Module 5 – L2:
Memory management

Paging - Demand Paging - Page Faults

Dr. Rishikeshan C A
Assistant Professor (Sr.)
SCOPE, VIT Chennai

Outline

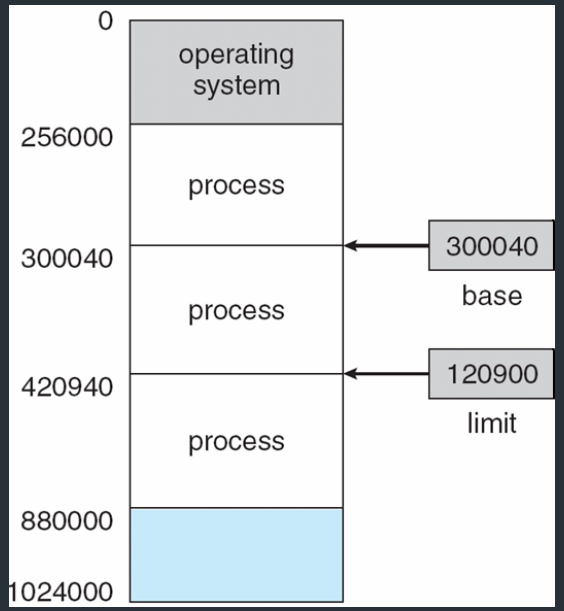
- Paging
- Address Translation Scheme
- Implementing paging, Paging H/W
- Paging Hardware With TLB
- Effective Access Time (EAT)
- Demand Paging
- Page Faults
- Page Fault Handling

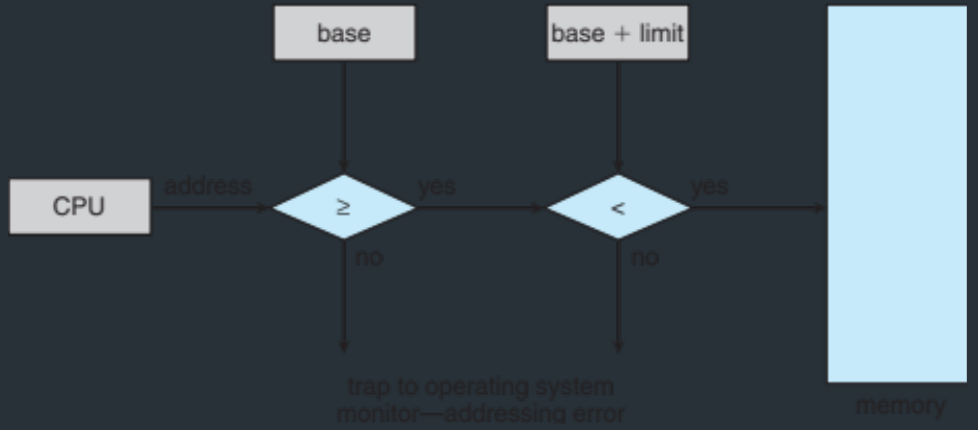
Background

- 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
- 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 access can take many cycles
- **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

Ref:

Logical vs. Physical Address Space

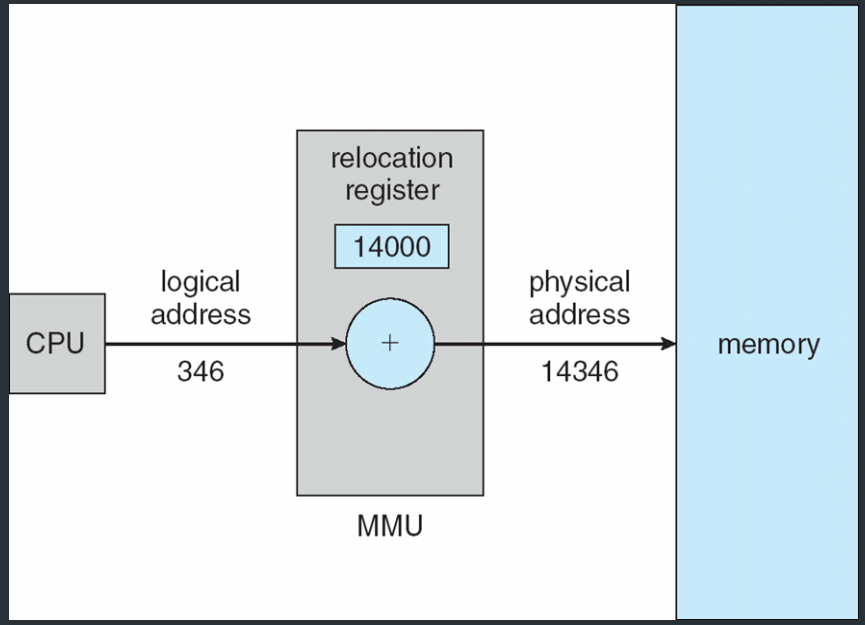
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical 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
- 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

- 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
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

