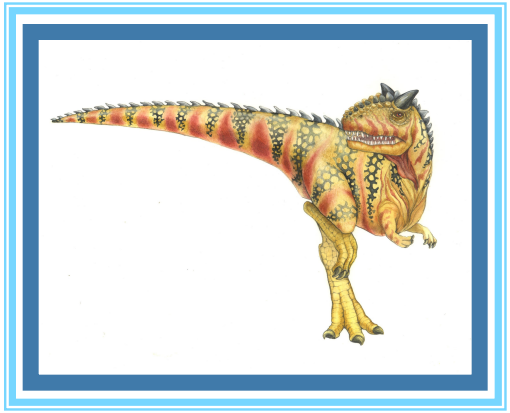
NOTE: Except slide 3, notes in blue are the notes taken in class and notes in green are the ones from chatgpt

Chapter 9: Main Memory

Main memory, also known as RAM (Random Access Memory), is a crucial part of the computer where the OS, application programs, and currently processed data are kept so they can be quickly reached by the computer's processor.



Chapter 9: Memory Management

- Background — basic concepts and importance of memory in a computer system
- Contiguous Memory Allocation — How memory is allocated in a continuous block.
- Paging — A memory management scheme that eliminates the need for contiguous allocation of physical memory
- Structure of the Page Table — The data structure used by the OS to manage paging
- Swapping — moving process between main memory and a storage device to ensure that each process has enough mem
- Example: The Intel 32 and 64-bit Architectures — How these architectures handle memory
- Example: ARMv8 Architecture — Memory management in ARM's 64-bit architecture.

Objectives

- To provide a detailed description of various ways of organizing memory hardware
  Understanding how memory hardware is structured and managed.
- To discuss various memory-management techniques,
  Learning about different methods for managing memory effectively
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

  Specific details about how the Intel Pentium processor handles memory, including both segmentation and paging.

Registers
What are they?
Registers are small, extremely fast storage locations within the CPU itself.
Purpose: They hold data that the CPU needs to access immediately for executing instructions.
Example: When performing a calculation, the CPU uses registers to store intermediate results.
Speed: Accessing data from registers is the fastest, typically done in one CPU clock cycle or less.
Size: Very small in size compared to main memory (usually only a few bytes to a few hundred bytes).

Main Memory (RAM)
What is it? Main memory, or Random Access Memory (RAM), is the larger but slower storage that holds the programs and data currently in use.
Purpose: It provides the working space for the CPU, storing the operating system, application programs, and current data.
Example: When you run a program, it gets loaded from the disk into main memory.
Speed: Slower than registers and cache, can take multiple CPU cycles to access data.
Size: Much larger than registers (usually in gigabytes).

Cache
What is it? Cache is a small, faster type of volatile memory that sits between the CPU and main memory.
Purpose: To store copies of frequently accessed data from main memory to reduce access time and improve overall system speed.
Example: Frequently used instructions and data are stored in the cache so that the CPU can access them quickly without going to main memory.
Speed: Faster than main memory but slower than registers.
Size:Smaller than main memory but larger than registers (typically in kilobytes to megabytes).

Memory Stream (Address Stream)
What is it? The sequence of addresses that the CPU reads from or writes to during program execution.
Purpose: It represents the flow of data between the CPU and memory.
Example: When a program accesses different variables stored in memory, it generates a stream of addresses corresponding to those variables.
Concept: The memory unit (controller) only sees a stream of addresses with read or write requests.

Stalls
What are they? Delays that occur when the CPU has to wait for data to be fetched from main memory or written back to it.
Purpose: To handle situations where the data needed by the CPU is not immediately available.
Example:
If the CPU requests data that is not in the cache and has to be fetched from main memory, the CPU might experience a stall.
Impact: Stalls can slow down the overall performance of the CPU

Protection of Memory
What is it?
Mechanisms to ensure that each process in the system operates within its own allocated memory space and cannot interfere with other processes' memory.
Purpose:To maintain system stability and security by preventing processes from accidentally or intentionally modifying each other's data.
Example:
Using memory management units (MMUs) to enforce access controls and boundaries between processes.
Implementation:
Achieved through techniques like segmentation and paging.

Refer to SLIDEONE previous slide notes for basics

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 is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

The CPU can only access data directly from main memory and registers.

cpu is not able to run instructions, it would be too slow. You can only instructions when it comes from main memory. The program is brought from main memory and then it runs.

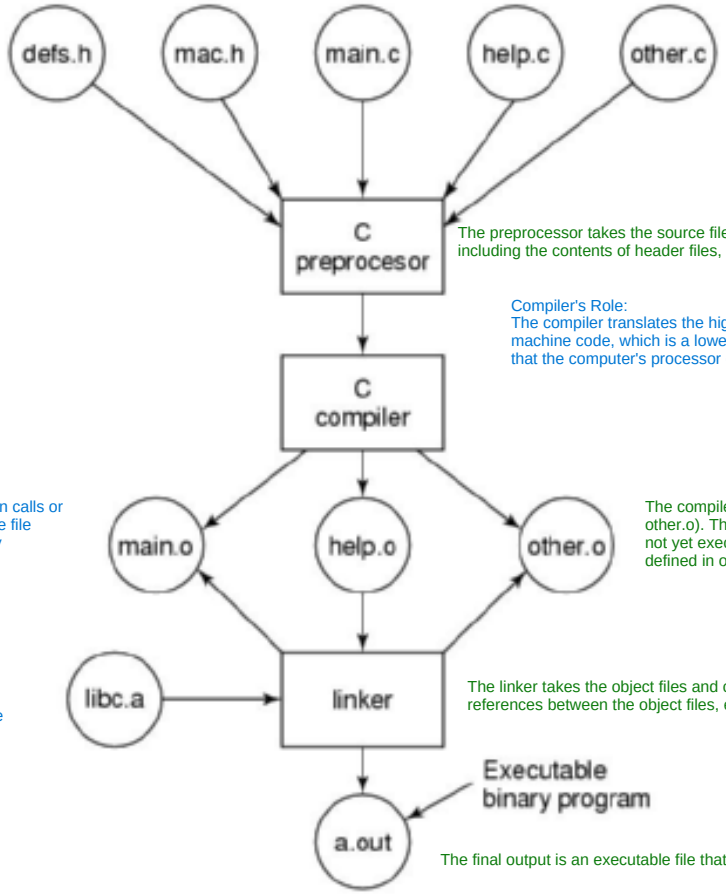
we have read adnd write for 2 reasons.

Registers are very fast, allowing access in one CPU clock cycle or less.

registers are fast, but we don't have many registers in the cpu but they are fast. CPU frequency is typically in order of giga hertz. Typically in order to read from main memory it will take a long time.

Registers: Small, fast storage locations directly within the CPU used for temporary storage of data.
Main Memory: Larger, slower storage used for running programs and storing data that the CPU needs to access quickly.
Cache: An intermediate storage that balances speed and size, making frequently accessed data quickly available to the CPU.
Address Stream: The sequence of memory addresses that the CPU accesses during program execution.
Stalls: Delays that occur when the CPU has to wait for data to be fetched from main memory.

The Model of Run Time



Source Files and Headers:
defs.h, mac.h, main.c, help.c, other.c: These are the source code files and header files written in C. Header files (like defs.h and mac.h) typically contain declarations for functions and macros, while source files (like main.c, help.c, and other.c) contain the actual code.

The preprocessor takes the source files and header files and processes them. This step includes expanding macros, including the contents of header files, and handling conditional compilation instructions.

Compiler's Role:
The compiler translates the high-level C code into machine code, which is a lower-level representation that the computer's processor can execute directly.

Memory Addresses in Object Files:
Each object file (main.o, help.o, other.o) may contain calls or references to functions or data not defined within the file itself. These references are represented by memory addresses that the linker resolves.

The compiler translates the preprocessed code into object files (main.o, help.o, other.o). This step converts the high-level C code into machine code, but the code is not yet executable because it may contain references to functions or variables defined in other files.

The linker is taking a notes of all this, and links.
Putting pieces together. All instructions are in the executable.Role of the Linker:
The linker is crucial because it combines all the pieces of the program into a single executable. It ensures that all instructions and data references are correctly linked, making the program ready to run.

The linker takes the object files and combines them into a single executable binary program (a.out). It resolves references between the object files, ensuring that all function calls and variable references are correctly linked.

The final output is an executable file that can be run on the computer.

Figure 1-30. The process of compiling C and header files to make an executable.

Protection

Address Space Protection:
The operating system needs to ensure that each process (running program) can only access its own allocated memory space. This prevents processes from interfering with each other, which could lead to crashes or security vulnerabilities.

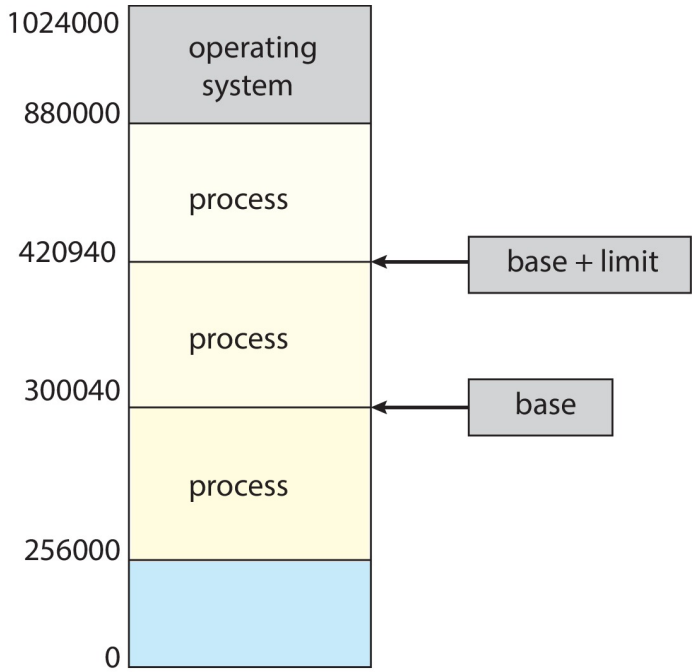
- Need to censure that a process can access only access those addresses in it address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process

Base and Limit Registers:
Base Register: This register holds the starting address of the process's memory space.
Limit Register: This register holds the size of the process's memory space.
Together, these registers define the logical address space of a process. The operating system uses these registers to check that any memory access by the process is within its allowed range.

Diagram Explanation:

Memory Layout:
The diagram shows a memory layout with the operating system at the top and three processes below it. Each process has its own allocated memory space.

Base and Limit Registers:
For each process, the base and limit registers define the range of memory addresses that the process can access. For example, if a process has a base address of 300040 and a limit of 120000, it can access memory addresses from 300040 to 420040.

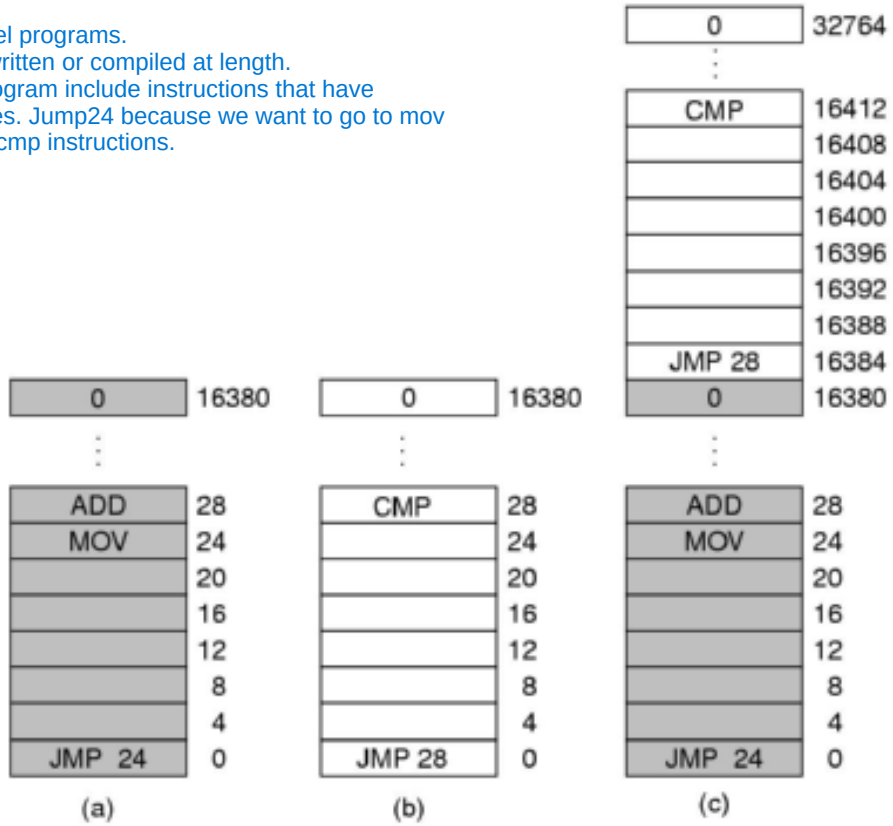


The OS here is placed at top but sometimes we can see its placed lower. But we see three processes assigned together in between. We need some constraints so that not all processes read and write together but have some limits.
Operating System Placement:
The operating system is typically placed at the top of the memory space, but it can also be placed at the bottom or in other configurations. The key point is that the OS manages the memory and ensures that processes do not interfere with each other.
Process Constraints:
The base and limit registers are used to enforce constraints on memory access. This ensures that processes can only read and write within their allocated memory space, providing protection and stability.

Multiple Programs Without Memory Abstraction

The slide shows two assembly-level programs. Each program has instructions that include memory addresses. Assembly language is a low-level programming language that is closely related to the machine code instructions that a computer's CPU executes.

We have two assembly level programs.
Every program has being written or compiled at length.
Both the gray and white program include instructions that have nothing to do with addresses. Jump24 because we want to go to mov instruction. jump 28 cuz of cmp instructions.



Assembly Instructions:
The instructions in the programs (ADD, MOV, CMP, JMP) are basic operations:
ADD: Adds two numbers.
MOV: Moves data from one place to another.
CMP: Compares two values.
JMP: Jumps to a different part of the program
These instructions are similar to commands you might give in a higher-level programming language, but they operate directly on the CPU's registers and memory.

Address Conflicts:
Since both programs use the same addresses, they cannot run simultaneously without interfering with each other. This is the relocation problem. Imagine two people trying to write on the same piece of paper at the same time—they would overwrite each other's work.

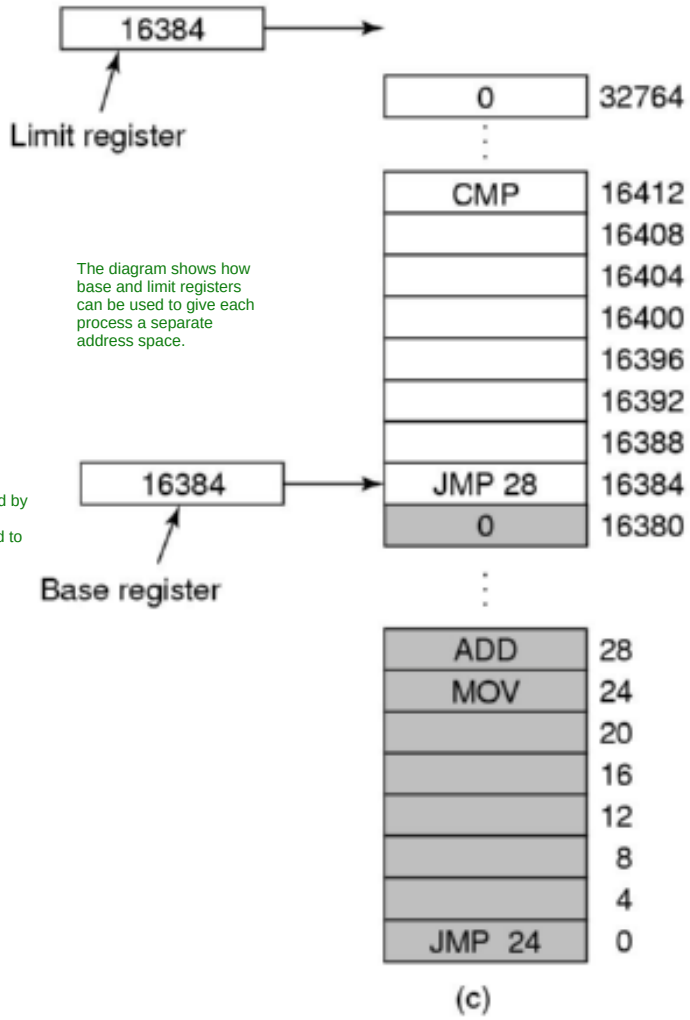
The JMP 24 and JMP 28 instructions tell the program to jump to a specific address to continue execution. Without proper relocation, these jumps will not work correctly if the programs are loaded at different addresses.

Figure 3-2. Illustration of the relocation problem.

Base and Limit Registers

The solution to the relocation problem is dynamic relocation. This involves using base and limit registers to adjust the addresses used by a program at runtime.

Base Register:
Holds the starting address of the process's memory space. Think of it as the starting point of a program's playground.
3. Limit Register:
Holds the size of the process's memory space. This defines the boundaries of the playground, ensuring the program doesn't wander into someone else's space.



The diagram shows how base and limit registers can be used to give each process a separate address space.

The base register is added to the addresses used by the program. For example, if the base register is 16384, an instruction like JMP 28 will be adjusted to 16384 + 28 = 16412.

Solution is dynamic relocation
where in the software you still write jump28 but you will still keep base register. The jmp 28 will be added to 16384.
Dynamic Relocation:
In dynamic relocation, the software still writes instructions like JMP 28, but the hardware adds the base register value to this address at runtime. This ensures that the program runs correctly regardless of where it is loaded in memory.
Base and Limit Registers:
These registers ensure that each process can only access its own memory space. The base register sets the starting address, and the limit register sets the maximum address the process can access.

Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Address Binding

Programs are stored on disk and need to be loaded into memory to execute. This process involves moving the program from disk to an input queue in memory.

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000

Without any support, programs would need to be loaded at address 0000. This is inconvenient because it would mean the first user process always starts at the same physical address, leading to conflicts.

- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Source Code Addresses: Usually symbolic, meaning they use names or labels instead of actual memory addresses.

Compiled Code Addresses: Bind to relocatable addresses, which are addresses relative to the start of the program module (e.g., "14 bytes from the beginning of this module").

Linker/Loader Addresses: Bind relocatable addresses to absolute addresses, which are the actual memory addresses where the program will reside (e.g., 74014).

Linker produces an executable.
we don't download a source. if the executables already knows the addr

The 0 location is too
crowded many programs want to stay at 0.
Source codes have symbolic addresses. All programs individually have addresses that would be conflicting if in main memory. The linker or the loader will decide the real and final addresses. The absolute address is the final address. Can be decide by the linker when you run the executable when you know where it will be stored.

