Chapter 7. Main Memory

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. (2018). *Operating System Concepts* (10th ed.). Wiley.

Contents

1. Background

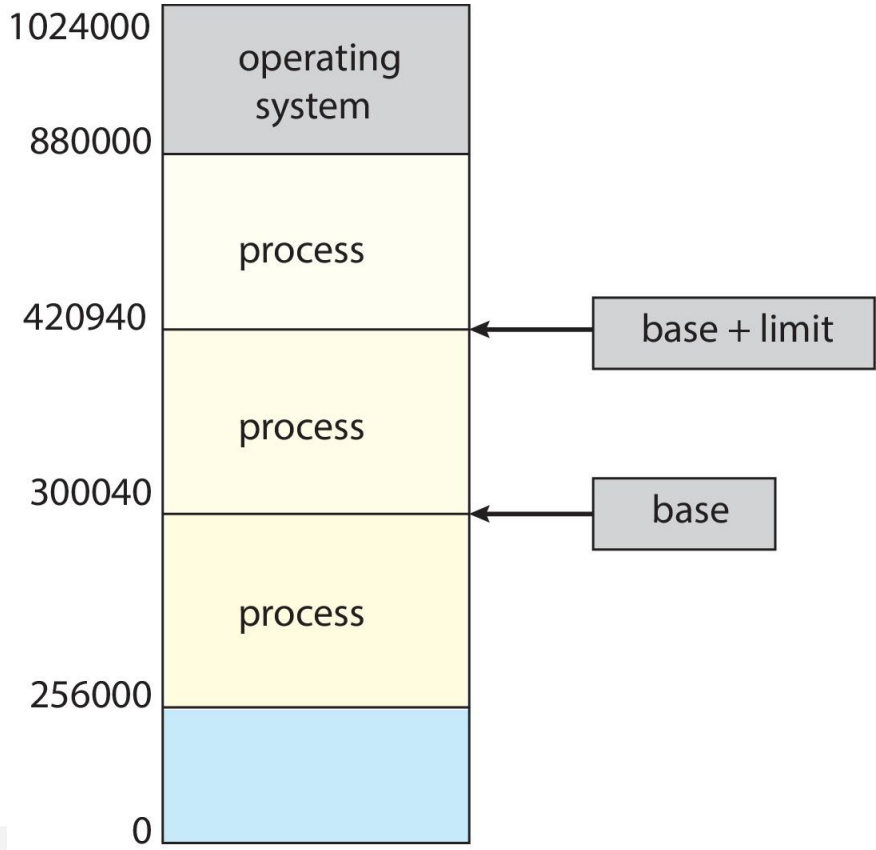
- Program must be brought (from disk (Secondary Memory)) into main memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
  - address + read request, or
  - address + data and write request

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

1. Background

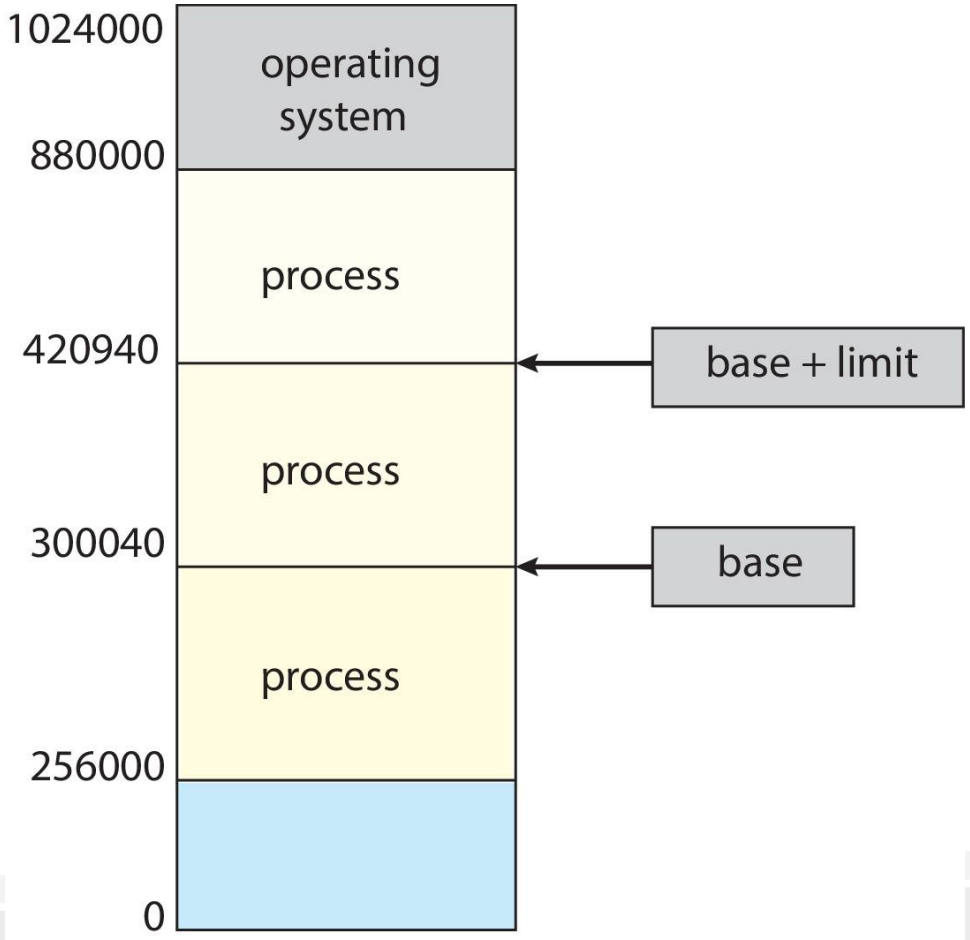
Protection: Base and Limit Registers

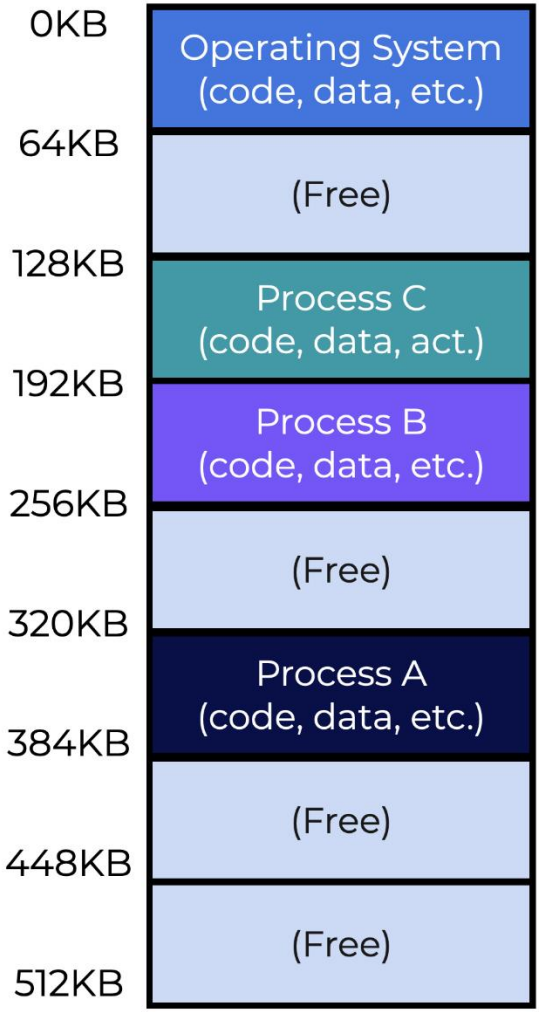
- Each process has a separate memory space.
- Separate per-process memory space protects the processes from each other
- Need to ensure that a process can access only those addresses in **its memory space** (also called **address space**)
- We can provide this protection by using a pair of **base** and **limit** registers
  - A pair of base and limit registers define the logical address space of a process
  - The base register holds the smallest legal physical memory address
  - The limit register specifies the size of the range.
- In this example, base register holds 300040 and the limit register holds 120900
  - So, the program can legally access all addresses from 300040 through 420939 (inclusive).



1. Background

Protection: Base and Limit Registers

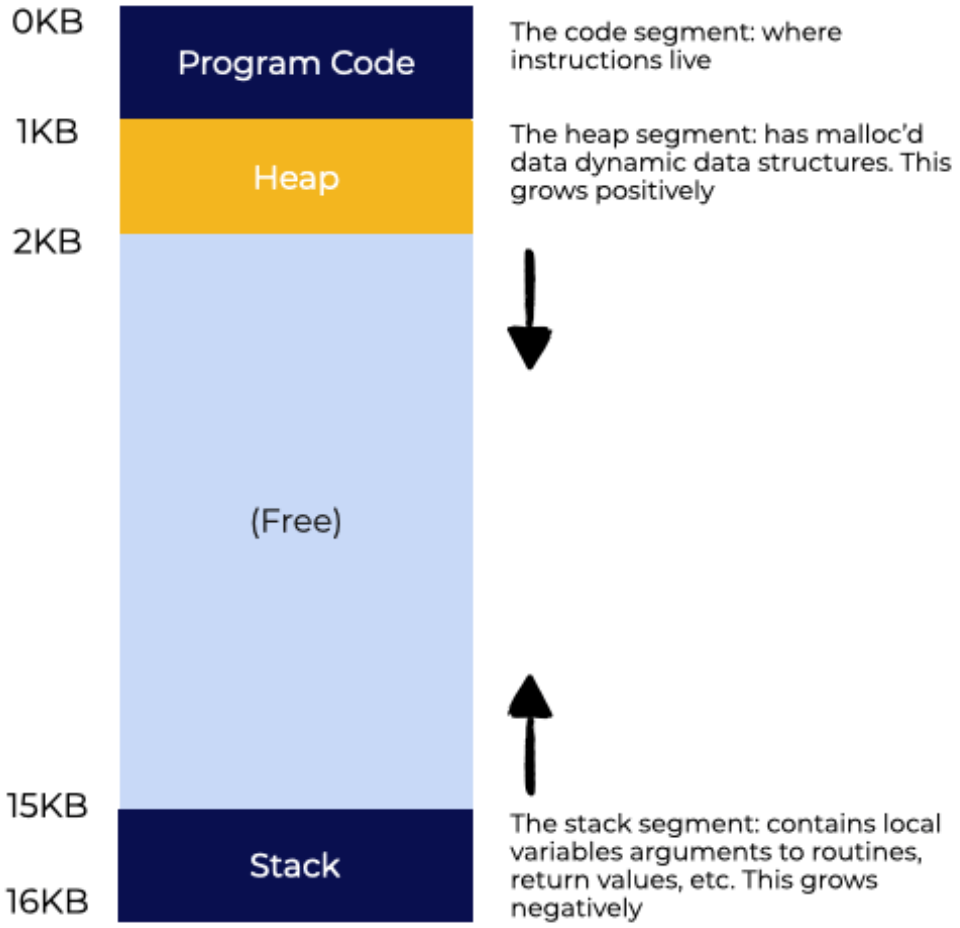




1. Background

A Process's Address Space

- A process' **address space** contains all of the program's memory state, including the code.
- The program uses a **stack** to monitor its position in the function call chain, allocate variables, and pass parameters.
- The **heap** is used for dynamically-allocated, user-managed memory.

