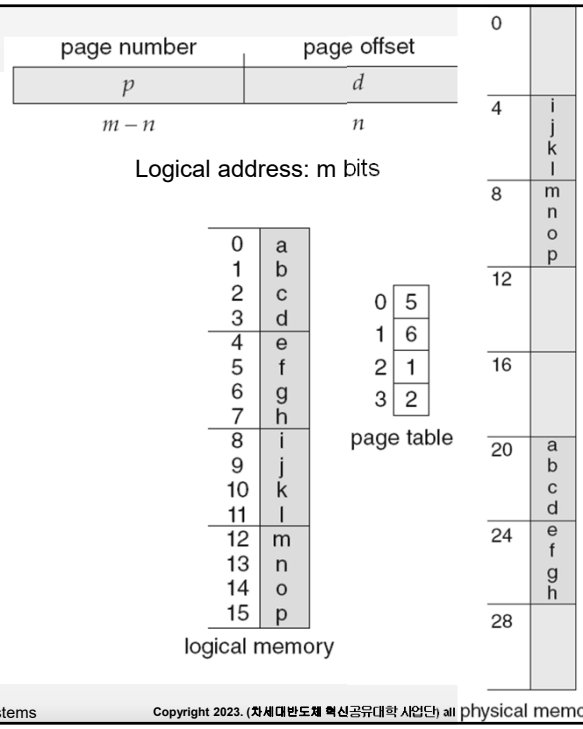
- Process's physical address space to be noncontiguous
- Basic Method
  - ►Memory blocks of fixed size: page vs. frame
  - ►page number: index of a per-process page table

