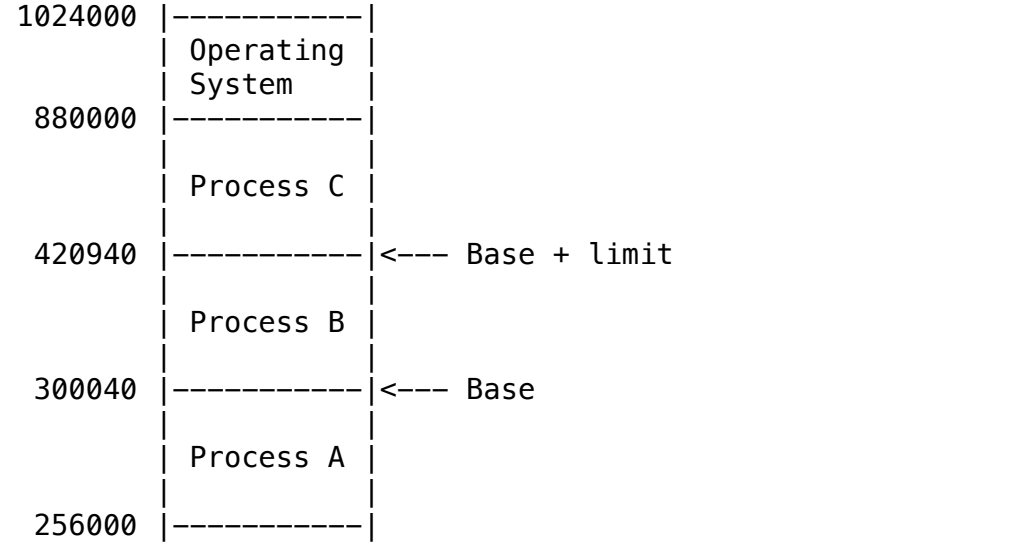
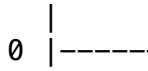
Lecture.9.Memory.Management.txt

- Background
    - Role in programming language
        - Instructions
            - Are usually read-only
                - In normal programming, instructions are read-only
                    - We do not allow code to rewrite itself
                - Some embedded systems allow rewriting of instructions
        - Variables
            - Read and write, only
        - Constants
            - Read only
    - To access memory, we need the address
    - When writing to memory, we need an address and data
    - Memory unit only sees a stream of:
        - Addresses + Read requests
        - Addresses + Data + Write requests
    - Main memory and registers are the only storage mediums that the CPU can access directly
        - The CPU cannot write into secondary storage
        - The CPU can only read and write data into registers and memory that is closest to the CPU, which is main memory
            - Main memory is cache and RAM
    - Protection of memory is required to ensure correct operation
        - We do not want different processes to overlap in memory space

- Protection
    - OS has to be protected from access by user processes, as well as protect user processes from one another
    - We can provide this protection by base and limit registers
        - i.e. Example of processes and operating system inside memory





- The operating system is protected, because no process can access its memory
  - No process can read or write to the operating system's memory
  - The operating system's memory is at the top of the memory address space
- Each process has its own space in memory
- The memory space of 'Process B' is defined by the base register, and the top, base plus limit
  - All addresses pertaining to 'Process B' are in the range 300040 and 420940
- A base and a limit register, pair, define a logical address space
  - The logical address space is on the CPU side
  - Real memory is on the physical address space
  - In a system, we need both physical and logical address space

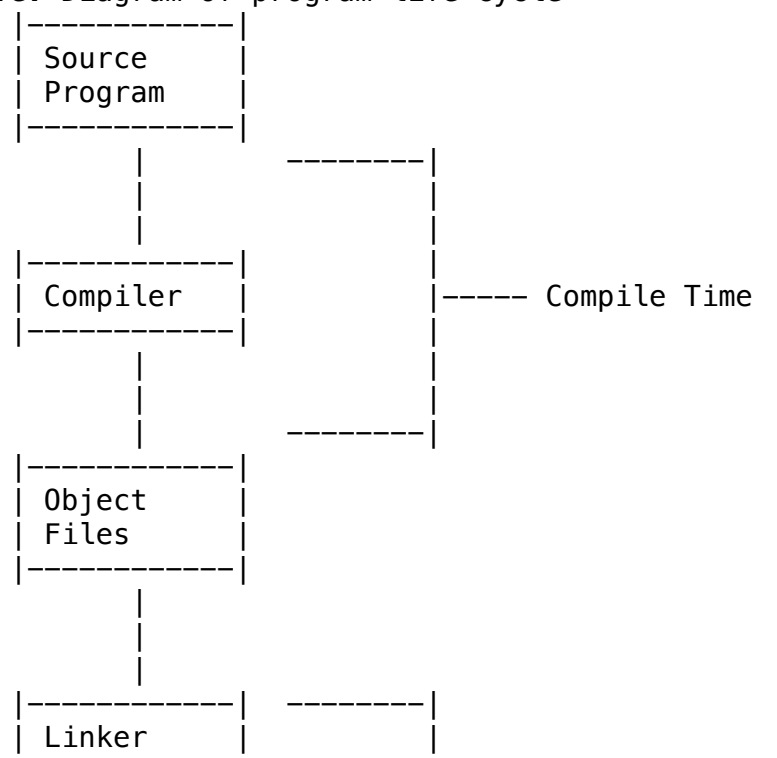
- Hardware Address Protection
  - Is used to ensure that the CPU does not access memory it is not allowed to access
    - This is achieved through base and base + limit
  - CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
    - If the generated memory address is lower than base, then an error is raised
    - If the address is not within base and base + limit, then memory access is denied
  - Protection of memory space is accomplished by having the CPU hardware
  - The instructions to loading the base and limit registers are privileged
    - Can be used only by the OS