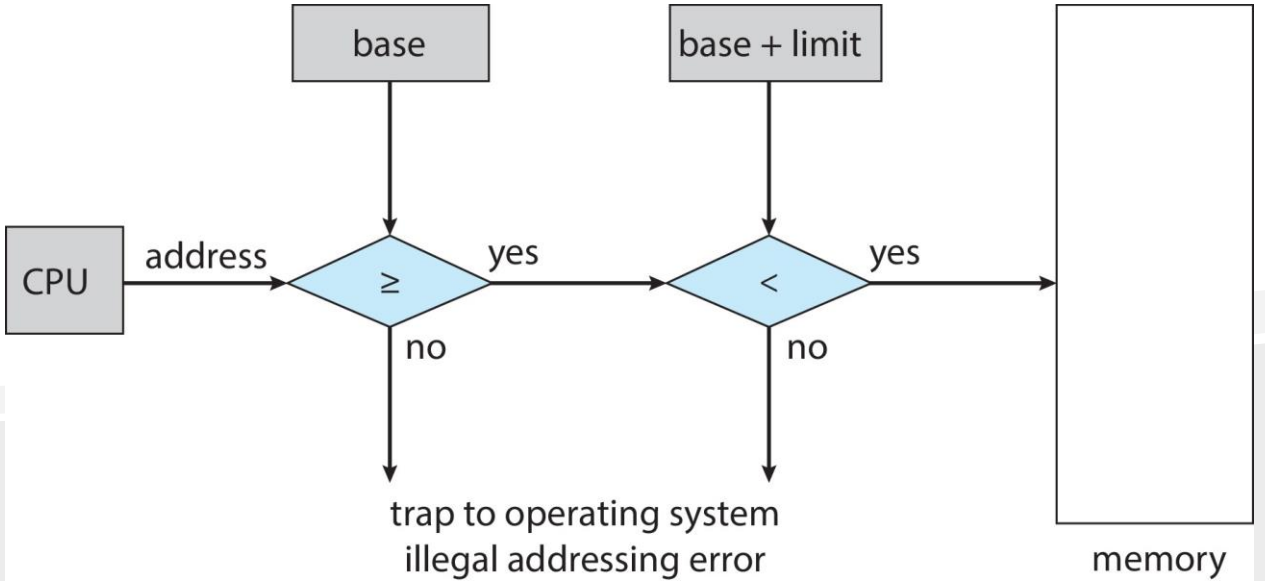


1. Background

Hardware Address Protection

- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user.

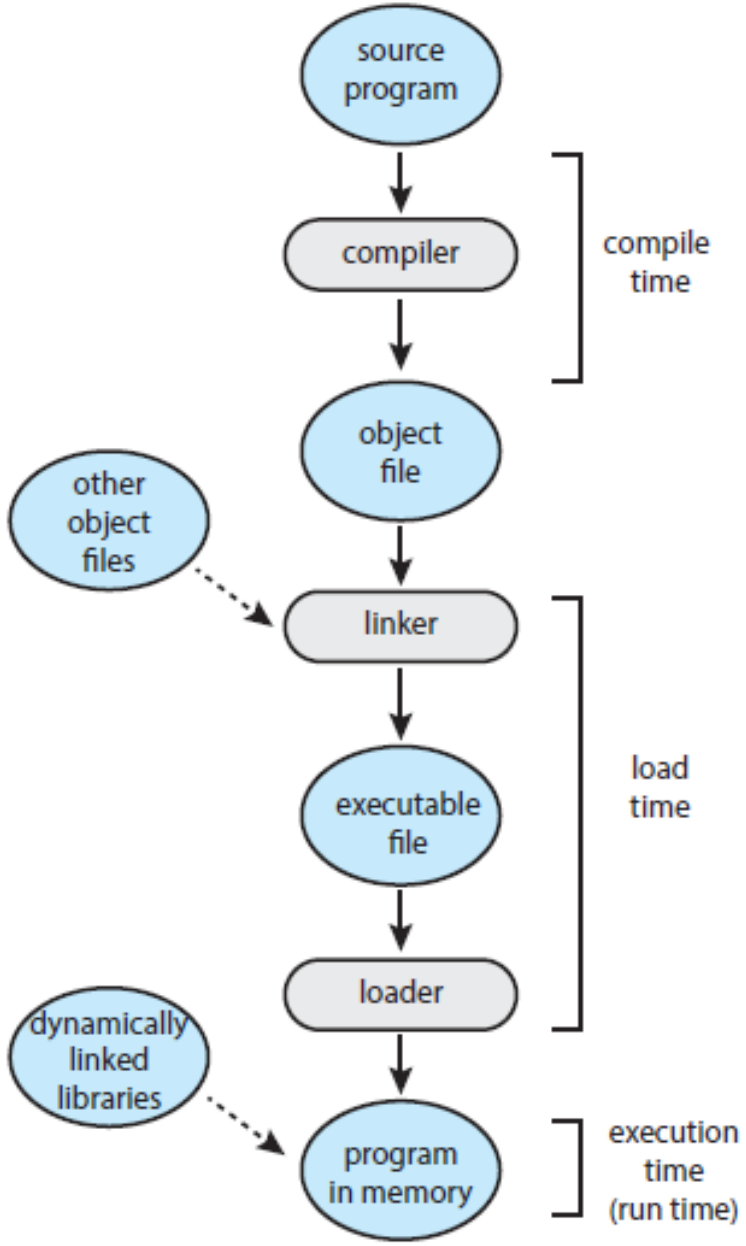
This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.



1. Background

Address Binding

- Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000.
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
    - Such as the variable `count`
  - Compiled code addresses bind to relocatable addresses
    - Example, "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - Example, 74014
  - Each binding maps one address space to another.



1. Background

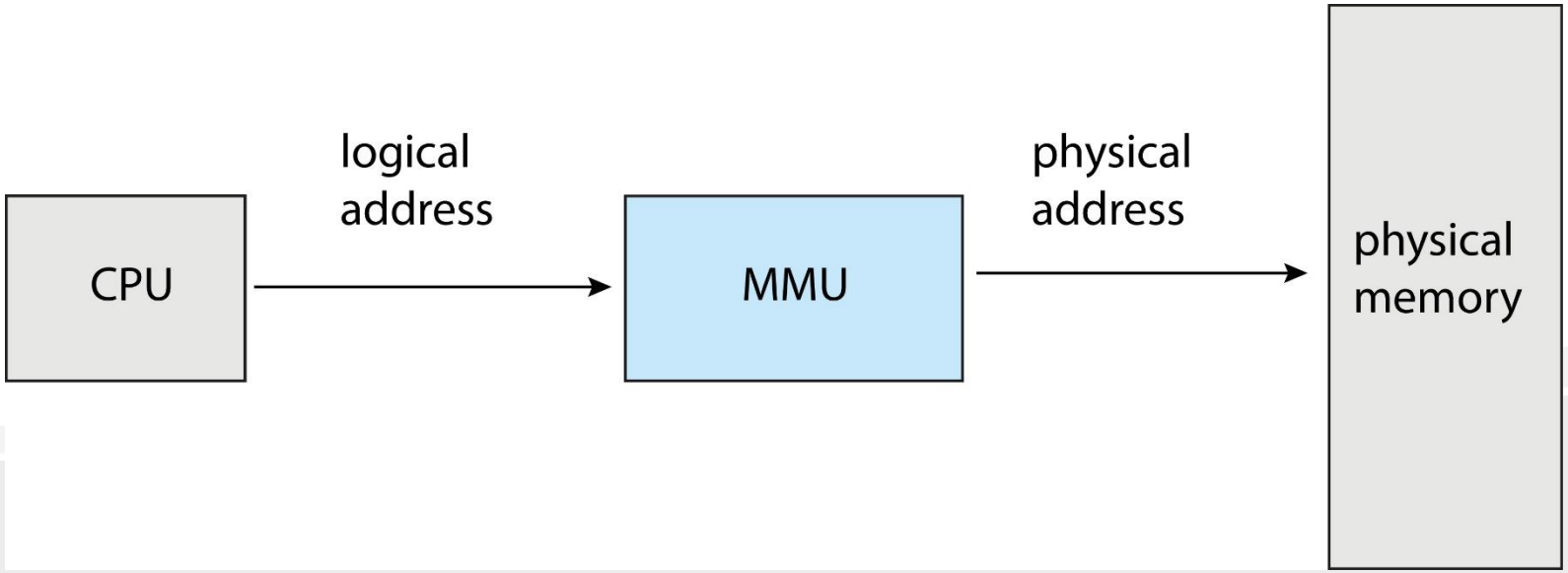
Address Binding

- Address binding of instructions and data to memory addresses can happen at three different stages
  - Compile time:
    - If you know at compile time, then **absolute code** can be generated. If, at some later time, the starting location changes, then it will be necessary to recompile this code.
  - Load time:
    - If it is not known at compile time, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
  - Execution time:
    - If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most operating systems use this method.