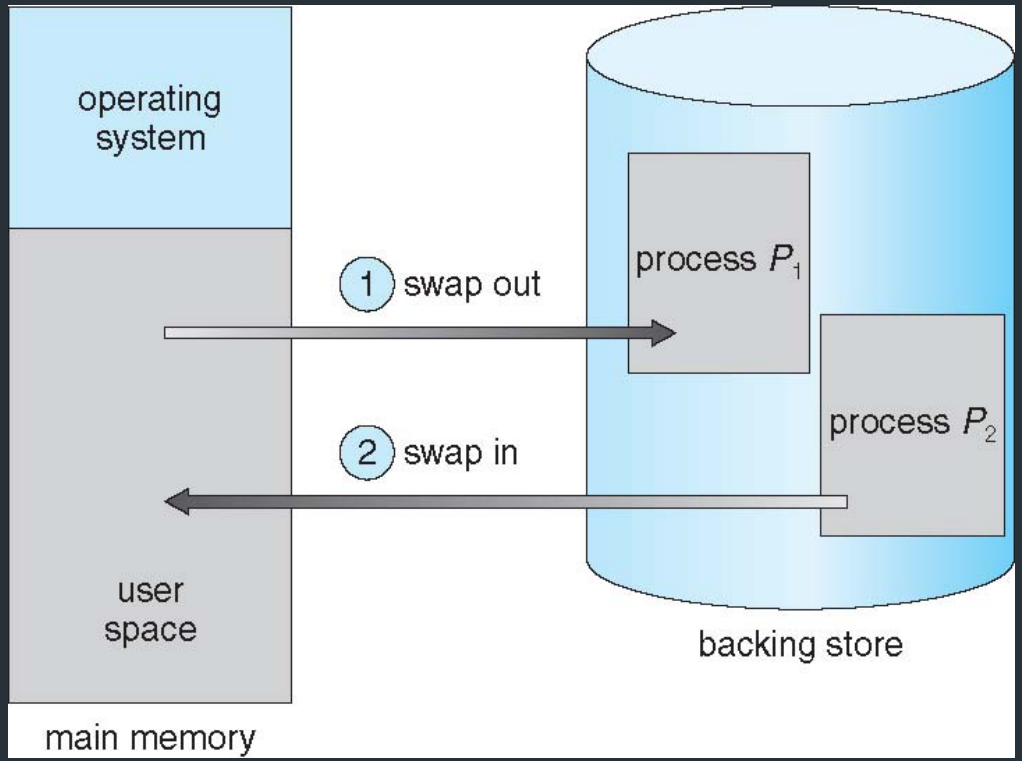


Ref: https://www.os-book.com/OS9/slide-dir/index.html

Swapping

- 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

Schematic View of Swapping



Ref: https://www.os-book.com/OS9/slide-dir/index.html

Swapping (Cont.)

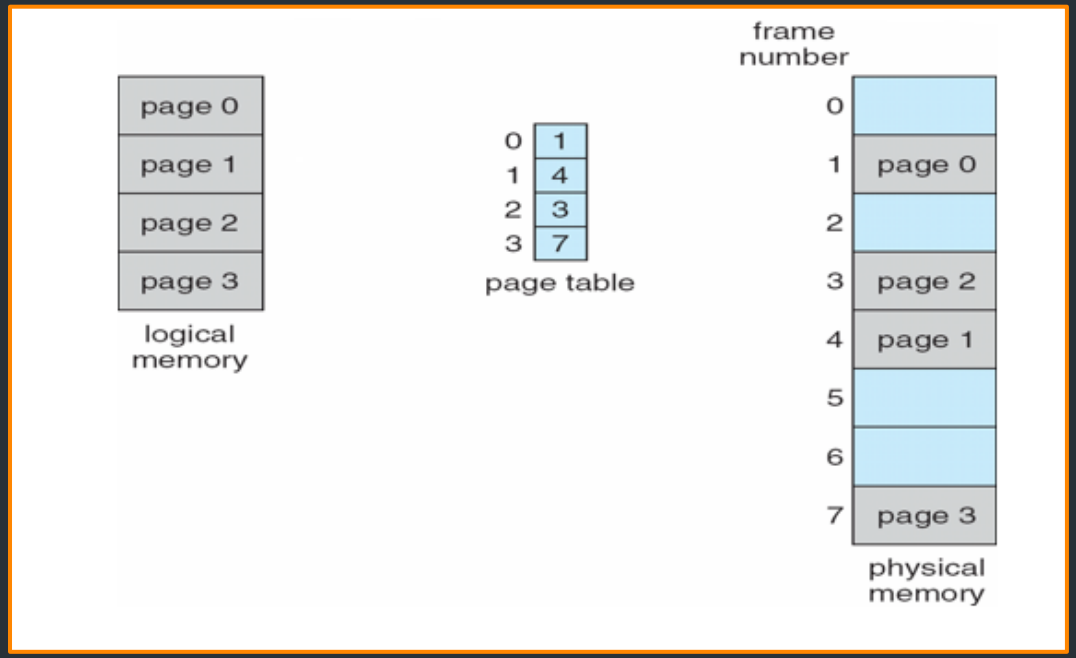
- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
    - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled. Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Paging

- Idea: Divide memory into fixed-sized **pages**
- Divide logical memory into blocks of same size **pages.**
  - Both virtual and physical memory are composed of pages
  - Divide physical memory into fixed-sized chunks/blocks called **frames** (size is power of 2, usually between 512 bytes and 16 MB).

- Goal
  - Eliminate external fragmentation
  - Don’t allocate memory that will not be used