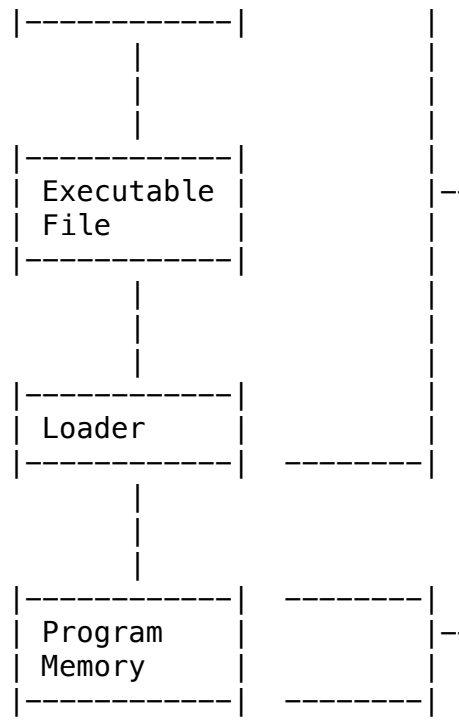
- Address Binding
  - Addresses are created during the stages of program development
  - Program resides on a disk as a binary executable file
    - Must be loaded into memory address 0000
      - Inconvenient
  - Addresses are represented in different ways at different stages of a program's life
    - Addresses in the source program are generally symbolic
      - i.e. The variable `count`
        - `count` references some space in address in memory
        - This notation is easier for humans to understand
    - 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
            - This is the address inside memory
    - Each binding maps one address space to another
        - Symbolic address -> Compiled code address -> Absolute address

- Binding Of Instructions & Data To Memory
    - Address binding of instructions and data to memory addresses can happen at three different stages:
        - Compile time: If memory location is known in advanced, then absolute address(es) can be generated
        - Load time: Must generate relocatable code if memory location is not known at compile time
        - Execution time: If the process can be moved during its execution from one memory segment to another, binding must be delayed until run time
            - Most (99%) of the address binding takes place on execution time
            - Need hardware support for address mapping

- Multistep Processing Of A User Program
    - Compile time
        - Absolute code
    - Load time
        - Relocatable code
    - Execution time
        - Run time binding
    - i.e. Diagram of program life cycle



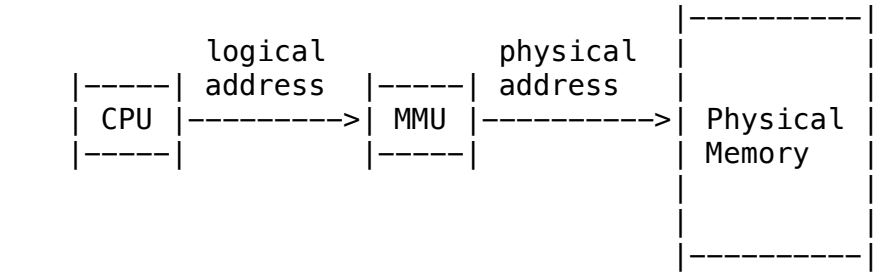


    - The first step is to compile source code to create object file
      - Multiple source files result in multiple object files
    - Then, the Linker takes all the object files and creates relocatable code
    - The job of the Loader is to load the code into memory
    - Execution time is also referred to as run time

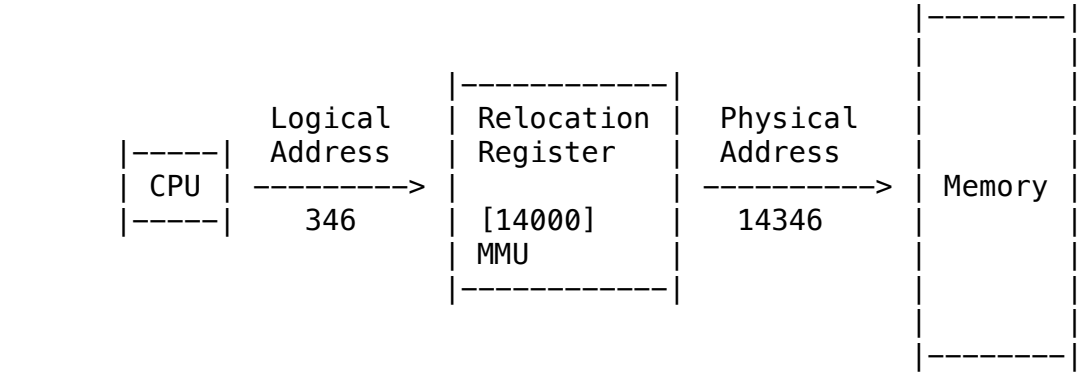
- Logical VS. Physical Address Space
  - The concept of logical address space that is bound to a separate physical address space is central to proper memory management
    - Logical address: Generated by the CPU
      - Also referred to as virtual address
    - Physical address: Address seen by the memory unit
  - Directly dealing with physical addresses is not an ideal solution
    - For instance, moving data in memory changes the physical address, but not the logical address
      - This is abstraction
  - 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 (MMU) (1)
  - Is used to translate logical addresses to physical addresses
  - The runtime mapping from virtual to physical addresses is done by a hardware device called memory-management unit (MMU)

    - We can choose from many different methods to accomplish runtime mapping
    - i.e. Diagram Of Memory Mapping



- Memory Management Unit (MMU) (2)
    - One way to implement a MMU is to use a relocation register
    - A generalization of the base-register scheme
    - The base register now called relocaion register
    - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
        - i.e.