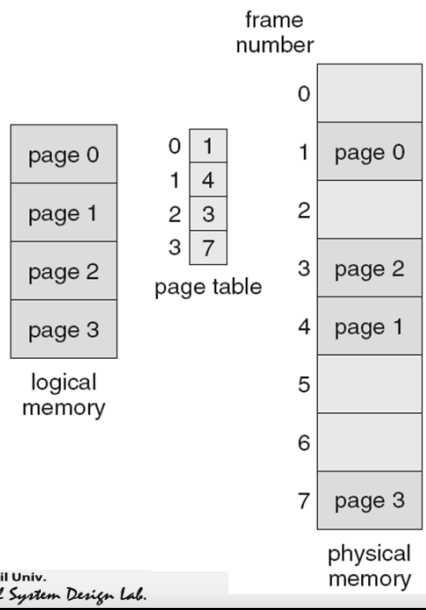




9.3 Paging

- ►Page size: power of 2, typically 4/8 KB or larger





11

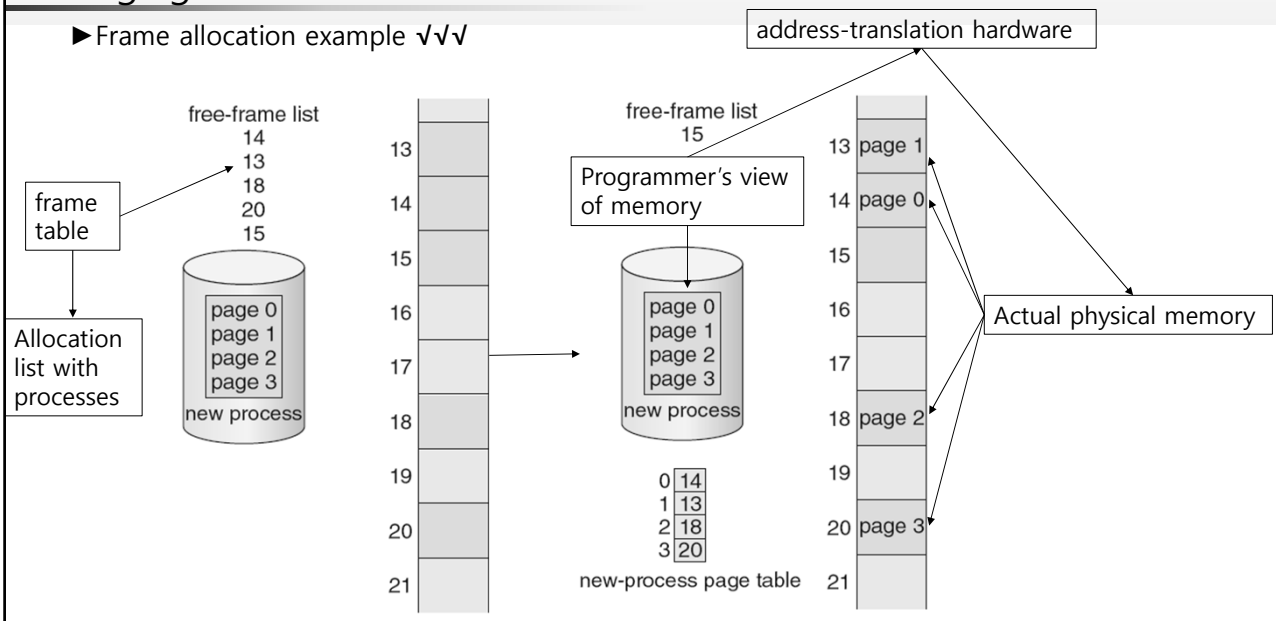
9.3 Paging

- ►No external fragmentation
- ►Internal fragmentation
  - Page size: 2kB, and a process of 72,766 bytes: 35 pages plus 1,086 bytes → 36 frames, internal fragmentation of 962 bytes
  - small page sizes are desirable for internal fragmentation
- ►Page sizes have grown as processes, data sets, and main memory have become larger
  - overhead is involved in each page table entry
  - Windows10: 4 KB and 2 MB
  - Linux: a default page size (typically 4 KB) and an architecture dependent larger page size called huge pages
    - System call `getpagesize()`
    - getconf `PAGESIZE`
- ►Page-table entry
  - 4 bytes long on a 32-bit CPU: 2^{32} physical page (4kB) → 2^{44} bytes (or 16 TB) of physical memory
- ►OS maintains a copy of the page table for each process
  - Used for a system call with an address as a parameter (e.g, buffer address): address translation
  - Used by CPU dispatcher
  - Paging increases the context-switch time