1. Background

Logical Versus Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
  - Logical address – generated by the CPU, also referred to as virtual address
  - Physical address – address seen by the memory unit —that is, the one loaded into the memory-address register of the memory



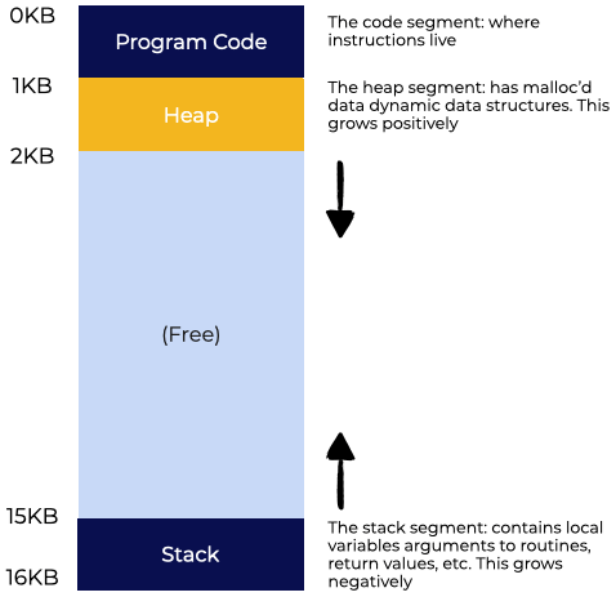
1. Background

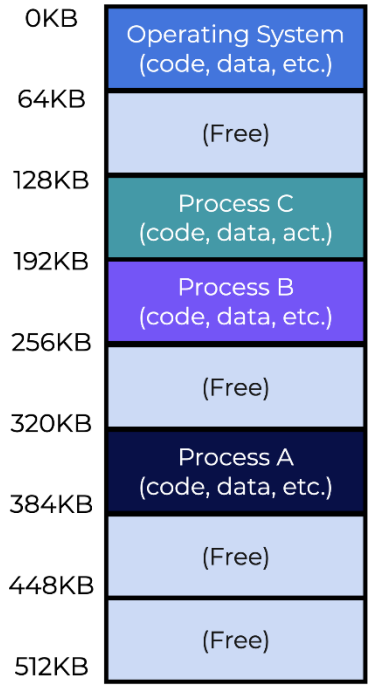
Logical Versus Physical Address Space

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- The execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a virtual address.
- Logical address space is the set of all logical addresses generated by a program.
  - The set of all physical addresses corresponding to these logical addresses is a physical address space.

1. Background

Logical Versus Physical Address Space



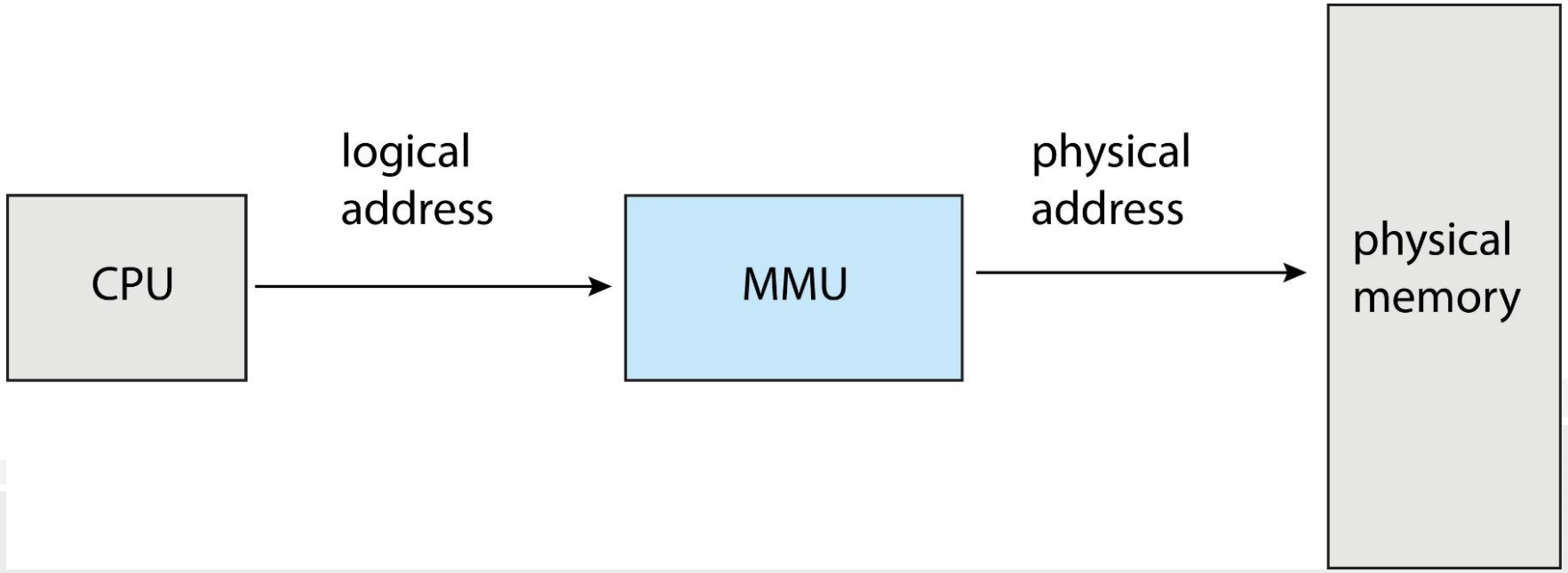


- Process A thinks that its program code resides at the beginning of the memory block (0kb). This is actually a virtual address.
- In real memory, each process is stored in various blocks of memory. Process A actually begins at address 320kb.
- The operating system and hardware work together to ensure that the process maps to the correct address (320kb) in physical memory from the virtual address

1. Background

Memory-Management Unit (MMU)

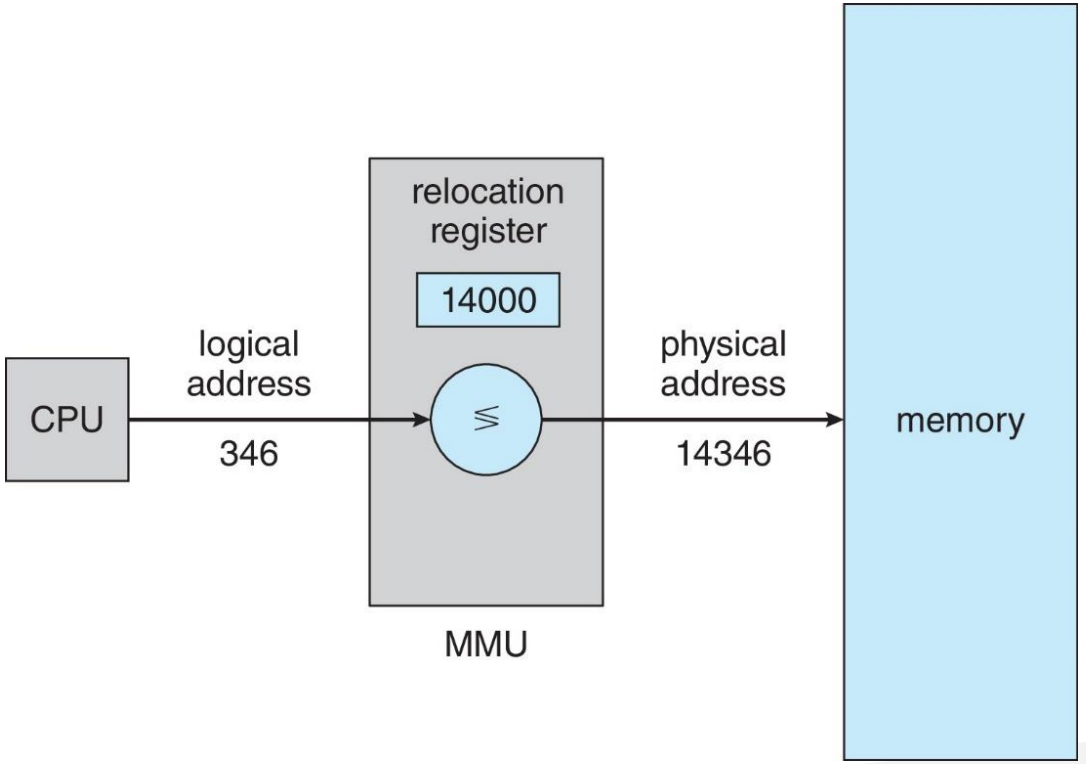
- MMU is hardware device that at run time maps virtual to physical address.
  - Many methods are possible to map from virtual to physical address.



1. Background

Memory-Management Unit (MMU)

- Consider a simple schema, 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.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
  - Execution-time binding occurs when reference is made to location in memory.
  - Logical address bound to physical addresses.



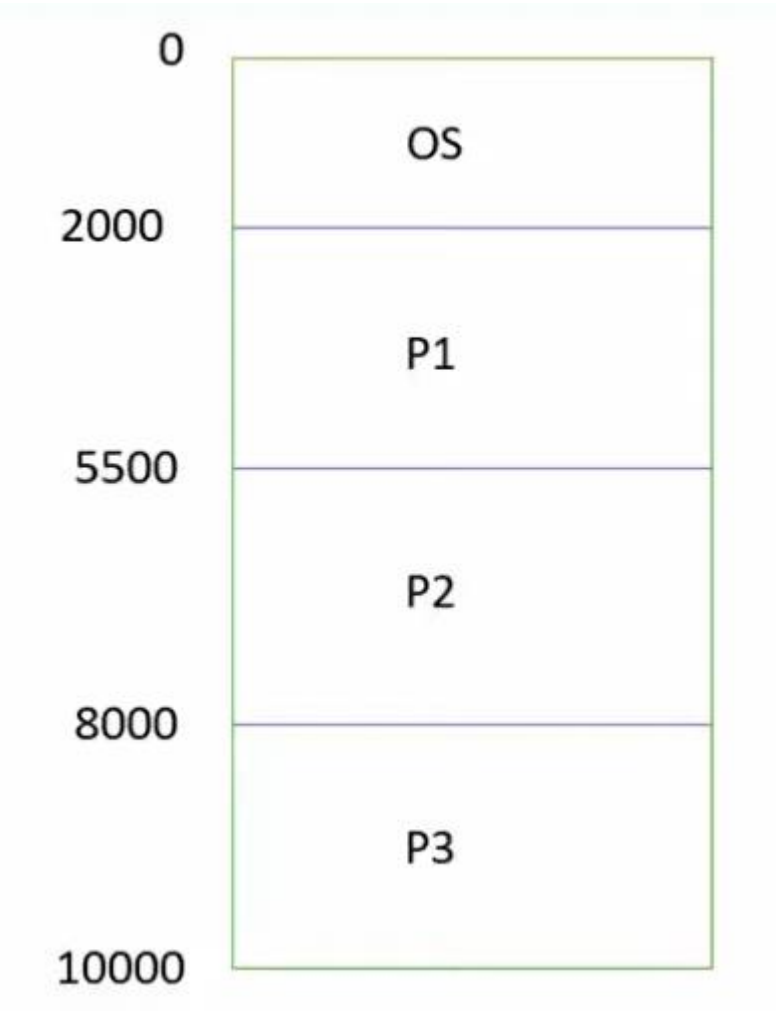
1. Background

Dynamic Loading

- It has been necessary for the entire program and all data of a process to be in physical memory for the process to execute.
  - The size of a process has thus been limited to the size of physical memory.
- To obtain better memory-space utilization, we can use dynamic loading.
  - A routine is not loaded until it is called. All routines are kept on disk in a relocatable load format.
  - The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