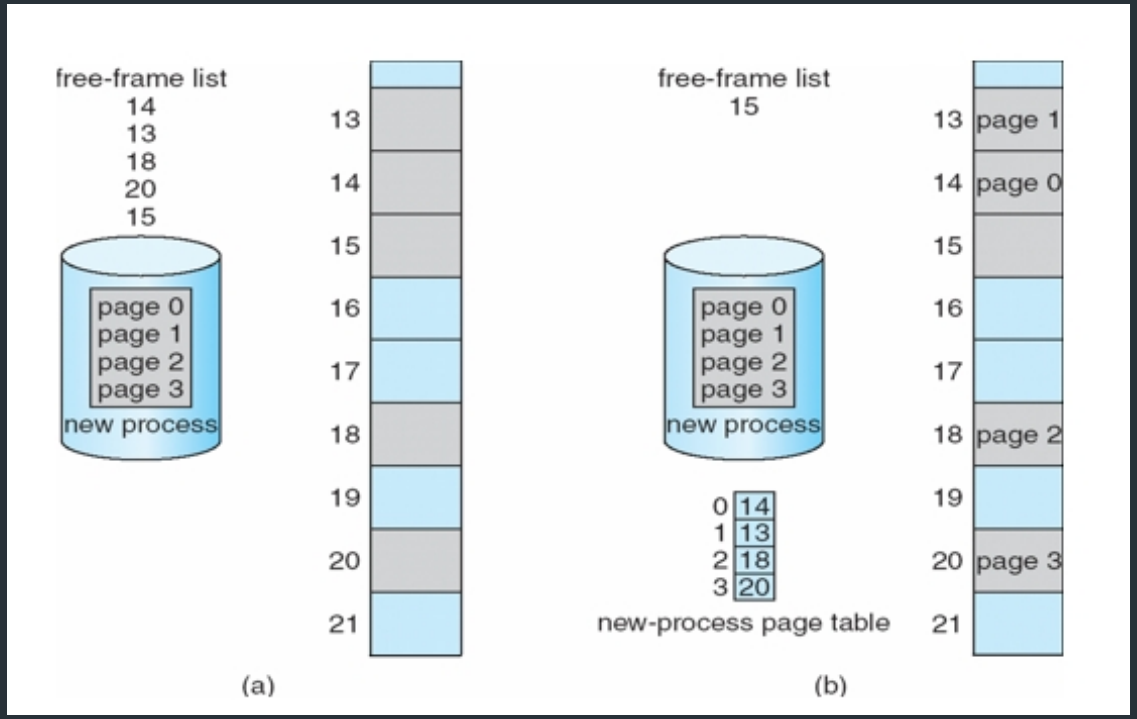
Paging: Example



Paging Implementation

- Keep track of all free physical page frames
- To run a program of size n pages, need to find n free frames and load program
  - Pages do not need to be contiguous!
- Set up mapping from virtual to physical in page table
- At memory reference time, translate virtual address to physical address using page table.

Paging: free-frame list



Before allocation

After allocation

- To run a program of size *n* pages, need to find any *n* free frames and load all the program (pages).
- So need to keep track of all free frames in physical memory use free-frame list.

*Source: Operating System Concepts, 9th Edition, **Abraham Silberschatz**, Peter B. **Galvin**, Greg Gagne*

Logical address used in paging

- Within each program, each logical address must consist of a page number and an offset within the page.
- A dedicated register always holds the starting physical address of the page table of the currently running process.
- Presented with the logical address (page number, offset) the processor accesses the page table to obtain the physical address (frame number, offset).

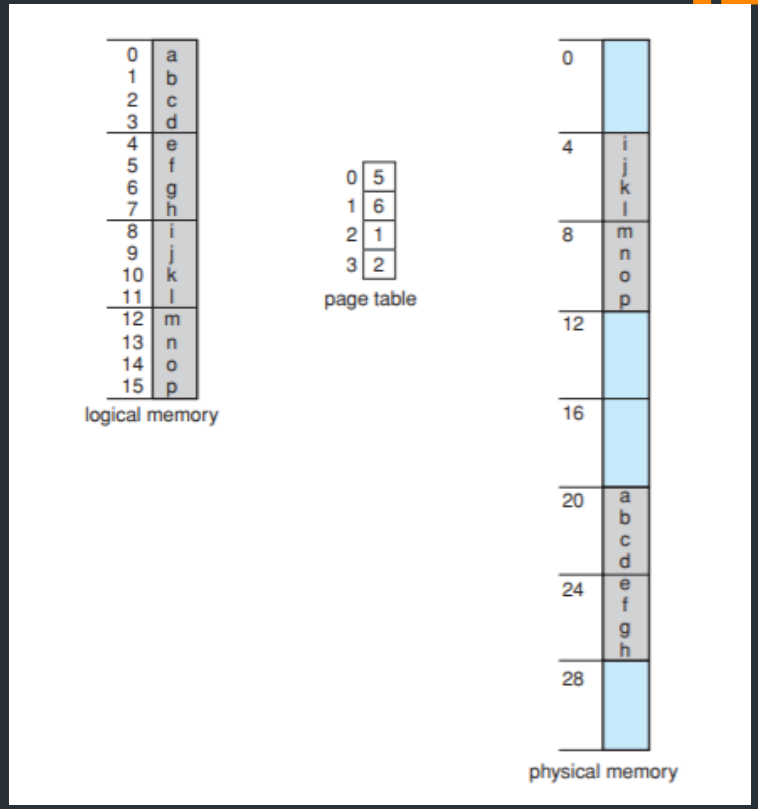
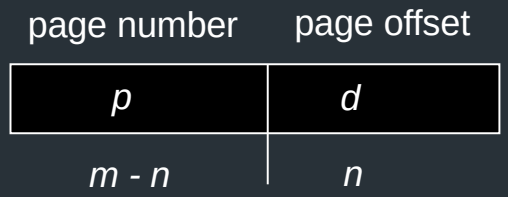