Symbolic Addresses:
In source code, addresses are often symbolic. For example, instead of specifying a memory address directly, you might use a variable name or a label.
Relocatable Addresses:
When the code is compiled, these symbolic addresses are converted to relocatable addresses. These are not fixed memory addresses but are relative to the start of the program.
Absolute Addresses:
The linker or loader converts relocatable addresses to absolute addresses, which are the actual memory locations where the program will be loaded.

Binding of Instructions and Data to Memory

Address binding means when are we going to define addresses of things like functions or data. Like names. The cpu wants to call things by mem addr and programmer wants names. When and how do we bind?

- Address binding of instructions and data to memory addresses can happen at three different stages

Compile Time: If the memory location is known ahead of time, absolute code can be generated. This means the addresses are fixed, and if the starting location changes, the code must be recompiled.

  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes

Load Time: If the memory location is not known at compile time, relocatable code must be generated. This allows the program to be loaded at any memory address.

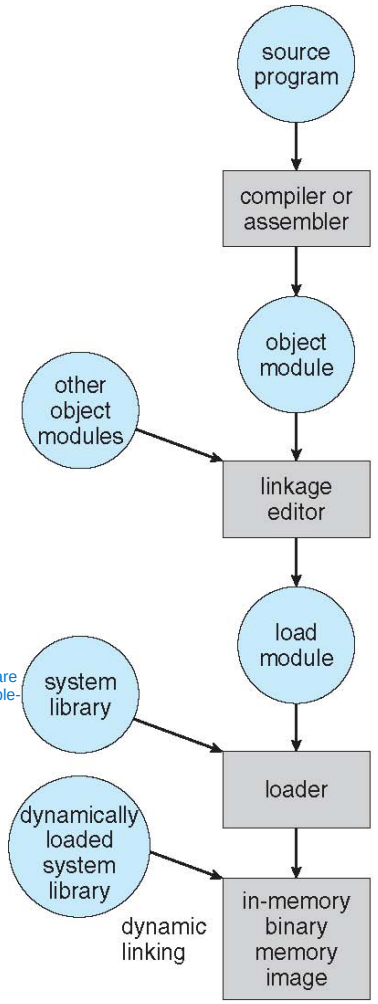
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time

Execution Time: Binding is delayed until runtime if the process can be moved during its execution from one memory segment to another. This requires hardware support for address maps, such as base and limit registers.

  - **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)

Binding of instruction to memory can happen at compile time. Sometimes compilation and links are put together and called build.
The executable knows the address. The load time mean the loader receives the request, start an instance of executable. Which means we should be able to do this.
at execution time: when we load we are immediately going to load. Load time and execution time is kinda similar. So what's the difference between 2, basically here we mean execution not of the full program but part of the program. Suppose your program can be seen as a set of func, at some point of time you are running inside a functions execution time means it could be that we want to just activate or load just part of the program or just to bind the part of the program that we are going to use.

Multistep Processing of a User Program



This is the original code written by the programmer in a high-level language like C or Java.

The source program is passed to the compiler (for high-level languages) or assembler (for assembly language). This step translates the source code into an object module, which is a machine code representation of the program.

The object module is the compiled version of the source program. It contains machine code but is not yet a complete executable because it may need to be linked with other object modules or libraries.

The linkage editor (or linker) combines the object module with other object modules and system libraries to create a load module. This step resolves references between different parts of the program and ensures everything is correctly linked.

The load module is the final version of the program that is ready to be loaded into memory. It includes all the necessary code and data.

The loader is responsible for loading the load module into memory. It places the program in the correct memory location and prepares it for execution.

The loaders module in loading in the main memory.

System Library:
System libraries are collections of pre-written code that programs can use. These libraries are loaded when needed, such as when you double-click an application to run it.

Dynamic linking means that some parts of the program (like certain libraries) are not loaded into memory until they are actually needed during execution. This can save memory and allow for updates to libraries without recompiling the entire program.

This is the final step where the program is loaded into memory and ready to run. The program is now an in-memory binary image, meaning it is in a format that the CPU can execute directly.

Logical vs. Physical Address Space

. Logical Address:
A logical address is generated by the CPU during program execution. It is also referred to as a virtual address. Logical addresses are used by the program and are independent of the actual physical memory location.

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management

A physical address is the actual location in the computer's memory hardware. It is the address seen by the memory unit and is used to access the physical RAM.

Logical address will start from 0. are the ones that can be written in the software, in machine code part of the system. for the software

  - **Logical address** – generated by the CPU; also referred to as **virtual address**

physical are used in the address bus and go in the main memory to we must change something. the difference from logic and physical is addition.

  - **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

Physical Address:
A physical address is the actual location in the computer's memory hardware. It is the address seen by the memory unit and is used to access the physical RAM.

- **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

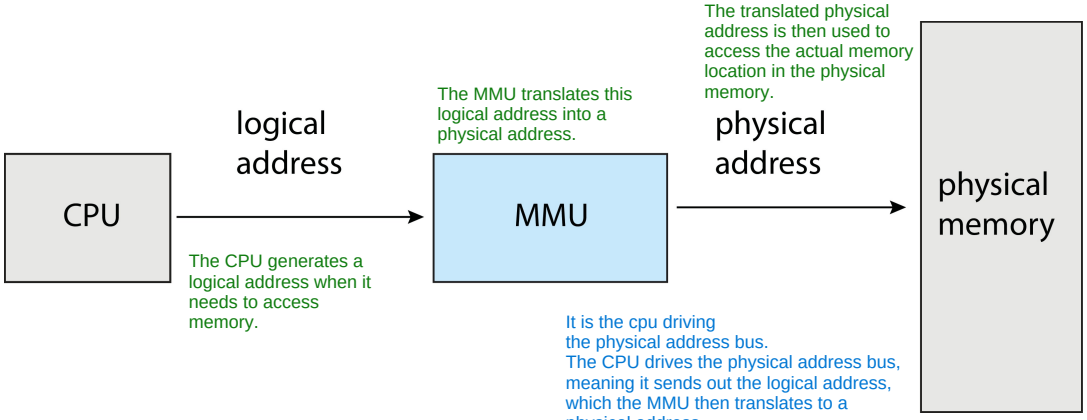
3. Address Binding:
The process of mapping a logical address to a physical address is called address binding. This can happen at different stages:
Compile Time: If the memory location is known ahead of time, the compiler generates absolute addresses.
Load Time: If the memory location is not known at compile time, the loader generates relocatable addresses.
Execution Time: If the program can be moved during execution, address binding happens dynamically using hardware support like base and limit registers.
Notes Explanation:
Logical Address Space:
The logical address space is the set of all logical addresses generated by a program. These addresses are used by the program to reference memory locations.
Physical Address Space:
The physical address space is the set of all physical addresses generated by a program. These addresses are used by the memory unit to access actual memory locations.
Compile-Time and Load-Time Binding:
During compile-time and load-time binding, logical and physical addresses are the same. However, during execution-time binding, logical addresses are translated to physical addresses dynamically

Memory-Management Unit (MMU)

MMU, is a hardware device that maps virtual (logical) addresses to physical addresses at runtime. This is crucial for modern operating systems to manage memory efficiently and securely. Logical Addresses are generated by the CPU during program execution. This is the address used by the program to reference memory locations.

- Hardware device that at run time maps virtual to physical address

Physical addr: The actual address in the computer's physical memory (RAM). This is where the data is actually stored.



- Many methods possible, covered in the rest of this chapter

When a given register is asking loads from the memory, the memory register are going to be loaded with the physical address. The MMU essentially acts as a translator or a middleman between the CPU and the physical memory. It ensures that the logical addresses used by the program are correctly mapped to the physical addresses in memory.

the mmu is basically a plus and comparator, you have logicall add and the relocation in .....

Memory-Management Unit (Cont.)

- Consider simple scheme. which is a generalization of the base-register scheme.

Relocation Register:
The base register is now called the relocation register. This register holds the base address of the process in memory.
The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. This ensures that the logical addresses are correctly mapped to physical addresses.

- The base register now called **relocation register**

Relocation Register:
The relocation register is used to adjust the logical addresses generated by the program. For example, if the relocation register holds the value 1000, and the program generates a logical address 200, the MMU will translate this to the physical address 1200 (1000 + 200).

- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Logical vs. Physical Addresses:
The user program deals with logical addresses and never sees the real physical addresses. This abstraction allows the operating system to manage memory more flexibly and securely.

  - Execution-time binding occurs when reference is made to location in memory

This type of binding allows the operating system to move processes in memory while they are running. The MMU handles the translation dynamically, ensuring that the program always accesses the correct physical memory locations.

Execution-time binding occurs when a reference is made to a location in memory. This means that the actual physical address is determined at the time the instruction is executed.

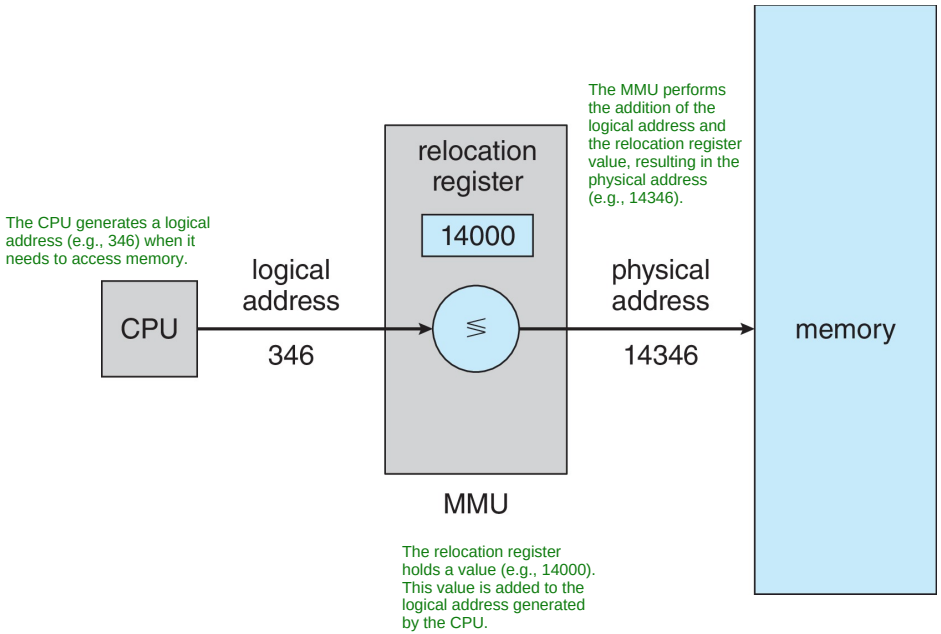
  - Logical address bound to physical addresses

The logical address is bound to a physical address at runtime. This binding ensures that the program can run correctly regardless of where it is loaded in physical memory.

Memory-Management Unit (Cont.)

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation 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

Relocation Register:
The base register is now referred to as the relocation register. This register holds the base address of the process in memory. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. This ensures that the logical addresses used by the program are correctly mapped to physical addresses



Simple Scheme:
This scheme is a generalization of the base-register scheme. It simplifies the process of translating logical addresses to physical addresses by using a single relocation register.
Address Translation:
The value in the relocation register is added to every logical address generated by the user process. For example, if the logical address is 346 and the relocation register value is 14000, the physical address will be 14346 (14000 + 346).

Dynamic Loading

Dynamic loading is a technique where the entire program does not need to be in memory to execute. Instead, routines (parts of the program) are loaded into memory only when they are called.

Routines:In programming, a routine is a sequence of instructions that perform a specific task. Routines are also known as functions, procedures, or subroutines. They are blocks of code that can be called and executed from different parts of a program.

Dynamic Loading of Routines:
In the context of dynamic loading, routines refer to parts of the program that are not loaded into memory until they are actually needed. This means that the entire program does not need to be loaded into memory at once. Instead, only the specific routines that are called during execution are loaded.

- The entire program does need to be in memory to excute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Benefits of Dynamic Loading:
1. Better Memory-Space Utilization: Unused routines are never loaded into memory, saving space.
2. Efficiency: Only the necessary parts of the program are loaded, which can improve performance.
3. Flexibility: Useful when large amounts of code are needed to handle infrequently occurring cases.

Dynamic loading can be implemented through program design without requiring special support from the operating system. However, the operating system can provide libraries to facilitate dynamic loading.

Routine Loading:
Routines are kept on disk in a relocatable load format. They are loaded into memory only when they are called by the program.
Handling Large Code:
Dynamic loading is particularly useful for programs that have large amounts of code to handle rare cases. Instead of loading all the code at once, only the necessary parts are loaded when needed.
OS Support:
While dynamic loading can be implemented by the program itself, the operating system can assist by providing libraries that make dynamic loading easier to implement.

Dynamic Linking

In static linking, system libraries and program code are combined by the loader into a single binary program image before execution.
Implication: This means that all the code needed by the program, including library functions, is included in the executable file. This can lead to larger executable sizes and less flexibility.

- **Static linking** – system libraries and program code combined by the loader into the binary program image

Dynamic linking postpones the linking of libraries until execution time.
Implication: This allows the program to load and link the necessary libraries only when they are needed during execution. This can save memory and allow for updates to libraries without recompiling the entire program.

- Dynamic linking –linking postponed until execution time

A stub is a small piece of code used to locate the appropriate memory-resident library routine.
Function: When a program calls a library function, the stub replaces itself with the address of the actual routine and then executes the routine.

- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address

Operating System Role:
Check Routine in Memory: The operating system checks if the routine is already in the process's memory address space.
Add to Address Space: If the routine is not in the address space, the operating system loads it into memory.

  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries

Dynamic linking is particularly useful for libraries because it allows multiple programs to share a single copy of the library code in memory, reducing overall memory usage.

- System also known as **shared libraries**

System libraries that are dynamically linked are also known as shared libraries.
Benefit: Shared libraries allow multiple programs to use the same library code, which can save memory and disk space.

- Consider applicability to patching system libraries
  - Versioning may be needed

When updating or patching system libraries, versioning may be needed to ensure compatibility with programs that use the library.

- See also:

  Linking & Loading

  CS-3013 Operating Systems - Hugh C. Lauer

  WPI

Contiguous Allocation

This means that memory needs to be allocated efficiently to ensure that both the OS and user applications can run smoothly.

- Main memory must support both OS and user processes

Efficient allocation ensures that the available memory is used in the best possible way without wastage.

- Limited resource, must allocate efficiently

In memory you should have both os and processes. The operating system is usually placed in the lower part of memory. This area is reserved for the OS and is not used by user processes.

In this method, each process is allocated a single contiguous block of memory. This means that all the memory for a process is in one continuous section.

- Contiguous allocation is one early method the simplest solution
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector . The interrupt vector is a table of pointers used to handle interrupts.
  - User processes then held in high memory User Processes: Held in high memory. Each user process is contained in a single contiguous section of memory.
  - Each process contained in single contiguous section of memory

Single Contiguous Section: Each process is allocated a single, continuous block of memory. This makes it easy to manage memory but can lead to inefficient use of memory if there are many small processes or if processes frequently start and stop.

Contiguous allocation is a method used in computer science and operating systems for allocating disk space to files. In this method, files are stored on disk as contiguous blocks, meaning that all blocks belonging to a file are stored together in consecutive sectors on the disk. This contrasts with non-contiguous allocation methods like linked allocation or indexed allocation, where files are stored in fragmented or scattered locations on the disk.

Contiguous Allocation (Cont.)

Relocation registers are used to protect user processes from each other and from changing the operating system's code and data. They help ensure that each process stays within its allocated memory space.

The base register sets the starting address for the process's memory, while the limit register sets the maximum address the process can access. Together, they define the process's memory boundaries.

The registers are key in order to provide relocation. The limit register contains the range of logical addresses. Each logical address generated by the process must be less than the value in the limit register. This ensures that the process does not access memory outside its allocated range.

- 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 *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

The base register contains the value of the smallest physical address allocated to a process. It acts as the starting point for the process's memory.

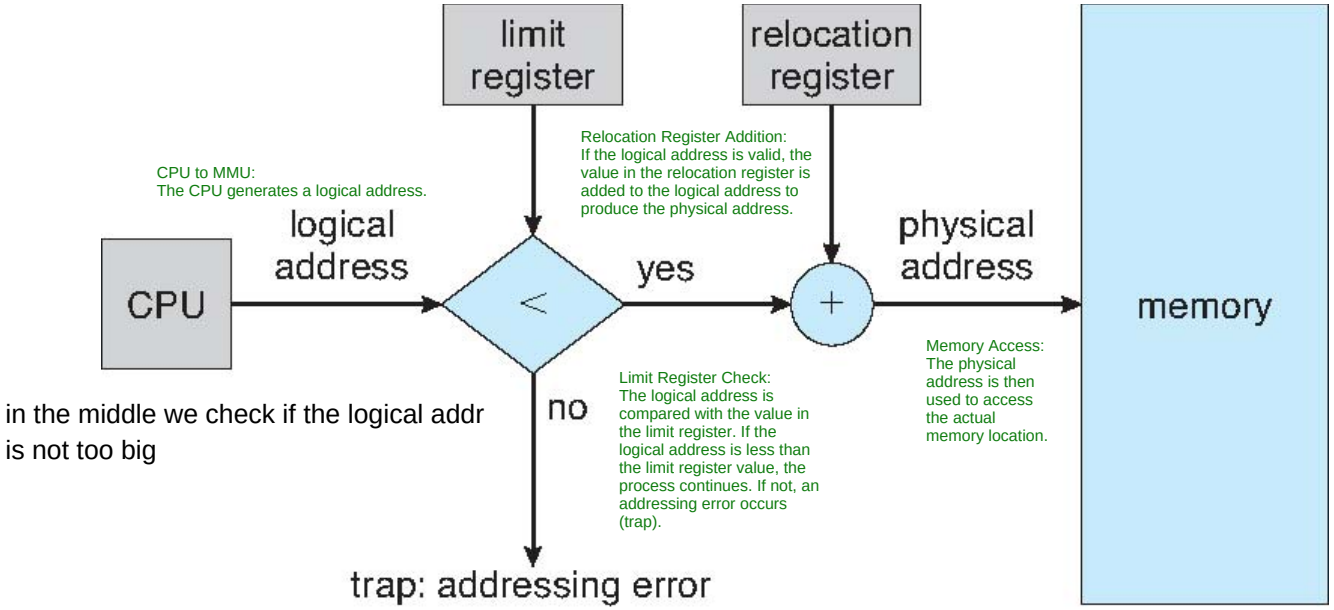
The Memory Management Unit (MMU) maps logical addresses to physical addresses dynamically. This allows the system to move processes in memory if needed. The MMU dynamically maps logical addresses to physical addresses. This means that the actual physical location of a process's memory can change, but the process continues to use the same logical addresses.