        - The value of the relocation register is added to the value of the logical address
            - Logical address + Relocation register = Physical address
                - 346 + 14000 = 14346
    - If the base is at 14000, then an attempt by the user to address location 346 is dynamically relocaed to location 14346
        - Refer to diagram above
    - The MMU is responsible for generating the relocation register address value

- Dynamic Loading
    - Initially we should have entire program and all data of a process to be in physical memory for the process to execute
        - Size of a process has been limited to the size of physical memory
            - If the size of the process is bigger than physical memory, then it cannot be executed
    - For better memory-space utilization unused routine is never loaded

        - Only routines that are used are loaded into memory
        - Unused routines are kept in secondary memory
    - All routines kept on disk in relocatable load format
    - Useful when large amounts of code are needed to handle infrequently
        - For instance, error handling is a lot of code, and is infrequently utilized. Therefore, it is better to keep it on secondary memory, instead of clogging up main memory with code that may never be used
    - Dynamic loading requires no special support from the OS
        - It is implemented through program design
        - The OS can help by providing libraries to implement dynamic loading

- Dynamic Linking
    - Dynamically linked libraries (DLLs) are system libraries that are linked to user programs when the programs are run
        - DLLs are added during execution time
    - Static linking: System libraries and program code combined by the loader into the binary program image
        - In other words, the libraries are linked in advance
    - Dynamic linking: Linking postponed until execution time
        - This is useful because the compiled program occupies less space in memory
    - With dynamic linking, multiple programs can share libraries
        - This is not possible with static linking
    - Without this facility, each program on a system must include a copy of its language library – or at least the routines referenced by the program – in the executable image
        - Decreases size of executable image
        - Libraries can be shared by multiple processes, only one instance of DLL in main memory
            - This saves space in memory
    - Due to limitation of resources, we need to optimize the use of these resources and prevent wasting

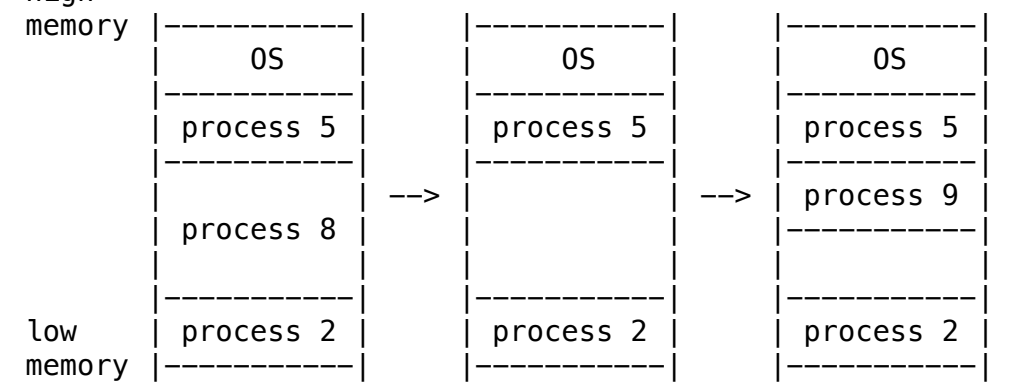
- Contiguous Memory Allocation
    - Main memory must support/store both OS and user processes
        - Main memory needs to be allocated in the most efficient way possible
            - Contiguous allocation is one way to achieve this
    - Main memory is usually divided into two partitions:
        - Resident OS usually held in low memory with interrupt vector
        - User processes then held in high memory
    - Each process is contained in a single contiguous section of memory
        - Each process has its own range of memory

- Contiguous Allocation
    - Relocation registers used to protect user processes from each

other, and from changing operating-system code and data
  - Base 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(es) dynamically
  - Can then allow actions such as kernel code being transient and kernel changing size

- Hardware Support For Relocation & Limit Registers
  - The logical address must be between the 'limit register' and the 'relocation register'
    - Used for memory protection
  - The relocation register contains the value of the smallest physical address
    - i.e. Relocation = 100040
      - If the logical address is greater than this value, then it will generate an error
  - The limit register contains the range of logical addresses
    - i.e. Limit = 74960
      - This is the maximum value of the address the CPU can use to access the memory

- Variable Partition
  - The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically
  - Operating system maintains information about:
    - Allocated partitions
    - Free partitions (hole)
  - i.e. Process Allocation Diagram



  - The operating system's memory is on top of all other memory
  - In the first block, there are 3 processes in memory: 5, 8, and 2
  - This method of allocation creates fragmentation
    - If process 5 completes, there are 2 free blocks of memory, but process 8 cannot run
    - Contiguous allocation does not work with fragmentation
      - Paging solves this problem

    - Allocation
        - What happens when there isn't sufficient memory to satisfy the demands of an arriving process?
            - You can try to rearrange the processes in memory and free up some space; apart from this, there isn't much that can be done

- 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
            - Takes more time than the other methods
        - 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
        - Best fit is better than first fit and worst fit in terms of storage utilization
    - Best fit and worst fit take more time than first fit
        - First fit is the best in terms of speed

- Fragmentation (1)
    - Both the first fit and best fit strategies for memory allocation suffer from external 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
        - Leads to loss of memory
        - Is a result of paging
    - Usually, external fragmentation is worse than internal fragmentation
    - First fit analysis reveals that given `N` blocks allocated, another `0.5N` blocks lost to fragmentation
        - 1/3 may be unusable -> 50% rule
            - i.e. Given 3 blocks, 1 may be lost due to fragmentation

- Fragmentation (2)
    - 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
    - The simplest compaction algorithm is to move all processes toward one end of memory
    - Another possible solution is to permit the logical address space of processes to be noncontiguous thus allowing a process to

      allocate physical memory wherever such memory is available
        - This is called Paging
            - Paging allows us to utilize all free space in memory

- Paging
    - Physical address space of a process can be noncontiguous
        - This means that a process can be divided into parts, and each part can be stored in memory at any location
            - i.e. For a given program, some instructions can be stored at address `N`, and the remaining instructions can be stored at address `N+1`
        - Avoids external fragmentation
        - Avoids problem of varying sized memory chunks
    - Paging refers to the logical address
    - Frames refer to the physical memory address
    - Paging is done by dividing physical memory into fixed-sized blocks called frames
        - Size is power of 2
            - Between 512 bytes and 16 Megabytes
    - The steps involved in paging are:
        - Divide logical memory into blocks of same size called pages
        - To run a program of size `N` pages, need to find `N` free frames
            - Pages are associated with frames, and both have the same size
        - Set up a page table to translate logical to physical addresses
            - This is the simplest implementation of the memory management unit (MMU) for non-contiguous processes
        - Backing store likewise split into pages