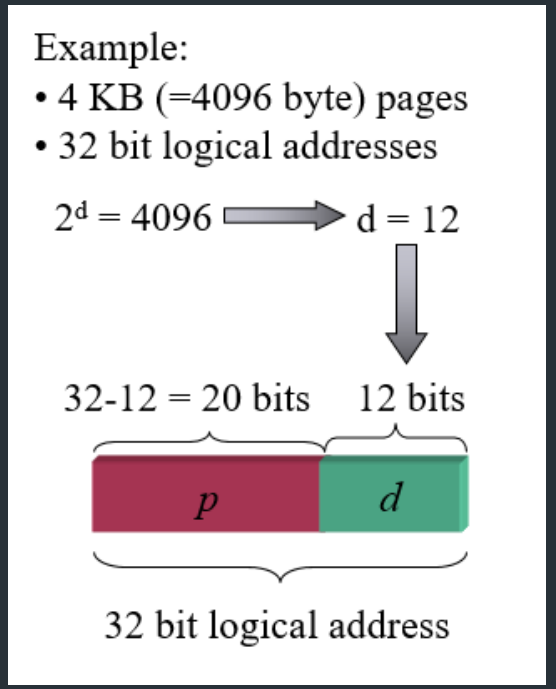
Figure: Paging example for a 32-byte memory with 4-byte pages

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/displacement (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



Logical-to-Physical Address Translation in Paging

By using a page size of a power of 2, the pages are invisible to the programmer, compiler/assembler, and the linker.

Address translation at run-time is then easy to implement in hardware: logical address (p, d) gets translated to physical address (f, d) by indexing the page table with p and appending the same displacement/offset d to the frame number f.

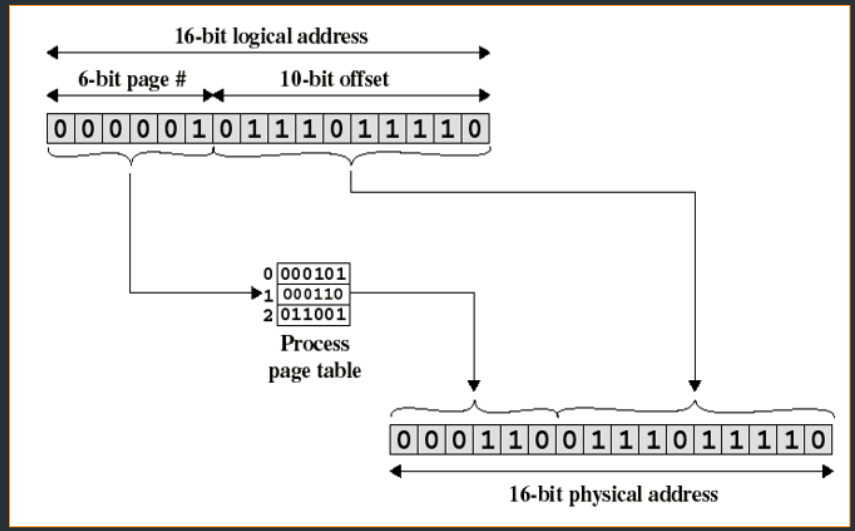
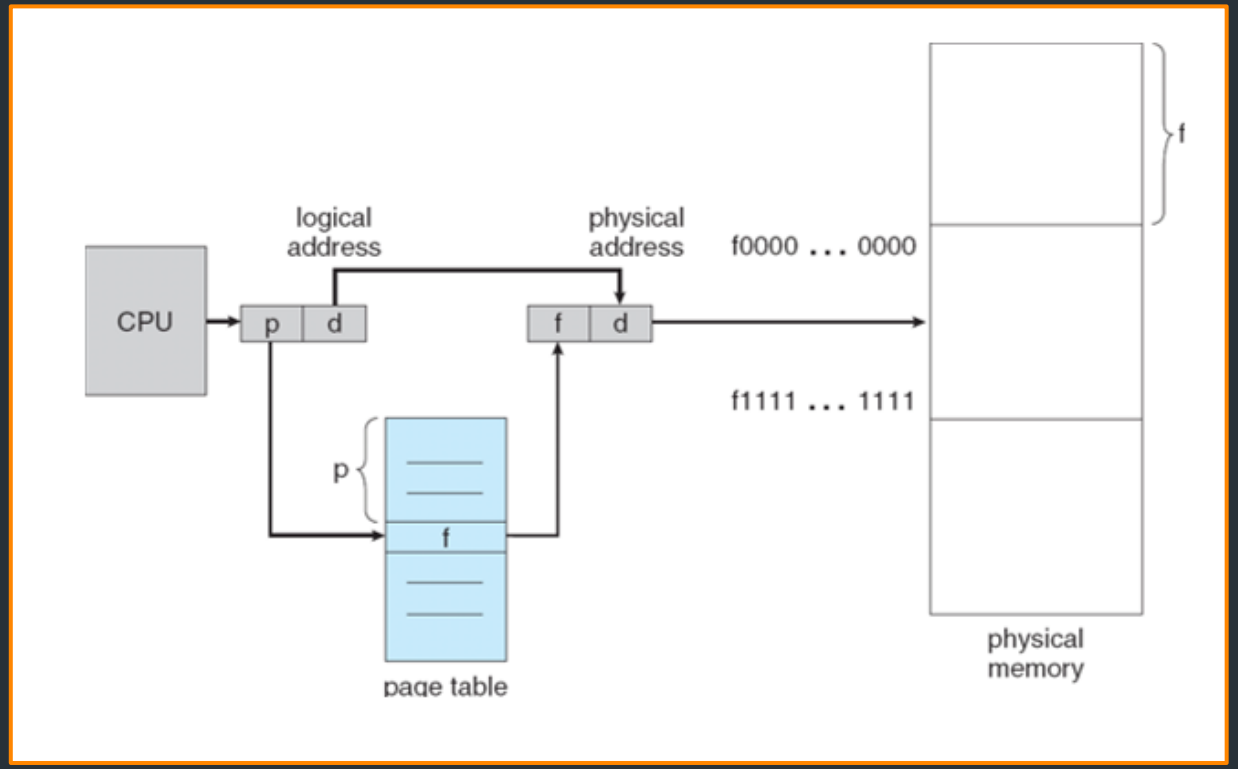