12

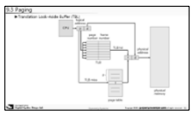
9.3 Paging

►Frame allocation example √√√



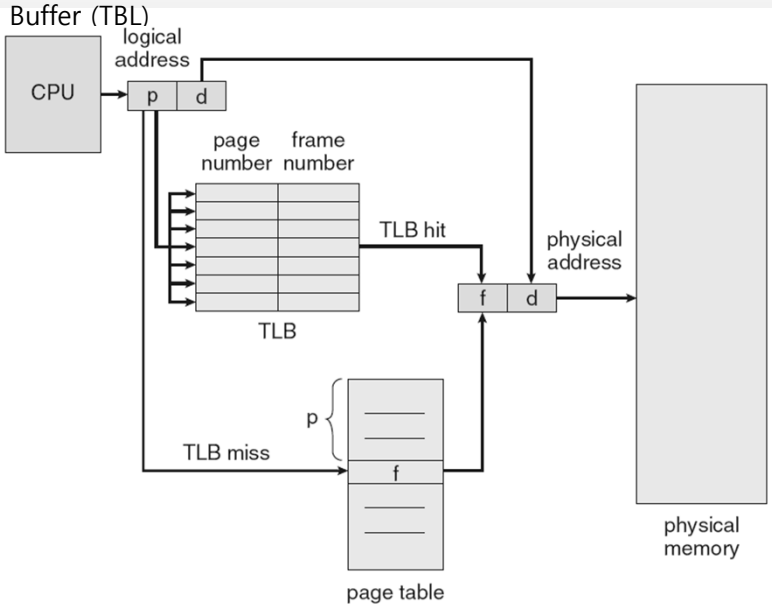
9.3 Paging

- Hardware Support
  - ►Dedicated hardware registers for page table: efficient
    - Not feasible for large page tables (e.g. 2^{20} for contemporary CPUs)
  - ►Reloading page table: increasing context-switching time
    - Large page tables in memory with page-table base register (PTBR)
    - Slow memory access time: two memory access (page table + data)
  - ►Translation Look-Aside Buffer (TBL)
    - special, small, fast-lookup hardware cache
    - part of the instruction pipeline, essentially adding no performance penalty
    - typically between 32 and 1,024 entries
    - TLB hit (no penalty), TLB miss (page number access and adding the entry to TLB) TLB
    - Replacement policy: from least recently used (LRU) through round-robin to random
      - By OS or by CPU
    - Wired down entries in some TLBs: never removed entries (especially for key kernel code)
    - address-space identifier (ASIDs) in each TLB entry in some TLB:
      - uniquely identifies each process
      - to provide address-space protection for that process



9.3 Paging

►Translation Look-Aside Buffer (TBL)



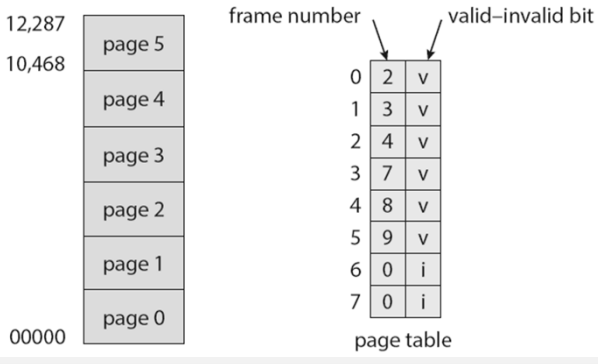
9.3 Paging

►Translation Look-Aside Buffer (TBL)

- hit ratio and effective memory-access time
  - effective access time = 0.80 × 10 + 0.20 × 20 = 12 ns (20% ↓)
  - effective access time = 0.99 × 10 + 0.01 × 20 = 10.1 ns (1% ↓)
- multiple levels of TLBs: ex. Intel Core i7
  - 128-entry L1 instruction TLB and a 64-entry L1 data TLB.
  - A miss at L1→ L2 512-entry TLB (six cycles)
  - A miss in L2 → page-table entries in memory (hundreds of cycles, or interrupt to the OS)
- TLB design (hardware) affects paging implementation of the OS

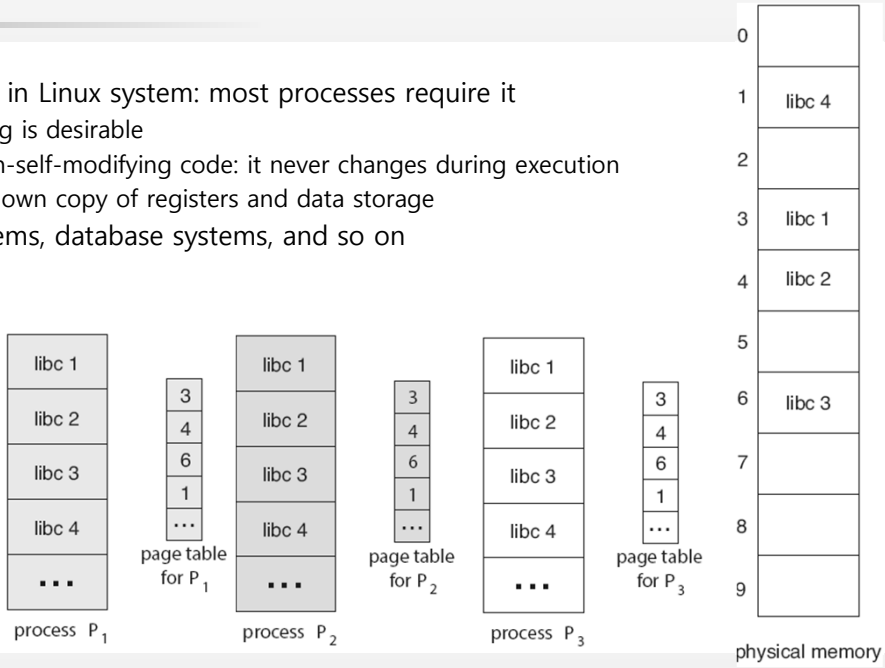
9.3 Paging

- Protection
  - ▶Protection bits in page table
    - Combinations of read, write, and execute
  - ▶Valid bit: valid pages are in the logical address space
    - Ex. 14 bit address (0 ~ 16383): a program with addresses of 0 ~ 10468
      → pages 0 ~ 5
  - ▶page-table length register (PTLR): to indicate the size of the page table
    → to check valid address range