- Address Translation Scheme
    - Address generated by CPU is divided into two parts:
        - 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
    - i.e. Diagram

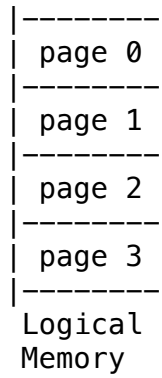
page number	page offset
p	d
m - n	n

    - For given logical address space 2^m and page size 2^n
        - The page number is `m - n`
        - The page offset is `n`
    - Logical address space is now toally separated from the physical address space

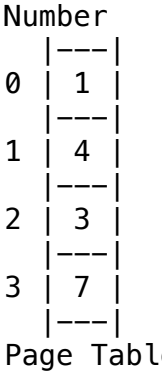
        - The CPU generates a logical address that consists of a page number and offset
        - The page table translates the logical address into a physical address

- Paging Hardware
    - The page number is used as an index into a per-process page table
    - The page table contains the base address of each frame in physical memory
    - Offset `d` does not change

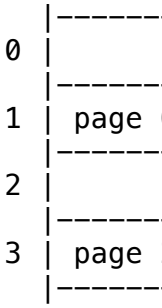
- Paging Model Of Logical & Physical Memory
    - i.e. Diagram of Logical Memory



    - i.e. Diagram of Page Table



    - i.e. Diagram of Physical Memory



```
      4 | page 1 |
        |--------|
      5 |        |
        |--------|
      6 |        |
        |--------|
      7 | page 3 |
        |--------|
      Physical Memory
```

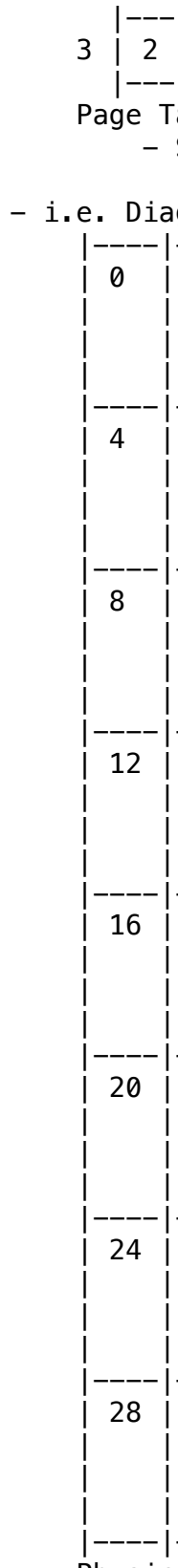
    - According to the diagrams above, logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 1
      - Paging is a form of dynamic relocation

- Paging Example
  - i.e. Diagram of Logical Memory

```
      |----|---|
      | 0  | a |
      | 1  | b |
      | 2  | c |
      | 3  | d |
      |----|---|
      | 4  | e |
      | 5  | f |
      | 6  | g |
      | 7  | h |
      |----|---|
      | 8  | i |
      | 9  | j |
      | 10 | k |
      | 11 | l |
      |----|---|
      | 12 | m |
      | 13 | n |
      | 14 | o |
      | 15 | p |
      |----|---|
      Logical Memory
```

        - $2^m = 16$; $m = 4$
        - Each page has 4 elements
          - Thus, $n = 4$
        - These (logical) addresses are generated by a CPU
        - On logical memory, each unit is called a page
  - i.e. Diagram of Page Table

```
      |---|
    0 | 5 |
      |---|
    1 | 6 |
      |---|
    2 | 1 |
```



- Size: 2^n
  - n = 2

- i.e. Diagram of Physical Memory

  - On physical memory, each unit is called a frame

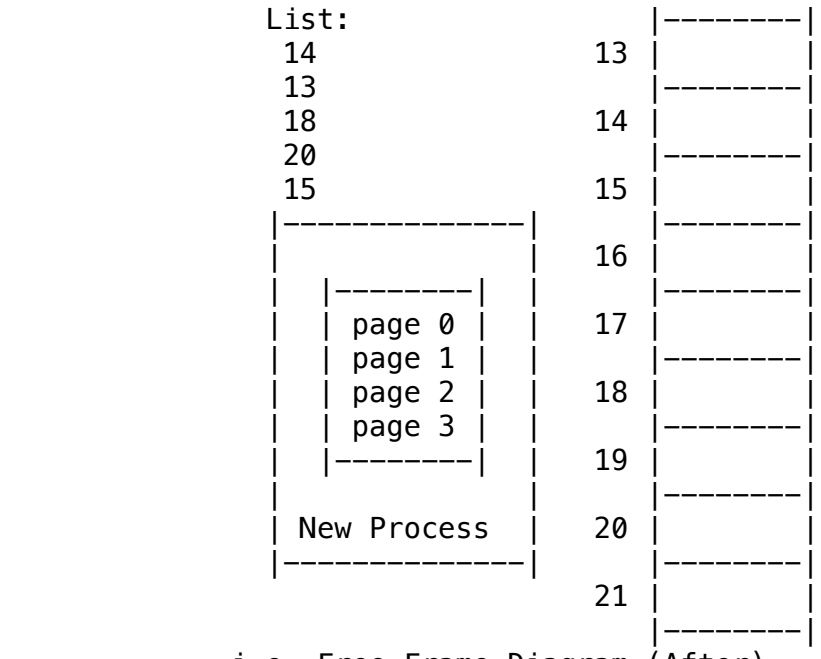
    - Refering to the diagrams above:
        - Page 0 on the logical memory is associated to frame 5 in the page table
            - Frame 5 is at index number 20 in physical memory
        - Logical address: n = 2 and m = 4
        - Page size of 4 bytes, physical memory of 32 bytes (8 pages)
        - Logical address 0 maps to physical address 20
            - [(5 x 4) + 0] = 20
        - 3 (page 0, offset 3) -> 23
            - [(5 x 4) + 3] = 23
        - 4 (page 1, offset 0) -> 24
            - [(6 x 4) + 0] = 24
    - A page table is used to associate a page to a particular frame
        - i.e. Page 0 on logical memory is translated to Frame 5 on physical memory