Fig. Source: cs.pitt.edu (by Ethan L. Miller and Scott A. Brandt)

Address Translation Architecture (Paging Hardware)



*Source: Operating System Concepts, 9th Edition, **Abraham Silberschatz**, Peter B. **Galvin**, Greg Gagne*

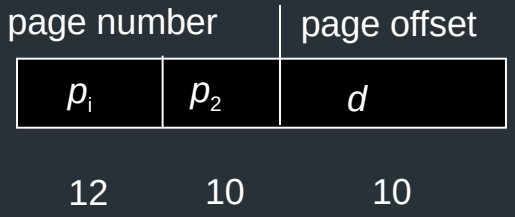
Internal Fragmentation & Frame size

- 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.
- Page sizes growing over time:
  - Solaris supports two page sizes – 8 KB and 4 MB.

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

  where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table
- Thus, a logical address is as follows:



Example

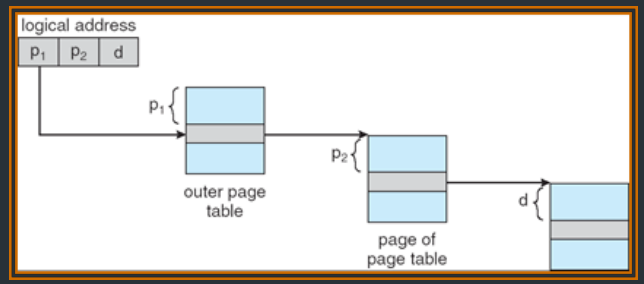


Illustration of Address-Translation Scheme

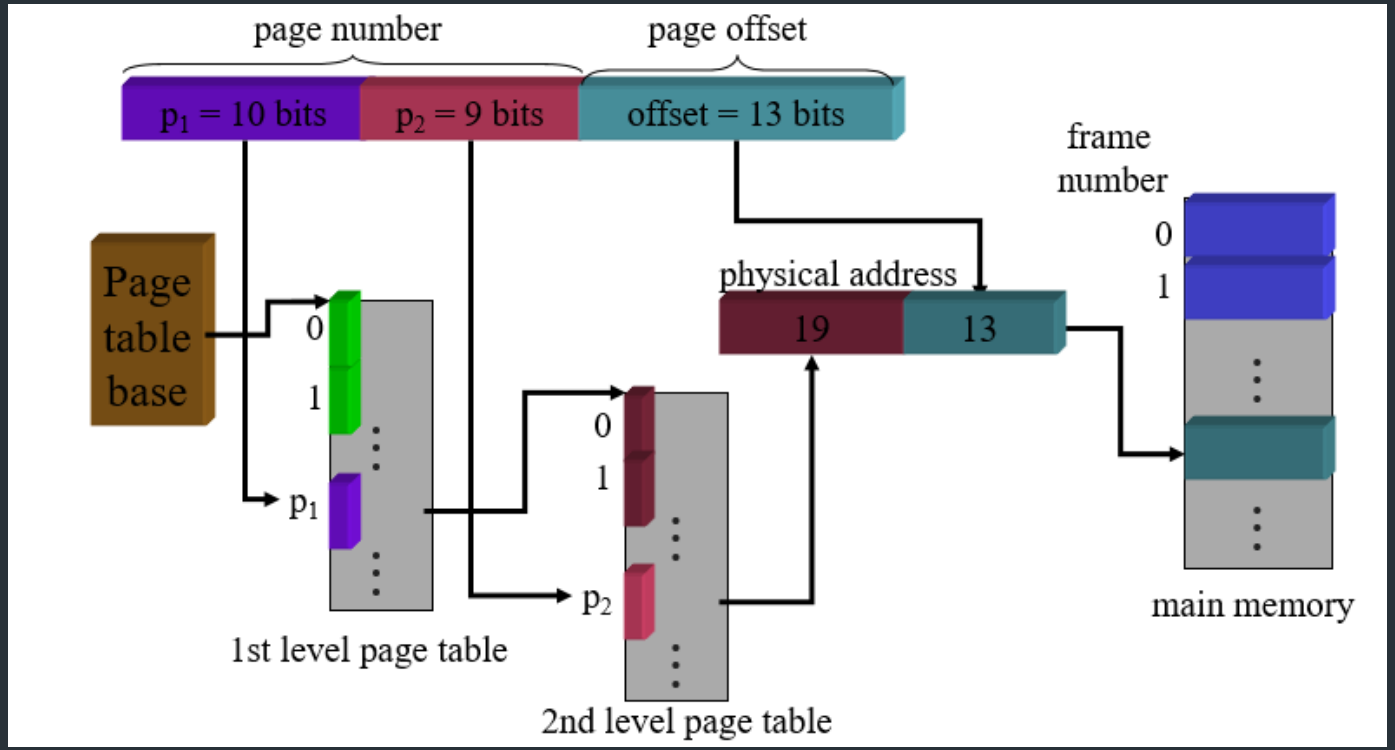
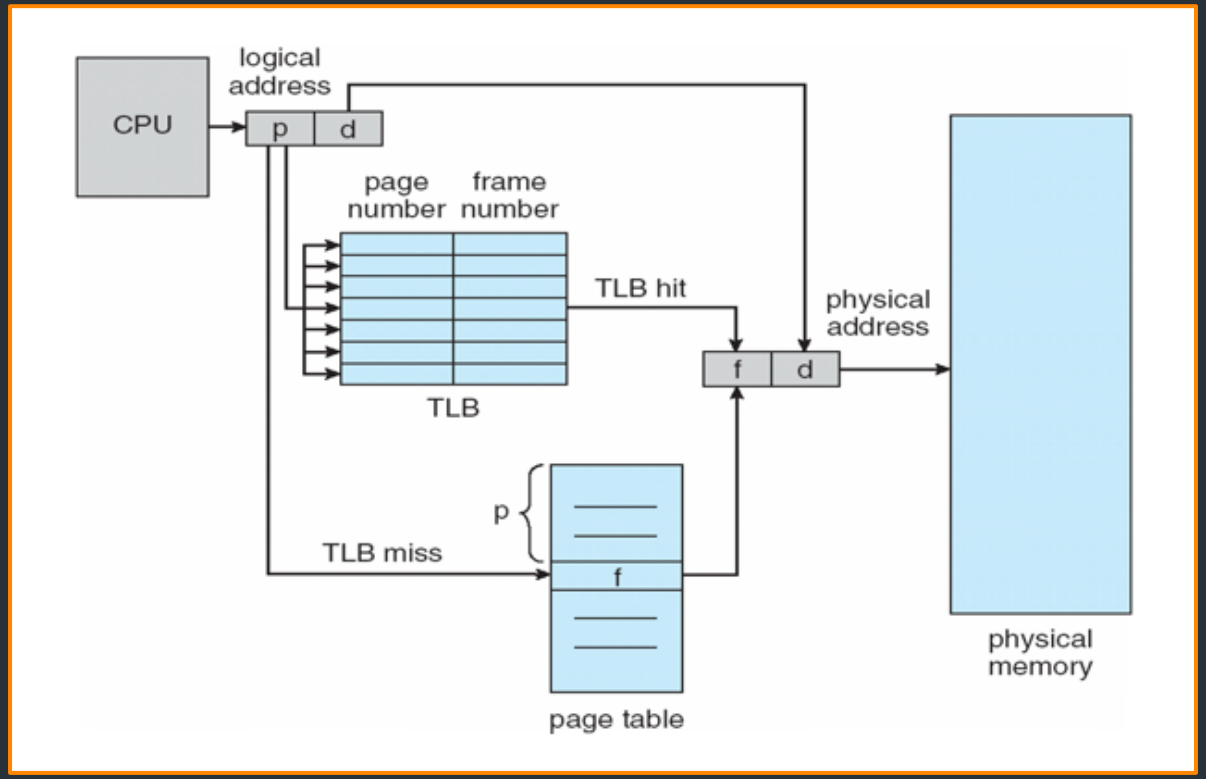