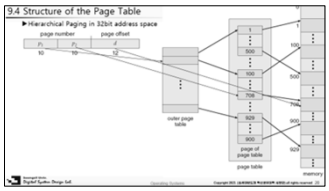
9.3 Paging

- Shared Pages
  - ▶Standard C library `libc` in Linux system: most processes require it
    - To save memory, sharing is desirable
      - Reentrant code : non-self-modifying code: it never changes during execution
      - Each process has its own copy of registers and data storage
  - ▶compilers, window systems, database systems, and so on

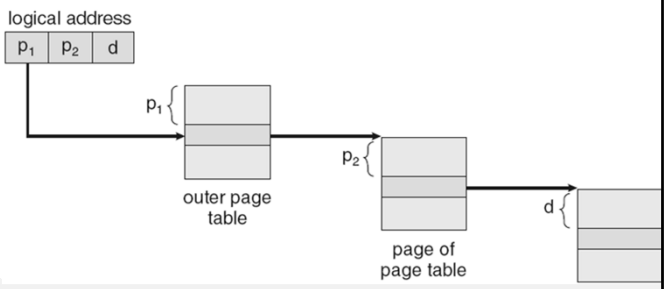


9.4 Structure of the Page Table

- Hierarchical Paging
  - ▶Page table size
    - 32-bit logical address space + page size 4 KB (2^{12}) → 1M entry page table ($2^{20} = 2^{32}/2^{12}$).
    - 4 bytes entry → 4 MB of physical address space
  - ▶One simple solution: to divide the page table into smaller pieces
    - p1: index into the outer page table
    - p2: displacement within the page of the inner page table



page number		page offset
p_1	p_2	d
10	10	12



  - ▶64bit address space
    - hierarchical page tables are inappropriate

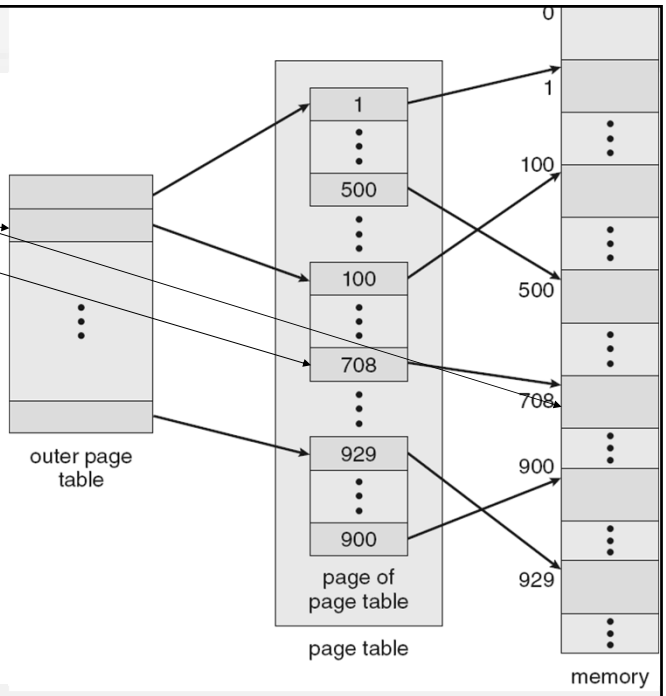
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

2^{34} bytes (16 GB)