The system can handle changes in the kernel's size and location because of the dynamic address mapping. This allows for more flexible and efficient memory management.
The system can handle changes in the kernel's size and location because of the dynamic address mapping. This allows for more flexible and efficient memory management.

We can also think about moving a process from one place to another its technically possible! By changing the base register to be added to logical addressess

1. Relocation Register: The relocation register holds the base address of the process in memory. This value is added to the logical address generated by the CPU to produce the physical address.
2. Limit Register: The limit register contains the range of logical addresses. It ensures that the logical address generated by the CPU does not exceed the allocated memory range for the process.

Hardware Support for Relocation and Limit Registers



Notes Explanation:

Addressing Error Trap:
If the logical address exceeds the limit register value, an addressing error occurs, and the process is trapped. This prevents the process from accessing memory outside its allocated range.

Dynamic Address Translation:
The MMU dynamically translates logical addresses to physical addresses using the relocation and limit registers. This ensures that each process stays within its allocated memory space.

Variable Partition

- Multiple-partition allocation

  Memory is divided into multiple partitions, each of which can hold a process. The degree of multiprogramming is limited by the number of partitions.

  - Degree of multiprogramming limited by number of partitions

    Partitions can have variable sizes to efficiently accommodate processes of different sizes. This helps in better memory utilization.

  - **Variable-partition** sizes for efficiency (sized to a given process' needs)

    A hole is a block of available memory. Holes of various sizes are scattered throughout memory. When a process terminates, it leaves a hole that can be used by another process.

  - **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

    When a process arrives, it is allocated memory from a hole large enough to accommodate it. This ensures that memory is used efficiently.

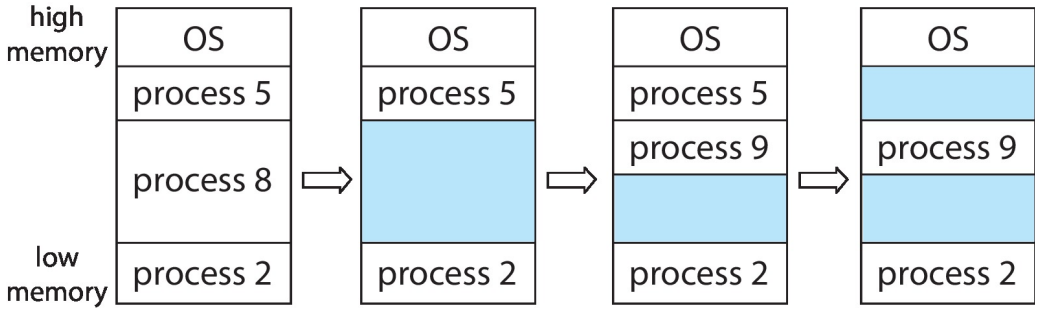
  - Process exiting frees its partition, adjacent free partitions combined

    When a process exits, it frees its partition. Adjacent free partitions are combined to form a larger hole, which can be used by other processes.

  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)

    The operating system maintains information about allocated partitions and free partitions (holes). This helps in efficient memory management.

The consequence of variable, when a p ends, there is free memory. You need another p to start which is small like p9. When you have random alternation of processes starting and ending and each is taking random variable size. Similar situtaion in dynamic memory allocation where each time you allocate the size can be random.



Efficiency:
Variable partition sizes allow for more efficient use of memory by fitting processes into appropriately sized partitions.
Combining Free Partitions:
When a process exits, its partition is freed. If there are adjacent free partitions, they are combined to form a larger hole, which can be used by other processes.
OS Information:
The operating system keeps track of both allocated partitions and free partitions. This information is crucial for efficient memory allocation and deallocation.

Variable partition refers to a memory management scheme used in operating systems to allocate memory to processes dynamically. In this scheme, the available memory is divided into variable-sized partitions, and each partition can accommodate a single process. The size of these partitions can vary depending on the memory requirements of processes.

The diagram shows a memory layout with the operating system and various processes. As processes start and end, the memory layout changes, creating holes of different sizes.
Example:
Initially, memory has the OS and processes 5, 8, and 2.
When process 8 ends, it leaves a hole.
Process 9 starts and is allocated the hole left by process 8.
When process 5 ends, it leaves another hole.
The memory layout changes dynamically as processes start and end.

Dynamic Storage-Allocation Problem

Dynamic Storage-Allocation: the problem involves how to satisfy a request of size n from a list of free holes (blocks of available memory).

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

Allocate the first hole that is big enough to accommodate the request. Advantage: Fast, as it stops searching as soon as it finds a suitable hole. Disadvantage: May lead to fragmentation, as it might leave behind smaller unusable 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

Allocate the smallest hole that is big enough to accommodate the request.
Advantage: Produces the smallest leftover hole, which can help in reducing fragmentation.
Disadvantage: Requires searching the entire list (unless ordered by size), which can be slower.

Allocate the largest hole available.
Advantage: Produces the largest leftover hole, which might be more useful for future allocations.
Disadvantage: Also requires searching the entire list and can lead to more fragmentation overall.

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

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

First-fit:

This algorithm allocates memory by selecting the first available hole (block of memory) that is large enough to accommodate the process.
It is relatively fast since it stops searching for a suitable hole as soon as it finds one that fits the process's size.
However, it may lead to fragmentation, as it might allocate a larger hole than necessary, leaving behind smaller unusable holes.

Best-fit:
The best-fit algorithm allocates memory by selecting the smallest hole that is large enough to accommodate the process.
This strategy requires searching through the entire list of free holes to find the smallest suitable one, unless the list is ordered by size.
It typically produces smaller leftover holes compared to first-fit, which helps in reducing fragmentation.
However, it may be slower than first-fit due to the need to search for the smallest suitable hole.

Worst-fit:
The worst-fit algorithm allocates memory by selecting the largest available hole to accommodate the process.
Similar to best-fit, it requires searching through the entire list of free holes unless they are ordered by size.
While worst-fit can potentially lead to larger leftover holes compared to first-fit and best-fit, it tends to produce more fragmentation overall.
This strategy is generally less efficient in terms of both speed and storage utilization compared to first-fit and best-fit.

Fragmentation

Definition: Total memory space exists to satisfy a request, but it is not contiguous. This means there are enough free memory blocks, but they are scattered and not in a single continuous block.
Example: Imagine having several small free blocks of memory scattered throughout, but none of them are large enough individually to satisfy a large memory request.

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous

Definition: Allocated memory may be slightly larger than the requested memory. The difference between the allocated size and the requested size is wasted within the allocated partition.
Example: If a process requests 18 KB of memory and the system allocates a 20 KB block, the 2 KB difference is wasted as internal fragmentation.

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

As a result of contiguous and variable size, we require fragmentation.

First-Fit Analysis:
Observation: Analysis of the first-fit strategy reveals that given
N blocks allocated, approximately
0.5N blocks are lost to fragmentation.
50-Percent Rule: This rule suggests that about one-third of memory may be unusable due to fragmentation.

External Fragmentation:
Occurs when free memory is fragmented into small blocks scattered throughout the memory. Even if the total free memory is sufficient, it cannot be used effectively because it is not contiguous.
Internal Fragmentation:
Occurs when the allocated memory block is larger than the requested memory, leading to wasted space within the allocated block.
50-Percent Rule:
This rule indicates that a significant portion of memory can be lost to fragmentation, making it unusable. This highlights the importance of efficient memory allocation strategies to minimize fragmentation.

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

Compaction: A technique used to reduce external fragmentation by shuffling memory contents to place all free memory together in one large block. This helps in making larger contiguous blocks of memory available for allocation.

Conditions for Compaction:
Dynamic Relocation: Compaction is only possible if relocation is dynamic, meaning that the addresses used by processes can be changed at execution time. This allows the system to move processes in memory without affecting their execution.

While a job (process) is involved in I/O operations, it must be latched in memory to ensure that it is not moved. This prevents the process from losing its place in memory while it is waiting for I/O operations to complete.

I/O into OS Buffers: To avoid issues with moving processes that are performing I/O, the I/O operations should be done into operating system buffers. This way, the process can be moved without disrupting the I/O operations.

The backing store (secondary storage like a hard disk) can also suffer from fragmentation problems similar to those in main memory. This means that free space on the disk can become fragmented, making it difficult to allocate large contiguous blocks of storage

Compaction
Analogy: Think of your computer's memory like an apartment that is full of furniture. If you want to renovate the apartment, the best solution would be to move everything out to another apartment, do the renovation, and then move everything back. However, in the context of computer memory, you don't have an extra "apartment" (extra memory) to move everything to.
Defragmentation: Instead, you have to shuffle things around within the same apartment to create enough space. This process is called defragmentation. It's like moving furniture around to make more room.
Expense: Defragmentation is expensive because you have to move things around within the limited space you have, which is a complex and time-consuming task.
I/O Problem
Single CPU and Concurrency: Initially, computers had a single CPU that could only handle one process at a time. If a process was not doing I/O (Input/Output operations), the CPU would be idle while waiting for I/O to complete.
Concurrency Motivation: The concept of concurrency (running multiple processes simultaneously) was introduced to make better use of the CPU. While one process is waiting for I/O, the CPU can work on another process.
I/O Operations: I/O operations involve copying data between memory (RAM) and storage (disk). For example, reading data from the disk into memory or writing data from memory to the disk.
Process State During I/O: When a process is performing I/O, it is in a wait state, meaning it is not using the CPU. The CPU can then be used by another process.
Compaction During I/O: If you try to perform compaction (moving processes around in memory) while a process is doing I/O, you risk corrupting the data. This is because the process is actively using the memory locations that you are trying to move.

Key Points
1. Compaction:
Purpose: To reduce external fragmentation by consolidating free memory into a single contiguous block.
Challenge: Requires moving processes around in memory, which is complex and time-consuming.
Dynamic Relocation: Necessary for compaction to be feasible, as it allows the system to change the memory addresses used by processes at runtime.
2. I/O Problem:
Concurrency: Allows the CPU to be used efficiently by running other processes while one process is waiting for I/O.
I/O Operations: Involve copying data between memory and storage.
Wait State: Processes performing I/O are in a wait state and not using the CPU.
Risk of Compaction During I/O: Moving processes in memory while they are performing I/O can lead to data corruption, as the process is actively using the memory locations being moved.

Paging

Paging means no variable size

if all processes had the same size then good, in the end not only non variable but also noncontiguous

Basic Concepts
Frames: Fixed-size blocks of physical memory.
Pages: Fixed-size blocks of logical memory (used by processes).
The key idea is that both frames and pages are of the same size, which allows the system to map pages to frames easily.
Why Use Frames and Pages?
Avoid External Fragmentation: By dividing memory into fixed-size blocks, we avoid the problem of scattered free spaces.
Efficient Memory Management: Any free frame can be used for any page, making it easier to allocate and deallocate memory.
Visual Representation
Physical Memory (Frames)
Imagine physical memory as a series of fixed-size blocks called frames:
Physical Memory:
+---------+---------+---------+---------+---------+---------+
| Frame 0 | Frame 1 | Frame 2 | Frame 3 | Frame 4 | Frame 5 |
+---------+---------+---------+---------+---------+---------+

Logical Memory (Pages)
Logical memory (used by processes) is also divided into fixed-size blocks called pages:

Process 1:
+---------+---------+
| Page 0 | Page 1 |
+---------+---------+

Process 2:
+---------+---------+---------+
| Page 0 | Page 1 | Page 2 |
+---------+---------+---------+

Mapping Pages to Frames
The operating system keeps track of which pages are loaded into which frames using a page table. Each process has its own page table.
Example:
Process 1:
Page 0 -> Frame 2
Page 1 -> Frame 4
Process 2:
Page 0 -> Frame 1
Page 1 -> Frame 3
Page 2 -> Frame 5

physical memory is called frams and logical memory is pages

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

  Noncontiguous Allocation: The physical address space of a process can be noncontiguous. This means that a process is allocated physical memory wherever free memory is available, rather than in a single continuous block. Advantages: Avoids External Fragmentation: Since memory is allocated in fixed-size blocks, there are no gaps between allocations, reducing external fragmentation. Avoids Varying Sized Memory Chunks: Memory is divided into fixed-size blocks, simplifying memory management.

  - 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

  Frames: Physical memory is divided into fixed-sized blocks called frames. The size of a frame is typically a power of 2, ranging between 512 bytes and 16 MB.
  Pages: Logical memory is divided into blocks of the same size called pages. Pages are the same size as frames.
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames

  Tracking Free Frames: The system keeps track of all free frames in physical memory.

  Loading Programs: To run a program of size N pages, the system needs to find N free frames and load the program into these frames.
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program

page size = frame size

  Page Table: The page table is a data structure that replaces the base register and is more expensive. It translates logical addresses to physical addresse
- Set up a **page table** to translate logical to physical addresses

  A page table is set up to translate logical addresses (used by the program) to physical addresses (used by the memory hardware). The page table maps each page to a corresponding frame in physical memory.
- Backing store likewise split into pages

  The backing store (secondary storage like a hard disk) is also divided into pages of the same size as the frames in physical memory.

logical is contigous and physical is not

- Still have Internal fragmentation

  Issue: While paging avoids external fragmentation, it can still suffer from internal fragmentation. This occurs when the allocated memory block (page) is slightly larger than the requested memory, leading to wasted space within the page.

  But we still have internal fragmentation, we have a process whose size is the remainder of the last page is called internal fragmentation. If you need to write a song in a notebook, the first few pages will be only written the left over is called internal fragmentation.

The page table is a data structure that replaces the base register and is more expensive. We don't have external fragmentiation because we only to allocate pages. If we have holes, they are always a multiple of page.

In contiguous fragmentation there is no internal fragmentation

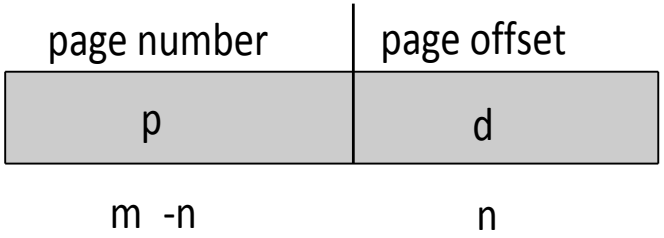
Address Translation Scheme

- Address generated by CPU is divided into:

Page Number (p): The address generated by the CPU is divided into a page number and a page offset. The page number is used as an index into a page table, which contains the base address of each page in physical memory.

- **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

Page Offset (d): The page offset is combined with the 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

How do we translate.

Address Format:
The logical address is divided into two parts:
Page Number (p): Identifies the page in the logical address space.
Page Offset (d): Identifies the specific location within the page.
For a given logical address space 2m and page size 2n the logical address is divided into m−n bits for the page number and n bits for the page offset.

Paging Hardware

translation is done this way where logical address will be split, the displacement in page will directly go to displacement in frame, byte 100 in page will be byte 100 in frame. cuz frame is exactly taking one page. only thing changes is relocation of page.
p is translated to f. p is index in page table and f is the content.

Translation Process:
Logical Address: The CPU generates a logical address, which is divided into a page number (p) and a page offset (d).
Page Table Lookup: The page number (p) is used to index into the page table, which provides the base address of the corresponding page in physical memory.
Physical Address: The base address from the page table is combined with the page offset (d) to form the physical address.

Paging Hardware:
The diagram shows how the logical address is translated to a physical address using the page table.
Page Table: Contains the base addresses of pages in physical memory.
Frames: Physical memory is divided into frames, and each page is mapped to a frame.

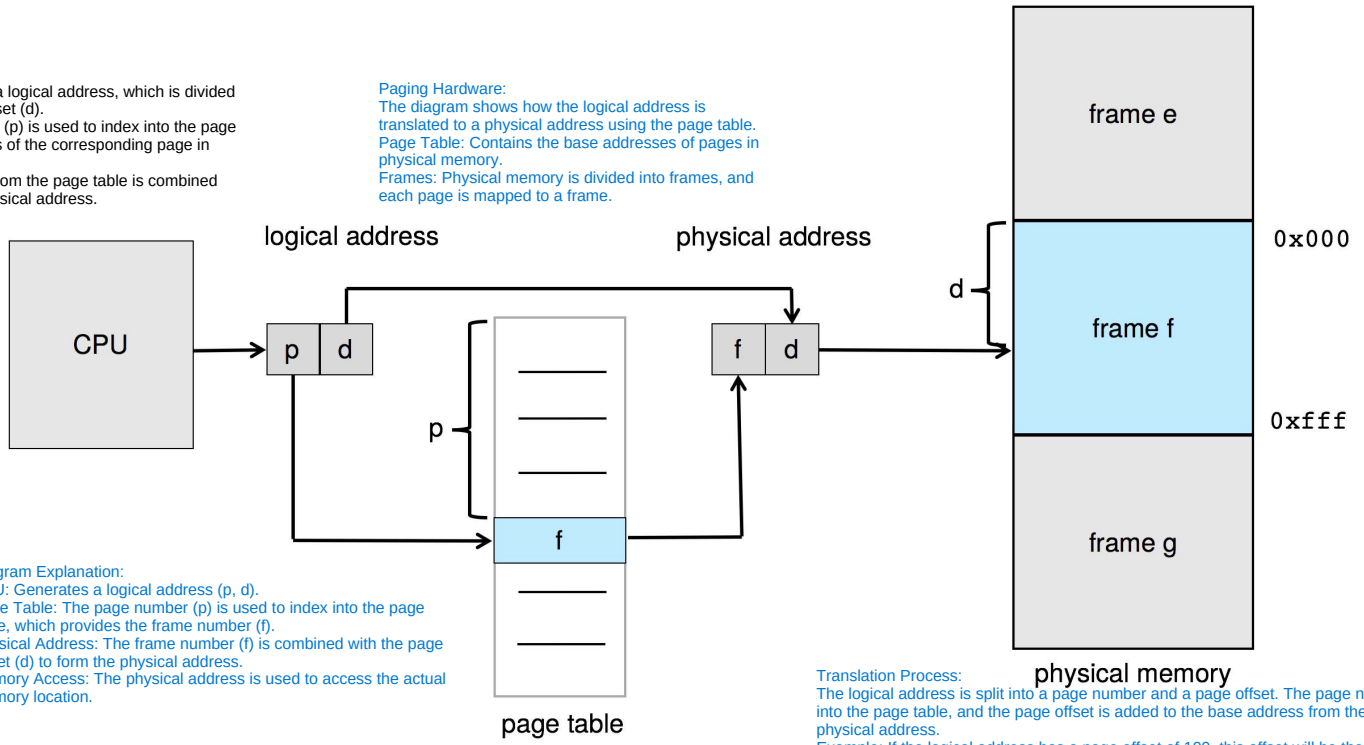


Diagram Explanation:
CPU: Generates a logical address (p, d).
Page Table: The page number (p) is used to index into the page table, which provides the frame number (f).
Physical Address: The frame number (f) is combined with the page offset (d) to form the physical address.
Memory Access: The physical address is used to access the actual memory location.