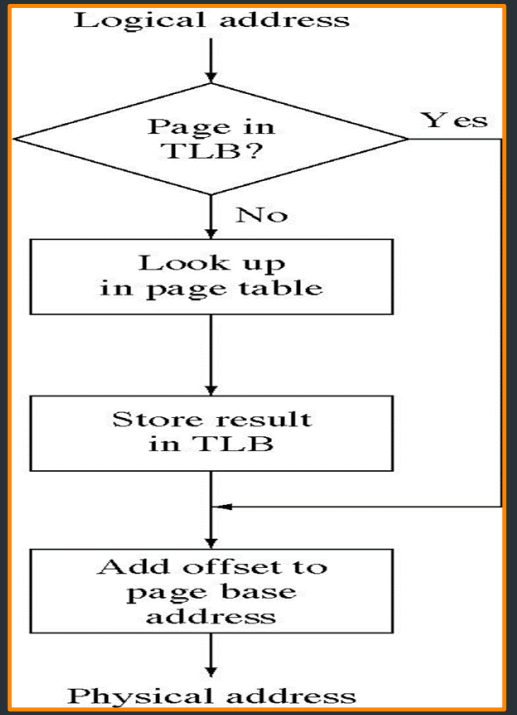
Fig. Source: cs.pitt.edu (by Ethan L. Miller and Scott A. Brandt)

Implementing page tables in hardware

- Page table resides in main (physical) memory
- CPU uses special registers for paging
  - Page table base register (PTBR) points to the page table
  - Page table length register (PTLR) contains length of page table: restricts maximum legal logical address
- Translating an address requires two memory accesses
  - First access reads page table entry (PTE)
  - Second access reads the data / instruction from memory
- Reduce number of memory accesses
  - Can't avoid second access (we need the value from memory)
  - Eliminate first access by keeping a hardware cache *Associative Memory* (*Registers*) (called a *translation lookaside buffer* or TLB) of recently used page table entries: enables fast parallel search

Paging Hardware With TLB





*Source: Operating System Concepts, 9th Edition, **Abraham Silberschatz**, Peter B. **Galvin**, Greg Gagne*

How TLB works

- TLB takes advantage of the **Locality Principle.**
- TLB uses associative mapping hardware to simultaneously interrogate all TLB entries to find a match/hit on page number.
- TLB hit rates are 90+%.
- The TLB must be flushed each time a new process enters the running state.
- Maybe keep/load TLB information in/from process context.