2. Contiguous Memory Allocation

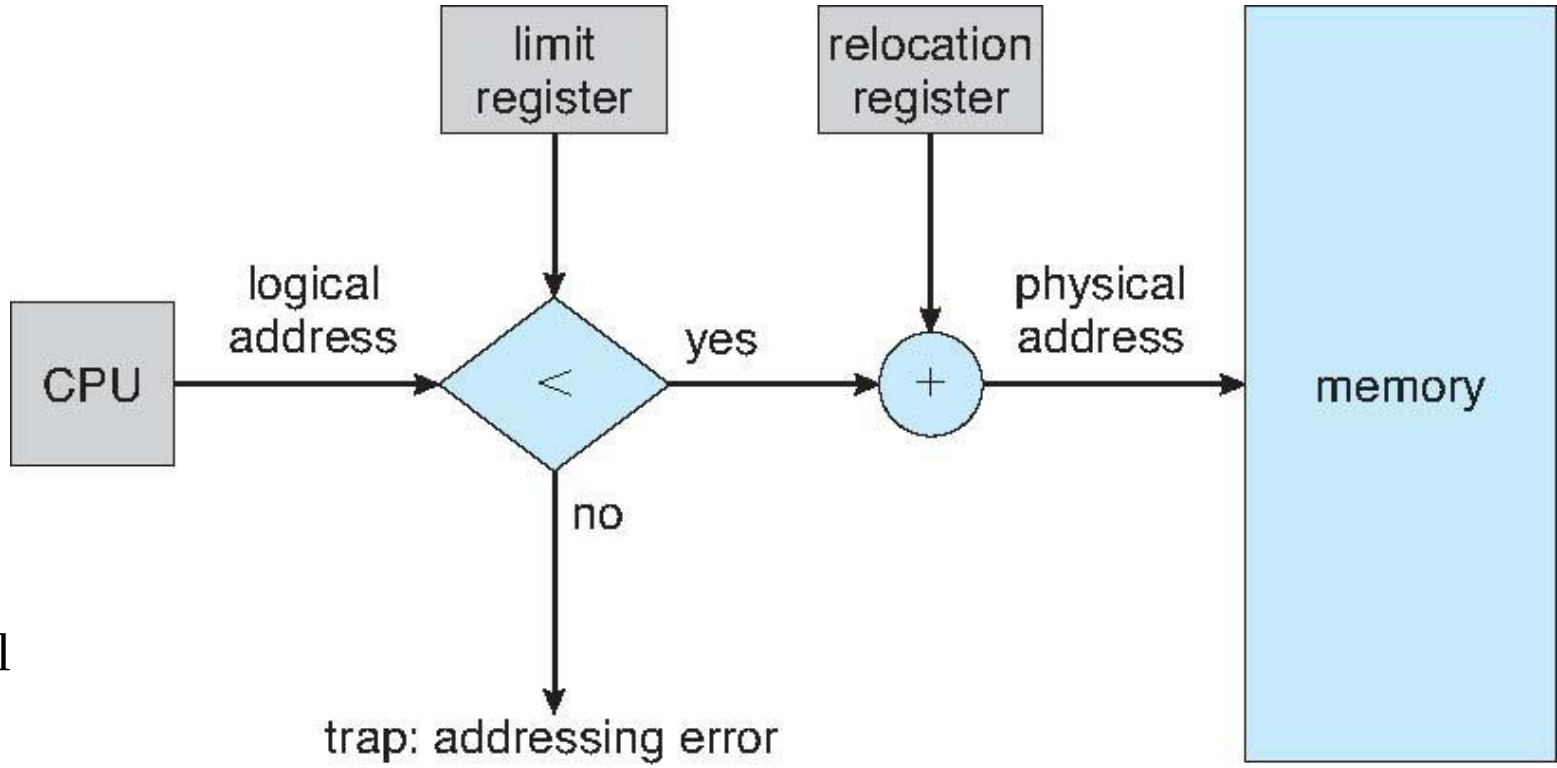
- Main memory must support both OS and user processes.
- We need to allocate main memory in the most efficient way possible. This section explains one early method, **contiguous memory allocation**.
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.



2. Contiguous Memory Allocation

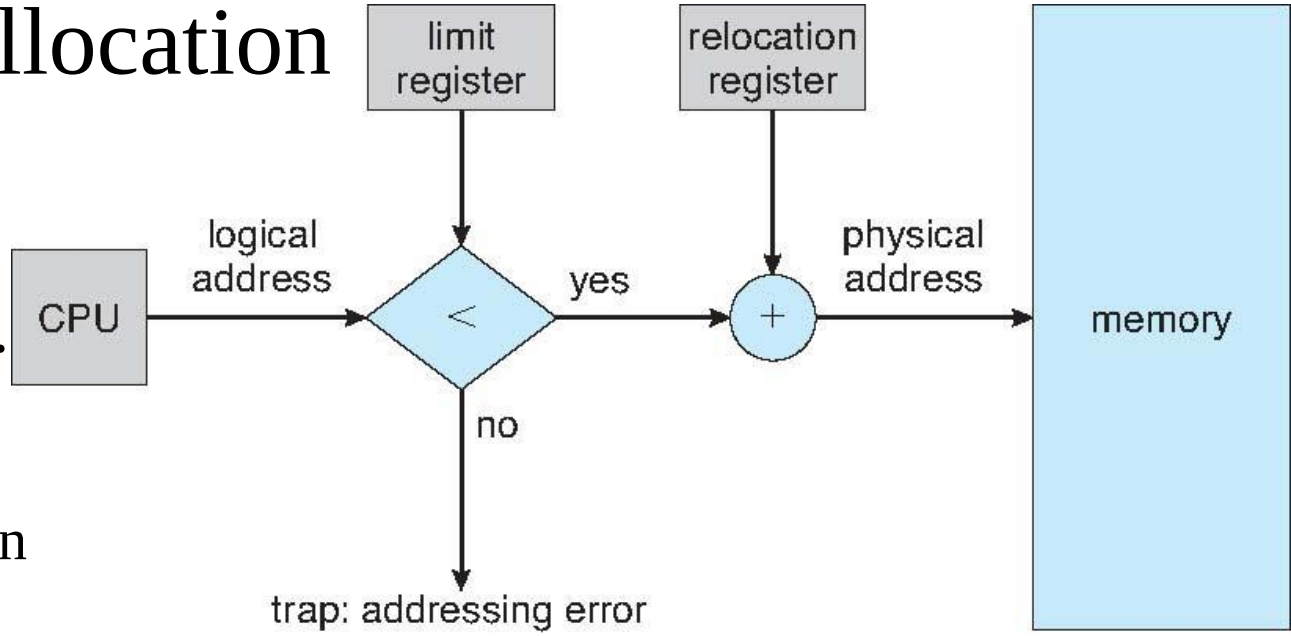
Memory Protection

- We can prevent a process from accessing memory that it does not own.
- Relocation register used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address dynamically by adding the value in the relocation register

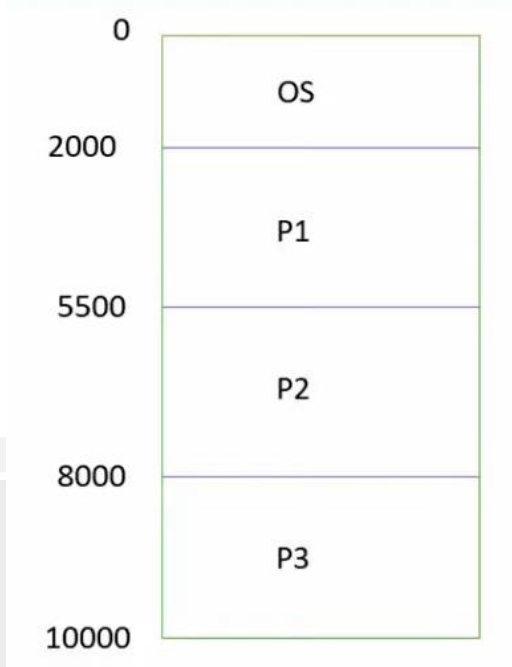


2. Contiguous Memory Allocation

Memory Protection



- P_1 has address space from 2000 to 5499.
  - Relocation register contains value of 2000
  - Limit register contains value of range. So, in this case, it will contain 5500-2000= 3500
  - Let's assume CPU wants to access data at 500 logical address.
    - MMU will check if 500 < 3500.
    - Answer is yes. So, 500 will be added with 2000. Then the logical address 500 will be mapped towards the physical address 2500. 2500 is within the physical address space of P_1.
    - So, protection is ensured

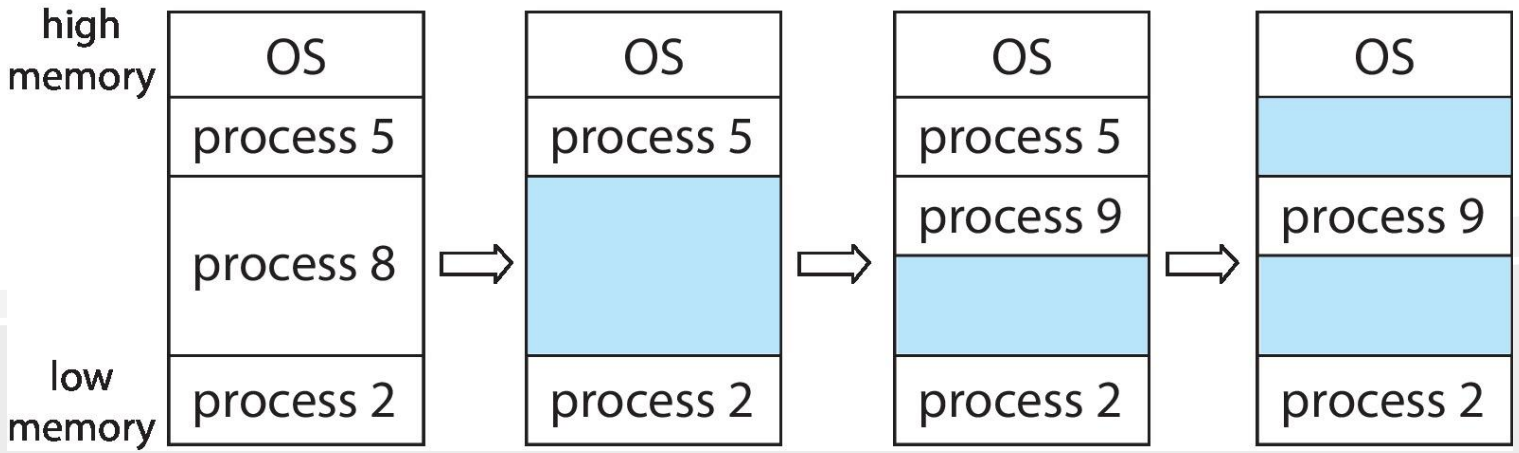


2. Contiguous Memory Allocation

Memory Allocation

Variable Partition

- One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process. → **Variable partition**
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about: a) allocated partitions b) free partitions (hole)



2. Contiguous Memory Allocation

Variable Partition

Dynamic Storage Allocation Problem

- When a new process arrives and multiple memory blocks are available then which block should be allocated to a process? There are three methods – **first fit**, **best fit** and **worst fit**.
  - 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
- First-fit and best-fit are better than worst-fit in terms of speed and storage utilization.