- Paging Calculating Internal Fragmentation
    - External fragmentation is when the system has enough free memory to satisfy the demands of a process, but the free memory is not contiguous
        - The problem of external fragmentation is solved by introducing pages
            - Pages allows us to store the content of a process across different locations inside physical memory
            - Paging causes internal fragmentation
            - Calculating internal fragmentation is shown below
    - Internal Fragmentation Example:
      Given page size = 2048 bytes
      For process size = 72766 bytes, we need 36 pages
      Internal fragmentation = 36 x 2048 - 72766 = 962 bytes
    - Worst case fragmentation = frame - 1 byte
    - Average fragmentation = 1/2 frame size
    - Pages are typically either 4 KB or 8 KB in size
    - Some CPUs and operating systems even support multiple page sizes
        - Windows 10 = 4KB and 2MB
        - Linux = 4KB and huge pages
        - Note: The combination of multiple page sizes works best

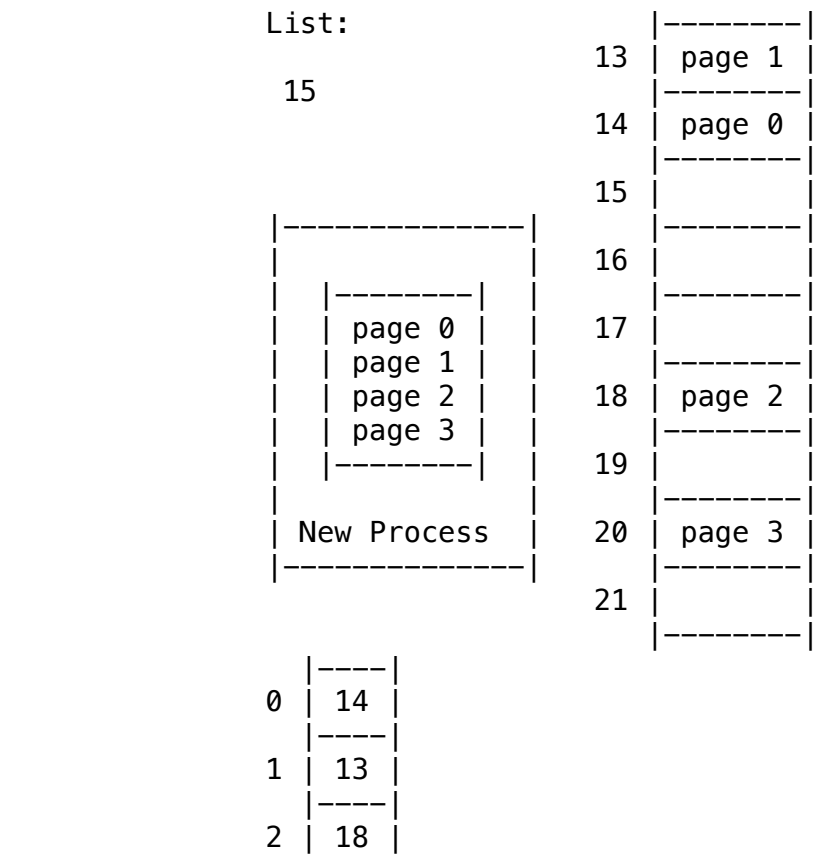
- Example
    - Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long
    - A 32-bit entry can point to one of 2^32 physical page frames
    - If the frame size is 4KB (2^12), then a system can address 16TB (2^44) bytes of physical memory

- Free Frames
    - A list of free frames needs to be maintained
        - Once a frame is used, it cannot be associated to a page anymore
    - When a process arrives in the system to be executed, its size,

expressed in pages, is examined
- Each page of the process needs one frame
- i.e. Free Frame Diagram (Before)



- i.e. Free Frame Diagram (After)



```
    |----|
  3 | 20 |
    |----|
```

  - The two diagrams above depict free frames before and after 4 pages allocation

- Implementation Of Page Table
  - A page table is a mapping from pages to frames
  - The hardware implementation of the page table can be done in several ways
  - 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
  - Problems with storing the page table in main memory:
    - If the page is very big, it will take up a lot of space in memory; contiguously as one portion
    - There may be 2 steps to access the page, and each access requires one step for the page table and another step for the actual data
  - In this scheme every access requires two steps
    - One for the page table, and one for the actual data
    - This approach is reused in many different circumstances
  - The two step memory access problem can be solved by the use of a "special fast lookup hardware cache" called translation look-aside buffers (TLBs)
    - This method puts the most used addresses in a separate buffer
      - As a result, performance is greatly increased

- Translation Look Aside Buffer
  - The TLB is a special hardware cache
  - Some TLBs store address space identifiers (ASIDs) in each TLB entry
    - Uniquely identifies each process used to provide address space protection for that process
  - TLBs are typically small
    - i.e. 64 - 1024 entries of all addresses
  - On a TLB miss, the regular page table is consulted, and the value is loaded into the TLB for faster access next time the same value is used
    - Replacement policies must be considered (if full)
      - Addresses that are not used for an extended period of time are replaced with newer addresses
    - Some entries can be wired down for permanent fast access