Effective Access Time (EAT)

- Effective Access Time (EAT) is between 1 and 2 access times – should be closer to 1.
- Assume memory cycle time is 1 microsecond.
- Associative (Memory) Lookup = ε time unit.
- Hit ratio = α – percentage of times that a page number is found in the associative memory; ratio related to number of associative registers.
- EAT = $\alpha(\varepsilon + 1) + (1 - \alpha)(\varepsilon + 2) = 2 + \varepsilon - \alpha$

EAT calculation example

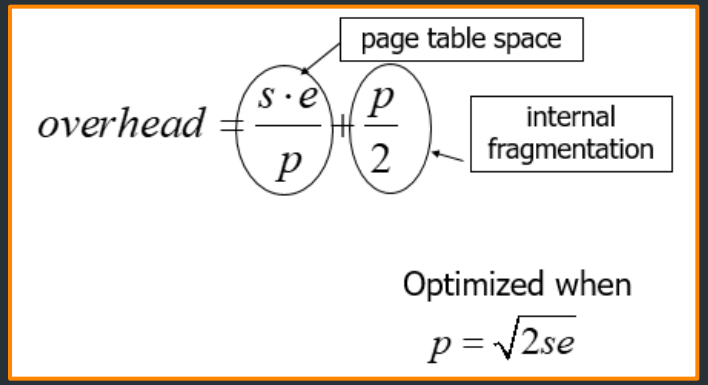
- Assume memory cycle time is 100 nanosecond.
- Associative memory lookup = 20 nanosecond.
- Hit ratio = 80%
  - EAT = 0.80 x 120 + 0.20 x 220 = 140 ns
  - So 40% slowdown in memory access time.
- Hit ratio = 98%
  - EAT = 0.98 x 120 + 0.02 x 220 = 122 ns
  - So only 22% slowdown in memory access time.

Page Size

Small page size

- Advantages
  - less internal fragmentation
  - better fit for various data structures, code sections
  - less unused program in memory
- Disadvantages
  - programs need many pages, larger page tables