2. Contiguous Memory Allocation

Fragmentation – External Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**.
- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe.

2. Contiguous Memory Allocation

Fragmentation – Internal Fragmentation

- Memory fragmentation can be internal as well as external.
- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. !!!
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation** — unused memory that is internal to a partition.

2. Contiguous Memory Allocation

Fragmentation - Compaction

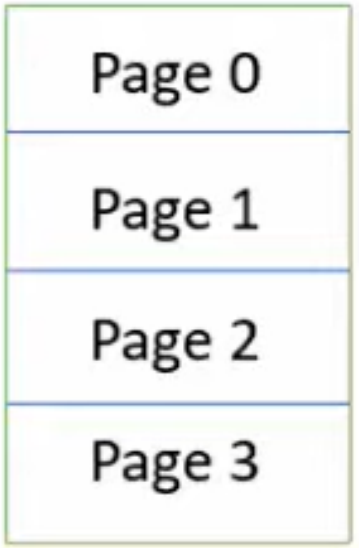
- One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- Compaction is not always possible, however.
  - If relocation is static and is done at assembly or load time, compaction cannot be done.
  - It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost.
  - The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of processes to be **noncontiguous**, thus allowing a process to be allocated physical memory wherever such memory is available.
  - This is the strategy used in **paging**, the most common memory-management technique for computer systems.

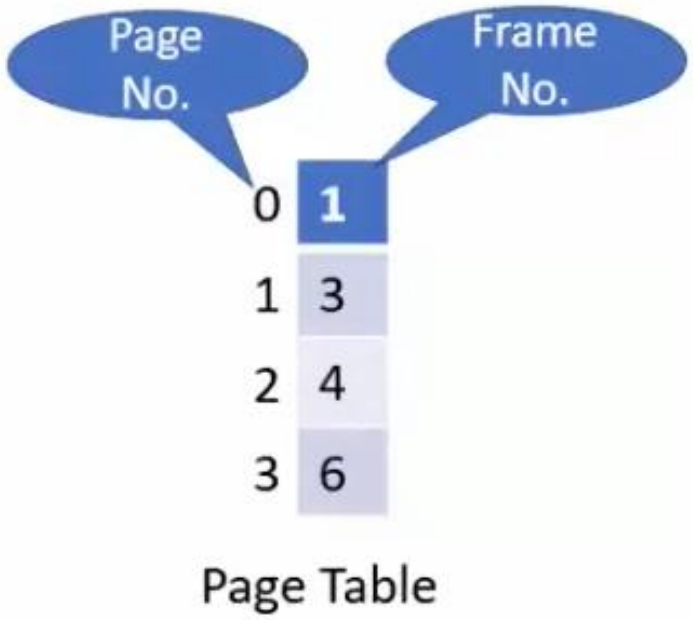
3. Paging

- Paging: A memory management scheme that permits a physical address space of a process can be noncontiguous
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames.**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**.
- Keep track of all free frames
- To run a program of size *N* pages, need to find *N* free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have internal fragmentation

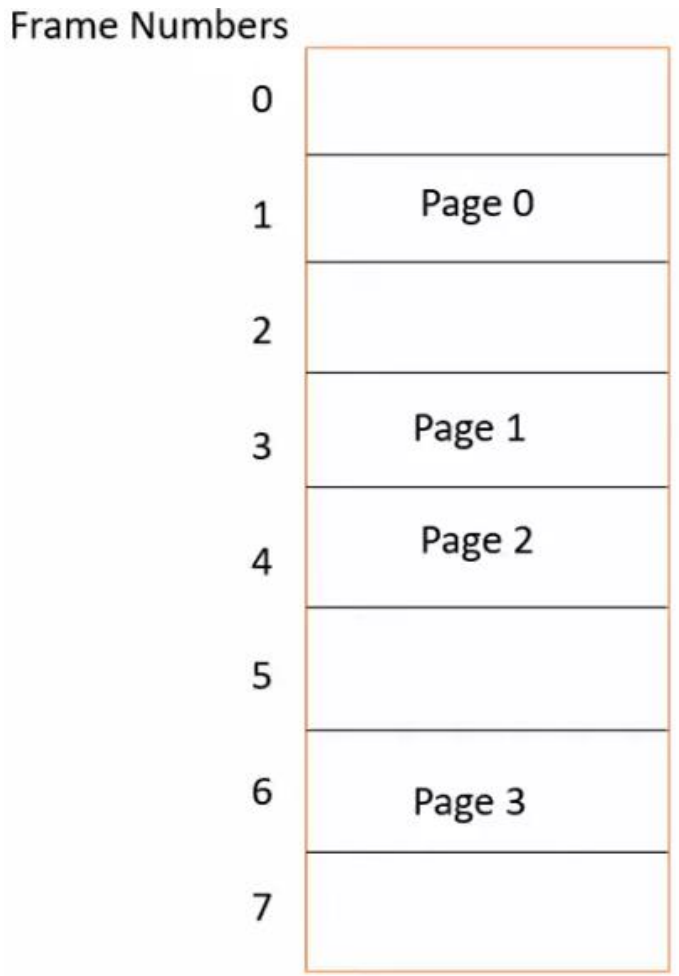
3. Paging

Page Size = Frame Size
Frame size = 512bytes





Page Table

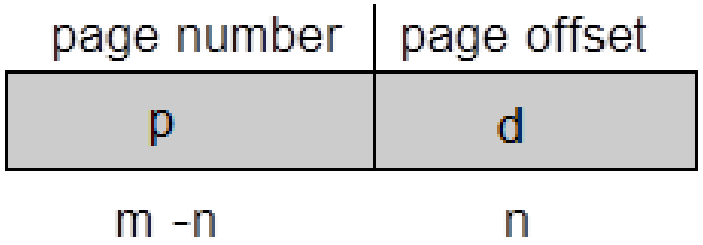


Main Memory

3. Paging
Address Translation Scheme

If logical address space size = 2^m
Page size = 2^n
Then
Page number = higher order (m-n) bits
Page offset = n low-order bits

- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit



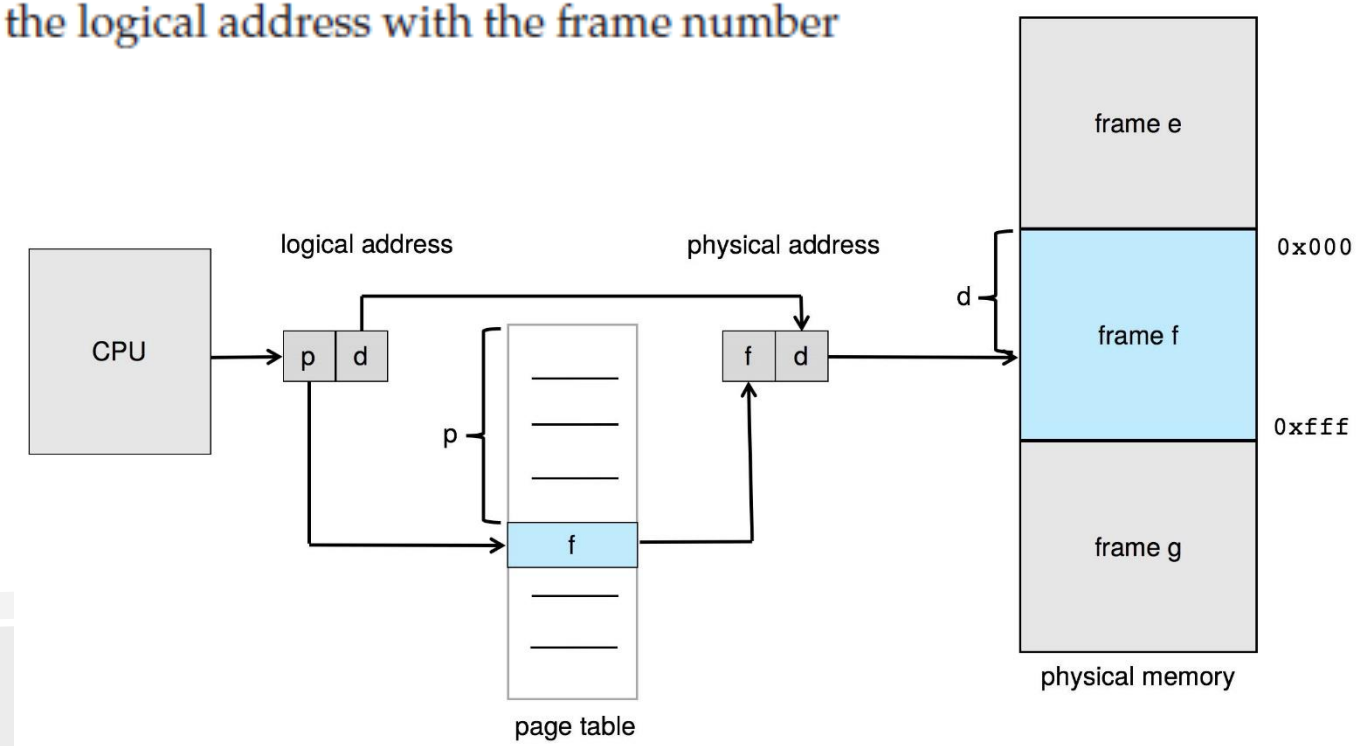
  - For given logical address space 2^m and page size 2^n

