Memory Management

CISC 3320 — CUNY Brooklyn College
Lecture Notes

Memory Management

In this chapter, we will:

1. Describe the hardware settings revolving around memory,
2. Define the concept of address space and explain how a program accesses an address in memory,
3. Discuss several memory management strategies, including Swapping, Contiguous Memory Allocation, Segmentation, and Paging, and
4. Introduce and analyze various structures that the page table could assume.

Basic Memory Hardware Settings

Since processes reside in memory while running, to run an instruction, we need to fetch it from memory first, which makes memory a critical part of the operating system.

The CPU can only access its registers and main memory. It can't access the hard drive or I/O devices directly without going through memory.

A CPU accesses its registers very quickly, usually within 1 clock tick, while memory accesses take longer time, usually a few clock ticks. One solution to decrease access time is cache, which is located closer to the CPU and therefore is accessed faster than memory.

A process 'sees' its own memory in a limited way: it is not able to access memory of other processes or OS-specific memory (the chunk of memory controlled by the kernel.)

Basic Memory Hardware Settings

Specifically, for the process, memory ranges from address 0 to the end address that was allocated to it (e.g., address 1024.) However, this chunk of memory is actually located somewhere in the midst of the computer's main memory, starting at some greater address than 0. In other words, the OS gives every process the illusion that its memory is the only existing one (as though other processes don't exist.)

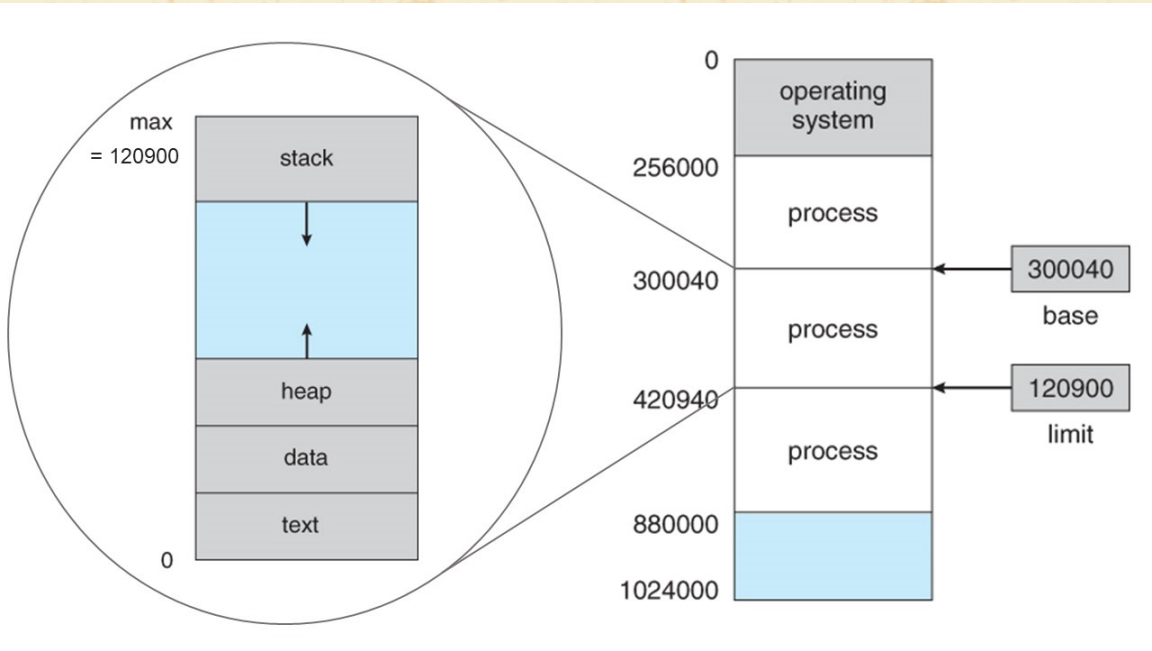
To know where every process's memory chunks are located, the operating system uses 2 registers: the **`base` register** (which tells at which address the memory chunk **starts**) and the **`limit` register** (telling **how large** the chunk is.)

What process is located at the memory's actual 0th address? This is the boot loader program, whose purpose is to boot the operating system as we learned in Topic 2, Slides 42-43.

In order to prevent a process from illegally accessing the memory of another process, the OS will check if the address that the process is trying to access is located within the following interval of addresses: `[base, base + limit)`.

Basic Memory Hardware Settings

Example: Although the process below sees its memory as ranging from 0 to 120900, the operating system actually puts it at addresses 300040 to 420940.