Overhead due to page table and internal fragmentation



Where,
s = average process size in bytes
p = page size in bytes
e = page entry

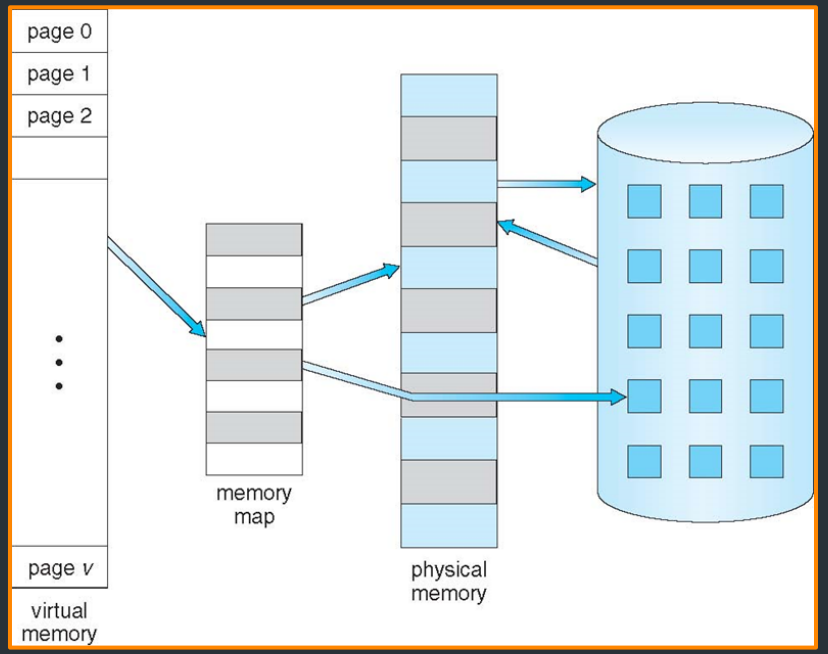
Demand paging

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paging

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

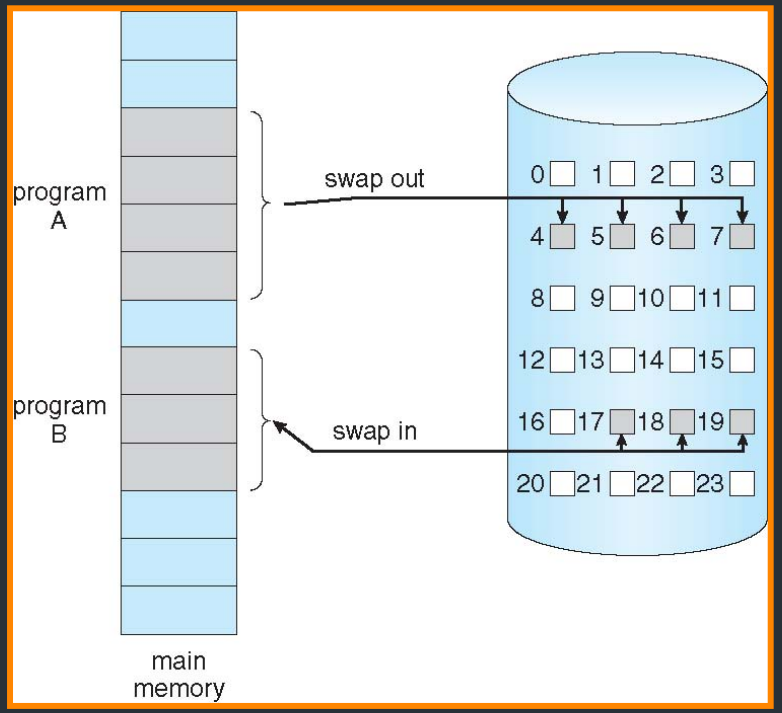


Virtual Memory That is Larger Than Physical Memory

Demand Paging

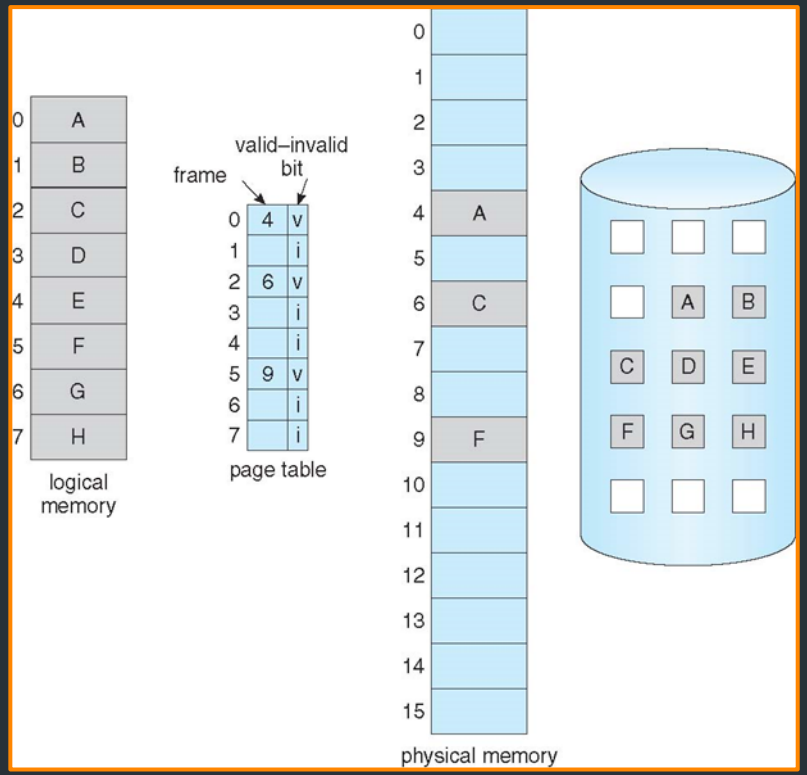
- Similar to paging system with swapping (diagram on right)
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code



Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

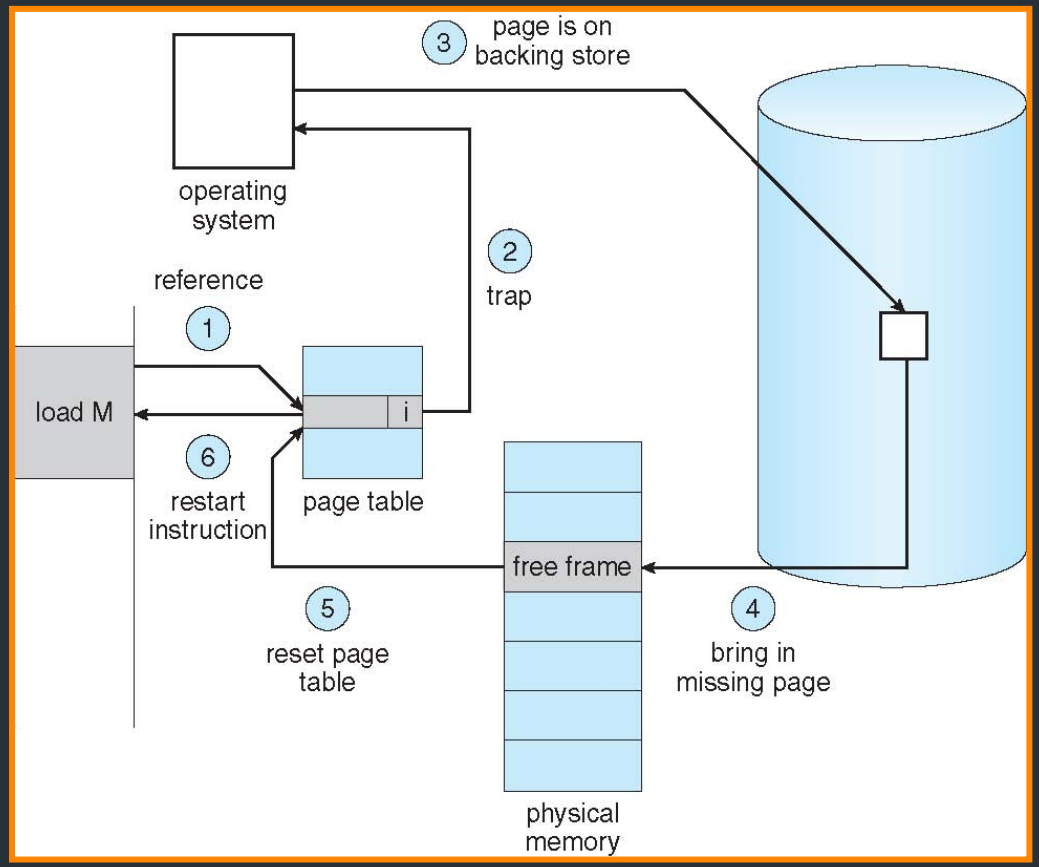


Page table when some pages are not in main memory

Page Fault & Page fault handling

1. If there is a reference to a page, first reference to that page will trap to operating system: **Page fault**
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out.
  - page replacement algorithms
  - performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access
  + p (page fault overhead
  + [swap page out]
  + swap page in
  + restart overhead)

Demand Paging Example

- Suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds (8,000,000 nanoseconds, or 40,000 times a normal memory access) with a page fault rate of p, (on a scale from 0 to 1), the effective access time (EAT) is now:

  (1 - p) * (200) + p * 8000000

  = 200 + 7,999,800 * p

- So, EAT clearly depends heavily on p! Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times.
- In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - Pure demand paging
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of locality of reference
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Instruction restart

Any Questions ?

References

- http://www.cs.nthu.edu.tw/~ychung/slides/CSC3150/Abraham-Silberschatz-Operating-System-Concepts---9th2012.12.pdf
- http://www.digilife.be/quickreferences/qrc/linux%20system%20call%20quick%
- Silberschatz, Gagne, Galvin: Operating System Concepts, 6th Edition

Thank You!