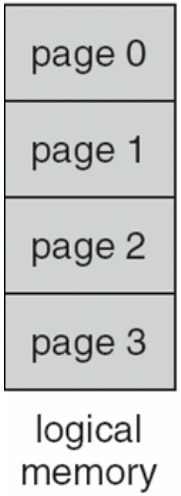
Translation Process:
The logical address is split into a page number and a page offset. The page number is used to index into the page table, and the page offset is added to the base address from the page table to get the physical address.
Example: If the logical address has a page offset of 100, this offset will be the same in the physical address. The only change is the relocation of the page.
Page Table Location:
The page table is not stored in the CPU because it is too large. Instead, it is stored in memory.
Memory Access: When the CPU needs to read or write data, it must first translate the logical address to a physical address using the page table. This requires an additional memory access to read the page table entry.Performance Impact: This additional memory access can slow down the system because it requires extra read/write operations.

Where is the page table? It is not in the cpu, the mmu is in cpu but the page table is not in the cpu cuz it doesn"t fit. too big. One page table of pages of 1 kb that's. imagine 1 mil entries in table.too big. The page table is in memory, it has a consequence. Suppose the cpu asks for fetch, read or write opp, in order to read you have to translate from p to f and read the page table. You need one memory opp you need to do extra mem opp to translate. The page table is the data structure of a kernel. The translation is done through a kernel in the structure. You need direct access. You need to calculate from p where to go. the allocation of kernel data structure from that of processes. For one r/w you need extra r/w. This is a slow down. A very dramatic slow down. So, you don't have fragmentation anymore but you have a slow down.

The page table is stored in memory, not in the CPU, which can impact performance due to additional memory accesses.

Paging Model of Logical and Physical Memory

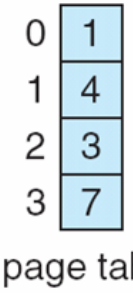
Logical Memory:
Logical memory is divided into pages. Each page is a fixed-size block of memory.
Example: Logical memory has pages 0, 1, 2, and 3.



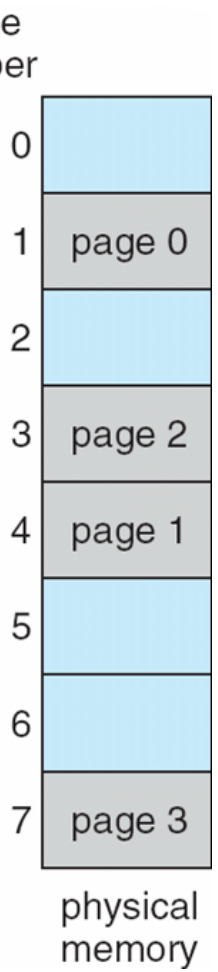
Logical Memory:
+---------+
| Page 0 |
+---------+
| Page 1 |
+---------+
| Page 2 |
+---------+
| Page 3 |
+---------+

Page Table:
+-----+-----+
| 0 | 1 |
+-----+-----+
| 1 | 4 |
+-----+-----+
| 2 | 3 |
+-----+-----+
| 3 | 7 |
+-----+-----+

Physical Memory:
+---------+
| Frame 0 |
+---------+
| Frame 1 | <- Page 0
+---------+
| Frame 2 |
+---------+
| Frame 3 | <- Page 2
+---------+
| Frame 4 | <- Page 1
+---------+
| Frame 5 |
+---------+
| Frame 6 |
+---------+
| Frame 7 | <- Page 3
+---------+



The page table maps logical pages to physical frames.
Each entry in the page table corresponds to a page in logical memory and contains the frame number where that page is stored in physical memory.
The page table shows that:
Page 0 is in Frame 1
Page 1 is in Frame 4
Page 2 is in Frame 3
Page 3 is in Frame 7



Physical memory is a set of frames, and each set can be empty or contain one page. No continuity and even the order can change. page 1 is in 4, page2 in 3. so nocontinguous allocation.
The allocation task is done page by page.

Each register for each page is too much, so we can't use registers because there are too many pages. Instead, we use a page table, which is an array that maps pages to frames. This array is saved in kernel memory.

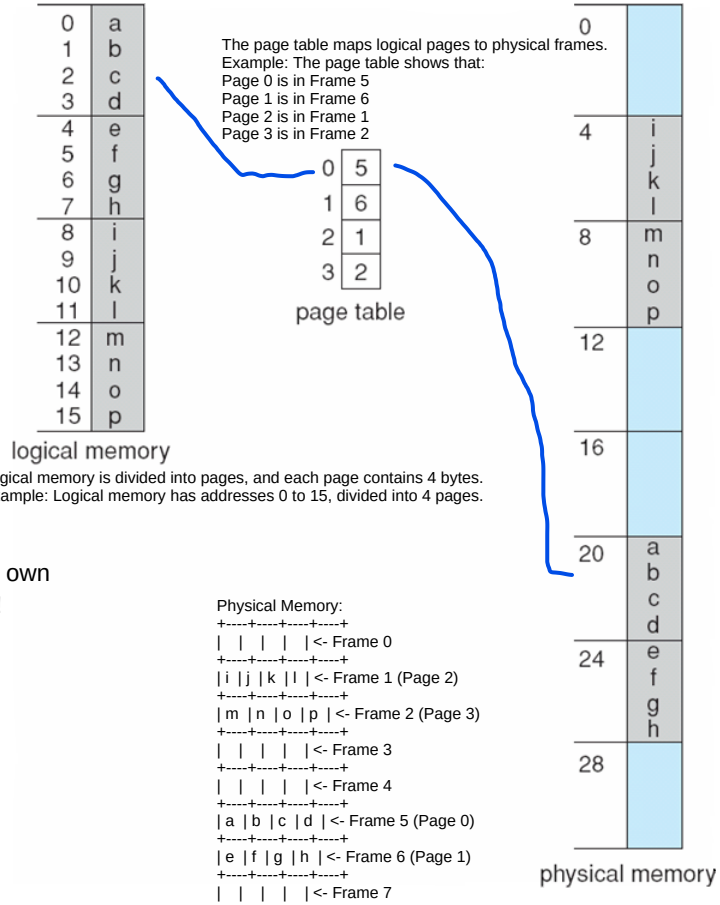
How do we organize the page table, the page 0 will correspond to entry 0.

Physical Memory:
Physical memory is divided into frames. Each frame can hold one page.
Example: Physical memory has frames 0 to 7, and the pages are stored in these frames as indicated by the page table.

Paging Example

Logical Address:
The logical address space is divided into pages, and each page is divided into bytes.
Example: Logical address space has n=2 and m=4, meaning a page size of 4 bytes and a physical memory of 32 bytes (8 pages).

- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



Logical memory is divided into pages, and each page contains 4 bytes.
Example: Logical memory has addresses 0 to 15, divided into 4 pages.

suppose you have a logical mem of 16 addr, and they are on 4 bits. from 0 to 15. suppose that the 16 addr are devided in 4 pages of 4 bytes for them. 0,1,2, and 3 the binary number on 4 bits start with 4 zeroes. 4,5,6,7 all top most bits are 01. 8,9,10,11 are ....
If you take the top 2 bits and group them which can be seen as the index of a page.
You can say you have four bits, the first 2 bits are the page numbers and the least significant bits are the offset of the page.

letter k is in page 2 with offset 2. L in page 2 with offset 3

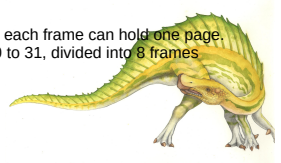
for better understanding solve this on your own on paper. Need to visualize with bits.!!!!!!!!!

Physical Memory:

```
+----+----+----+----+
|    |    |    |    | <- Frame 0
+----+----+----+----+
| i  | j  | k  | l  | <- Frame 1 (Page 2)
+----+----+----+----+
| m  | n  | o  | p  | <- Frame 2 (Page 3)
+----+----+----+----+
|    |    |    |    | <- Frame 3
+----+----+----+----+
|    |    |    |    | <- Frame 4
+----+----+----+----+
| a  | b  | c  | d  | <- Frame 5 (Page 0)
+----+----+----+----+
| e  | f  | g  | h  | <- Frame 6 (Page 1)
+----+----+----+----+
|    |    |    |    | <- Frame 7
+----+----+----+----+
```

Logical Memory: Suppose you have a logical memory of 16 addresses, represented by 4 bits (0 to 15). These addresses are divided into 4 pages of 4 bytes each.
Page Indexing: The top 2 bits of the 4-bit address represent the page number, and the bottom 2 bits represent the offset within the page.
Example: The address k (10 in binary) is in Page 2 with an offset of 2. The address l (11 in binary) is in Page 2 with an offset of 3.

Physical Memory:
Physical memory is divided into frames, and each frame can hold one page.
Example: Physical memory has addresses 0 to 31, divided into 8 frames.



Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes

In order to fit this process we need 35 fully used page and last one partially used (1086 bytes. the rest 962 bytes that is left is internal fragmentation.

- Internal fragmentation of 2,048 - 1,086 = 962 bytes

Internal fragmentation is the unused part, don't confuse with the used part.

- Worst case fragmentation = 1 frame – 1 byte

The worst-case scenario is when a process requires just over a multiple of the page size, leading to almost a full page of internal fragmentation.
Worst case fragmentation = 1 frame - 1 byte.

- On average fragmentation = 1 / 2 frame size

On average, internal fragmentation is half the size of a frame.

- So small frame sizes desirable?

Smaller frame sizes reduce internal fragmentation but increase the number of page table entries.Larger frame sizes increase internal fragmentation but reduce the number of page table entries.

- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

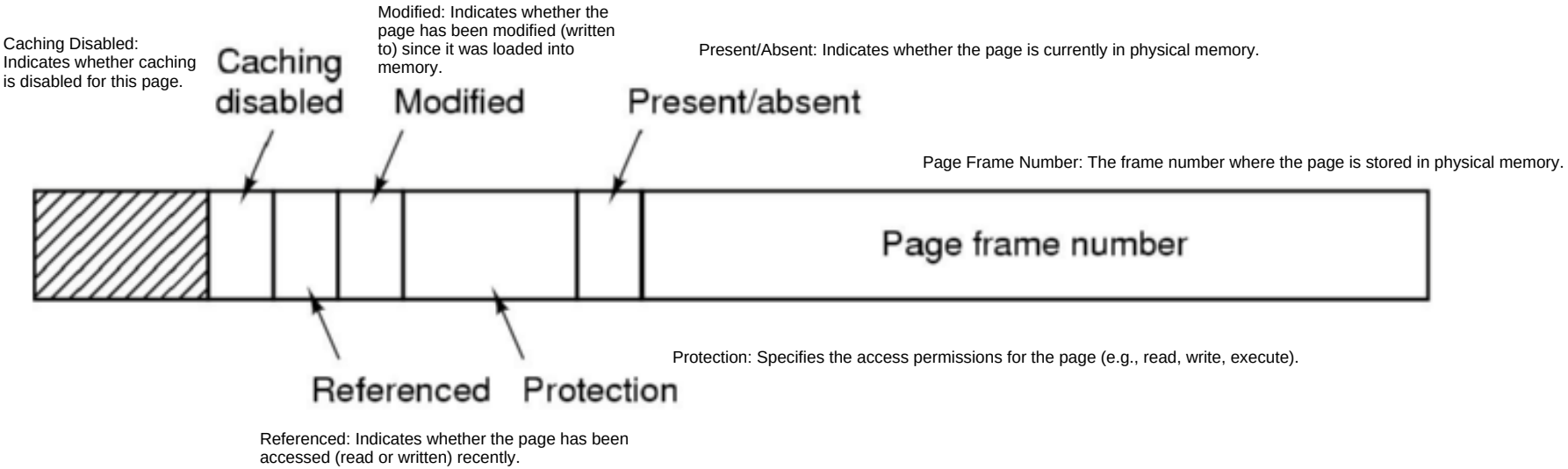
Page sizes have been growing over time to balance the trade-offs between fragmentation and page table size.
Example: Solaris supports two page sizes – 8 KB and 4 MB.

In order to minimize fragmentation, small pages is better. Fragmentation is asking for small pages, page table is asking for larger pages. Since, every page in a process needs one entry. So in the end the page table would prefer larger pages, worst fragmentation. These are two criteria which have to be given priority to one accordingly. The size of processes is increasing over the years. Larger pages are going to be considered.

Small Pages vs. Large Pages: Smaller pages reduce fragmentation but require more page table entries. Larger pages increase fragmentation but require fewer page table entries.
Page Table Memory: Each page table entry takes memory to track, so there is a trade-off between page size and the number of entries.

Structure of Page Table Entry

Frame Number: The frame number can be a large number, potentially in the range of millions. Typically, the entry in an array of page tables is in bytes.
Page Table Entry Size: Page table entries are typically 32 bits. If the frame number uses 24-26 bits, the remaining bits can be used for additional information.
Extra Bits: The extra bits can be used to add characteristics to each page, such as caching, modification status, presence, reference status, and protection.
Implementation: The page table is implemented in a way that allows specifying different characteristics for each page, such as whether it is cached or not.



Since pages are allowed to have one entry for each page. Frame number is a number that could be in the range of millions. Typically entry in an array of page tables should be in bytes. Page table entries of 32 bits (imagine), lets say frame number is happy with 24-26 bits. we can use the extra bits to add extra info associated to those pages. Instead of having a protection for all we can protection for each. You can spcify differenct characteristic for each page. You can have pages that are cached and pages that are not cached. Page table is implemented the following way:

Figure 3-11. A typical page table entry.

Implementation of Page Table

Page Table in Main Memory: The page table is kept in main memory. This table maps logical pages to physical frames

- Page table is kept in main memory
  - **Page-table base register** (**PTBR**) points to the page table
    The PTBR points to the starting address of the page table in main memory.
  - **Page-table length register** (**PTLR**) indicates size of the page table
    The PTLR indicates the size of the page table, i.e., the number of entries it contains.
- In this scheme every data/instruction access requires two memory accesses

In this scheme, every data or instruction access requires two memory accesses:
One to access the page table to get the frame number.
One to access the actual data or instruction in physical memory.

  - 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 **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

TLBs store a small number of page table entries to speed up address translation.

Reading the Page Table:
Reading the page table is done through dedicated hardware, exploiting two registers (PTBR and PTLR).
The TLB acts as a cache for the page table, reducing the need to access main memory for every address translation.
Without the TLB, the system would experience a slowdown by a factor of 2 due to the additional memory access required for the page table.

Translation Look-Aside Buffers (TLBs):
To solve the two memory access problem, a special fast-lookup hardware cache called TLBs (also called associative memory) is used.
TLBs store a small number of page table entries to speed up address translation.

Reading in the page table is done through a dedicated hardware. That hardware is expoiting two register. Plus a piece in the cpu a very small table, which is working as a cache for the page table. Slow down a factor of 2 when you cant use this cache.

Translation Look-Aside Buffer

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

Address-Space Identifiers (ASIDs): Some TLBs store ASIDs in each entry to uniquely identify each process. This provides address-space protection for each process. Without ASIDs, the TLB would need to be flushed at every context switch to avoid address conflicts between processes.

On a TLB miss (when the required page table entry is not in the TLB), the value is loaded into the TLB for faster access next time. Replacement policies must be considered to decide which entry to replace when the TLB is full.

Dedicated TLB:
If the TLB is dedicated to a single process, it does not need ASIDs. However, if the TLB is shared among different processes, each entry must include an identifier for the process (ASID).
This allows the TLB to distinguish between entries for different processes, preventing address conflicts.
Example: You can have address page number 10 for process 1 and page number 10 for process 2. The ASID helps the TLB identify which process each entry belongs to.

If the tlb is dedicated to a single process

You can have a tlb shared among diff processes, but in order to do that you need to atleast in each of the line an identifier of the process. You can have addresss, page number 10 for process and page number 10 for process 2. You need to be able to detect which one is which. For this you need address-space identifier (ASIDS). which means entry of tlb is not only page num or frame num but also asids.

Valid bit here just means in the tlb you can have entries valid or not valid. Suppose you have a tlb of 100 numbers. You have two possibilities. When you start with an empty tlb all 0s. Then you will have 1s. It just means usable or not (valid). Modified means that page written has been modified with respect to the copy on the disc cuz sooner or later it will be need to written, so needs to be save again.

Translation Lookaside Buffers

TLB Entries:
Valid Bit: Indicates whether the TLB entry is valid (usable) or not. A valid bit of 1 means the entry is valid, while 0 means it is not.
Virtual Page: The page number in the virtual address space.
Modified Bit: Indicates whether the page has been modified (written to) since it was loaded into memory. A modified bit of 1 means the page has been modified.
Protection: Specifies the access permissions for the page (e.g., Read/Write, Read/Execute).
Page Frame: The frame number in physical memory where the page is stored.