3. Paging

Paging Hardware

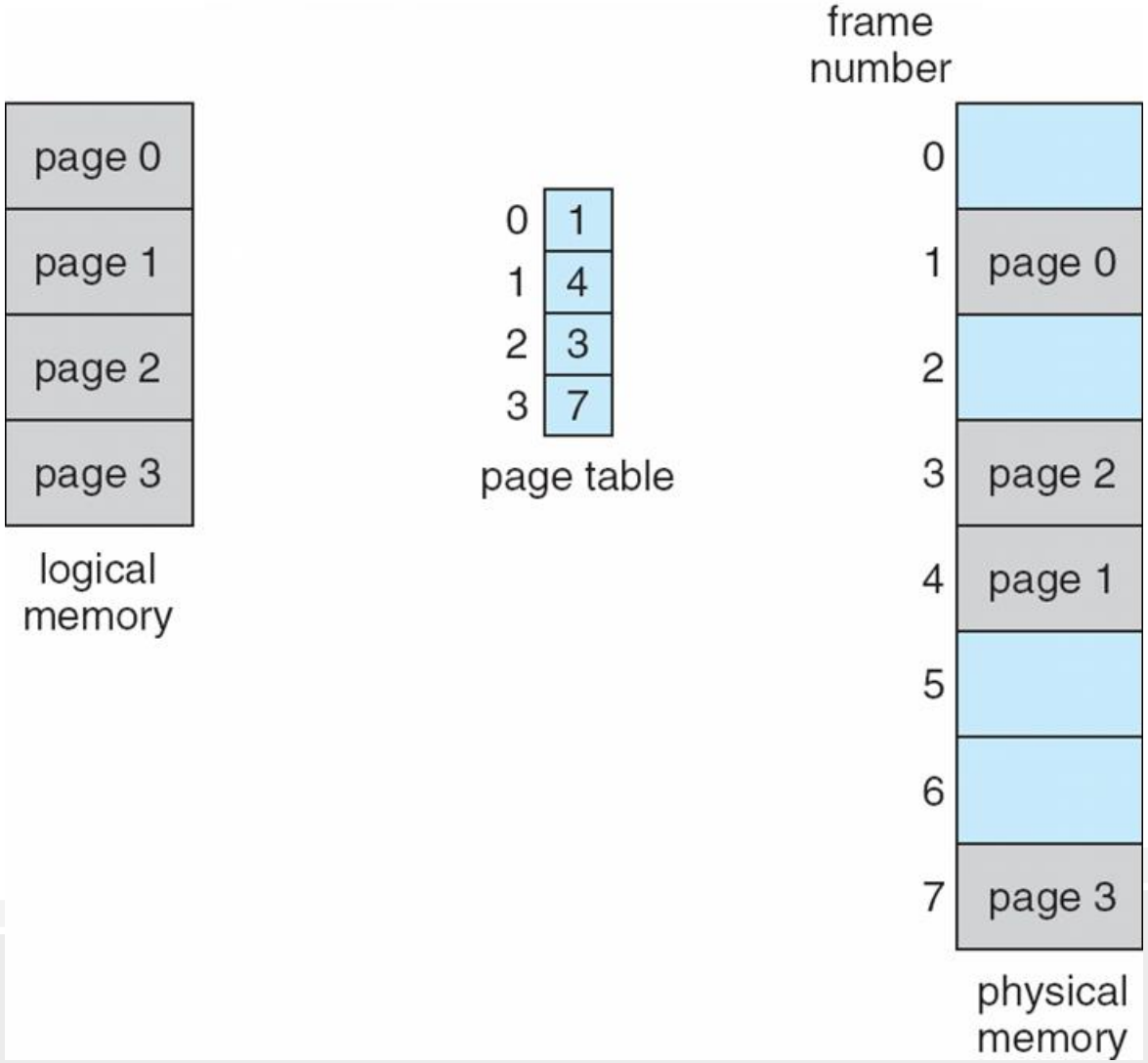
The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number *p* and use it as an index into the page table.
2. Extract the corresponding frame number *f* from the page table.
3. Replace the page number *p* in the logical address with the frame number *f*.



3. Paging

Paging Model of Logical and Physical Memory



3. Paging

Address Translation

Translate **logical address** into **physical address** using paging memory allocation

Notation: Page (P), frame (f), d (offset), ps (page size), logical address (LA), physical address (PA)

Procedure:

P = LA div ps

d = LA mod ps

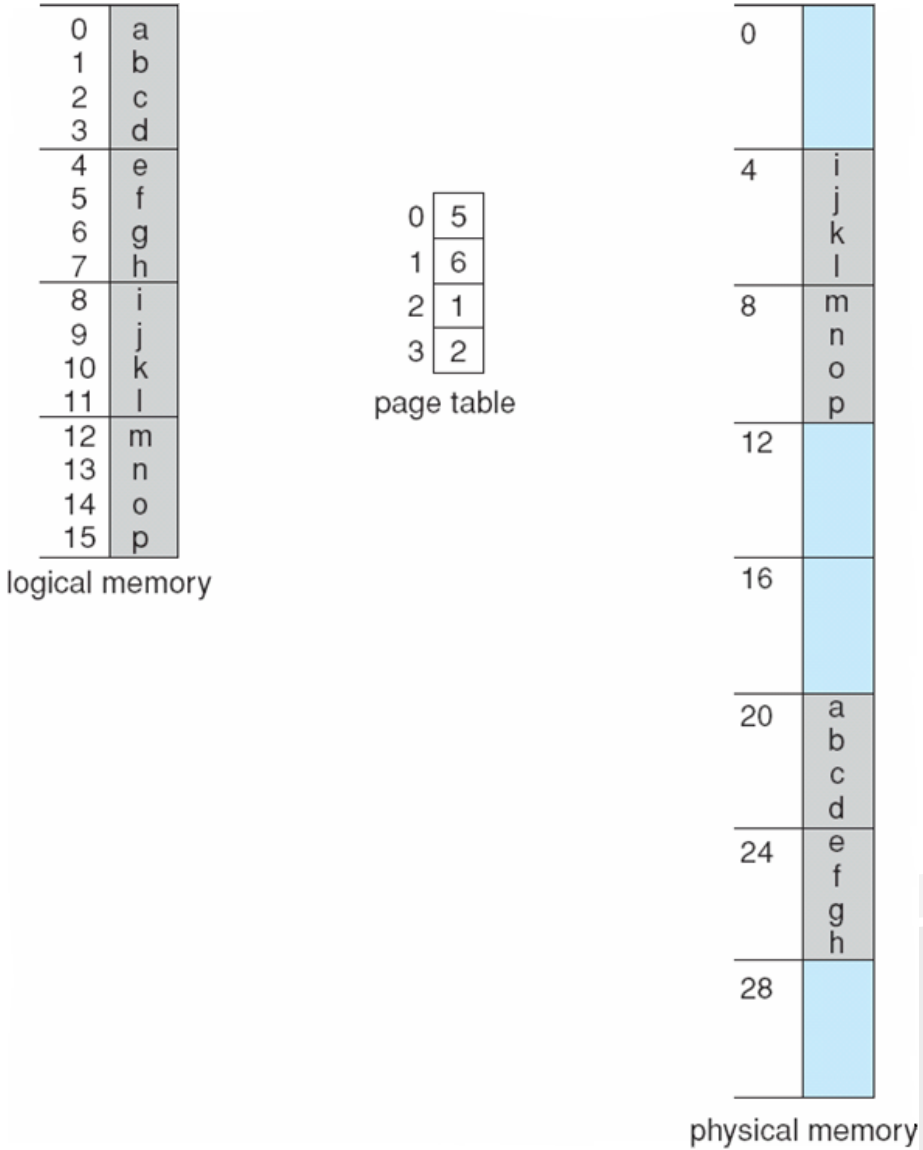
lookup page table to **find f** from p

PA = f*ps + d

3. Paging
Address Translation

32-byte memory and 4-byte pages

- Logical address 3 => physical address ???
  - (page 0, offset 3) maps to physical address 23 [= (5×4) + 3]
- Logical address 4 => physical address ???
  - physical address 24
- Logical address 13 => physical address ???
  - maps to physical address 9



3. Paging

Calculating Internal Fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- **Internal fragmentation of 2,048 - 1,086 = 962 bytes**
- Worst case fragmentation = 1 frame size – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

3. Paging

Hardware Support

- Page tables are **per-process** data structures.
  - A pointer to the page table is stored with the other register values (like the instruction pointer) in the process control block of each process.
  - When the CPU scheduler selects a process for execution, it must reload the user registers and the appropriate hardware page-table values from the stored user page table.
- The hardware implementation of the page table can be done in several ways.
  - In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers.
    - Makes the page-address translation very efficient. However, this approach increases context-switch time, as each one of these registers must be exchanged during a context switch.
    - The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary CPUs, however, support much larger page tables (for example, 220 entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

3. Paging

Translation Look-Aside Buffer

- Storing the page table in main memory can yield faster context switches, it may also result in slower memory access times.
- With this scheme, two memory accesses are needed to access data (one for the page-table entry and one for the actual data)
  - 1. Suppose we want to access location *i*. We must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires one memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address.
  - 2. We can then access the desired place in memory.

3. Paging

Translation Look-Aside Buffer

- The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a **translation look-aside buffer (TLB)**.
- The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.
  - When the associative memory is presented with an item, **the item is compared with all keys simultaneously**. If the item is found, the corresponding value field is returned.
  - The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty.
  - To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size.