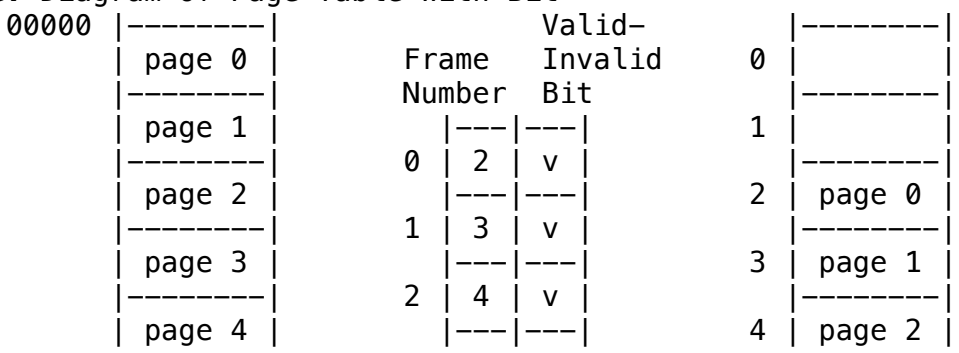
- Paging Hardware With TLB
  - When the frame number is obtained, we can use it to assess memory
  - We add the page number and frame number to the TLB, so that they

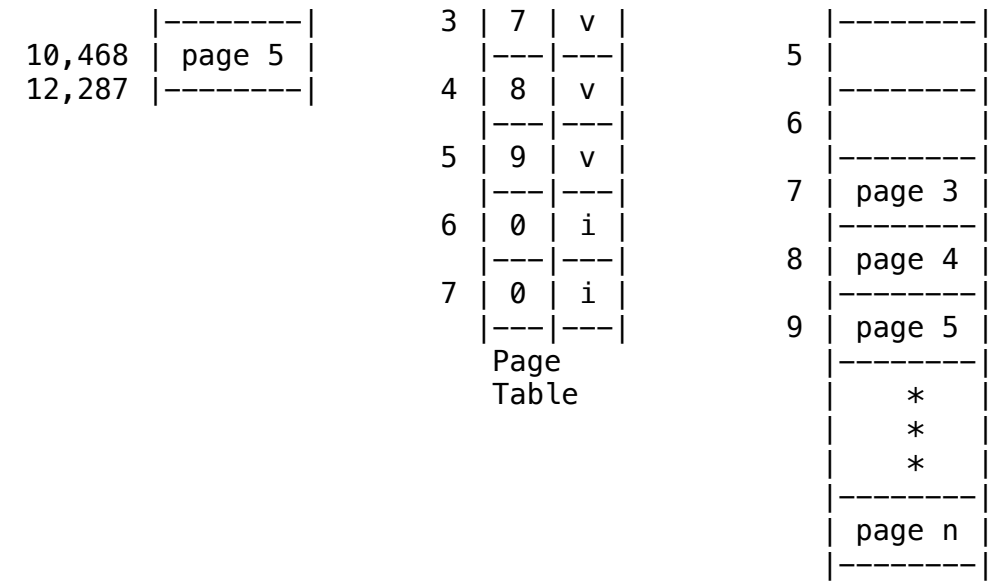
  will be found quickly on the next reference
- A system can, and will, work without a TLB
  - However, it will be slower
- The TLB can only store a subset of the page table

- Effective Access Time
  - Percentage of times that a page number is found in the TLB is called hit ratio
  - An 80% hit ratio means that we find the desired page number in the TLB 80% of the time
    - This is a relatively good hit ratio
  - Example
    - Question: Suppose that takes 10ns to access memory if we find the desired page in TLB, otherwise we need two memory access so it is 20 ns
    - Answer: Effective Access Time (EAT) =
      (0.80 x 10) + (0.20 x 20) = 12ns
      Therefore, 20% slowdown in access time
  - Question: Calculate slowdown in access time for hit ratio of 99%
    - The EAT will be much lower than the hit ratio for 80%

- Memory Protection
  - Some pages in memory need to be dedicated to the operating system, and some pages need to be protected so other processes cannot read them
    - Each process should only be able to access its own memory
  - Memory protection in paged environment is accomplished by protection bits associated with each frame
    - These bits are kept in the page table
    - One bit can define a page to be read-write or read-only
  - 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
  - i.e. Diagram of Page Table With Bit





  - Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the OS
    - The only valid pages are from 0 to 5
    - If the process has a page 6, then it will not get the corresponding frame associated to page 6
      - Because there is nothing in the frame
    - This is kind of like a page fault
      - A page fault results in trying to access a page that is not valid
- Shared Pages
  - Shared code
    - One copy of read-only (re-entrant) code shared among processes
      - i.e. Text editors, Compilers, Window systems, Etc.
    - Similar to multiple threads sharing the same process space
    - Also useful for interprocess communication if sharing of read-write pages is allowed
    - i.e. Multiple processes generate their own logical address space, and all of them are mapped to the same physical address space
  - Private code & data
    - Each process keeps a separate copy of the code and data
      - Nothing is shared
  - The pages for the private code and data can appear anywhere in the logical address space

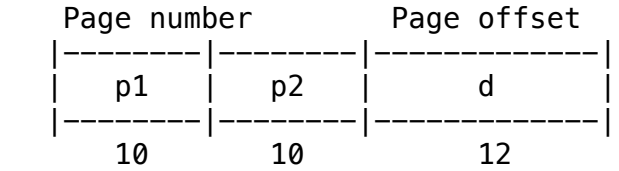
- Shared Pages Example
  - Shared code helps processes reuse code
    - i.e. Multiple processes using the same library, like the standard C library
  - Three processes sharing the pages for the standard C library 'libc'

        - Only one copy of the standard C library need to be kept in physical memory
        - The page table for each user process maps onto the same physical copy of 'libc'
        - Only reading is allowed; no writing
    - Note: On a theoretical level, physical memory can refer to either cache or RAM

- 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 = 4KB
            - 2^12 = 4096 B
        - Page table would have 1 million entries (2^20 = 2^32 / 2^12)
        - If each entry is 4 bytes -> Each process 4MB of physical address space for the page table alone
        - We don't want to allocate the page table contiguously in main memory, because it takes a lot of space
        - One simple solution is to divide the page table into smaller units
            - i.e.
                - Hierarchical paging
                - Hashed page Tables
                - Inverted page tables