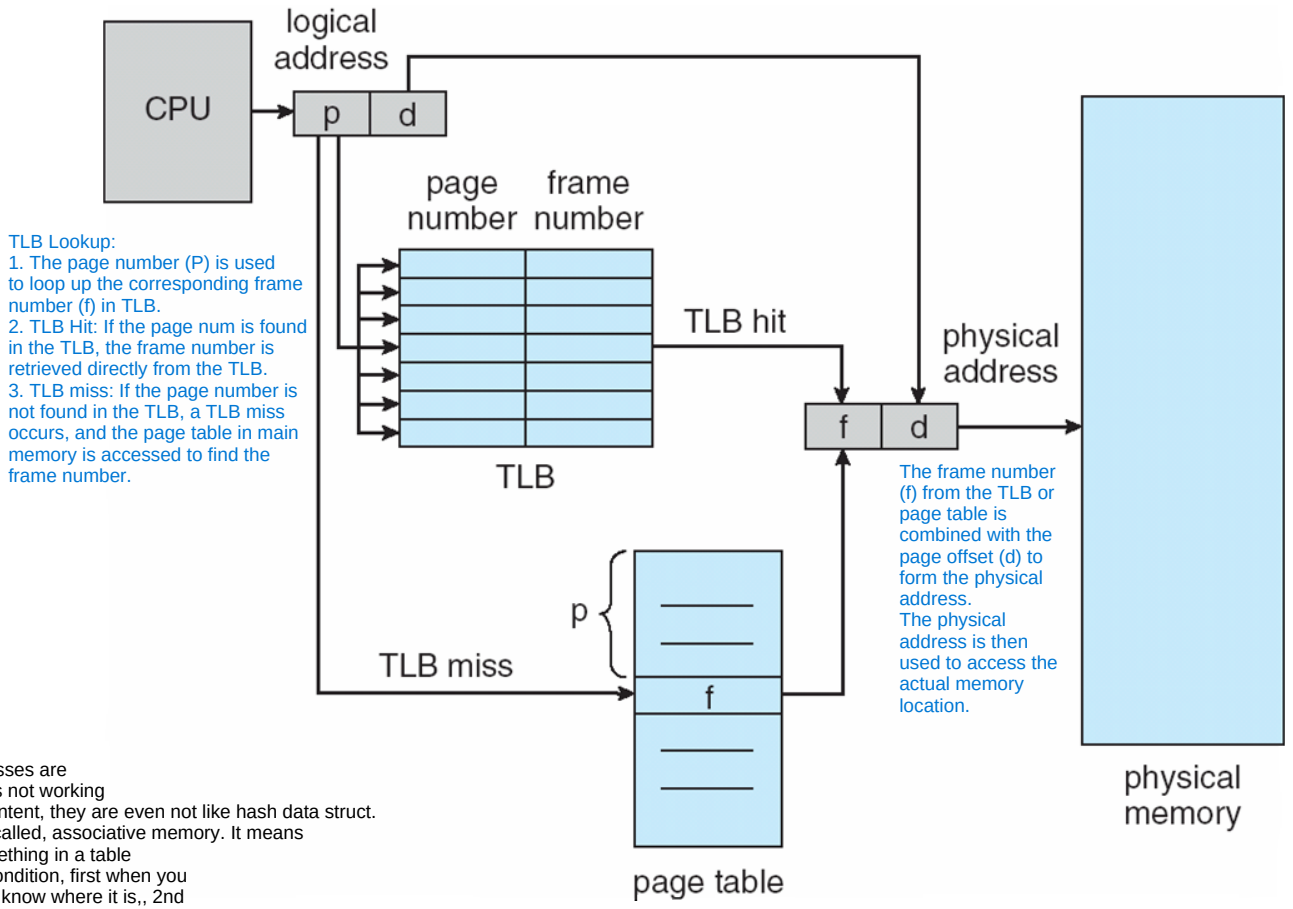
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Valid Bit: The valid bit indicates whether the TLB entry is valid. If the TLB is empty, all valid bits are 0. When entries are added, the valid bits are set to 1.
Modified Bit: The modified bit indicates whether the page has been written to. If the page has been modified, it needs to be written back to disk eventually.
Associative Memory: TLBs use associative memory, meaning that the search for a page number is done by comparing all entries

Figure 3-12. A TLB to speed up paging.

Paging Hardware With TLB

Logical Address:
The CPU generates a logical address, which is divided into a page number (p) and a page offset (d).



TLB Lookup:
1. The page number (P) is used to loop up the corresponding frame number (f) in TLB.
2. TLB Hit: If the page num is found in the TLB, the frame number is retrieved directly from the TLB.
3. TLB miss: If the page number is not found in the TLB, a TLB miss occurs, and the page table in main memory is accessed to find the frame number.

The frame number (f) from the TLB or page table is combined with the page offset (d) to form the physical address.
The physical address is then used to access the actual memory location.

Diagram Explanation:
CPU: Generates a logical address (p, d).
TLB: The page number (p) is used to look up the frame number (f) in the TLB.
TLB Hit: The frame number is found in the TLB, and the physical address is formed.
TLB Miss: The frame number is not found in the TLB, so the page table is accessed to find the frame number.
Page Table: If a TLB miss occurs, the page table is accessed to find the frame number (f).
Physical Memory: The physical address is used to access the actual memory location.

highly probable addresses are
in the cache. The tlb is not working
p is the index and f content, they are even not like hash data struct.
Here we have the so called, associative memory. It means
when you search something in a table
you typically have 2 condition, first when you
know the location and know where it is,, 2nd
to search you do several comparisons. Associative mem
are the ones where you simply tell the memory, you don't
tell the memory the addr but .....
It is basically the only row that is corresponding to p. if p corresponds to one of the left entries on the table you will get the right f.

Associative Memory: TLBs use associative memory, meaning that the search for a page number is done by comparing all entries simultaneously. This allows for fast lookup.
Highly Probable Addresses: The TLB stores highly probable addresses, meaning the addresses that are most likely to be accessed soon.
TLB Miss Handling: On a TLB miss, the page table is accessed to find the frame number, and the TLB is updated with this new entry for faster access next time.

Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
- If we find the desired page in TLB then a mapped-memory access take 10 ns
- Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**)

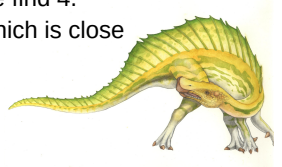
  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

what is the effect of tlb? It is a cache. What is the final result EAT. You have a certain hit ratio suppose 80% means every 5 you have find 4. The EAT is equal to = the prob of hit (80%) x 10 nanoseconds + the prob miss (20%) + 20 nanoseconds. If you want a performace which is close to the original performance you need to have prob close to 1. or high 90s.



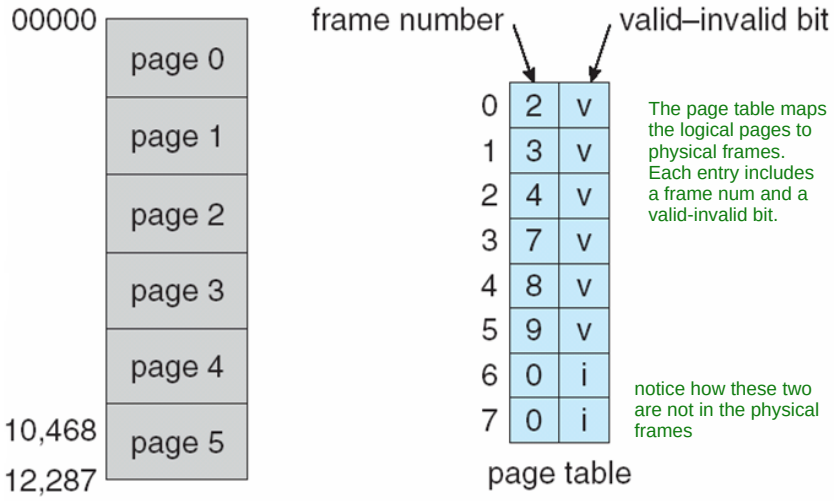
Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

Both the page table and tlb can provide protection in terms of specifying page if read only writable or execute. Execute is diff from reading, execute means only fetch and you cannot copy to somewhere else. The executable code is not readable but can with fetch. Valid is another way where you could use in the page table some validity bit where you can say each bit is not valid. For now a page is not valid when a page does not exist. Suppose your page table of 8 entries but your process only has 6 entries. You have only one possibility to protect valid address. An invalid pointer is an invalid address.

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

one way to protect page 6 is not table is to say yes page table has only 6 entries. The non valid pages could also be in the middle in the future that the addr space could have things on top or bottom of stack and empty in the middle. You could have page table with 100 entries and invalid ones are not only 101..100+ but it could be in the middle.



Each entry in the table has a valid-invalid bit.
1. Valid (v): Indicates that the page is in the process's logical addr space and is currently in physical memory.
2. Invalid(v): Indicates that the page is not in the process's logical addr space or is not currently in physical address.

Protecting Pages: One way to protect pages is to use the valid-invalid bit. For example, if a page table has only 6 entries, pages beyond this range are marked as invalid.
Non-Valid Pages: Non-valid pages could be in the middle of the address space, not just at the end. This means that the address space could have valid pages at the top and bottom, with invalid pages in the middle.
Page Table Size: A page table could have many entries, and invalid entries are not necessarily at the end. They could be scattered throughout the table.

Shared Pages

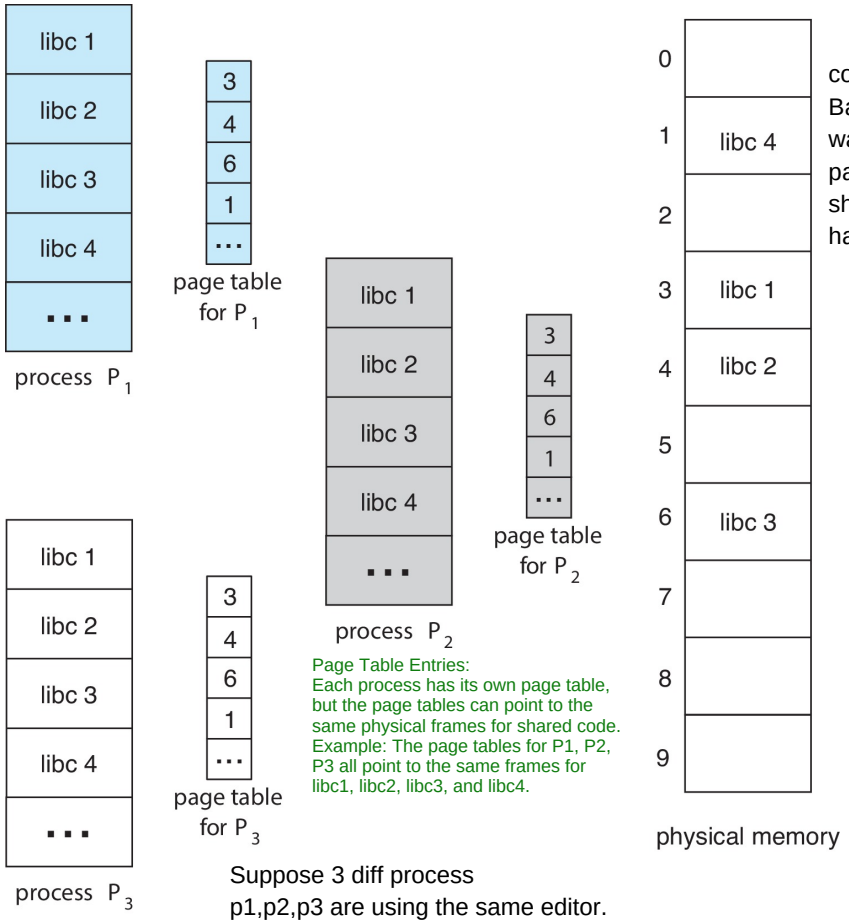
Shared Libraries: Shared pages are similar to shared libraries, where multiple processes can use the same code without duplicating it in memory.
Efficiency: Sharing code among processes reduces memory usage and improves efficiency, as only one copy of the code needs to be loaded into memory.

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared pages are what we have been asking for shared libraries.

Shared Pages:
Shared Code: One copy of read-only (reentrant) code is shared among multiple processes, reducing memory usage and improving efficiency.
Private Code and Data: Each process keeps a separate copy of its private code and data, which can appear anywhere in the logical address space.
Shared Libraries: Shared pages are similar to shared libraries, allowing multiple processes to use the same code without duplicating it in memory.

Shared Pages Example



1. Shared Code:
Multiple processes can share the same code pages to save memory. This is particularly useful for read-only code, such as libraries.
Example: Processes P1, P2 and P3 are using the same shared library (libc).

Page Table Entries:
Each process has its own page table, but the page tables can point to the same physical frames for shared code.
Example: The page tables for P1, P2, P3 all point to the same frames for libc1, libc2, libc3, and libc4.

cont of text below:
Basically you are sharing the editor, this is the way pages can be mapped to share frames. each page table will contain frame that can be shared. shouldn't contain data for one the process they don't have global variables.

Shared Editor Example: Suppose three different processes P1, P2, and P3 are using the same editor. Each process has its own page table, but the page tables point to the same frames for the shared editor code.
Shared Frames: Pages can be mapped to shared frames, allowing multiple processes to use the same code without duplicating it in memory.
No Global Variables: Shared pages should not contain data specific to one process, such as global variables, to avoid conflicts.

Suppose 3 diff process
p1,p2,p3 are using the same editor.
lets say 3 editing word or ppt or notepad.
so process p1 has a copy virtually of libc1 ,
libc2, libc3 where libc3 here is the editor. If the program was statically linked and loaded it will be a copy in my executable. So logically i see 4 pages libc1,2,3,4.
Suppose same thing is done my process 1 or 2 or 3.

Structure of the Page Table

Basics Of Mem Sizes: Bit -> The smallest unit of data in computer, represented as 1 or 0. Byte -> A group of 8 bits. It is the basic unit of data storage. 1KB (Kilobyte): 1024 bytes (2^10 bytes) (which is very close to 1000 hence the word kilo. 1MB (Megabyte): 1024 KB (2^10KB) or 1,048,576 bytes (2^20 bytes). 1GB (Gigabyte): 1024 MB (2^10 MB) or 1,073,741,824 bytes (2^30 bytes)

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers

1. 32-bit Logical Address Space:
Modern computers often use a 32-bit logical address space, meaning the address space can have 2^32 unique addresses.
Example: A 32-bit address space can address up to 4 GB of memory (2^32 bytes).
Remember 1GB is 2^30 bytes.
4x1GB = 4 x 2^30 = 2^2 x 2^30 = 2^32 bytes

  - Page size of 4 KB (2^{12})

Page Size:
A common page size is 4 KB (2^12 bytes).
Example: Each page in memory is 4,096 bytes.

  - Page table would have 1 million entries ($2^{32} / 2^{12}$)

The page table maps logical pages to physical frames.
Calculation: Total number of pages 2^32/ 2^12 = 2^20 pages.
This means there are 1,048,576 pages in a 32-bit address space with 4 KB pages.

  - If each entry is 4 bytes ➔ each process 4 MB of physical address space for the page table alone

Page Table Entry Size:
Each entry in the page table is typically 4 bytes.
Calculation:
Total size of the page table = Number of pages Size of each entry.
Total size = 2^20 pages 4 bytes = 4,194,304 bytes = 4 MB.

    - Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables

Hierarchical Paging: Divides the page table into smaller units to manage large page tables more efficiently.
Hashed Page Tables: Uses a hash table to handle large address spaces.
Inverted Page Tables: Uses a single page table for all processes, reducing the memory needed for page tables.

How can we implement page table? They are a kernel data structure and continuous. You don't read page tables by fetch or r/w. Before being used implicitly, they need to be loaded and need to be prepared.
Kernel Data Structure: The page table is a kernel data structure and is continuous. It is not read by fetch or read/write operations but is used implicitly by the system.
Page Table Creation: The kernel creates the page table when creating a process.
Entry Size: Each entry in the page table is 4 bytes, containing information such as the page number, protection, validity, and other attributes.

Summary
32-bit Logical Address Space: Can address up to 4 GB of memory.
Page Size: Commonly 4 KB, meaning each page is 4,096 bytes.
Page Table Size: For a 32-bit address space with 4 KB pages, the page table would have 1,048,576 entries, each 4 bytes, totaling 4 MB.
Contiguous Allocation: Allocating 4 MB contiguously in main memory is impractical.
Solutions: Hierarchical Paging, Hashed Page Tables, and Inverted Page Tables are methods to manage large page tables more efficiently

Cpu is doing load instruction.

so each entry in page table, in 4 bytes we fit page nm, protection, validity and something else.

Hierarchical Page Tables

Breaking Up the Logical Address Space:
Instead of having one large page table, the logical address space is broken up into multiple smaller page tables.
This approach helps manage large page tables more efficiently.

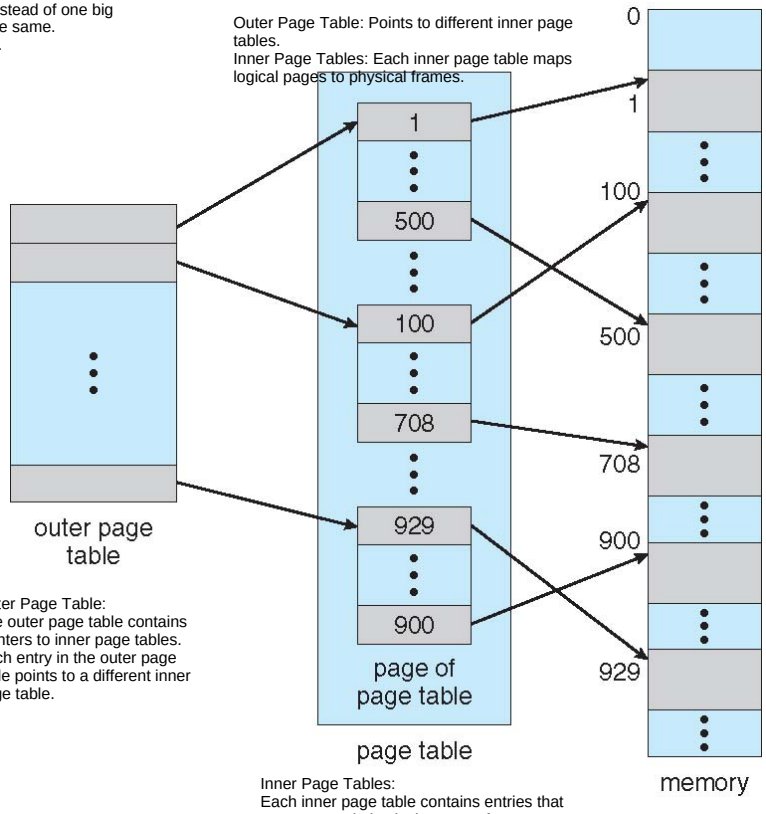
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

wo-Level Page Table:
A simple technique to implement hierarchical paging is to use a two-level page table.
The first level is the outer page table, and the second level consists of multiple inner page tables

Paging the Page Table:
The outer page table points to the inner page tables, which in turn point to the actual pages in memory.
This way, the overall space required for the page table is the same, but it is divided into smaller, more manageable units.

Suppose 4megs is too much we split into 4 smaller pages,
same as you do with directories. An outer page table and inner page table instead of one big contiguous page table we split into many page tables. the overall space is the same.
each second level page table is contiguous inside but not contiguous overall.

Splitting Large Page Tables:
Suppose 4 MB is too much to allocate contiguously. We split it into smaller pages, similar to how directories are organized.
Outer and Inner Page Tables:
Instead of one big contiguous page table, we split it into many smaller page tables. The overall space required is the same, but each second-level page table is contiguous inside but not contiguous overall.



Outer Page Table: Points to different inner page tables.
Inner Page Tables: Each inner page table maps logical pages to physical frames.

Outer Page Table:
The outer page table contains pointers to inner page tables.
Each entry in the outer page table points to a different inner page table.

Inner Page Tables:
Each inner page table contains entries that map to actual physical memory frames.
The inner page tables are contiguous within themselves but are not contiguous overall.

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

Logical Address Division:
A logical address on a 32-bit machine with a 1 KB page size is divided into:
A page number consisting of 22 bits.
A page offset consisting of 10 bits.

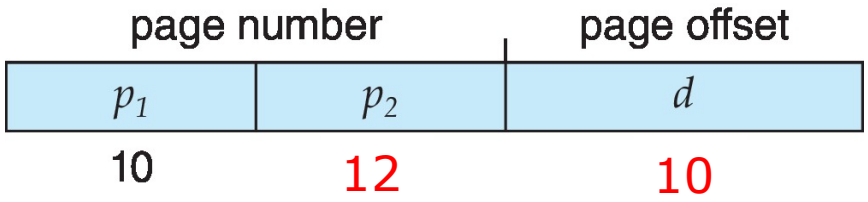
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 12-bit page offset

Since the page table is paged, the page number is further divided into:
A 10-bit page number (p1).
A 12-bit page offset (p2).