3. Paging
Translation Look-Aside Buffer

- The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB.
  - If the page number is found, its frame number is immediately available and is used to access memory.
  - If the page number is not in the TLB (known as a **TLB miss**), address translation proceeds steps (as the paging model). When the frame number is obtained, we can use it to access memory. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

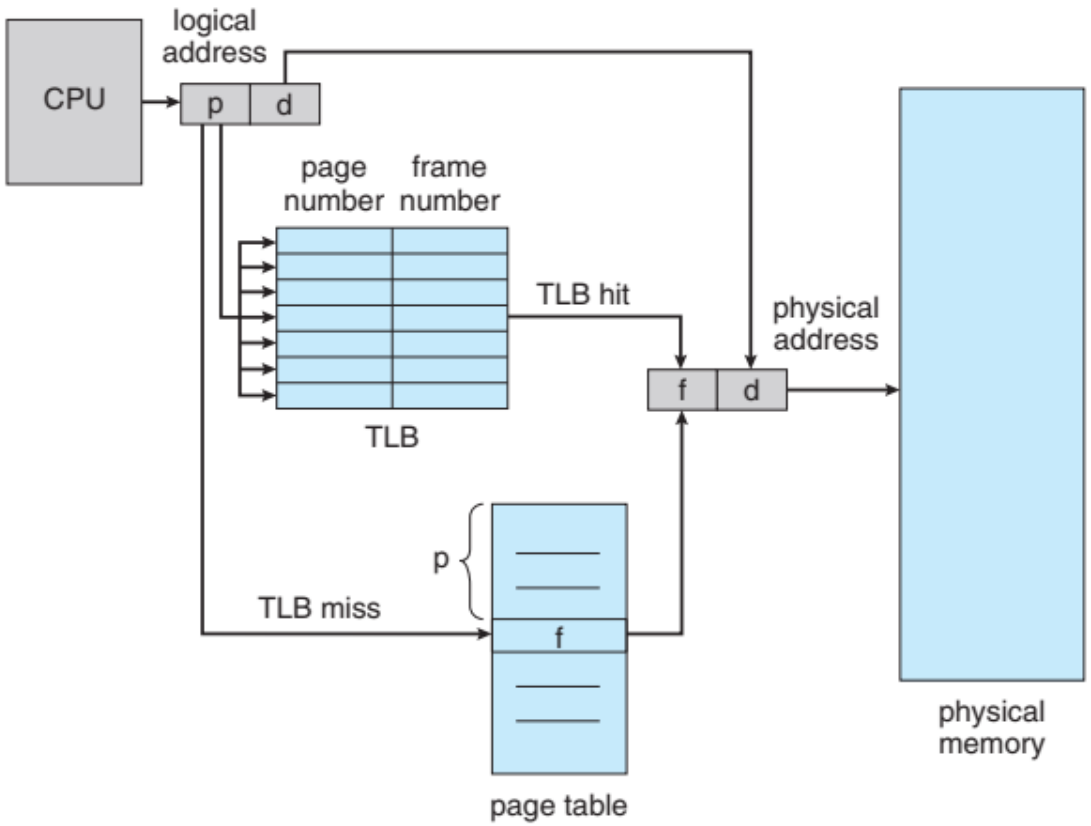


Figure 9.12 Paging hardware with TLB.

3. Paging
Translation Look-Aside Buffer

- If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random.
  - Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves.
- Some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

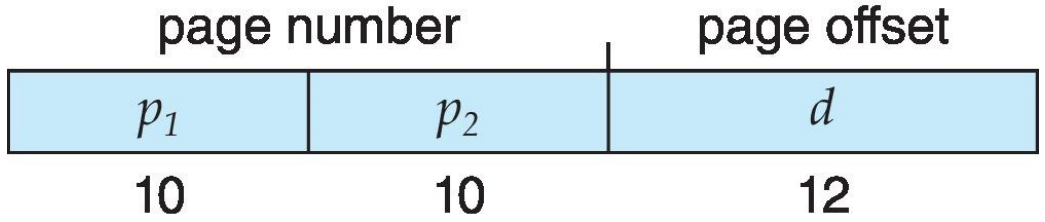
4. Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12} = 2^{20}$)
  - If each entry is 4 bytes → each process needs 4 MB of physical address space for the page table alone
    - Don’t want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller pieces.
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables

4. Structure of the Page Table

Hierarchical Paging

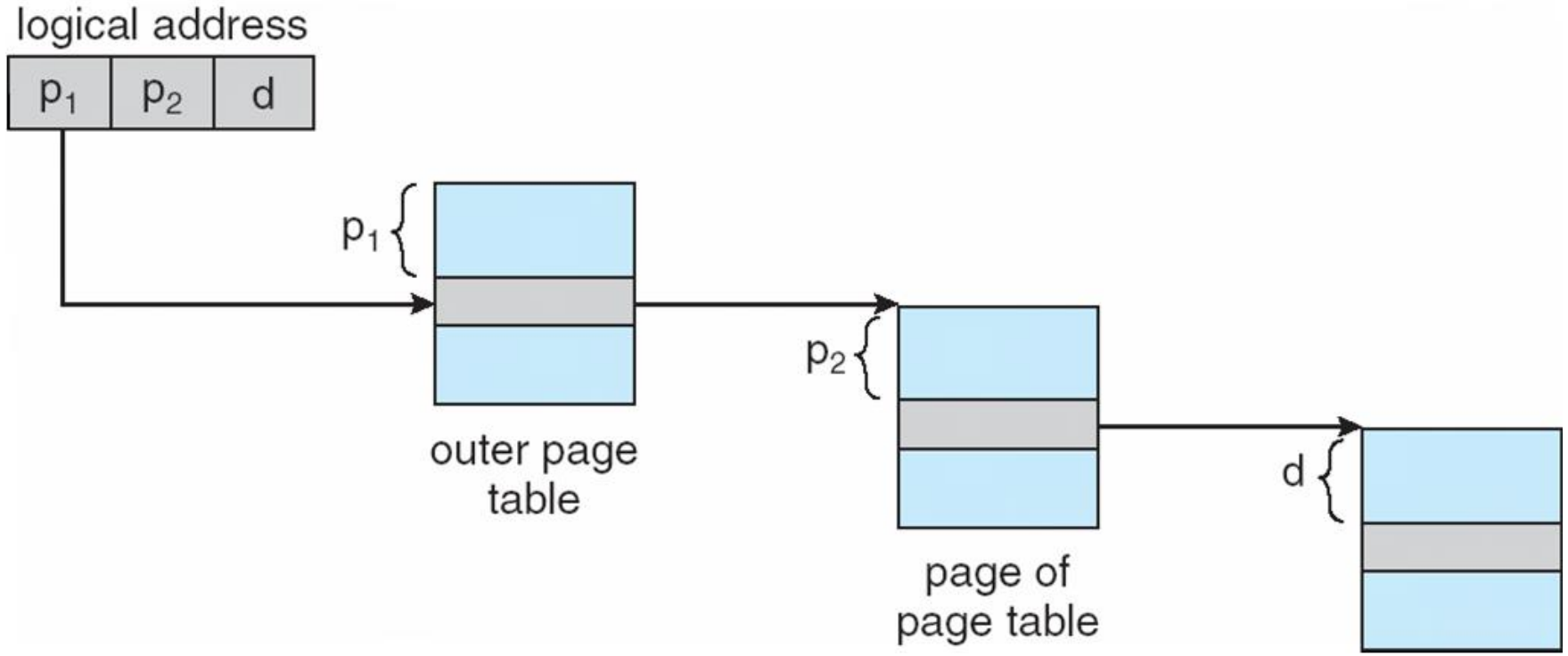
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



  - 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**

4. Structure of the Page Table

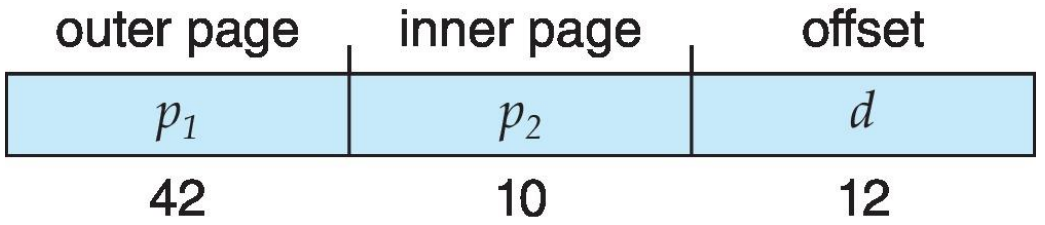
Hierarchical Paging



4. Structure of the Page Table

Hierarchical Paging

- 64-bit logical address space → 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

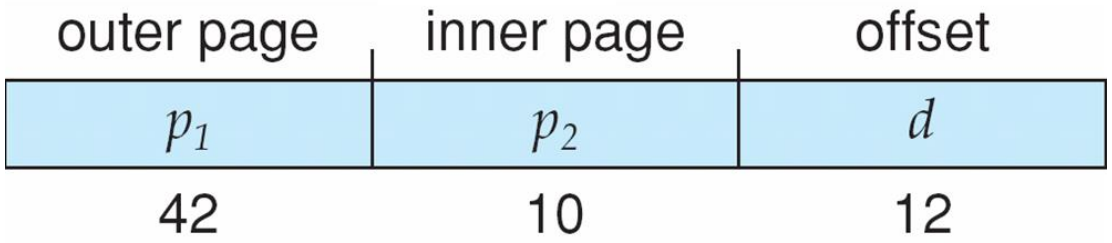


  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

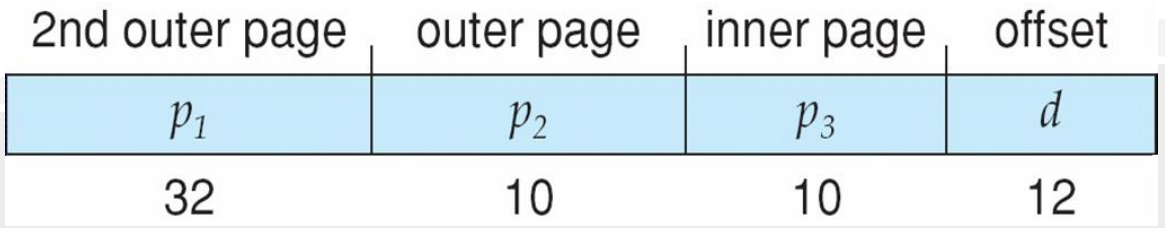
4. Structure of the Page Table

Hierarchical Paging

- Two-level paging scheme: Outer page table has 2^{42} entries or 2^{44} bytes (too large table)