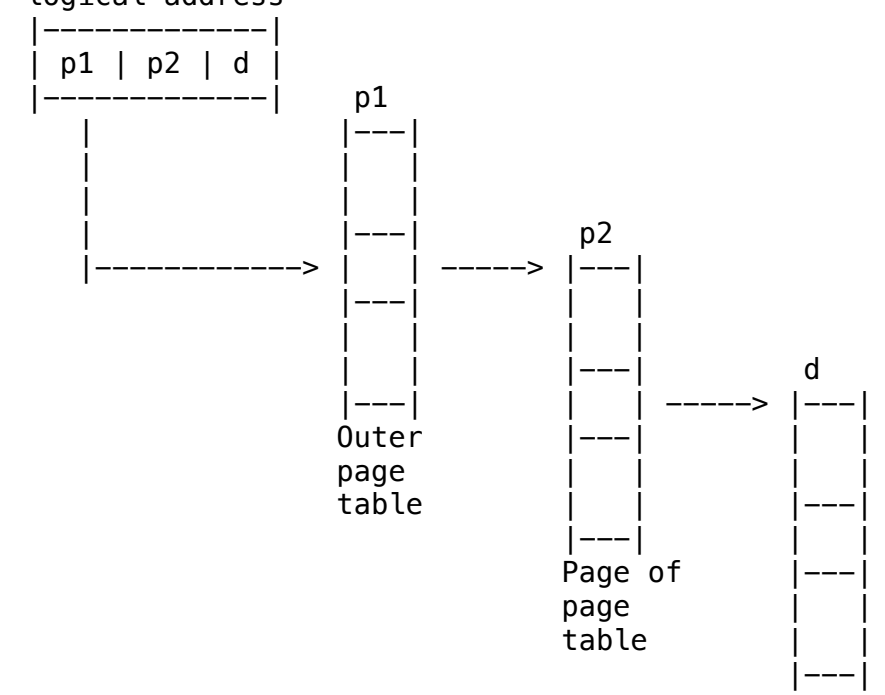
- Hierarchical Page Tables
    - Break up the logical address space into multiple page tables
        - A simple technique is a two-level page table
            - i.e. Outer page table, and inner page table
                - The outer page table points to the inner page table
                - Inner page table points to location in physical memory
                - The page tables do not have to be contiguous and can be placed anywhere in memory
        - We then page the page table
    - Note: This does not save any space in memory

- Two Level Paging Example
    - A logical address (on 32-bit machine with 4K page size) is divided into two numbers:
        - A page number consisting of 20 bits
            - `p1` and `p2` occupy 10 bits
        - A page offset consisting of 12 bits
            - 2^12 = 4096 B = 4 KB
    - Since the page table is paged, the page number is further divided into:
        - A 10-bit page number
        - A 12-bit page offset
    - Thus, a logical address is as follows:



- Where `p1` is an index into the outer page table, and `p2` is the displacement within the page of the inner page table
  - Known as forward-mapped page table
- Note: Hierarchical paging helps the system to better utilize memory space

- Address Translation Scheme
  - i.e. Diagram of Address Translation



  - Address translation for a two level 32-bit paging architecture
    - `p1` is used to go to the outer table
    - Outer table points to the inner table
    - `p2` gives offset to the elements
    - `p1` is used as an offset to find the pointer to the outer table

- 64 Bit Logical Address Space
  - Even two level paging scheme is not sufficient!
  - If page size is 4KB (2^12), then page table has 2^52 entries
  - If two level scheme, inner page tables could be 2^10 with 4-byte entries
  - Outer page table has 2^42 entries
    - One solution is to add a 2nd outer page table

- Three Level Paging Scheme
  - In the following example, the 2nd outer page table is still 2^32 bytes in size
  - i.e. Diagram of Two Level Paging

outer page	inner page	offset
p1	p2	d
42	10	12

    - This is the previous paging scheme
  - i.e. Diagram Three Level Paging

2nd outer page	outer page	inner page	offset
p1	p2	p3	d
32	10	10	12

    - This is the new paging scheme for 64-bit logical addresses
    - The '2nd outer page' can be divided into many pages
      - In modern microprocessors, it goes up to 4 tables
  - The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth

- Hashed Page Tables
  - Common in address spaces with more than 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. Virtual page number
    2. 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

- Hashed Page Table
  - The hash function gives the chain of the page frames
  - 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

- 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
    - An inverted page table has one entry for each real page, or frame, of memory

    - Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of 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 Table Architecture
    - < pid, pagenumber > is presented to the memory subsystem
        - The inverted page table is then searched for a match. If a match is found at entry `i` then the physical address < i, offset > is generated
            - The 'offset' in the page table is the frame
    - In the inverted page table architecture, we look for the content, and the 'offset' is the frame
        - In hashed page table, we look for the offset and the content is the frame
    - Observation
        - How to implement shared memory?
            - This is not possible, because inverted page tables always point to the particular page
                - This is a limitation of inverted page table architecture
        - Map multiple virtual addresses to the same physical address

- Swapping
    - There are two types of addresses:
        - Virtual Address
            - Created by the CPU
        - Physical Address
            - The address on which the data is stored in memory
    - The job of the memory management unit (MMU) is to translate virtual addresses to physical addresses, and vice versa
    - A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
        - The backing store is usually a hard drive (HDD)
    - Backing store: Fast disk large enough to accommodate copies of all memory images for all users
    - Roll out, roll in: Swapping variant used for priority-based scheduling algorithms
    - Major part of swap time is transfer time
        - Total transfer time is directly proportional to the amount of memory swapped

- Schematic View Of Swapping
    - A process' memory can be swapped out, from the main memory into the backing store, for another process
        - i.e. Process P1 is swapped for Process P2
    - The backing store is greater in size than main memory
        - But processes need to be in main memory, in order for them

to be executed
  - Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system!
  - Swapping can take a lot of time