Suppose you have 22 bit of page numbers, we further split to 10 and 12. we have outer page table driver by 10 bits. outer page table can have upto.

- Thus, a logical address is as follows:

Logical Address Format:
The logical address is divided as follows:
p1: Index into the outer page table.
p2: Displacement within the page of the inner page table.
d: Page offset.



Logical Address Division:
The logical address is divided into three parts:
p1: Index into the outer page table (10 bits).
p2: Index into the inner page table (12 bits).
d: Page offset (10 bits).
The outer page table points to inner page tables.
The inner page tables map to physical memory frames.

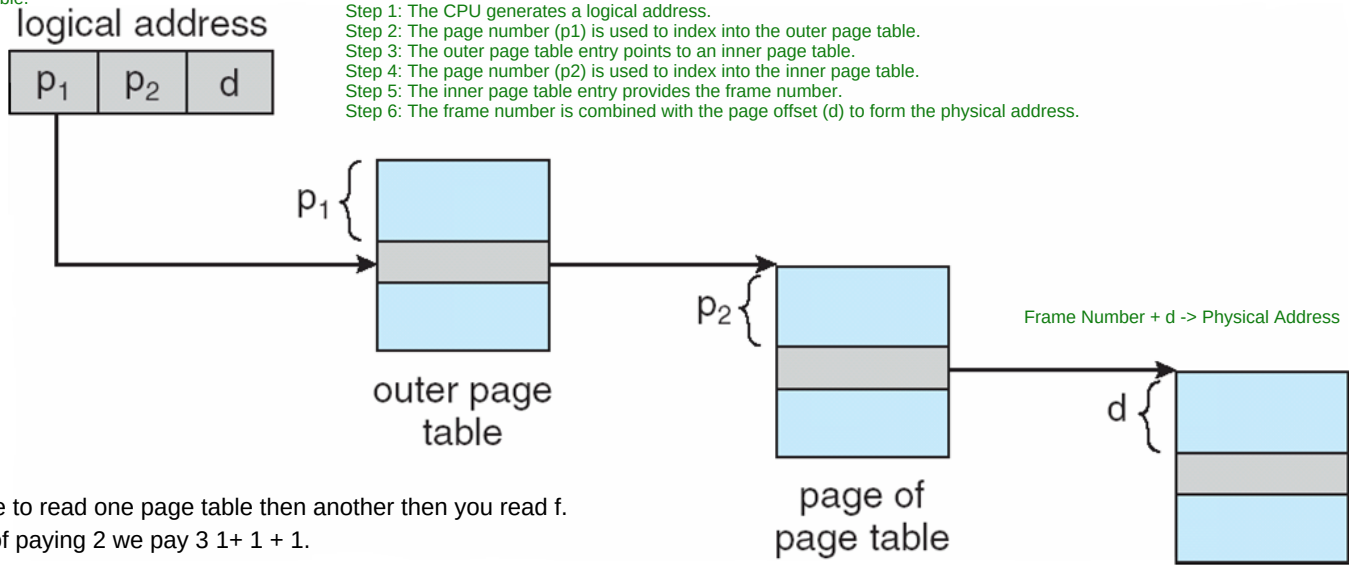
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Splitting Page Numbers: Suppose you have 22 bits of page numbers. We further split them into 10 bits for the outer page table and 12 bits for the inner page table.
Outer Page Table: The outer page table is indexed by 10 bits, allowing it to have up to 1,024 entries.
Forward-Mapped Page Table: This method is known as a forward-mapped page table, where the logical address is divided into multiple levels for efficient management.

Address-Translation Scheme

Logical Address:
The logical address is divided into three parts:
p1: Index into the outer page table.
p2: Index into the inner page table.
d: Page offset within the frame.

Translation Process:
Step 1: The CPU generates a logical address.
Step 2: The page number (p1) is used to index into the outer page table.
Step 3: The outer page table entry points to an inner page table.
Step 4: The page number (p2) is used to index into the inner page table.
Step 5: The inner page table entry provides the frame number.
Step 6: The frame number is combined with the page offset (d) to form the physical address.



You have to read one page table then another then you read f.
instead of paying 2 we pay 3 1+ 1 + 1.

Logical Address: The logical address is divided into three parts: p1, p2, and d.
Outer Page Table: The outer page table is indexed by p1.
Inner Page Table: The inner page table is indexed by p2.
Physical Address: The frame number from the inner page table is combined with the page offset (d) to form the physical address.

64-bit Logical Address Space

Address Space Size:
A 64-bit logical address space is much larger than a 32-bit address space.
Page Size: If the page size is 4 KB (2^12 bytes), the page table would have 2^52 entries.

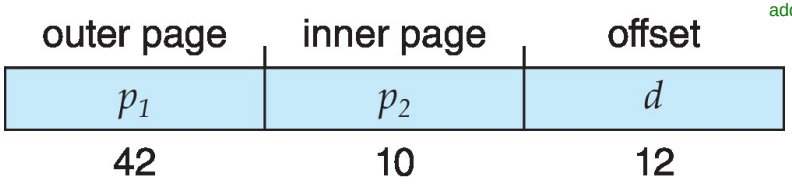
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

Two-Level Paging:
If a two-level paging scheme is used, the inner page tables could have 2^10 4-byte entries.
Address Division:
p1: 42-bit outer page table index.
p2: 10-bit inner page table index.
d: 12-bit page offset.

addresses going from 32 to 64 were done.4

Outer Page Table Size:
The outer page table has 2^42 entries or 2^44 bytes.
One solution is to add a second outer page table to manage the large address space.

Logical Address: The logical address is divided into three parts: p1, p2, and d.
Outer Page Table: The outer page table is indexed by p1.
Inner Page Table: The inner page table is indexed by p2.
Physical Address: The frame number from the inner page table is combined with the page offset (d) to form the physical address.



Memory Accesses:
In this example, the second outer page table is still 2^34 bytes in size.
This could require up to 4 memory accesses to get to one physical memory location.

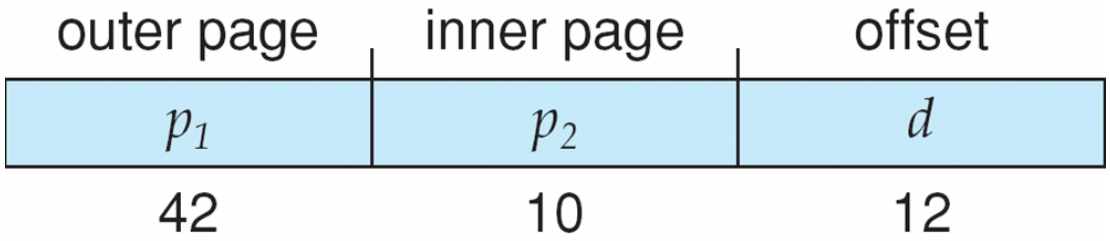
- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2^{nd} outer page table
- But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
  - And possibly 4 memory access to get to one physical memory location

Notes Explanation:
Address Expansion: As addresses expand from 32-bit to 64-bit, the size of the page tables increases significantly.
Multiple Levels: To manage the large address space, multiple levels of page tables are used.
Memory Accesses: Multiple memory accesses are required to translate a logical address to a physical address, which can impact performance.

Three-Level Paging: When two-level paging is not sufficient to handle large address spaces, a three-level paging scheme can be used. The logical address is divided into multiple parts to index through multiple levels of page tables.

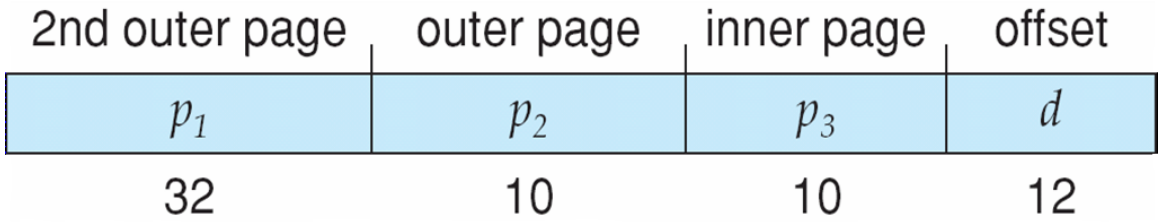
Three-level Paging Scheme

Address Division:
First Example:
p1: 42-bit outer page table index.
p2: 10-bit inner page table index.
d: 12-bit page offset.



First Example:
The logical address is divided into three parts: p1, p2, and d.
p1 indexes into the outer page table.
p2 indexes into the inner page table.
d is the page offset within the frame.

Second Example:
p1: 32-bit second outer page table index.
p2: 10-bit outer page table index.
p3: 10-bit inner page table index.
d: 12-bit page offset.



The logical address is divided into four parts: p1, p2, p3, and d.
p1 indexes into the second outer page table.
p2 indexes into the outer page table.
p3 indexes into the inner page table.
d is the page offset within the frame.

Notes Explanation:
Splitting Large Page Tables: When 4 MB is too much to allocate contiguously, we split it into smaller pages, similar to how directories are organized.
Multiple Levels: Instead of one big contiguous page table, we split it into many smaller page tables. The overall space required is the same, but each second-level page table is contiguous inside but not contiguous overall.

Hashed Page Tables

Hashing is considered in software a way to logically compact a big direct access array in a smaller array but we need a func to convert a big index to small index. The potential addr are very big but we know we know all are not gonna be used.

- 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

Hashing the Virtual Page Number:
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

Searching for a Match:
Virtual page numbers are compared in this chain to search for a match.
If a match is found, the corresponding physical frame is extracted.

  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

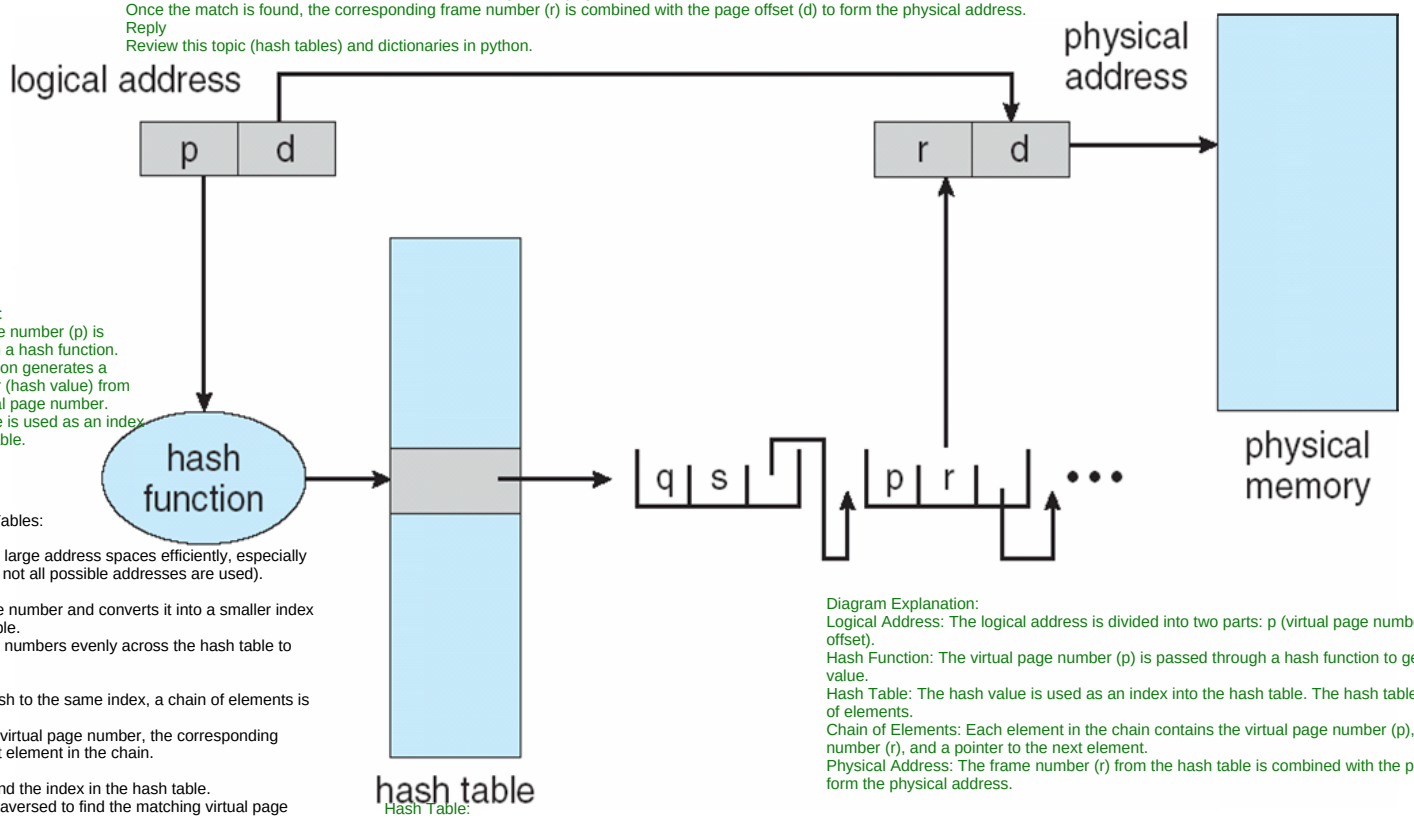
Hashed Page Table:

```
+------------------+
| Hash Function    |
+------------------+
| Chain of Elements |
+------------------+
| Element 1        |
| - Virtual Page   |
| - Mapped Frame   |
| - Next Pointer   |
+------------------+
| Element 2        |
| - Virtual Page   |
| - Mapped Frame   |
| - Next Pointer   |
+------------------+
```

Clustered Page Tables:
A variation for 64-bit addresses is clustered page tables.
Each entry refers to several pages (such as 16) rather than just one.
Especially useful for sparse address spaces where memory references are non-contiguous and scattered.

Hashed Page Table

Address Translation:
The logical address is divided into the virtual page number (p) and the page offset (d).
The virtual page number (p) is hashed to find the corresponding entry in the hash table.
The chain of elements is traversed to find the matching virtual page number.
Once the match is found, the corresponding frame number (r) is combined with the page offset (d) to form the physical address.
Reply
Review this topic (hash tables) and dictionaries in python.



Hash Function:
The virtual page number (p) is passed through a hash function.
The hash function generates a smaller number (hash value) from the larger virtual page number.
This hash value is used as an index into the hash table.

Detailed Explanation of Hashed Page Tables:
1. Purpose:
Hashed page tables are used to handle large address spaces efficiently, especially when the address space is sparse (i.e., not all possible addresses are used).
2. Hash Function:
The hash function takes the virtual page number and converts it into a smaller index that can be used to access the hash table.
The goal is to distribute the virtual page numbers evenly across the hash table to minimize collisions.
3. Handling Collisions:
When multiple virtual page numbers hash to the same index, a chain of elements is created at that index.
Each element in the chain contains the virtual page number, the corresponding frame number, and a pointer to the next element in the chain.
4. Address Translation Process:
The virtual page number is hashed to find the index in the hash table.
The chain of elements at that index is traversed to find the matching virtual page number.
Once the match is found, the corresponding frame number is retrieved.
The frame number is combined with the page offset to form the physical address.

Diagram Explanation:
Logical Address: The logical address is divided into two parts: p (virtual page number) and d (page offset).
Hash Function: The virtual page number (p) is passed through a hash function to generate a hash value.
Hash Table: The hash value is used as an index into the hash table. The hash table contains chains of elements.
Chain of Elements: Each element in the chain contains the virtual page number (p), the frame number (r), and a pointer to the next element.
Physical Address: The frame number (r) from the hash table is combined with the page offset (d) to form the physical address.

Hash Table:
The hash table contains chains of elements that hash to the same location.
Each element in the chain contains:
The virtual page number (p).
The corresponding frame number (r).
A pointer to the next element in the chain.

Chaining: Given a virtual page number (p), which is not used directly as the index in a table, it is used as an index in a table after translation through the hash function. The hash function generates a smaller number with fewer bits from a larger one under some statistical assumption. The small portion is such that they will be well distributed over the array. A linked list may be used.
Two Values Mapped to Single Value: Two values mapped to a single value in the hash table then a list is used. The left is the page number, and the right is the frame number.
Translation Example: In the list, you will have the translation of page numbers 100 and 200. When you translate, you need to traverse this list and find the right page number (p).

Inverted Page Table

Instead of each process having its own page table to keep track of all possible logical pages, an inverted page table tracks all physical pages.
This means there is one entry for each real page of memory, rather than one entry for each virtual page.

- 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

Each entry in the inverted page table consists of:
The virtual address of the page stored in that real memory location.
Information about the process that owns that page.

- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

This approach decreases the memory needed to store each page table because there is only one page table for all processes.
However, it increases the time needed to search the table when a page reference occurs because the table must be searched to find the corresponding entry.

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access

Hash Table for Search Efficiency:
A hash table is used to limit the search to one or at most a few page-table entries.
The Translation Lookaside Buffer (TLB) can accelerate access by caching recent translations.