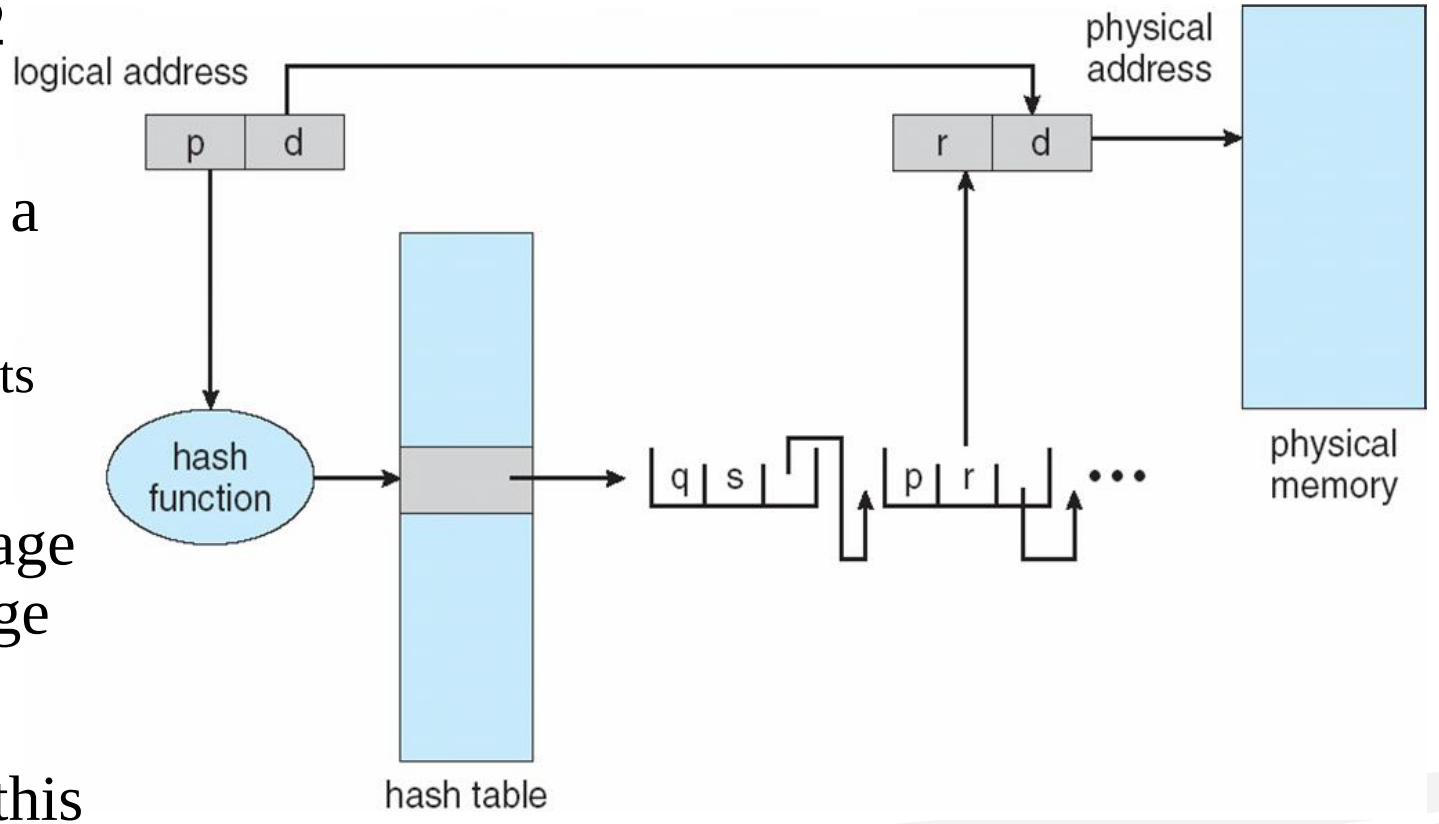
- Three-level paging scheme: The 2^{nd} outer page table is still 2^{34} bytes in size (16 GB)
  - 4 memory access to get to one physical memory location



4. Structure of the Page Table

Hashed Page Tables

- Commonly used in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted



4. Structure of the Page Table

Inverted Page Tables

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages → Inverted page table
  - An inverted page table has one entry for each real page of memory.
- 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
  - Thus, only one page table is in the system, and it has only one entry for each page of physical memory.
- 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
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

4. Structure of the Page Table

Inverted Page Tables

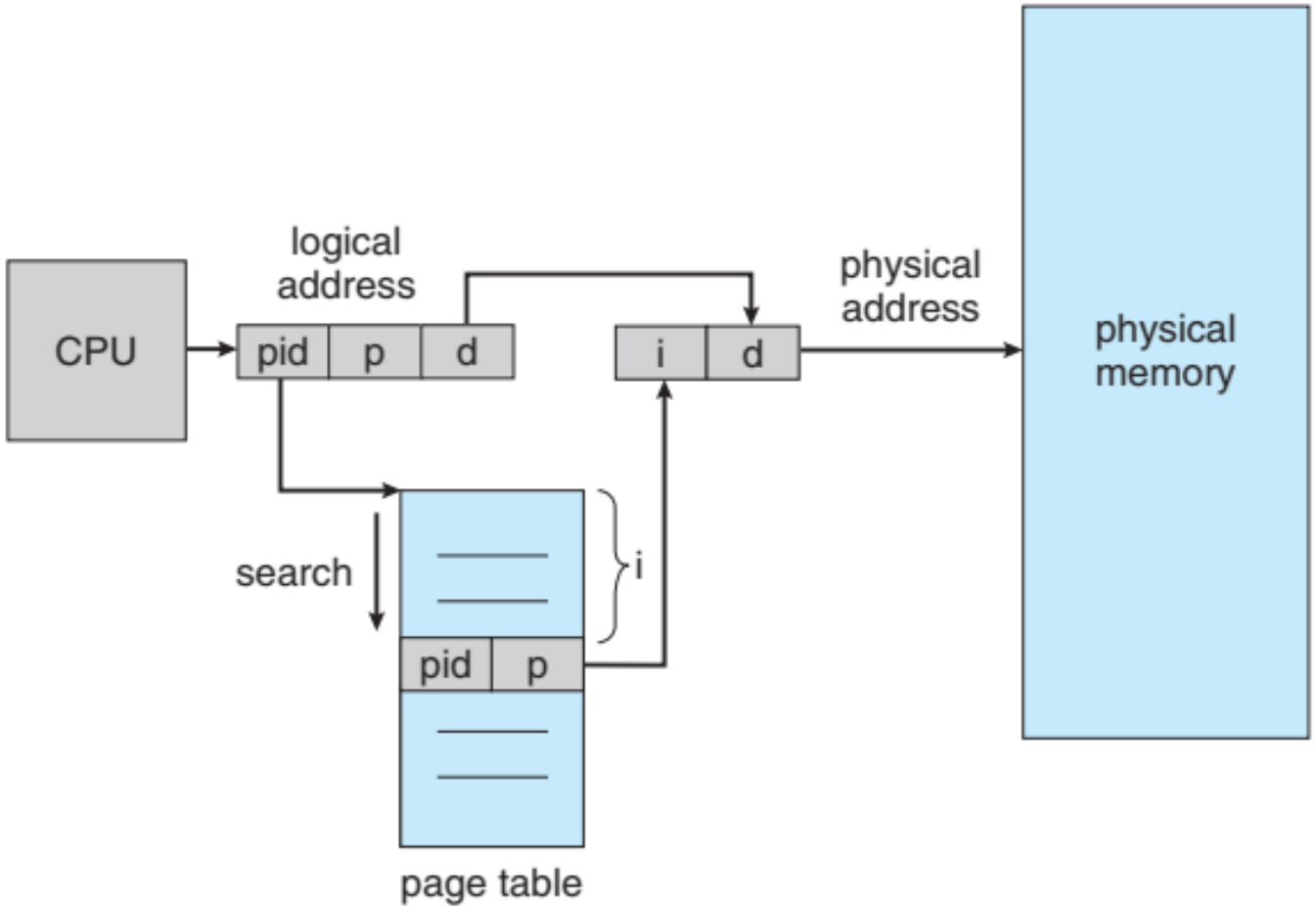
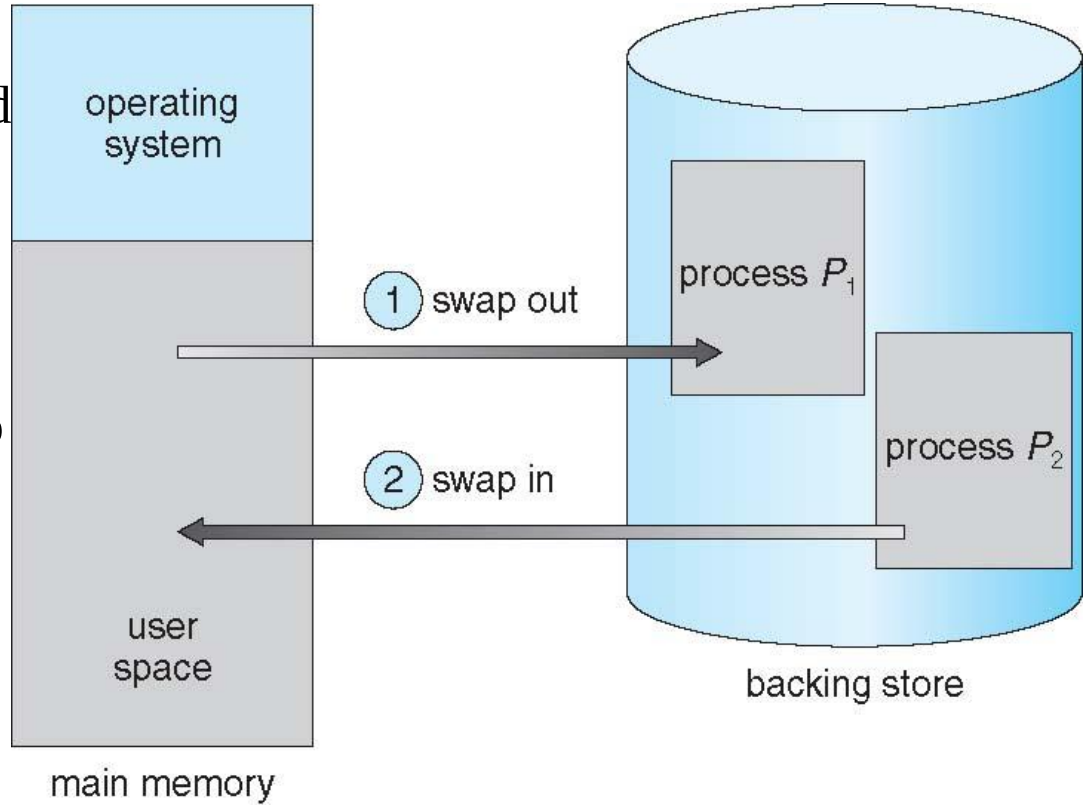


Figure 9.18 Inverted page table.

5. Swapping

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



5. Swapping

- 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

5. Swapping

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()`