9.4 Structure of the Page Table

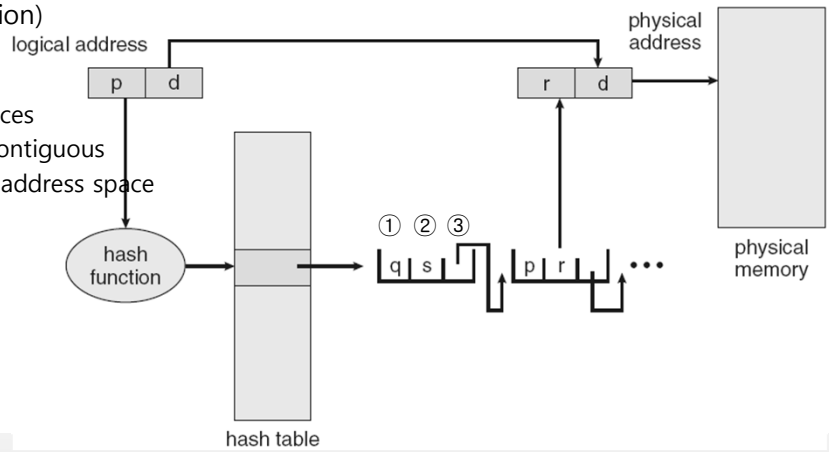
▶Hierarchical Paging in 32bit address space

page number		page offset
p_1	p_2	d
10	10	12



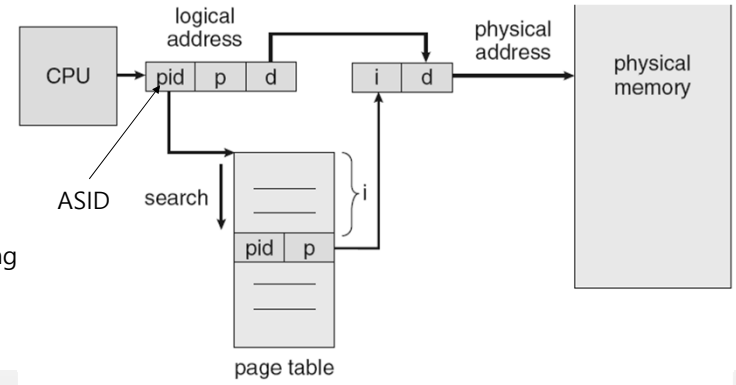
9.4 Structure of the Page Table

- Hashed Page Tables
  - ① virtual page number, ② mapped page frame, and ③ a pointer to the next element
    - Compare page number with field 1(①) in the first element in the linked list
    - If match → corresponding page frame (field 2, ②)
    - Otherwise, subsequent entries in the linked list are searched
  - clustered page tables (variation)
    - ② refers to several pages (such as 16)
    - useful for sparse address spaces
      - Memory references: noncontiguous
      - scattered throughout the address space



9.4 Structure of the Page Table

- Inverted Page Tables
  - Drawback of normal page tables: consist of millions of entries
  - One entry for each real page (or frame) of memory
    - Each entry consists of the virtual address + information about the process
    - only one page table and only one entry for each frame
    - often require the ASID (address-space identifier)
    - Ex. 64-bit UltraSPARC and PowerPC √
  - Increased time to search the table
    - hash table: search to one—or at most a few—page-table entries → at least two real memory reads (hash-table and page table)
    - TLB improves performance
  - Shared memory problem
    - Replacement for each context-switching after page fault