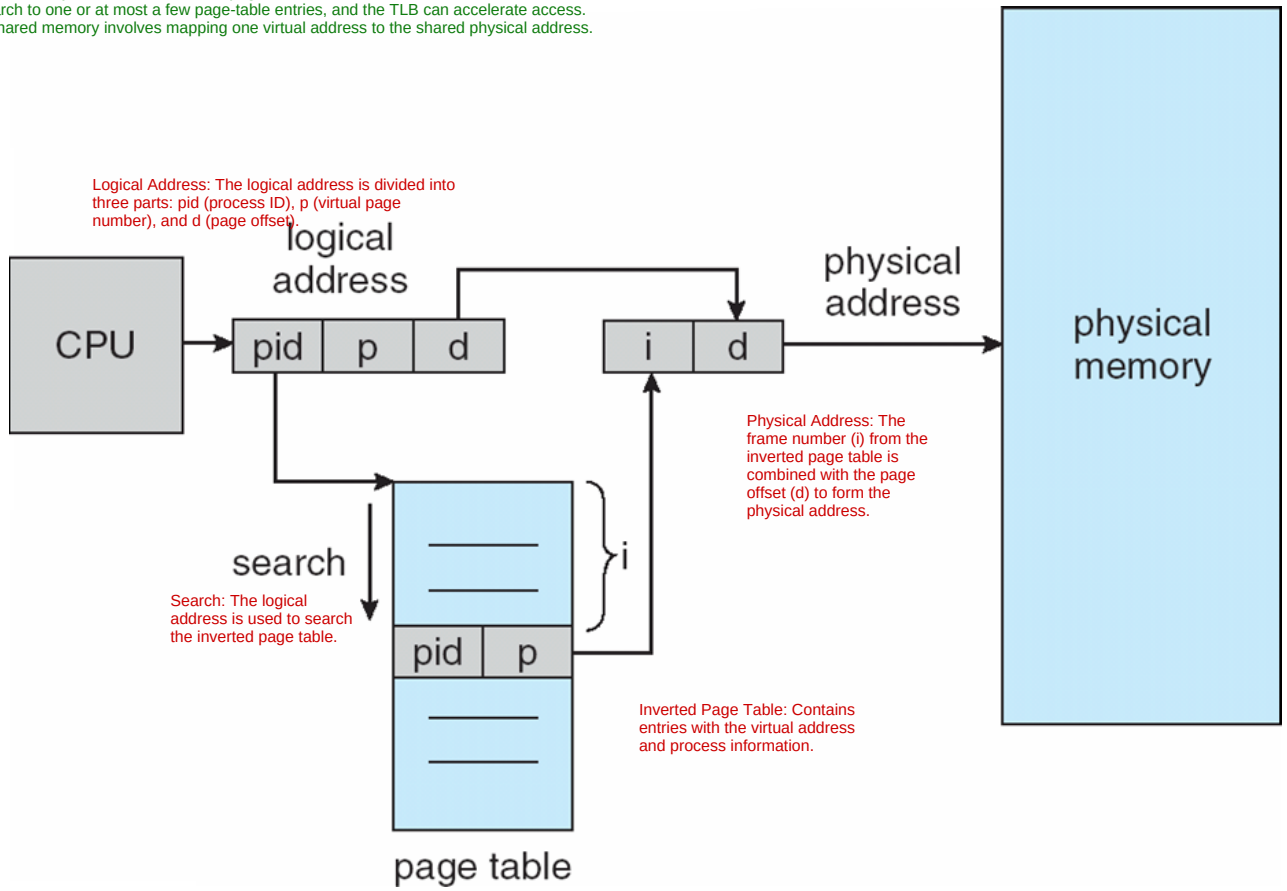
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

5. Shared Memory:
Implementing shared memory with an inverted page table involves mapping one virtual address to the shared physical address.
This allows multiple processes to share the same physical memory.

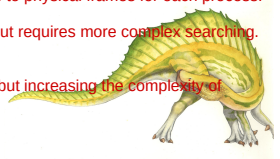
Tracking Physical Pages: Rather than each process having a page table and keeping track of all possible logical pages, the inverted page table tracks all physical pages.
Entry Structure: Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
Memory Efficiency: This decreases the memory needed to store each page table but increases the time needed to search the table when a page reference occurs.
Hash Table: A hash table is used to limit the search to one or at most a few page-table entries. The TLB can accelerate access.
Shared Memory: Implementing shared memory involves one mapping of a virtual address to the shared physical address.

Inverted Page Table Architecture

Inverted Page Tables:
Tracking Physical Pages: Instead of each process having its own page table, an inverted page table tracks all physical pages.
Entry Structure: Each entry contains the virtual address of the page stored in that real memory location and information about the process that owns that page.
Memory Efficiency: Decreases the memory needed to store each page table but increases the time needed to search the table.
Hash Table: Used to limit the search to one or at most a few page-table entries, and the TLB can accelerate access.
Shared Memory: Implementing shared memory involves mapping one virtual address to the shared physical address.

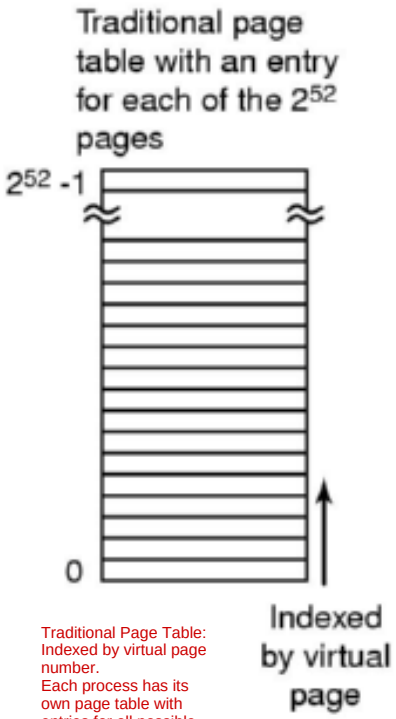


Relation to Previous Concepts:
Single-Level, Two-Level, and Three-Level Paging: These methods focus on managing large page tables by breaking them into smaller, more manageable units. They are primarily concerned with mapping logical pages to physical frames for each process.
Hashed Page Tables: Introduce a way to handle large address spaces by using a hash function to map virtual page numbers to a smaller index, reducing the size of the page table and improving search efficiency.
Inverted Page Tables: Take a different approach by tracking physical pages instead of logical pages. This means there is only one page table for all processes, which reduces the memory needed to store page tables but requires more complex searching.
Key Differences:
Traditional Page Tables: Each process has its own page table, which maps logical pages to physical frames. This can lead to large page tables, especially for large address spaces.
Inverted Page Tables: There is only one page table for all processes, which tracks physical pages. Each entry contains the virtual address and process information, reducing the overall memory needed for page tables but increasing the complexity of searching.



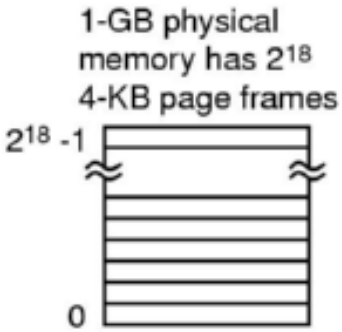
Inverted Page Tables

Traditional Page Tables:
Each process has its own page table with an entry for each possible virtual page.
For large address spaces, this can result in very large page tables.

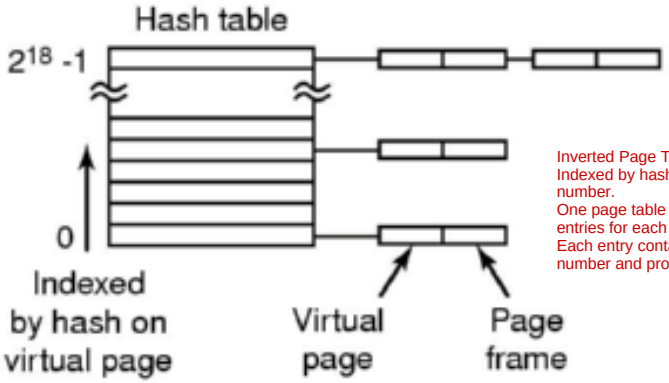


Traditional Page Table:
Indexed by virtual page number.
Each process has its own page table with entries for all possible virtual pages.

Inverted Page Tables:
There is only one page table for all processes, which tracks physical pages.
Each entry in the inverted page table contains the virtual address and process information for the page stored in that physical frame.



. Hash Table for Search Efficiency:
A hash table is used to limit the search to one or at most a few page-table entries.
This helps to manage the increased complexity of searching the inverted page table.



Inverted Page Table:
Indexed by hash on virtual page number.
One page table for all processes, with entries for each physical frame.
Each entry contains the virtual page number and process information

Memory Efficiency:
Inverted page tables reduce the memory needed to store page tables because there is only one page table for all processes.
However, searching the inverted page table can be more complex and time-consuming.

Figure 3-14. Comparison of a traditional page table with an inverted page table.

Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

Modern 64-bit Operating System:
The Oracle SPARC Solaris is a modern 64-bit operating system with tightly integrated hardware.
The goals are efficiency and low overhead.

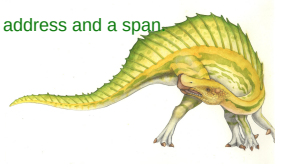
Based on Hashing:
The memory management system is based on hashing but is more complex than simple hashed page tables.

Two Hash Tables:
There are two hash tables:
One for the kernel.
One for all user processes.
Each hash table maps memory addresses from virtual to physical memory.

Contiguous Area of Mapped Virtual Memory:
Each entry in the hash table represents a contiguous area of mapped virtual memory.
This is more efficient than having a separate hash-table entry for each page.

Each entry has a base address and a span (indicating the number of pages the entry represents).

Key Points:
Efficiency and Low Overhead: The system is designed for efficiency and low overhead.
Two Hash Tables: Separate hash tables for the kernel and user processes.
Contiguous Memory Mapping: Each entry represents a contiguous area of virtual memory, improving efficiency.
Entry Structure: Each entry includes a base address and a span.



Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

Translation Lookaside Buffer (TLB):
The TLB holds translation table entries (TTEs) for fast hardware lookups.
A cache of TTEs resides in a translation storage buffer (TSB), which includes an entry for each recently accessed page.

Virtual Address Reference:
When a virtual address is referenced, the TLB is searched.
If there is a TLB miss, the hardware walks the in-memory TSB looking for the TTE corresponding to the address.

Handling TLB Misses:
If a match is found in the TSB, the CPU copies the TSB entry into the TLB, and the translation completes.
If no match is found, the kernel is interrupted to search the hash table.

Kernel Handling:
The kernel creates a TTE from the appropriate hash table and stores it in the TSB.
The interrupt handler returns control to the Memory Management Unit (MMU), which completes the address translation.

TLB and TSB:
The TLB holds TTEs for fast hardware lookups.
The TSB caches recently accessed TTEs.
When a virtual address is referenced, the TLB is searched. If there is a miss, the hardware searches the TSB. If no match is found, the kernel searches the hash table and creates a TTE.

How It Fits In:
Relation to Previous Concepts:
The Oracle SPARC Solaris system builds on the concepts of hashed page tables and TLBs.
It introduces additional complexity with two hash tables and the use of a TSB to cache TTEs.
The system aims to improve efficiency and reduce overhead by using contiguous memory mapping and caching mechanisms.

Swapping

Swapping:
A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
This allows the total physical memory space of processes to exceed the physical memory available.

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping: Temporarily moves processes out of memory to a backing store and brings them back for execution.
Backing Store: A fast disk that stores copies of all memory images.
Roll Out, Roll In: Used for priority-based scheduling, swapping out lower-priority processes.
Swap Time: Mainly consists of transfer time, proportional to the amount of memory swapped.
Ready Queue: Contains ready-to-run processes with memory images on disk.

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

Swapping Back to Physical Addresses:
Whether a swapped-out process needs to swap back into the same physical addresses depends on the address binding method.
Must also consider pending I/O to/from the process memory space.

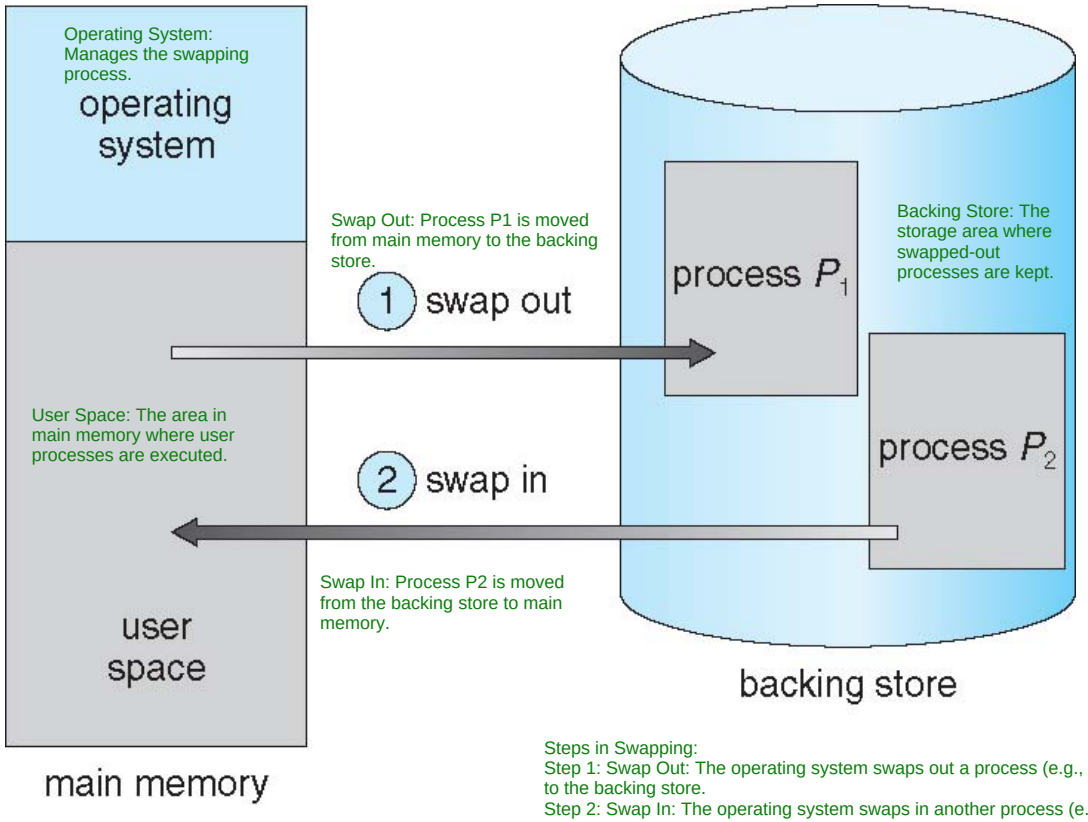
Modified Versions of Swapping:
Found on many systems (e.g., UNIX, Linux, Windows).
Swapping is normally disabled.
It is started if more than a threshold amount of memory is allocated.
Disabled again once memory demand is reduced below the threshold.

Key Points:
Swapping Back: Depends on the address binding method and pending I/O operations.
Modified Swapping: Common in many systems, usually disabled but activated when memory demand exceeds a threshold and disabled again when demand is reduced.

Summary:
Swapping:
Allows processes to be temporarily moved out of memory to a backing store and brought back for execution.
Enables the total physical memory space of processes to exceed the available physical memory.
Uses a backing store (fast disk) to store memory images.
Roll out, roll in is used for priority-based scheduling, swapping out lower-priority processes.
Swap time is mainly transfer time, proportional to the amount of memory swapped.
The system maintains a ready queue of processes with memory images on disk.
Swapping Back to Physical Addresses:
Depends on the address binding method and pending I/O operations.
Modified versions of swapping are common in many systems, usually disabled but activated when memory demand exceeds a threshold and disabled again when demand is reduced.

Schematic View of Swapping

Swapping involves moving a process out of main memory to a backing store (swap out) and then bringing another process from the backing store into main memory (swap in).



Steps in Swapping:
Step 1: Swap Out: The operating system swaps out a process (e.g., Process P1) from main memory to the backing store.
Step 2: Swap In: The operating system swaps in another process (e.g., Process P2) from the backing store to main memory.

Main Memory and Backing Store:
Main Memory: The primary memory where processes are executed.
Backing Store: A secondary storage (usually a hard disk) where processes are temporarily stored when swapped out.

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 then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch and Swapping:
If the next process to be put on the CPU is not in memory, the system needs to swap out a process and swap in the target process.
This can make the context switch time very high.

Example Calculation:
Consider a 100MB process being swapped to a hard disk with a transfer rate of 50MB/sec.
Swap Out Time: 100MB / 50MB/sec = 2000 ms (2 seconds).
Swap In Time: 100MB / 50MB/sec = 2000 ms (2 seconds).
Total Context Switch Time: 2000 ms (swap out) + 2000 ms (swap in) = 4000 ms (4 seconds).

Reducing Swap Time:
The size of the memory swapped can be reduced by knowing how much memory is really being used.
System calls like request_memory() and release_memory() can inform the OS of memory usage.Reducing Swap Time:
The size of the memory swapped can be reduced by knowing how much memory is really being used.
System calls like request_memory() and release_memory() can inform the OS of memory usage.

Context Switch and Swapping: Swapping can significantly increase context switch time if the next process is not in memory.
Example Calculation: A 100MB process with a transfer rate of 50MB/sec results in a total context switch time of 4 seconds.
Reducing Swap Time: Reducing the size of memory swapped by knowing actual memory usage can help reduce swap tim

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Other Constraints on Swapping:
Pending I/O: If there is pending I/O, you can't swap out a process because the I/O would occur to the wrong process.
Double Buffering: To handle pending I/O, you can always transfer I/O to kernel space first, then to the I/O device. This is known as double buffering, but it adds overhead.

Modern Operating Systems:
Standard swapping is not used in modern operating systems.
A modified version of swapping is common, where swapping occurs only when free memory is extremely low.

Pending I/O: Can't swap out a process with pending I/O as it would cause I/O to occur to the wrong process.
Double Buffering: Transfers I/O to kernel space first, then to the I/O device, adding overhead.
Modern OS Swapping: Standard swapping is not used; instead, a modified version is used where swapping occurs only when free memory is extremely low.

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - 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 OSes support paging as discussed below

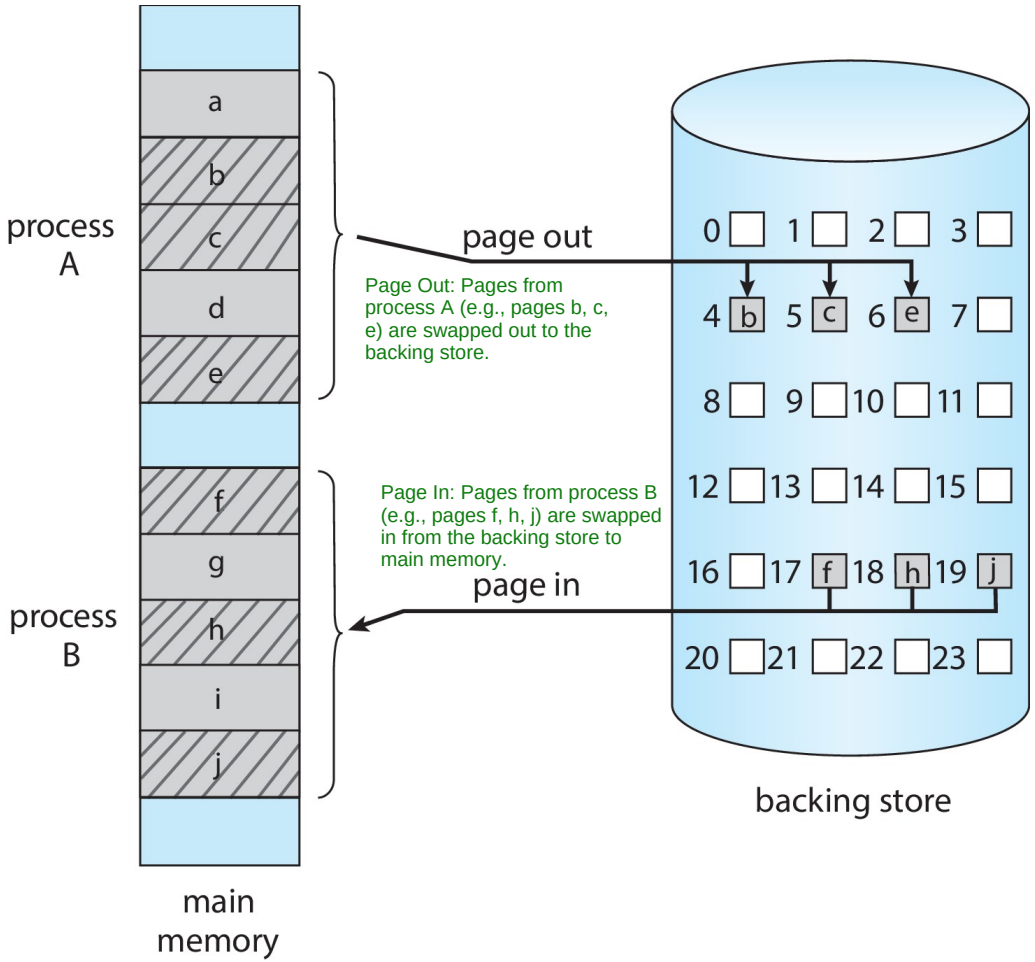
Swapping Not Typically Supported:
Mobile systems are often flash memory-based.
Flash memory has a small amount of space, a limited number of write cycles, and poor throughput between flash memory and the CPU on mobile platforms.