- Context Switch Time Including Swapping
  - 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 be very high
  - Example: 100MB process swapping to hard disk with transfer rate of 50MB/sec
    - Swap out time of 2000ms
    - 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
  - Context switching with swapping increases the time by a huge factor
    - Normally, context switching is very fast and takes micro-seconds. But swapping makes the entire thing very slow

- Swapping In Modern OS
  - Modified versions of swapping are found on many systems
    - i.e. UNIX, Linux, and Windows
  - Normally, swapping is disabled
    - But, it is started if more than threshold amount of memory is allocated
      - And then, disabled again once memory demand reduced below threshold
  - i.e. If 80% or more of main memory has been allocated, then swapping is enabled

- Schematic View Of Swapping With Paging
  - A subset of pages for processes A and B are being paged-out and paged-in respectively
  - Pages of a process rather than an entire process can be swapped
    - Unused parts of a process are moved out
  - Most systems, including Linux and Windows, now use swapping with paging

- Swapping On Mobile Systems
  - Not typically supported
    - Due to lack of huge hard drive (HDD)
  - Flash memory based: Small amount of space, limited number of write cycles, poor throughput between flash memory and CPU on mobile platform
    - Thus, swapping is not a good option
  - iOS asks apps to voluntarily relinquish allocated memory:

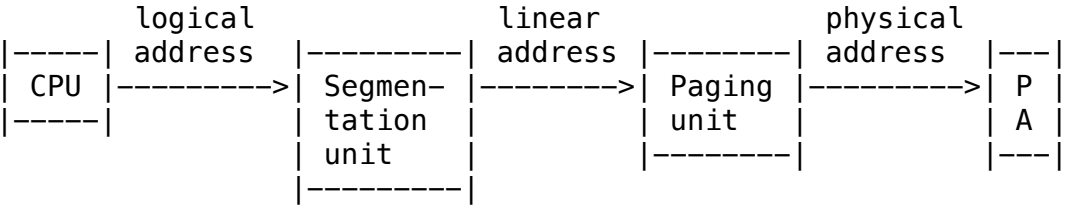
        - Read-only data thrown out and reloaded from flash
        - Failure to free can result in termination
    - Android terminates apps if low free memory, but first writes application state to flash for fast restart
    - Both iOS and Android support paging

- Implementation
    - The Intel 32 and 64-bit architectures
        - These are the current most popular architectures
    - Intel x86-64
    - ARM architecture
        - Mostly used on mobile devices

- Example: The Intel 32 & 64 Bit Architectures (1)
    - The dominant industry chips are:
        - Pentium CPUs are 32-bit
            - Called IA-32 architecture
        - Current Intel CPUs are 64-bit
            - Called IA-64 architecture
        - Many variations in the chips, cover the main ideas here

- Example: The Intel 32 & 64 Bit Architectures (2)
    - The pages may be different sizes when translating from logical to physical addresses
        - This is how it works on most CPUs
    - 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 4KB or 4MB
            - These are typical page sizes
            - 4KB = small
            - 4MB = large

- Logical To Physical Address Translation In IA-32
    - i.e. Diagram of Address Translation



        - Note: PA = Physical Address
    - Between logical and physical address conversion, there are two steps:
        - Segmentation
            - Creates linear address
        - Paging

        - Creates physical address
    - Memory management divided into two components:
        - Segmentation
        - Paging
    - The CPU generates logical addresses, which are given to the segmentation unit
    - The segmentation unit produces a linear address for each logical address
    - The linear address is then given to the paging unit, which in turn generates the physical address in main memory

- Intel IA-32 Segmentation (1)
    - The logical address comprises of two parts
        - Selector
        - Offset
    - The logical address space of a process is divided into two partitions
        - Up to 8K segments that are private to that process
            - Information kept in the local descriptor table (LDT)
        - Up to 8K segments that are shared among all the processes
            - Global descriptor table (GDT)

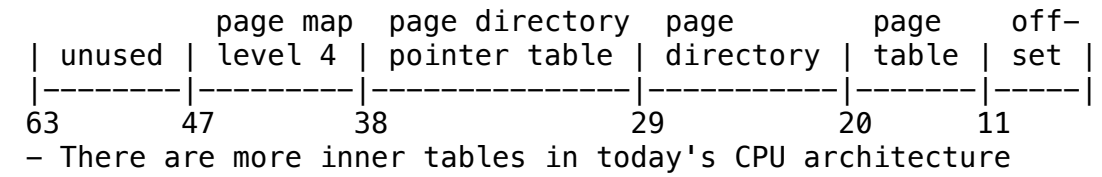
- Intel IA-32 Segmentation (2)
    - The logical address is a pair (selector, offset)
    - The selector is a 16-bit number:

s	g	p
13	1	2

        - Where:
            - `s`: The segment number
            - `g`: Indicates whether the segment is in the GDT or LDT
            - `p`: Deals with protection
    - The offset is a 32-bit number specifying the location of the byte within the segment in question

- Intel IA-32 Paging Architecture
    - The address translation scheme for this architecture is similar to the scheme shown in Figure on Page 39

- Intel x86-64
    - Current generation Intel x86 architecture
        - 64 bits is ginormous (> 16 exabytes)
            - 1 EB = 1,000,000,000,000,000,000 B
        - In practice only implement 48-bit addressing
            - Page sizes of 4 KB, 2 MB, 1 GB
                - Four levels of paging hierarchy
    - Can also use page address extension (PAE) so virtual addresses are 48-bits and physical addresses are 52-bits
    - i.e. Diagram of Current Paging Table



    - There are more inner tables in today's CPU architecture

- Example: ARM Architecture
    - Dominant mobile platform chip
        - i.e. Apple iOS, Google Android, Etc.
    - Modern, energy efficient 32-bit CPU
    - 4 KB and 16 KB pages
        - An architecture should be able to address different sizes
    - 1 MB and 16 MB pages
        - Termed sections
    - One level paging for sections, two level for smaller pages

- End
    - Operating Systems are among the most complex pieces of software ever developed