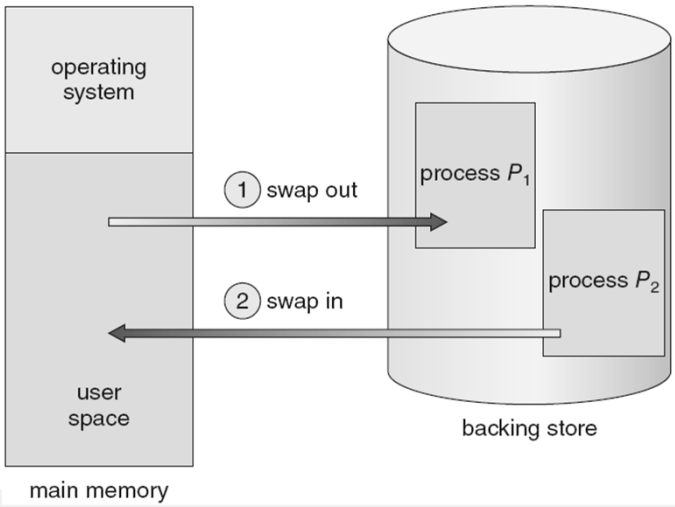


9.4 Structure of the Page Table

- Oracle SPARC Solaris
  - ►two hash tables—one for the kernel and one for all user processes
  - ►hash-table entry represents a contiguous area of mapped virtual memory
    - ▪ more efficient than having a separate hash-table entry for each page
  - ►TLB holds translation table entries (TTEs)
    - ▪ A cache of TTEs in a translation storage buffer (TSB), which includes an entry per recently accessed page.
  - ►virtual address reference → TLB
    - ▪ If failure → TLB walk: walk through in-memory TSB looking for the TTE → copies the TSB entry into the TLB
    - ▪ If failure in TSB → kernel is interrupted to search the hash table → TSB → TLB

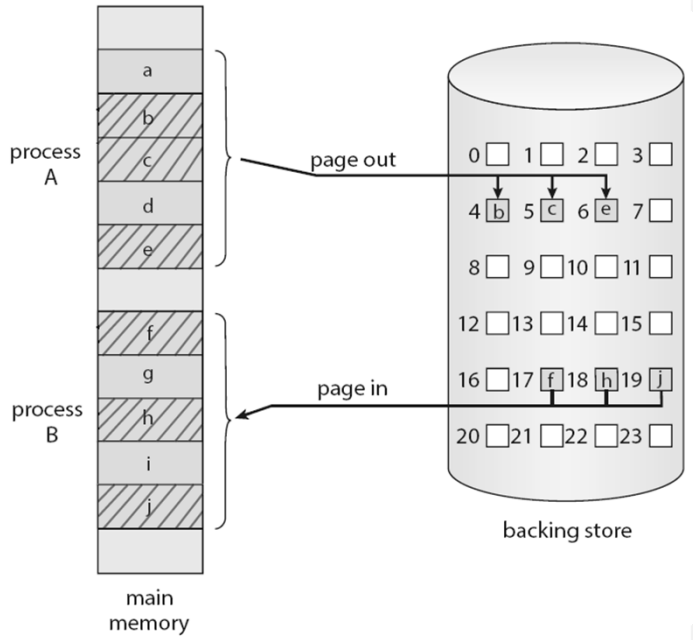
9.5 Swapping

- Temporary swapping between memory and backing store
  - ►total physical address space of all processes to exceed the real physical memory
  - ►increasing the degree of multiprogramming
- Standard Swapping



9.5 Swapping

- Swapping with Paging



9.6 Example: Intel 32- and 64-bit Architectures

- IA-32 Architecture
  - ►IA-32 Segmentation
    - Segment: ~ 4 GB, and the maximum number of segments per process: 16 K
    - logical address space of a process: two partitions.
      - 1st partition: up to 8 K segments, private. Information in local descriptor table (LDT)
      - 2nd partition: up to 8 K segments, shared. Information in global descriptor table (GDT)
      - Entry in the tables: 8-byte segment descriptor, base and limit

Summary

- A memory allocation method for a process: base and limit registers.
- Binding symbolic address to physical addresses may occur during
  - ►(1) compile, (2) load, or (3) execution time.
- Logical address and physical address.
- Contiguous memory allocation of varying sizes.
  - ►(1) first fit, (2) best fit, and (3) worst fit.
- Paging
  - ►Page and frame
  - ►page table: TLB
  - ►Hierarchical paging, hashed page tables and inverted page tables
- Swapping

Exercises, problems and projects

- Exercises
  - ►9.2, 9.4, 9.5, 9.7, 9.8, 9.9
- Problems
  - ►9.12

9.12 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?