Alternative Methods to Free Memory:
iOS:
Asks apps to voluntarily relinquish allocated memory.
Read-only data is thrown out and reloaded from flash if needed.
Failure to free memory can result in app termination.
Android:
Terminates apps if free memory is low but first writes the application state to flash for a fast restart.
Both iOS and Android support paging as discussed previously.

Flash Memory Constraints: Small space, limited write cycles, and poor throughput make swapping less feasible on mobile systems.
iOS Memory Management: Asks apps to relinquish memory voluntarily, throws out read-only data, and may terminate apps if they fail to free memory.
Android Memory Management: Terminates apps if memory is low but saves the application state to flash for a fast restart.
Paging Support: Both iOS and Android support paging.

Swapping with Paging

Swapping with Paging:
Swapping can be combined with paging to manage memory more efficiently.
Instead of swapping entire processes, individual pages of a process can be swapped in and out of memory.



Backing Store: A secondary storage where pages are stored when swapped out.

Main Memory: Contains pages of processes A and B.

Example: The Intel 32 and 64-bit Architectures

- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

1. Dominant Industry Chips:
Intel CPUs are dominant in the industry, widely used in various computing devices.

2-bit Architecture (IA-32):
Pentium CPUs are 32-bit and are referred to as IA-32 architecture.
IA-32 architecture supports 32-bit addressing and operations.

64-bit Architecture (IA-64):
Current Intel CPUs are 64-bit and are referred to as IA-64 architecture.
IA-64 architecture supports 64-bit addressing and operations, allowing for larger address spaces and more memory.

Variations in Chips:
There are many variations in Intel chips, but the main ideas and principles of the architectures are covered here.

Key Points:
Dominant Industry Chips: Intel CPUs are widely used and dominant in the industry.
IA-32 Architecture: 32-bit architecture used in Pentium CPUs.
IA-64 Architecture: 64-bit architecture used in current Intel CPUs.
Variations in Chips: Many variations exist, but the main ideas are consistent.

Example: The Intel IA-32 Architecture

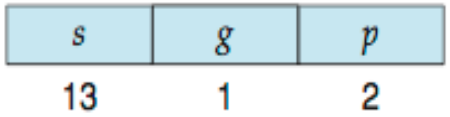
- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))

Segmentation and Paging:
The Intel IA-32 architecture supports both segmentation and segmentation with paging.
This allows for flexible and efficient memory management.
2. Segment Size and Count:
Each segment can be up to 4 GB in size.
A process can have up to 16,000 segments.
3. Segment Partitions:
Segments are divided into two partitions:
Local Descriptor Table (LDT): Contains up to 8,000 segments that are private to the process.
Global Descriptor Table (GDT): Contains up to 8,000 segments that are shared among all processes.

Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses

Logical Address: The logical address is divided into three parts: s (segment), g (global bit), and p (page).
Segmentation Unit: Converts the logical address to a linear address.
Paging Unit: Converts the linear address to a physical address in main memory.



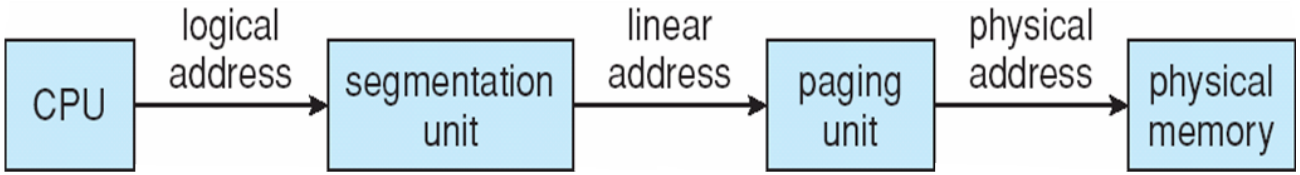
Logical Address Generation:
The CPU generates a logical address.
The logical address includes a selector that is given to the segmentation unit.
2. Segmentation Unit:
The segmentation unit uses the selector to produce a linear address.
The linear address is a combination of the segment (s), the global bit (g), and the page (p).
3. Paging Unit:
The linear address is given to the paging unit.
The paging unit generates the physical address in main memory.
The paging units form the equivalent of the Memory Management Unit (MMU).
Page sizes can be 4 KB or 4 MB.

  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32

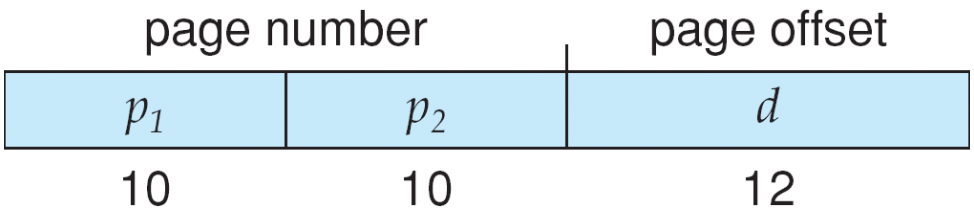
Paging Unit: Converts the linear address to a physical address.

CPU: Generates the logical address.



Segmentation Unit: Converts the logical address to a linear address.

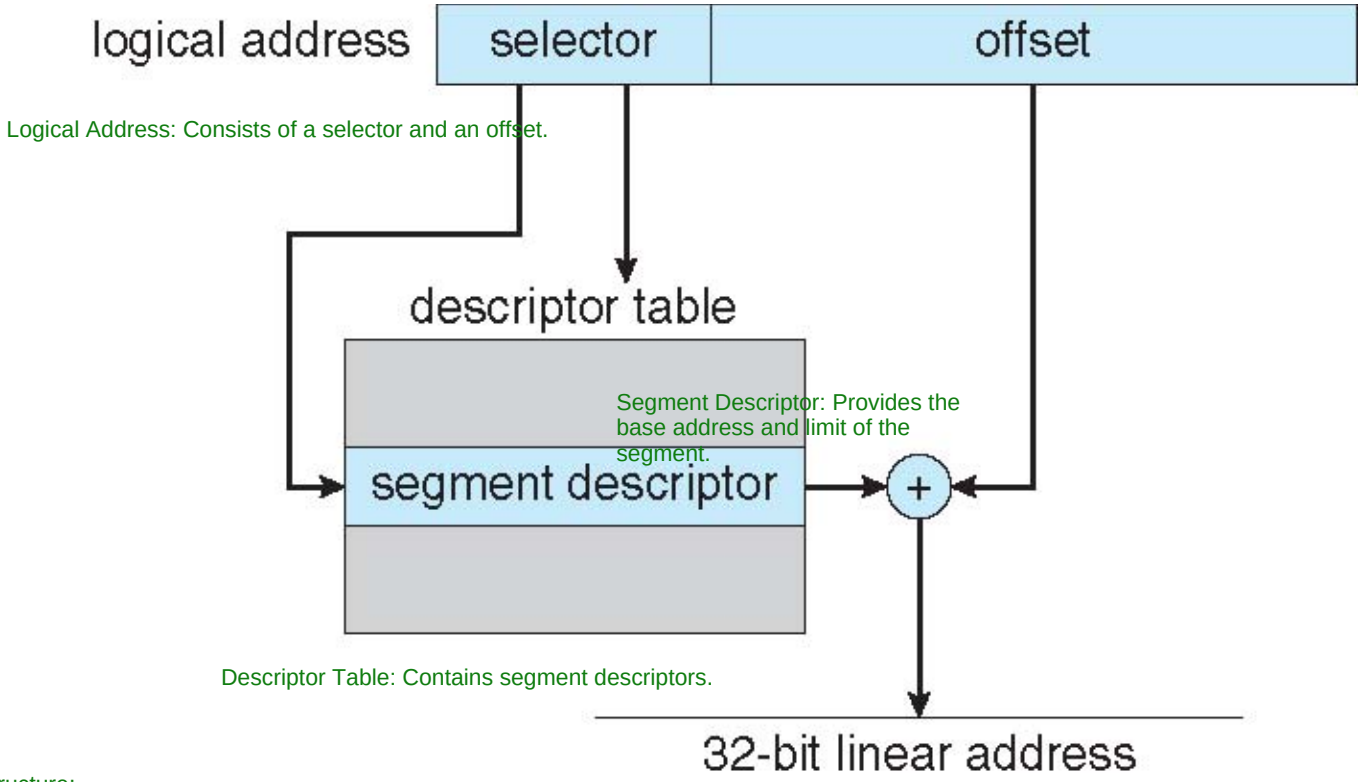
Physical Memory: The physical address is used to access the memory location.



1. Address Translation Process:
The process of translating a logical address generated by the CPU to a physical address in main memory involves several steps.
2. Steps in Address Translation:
Logical Address: Generated by the CPU.
Segmentation Unit: Converts the logical address to a linear address.
Paging Unit: Converts the linear address to a physical address.
Physical Memory: The physical address is used to access the actual memory location.
3. Address Division:
The logical address is divided into multiple parts:
Page Number (p1): 10 bits.
Page Number (p2): 10 bits.
Page Offset (d): 12 bits.

Intel IA-32 Segmentation

Selector: Used to index into the descriptor table.



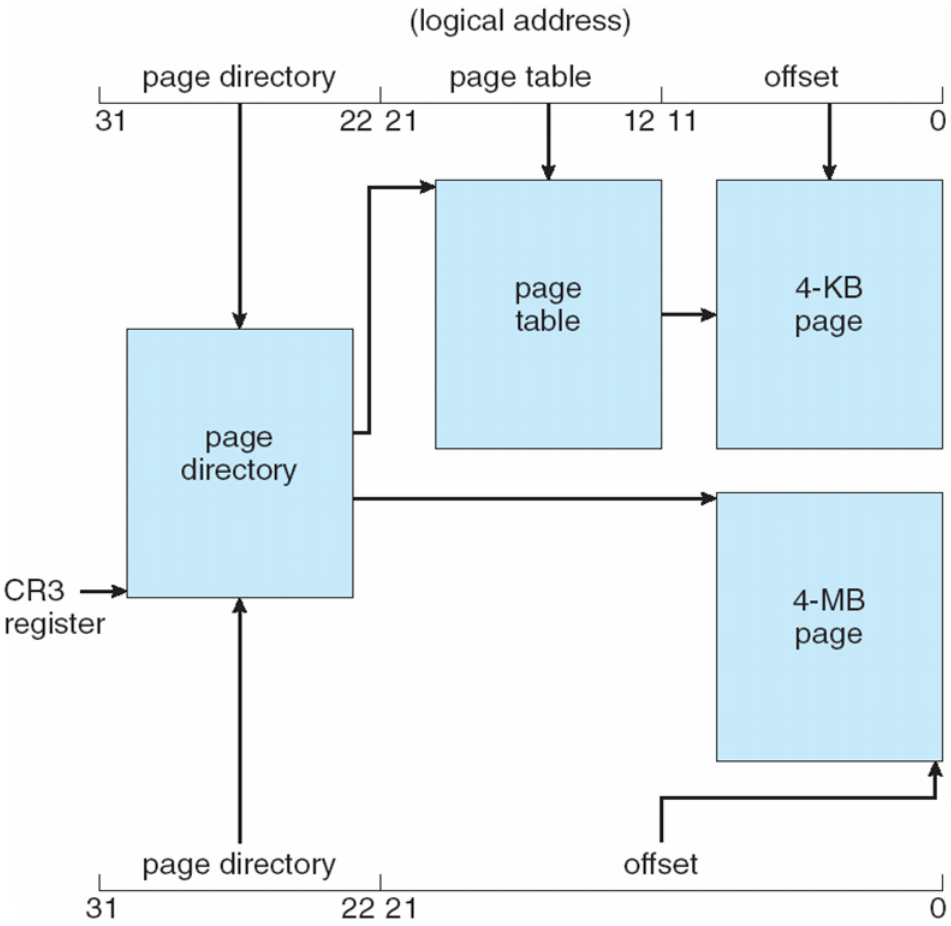
Logical Address: Consists of a selector and an offset.

Segment Descriptor: Provides the base address and limit of the segment.

Descriptor Table: Contains segment descriptors.

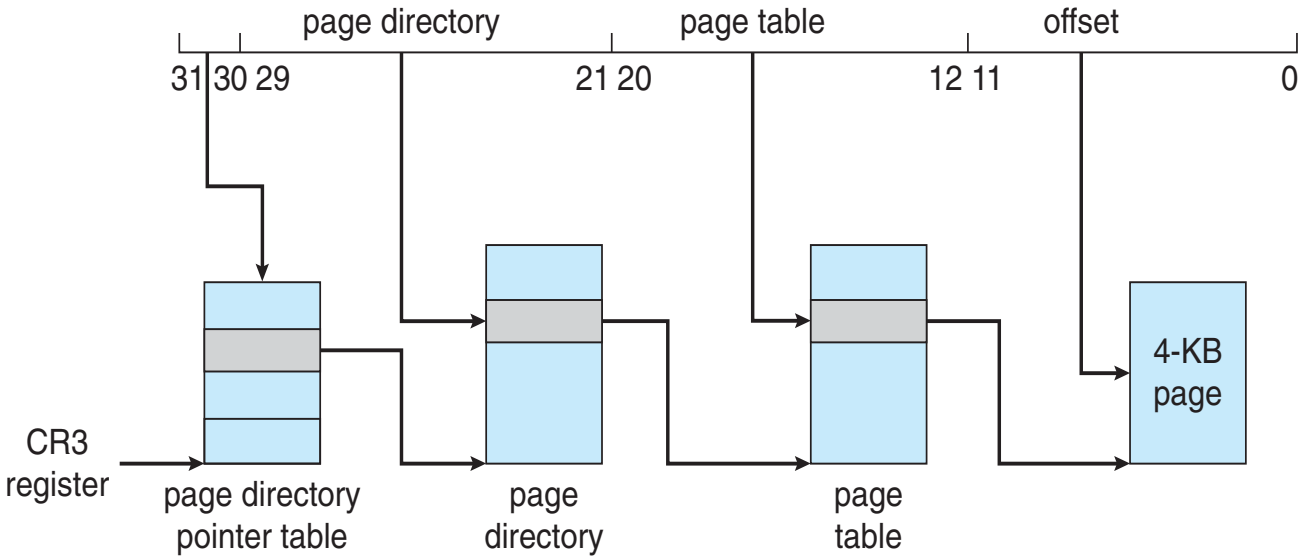
Logical Address Structure:
The logical address in the IA-32 architecture consists of two parts:
Selector: Identifies the segment.
Offset: Specifies the location within the segment.
2. Segmentation Process:
Selector: The selector is used to index into the descriptor table.
Descriptor Table: Contains segment descriptors that provide the base address and limit of the segment.
Segment Descriptor: The segment descriptor is used to calculate the linear address by adding the base address to the offset.
3. Linear Address:
The result of the segmentation process is a 32-bit linear address.

Intel IA-32 Paging Architecture



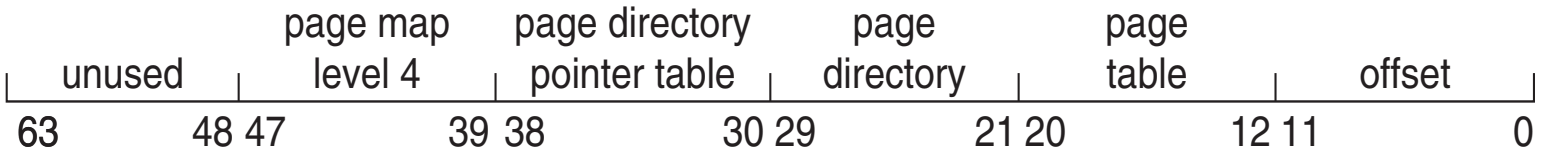
Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



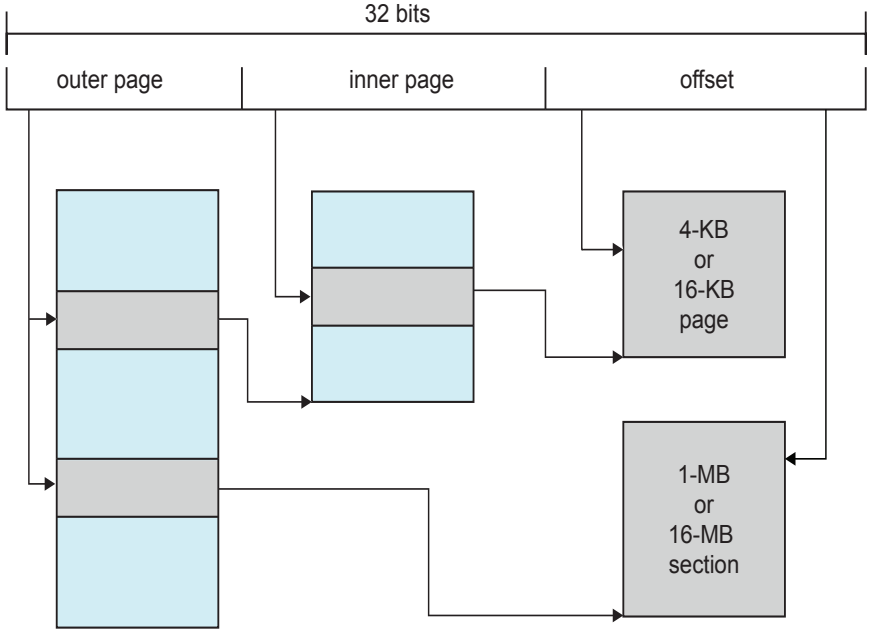
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



End of Chapter 9