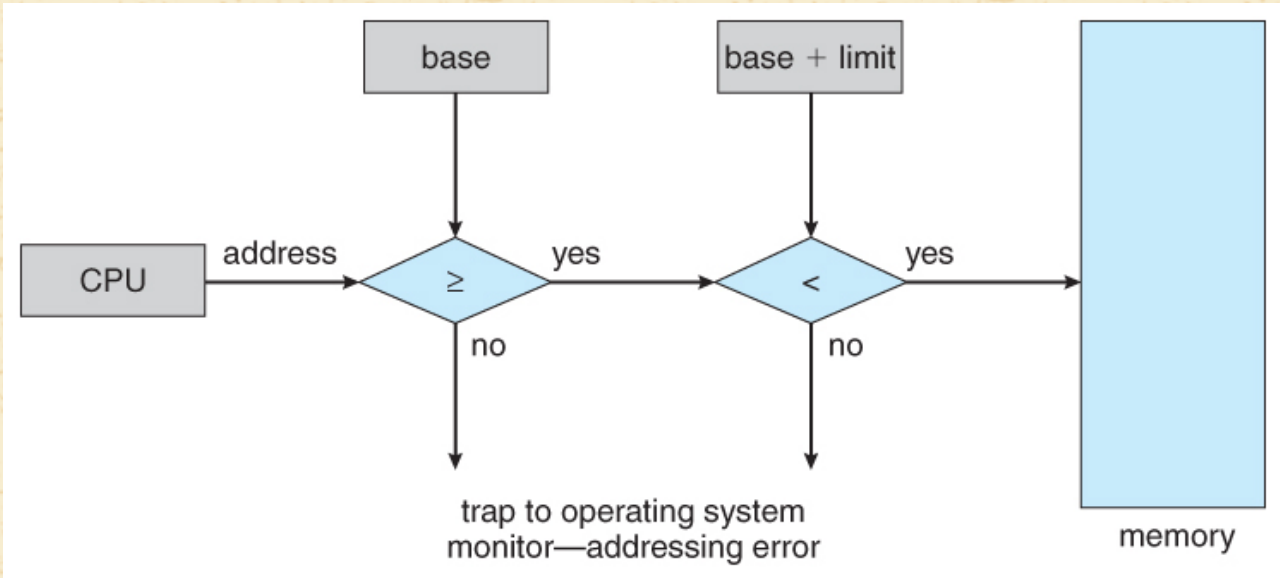
The memory illusion of every process. Derived from Bell, John T. "Main Memory." University of Illinois, Chicago.

Basic Memory Hardware Settings

Any questions?

Basic Memory Hardware Settings

When a process attempts to access an address in memory, the OS checks that the address is not outside of the process's memory section by checking that it is between the start (`base`) and end (`base + limit`) addresses:



Hardware address protection with base and limit registers. Derived from Bell, John T. "Main Memory." University of Illinois, Chicago.

Address Binding

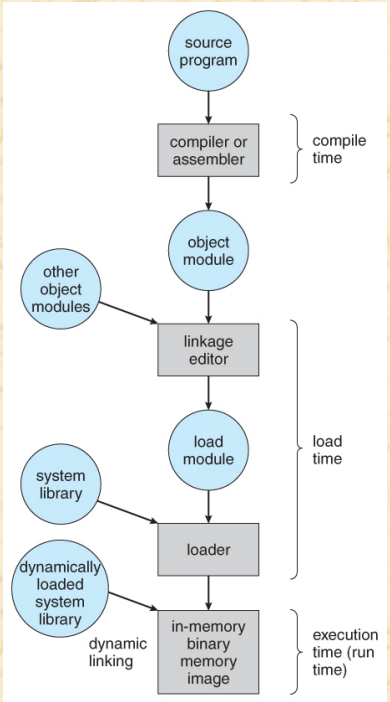
Inside a program's source code, memory addresses are usually referred to using symbolic names such as `i`, `count`, and `maxItem`, which we call **variables**.

These symbolic names must be **mapped** or **bound** to physical memory addresses, which occurs in one or more of the following several stages:

1. **Compilation Time**: If it is known at compilation time where a program will reside in physical memory, then **absolute code** can be generated by the compiler, containing actual physical addresses.
2. **Load Time**: If the location at which a program will be loaded is not known at compile time, then the compiler must generate **relocatable code**, which references addresses relative to the start of the program.
3. **Execution Time**: If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OSes.

Address Binding

This diagram illustrates the address binding stages:



Multistep processing of a user program. Derived from Bell, John T. "Main Memory." University of Illinois, Chicago.

Logical Address Space vs. Physical Address Space

The address used by the CPU is a **logical address**, whereas the address actually seen by the memory hardware is a **physical address**.

Addresses that are bound at compilation time or load time have identical logical and physical addresses, while those bound at execution time have different logical and physical addresses.

When the addresses are different, we also call logical addresses **virtual addresses**.

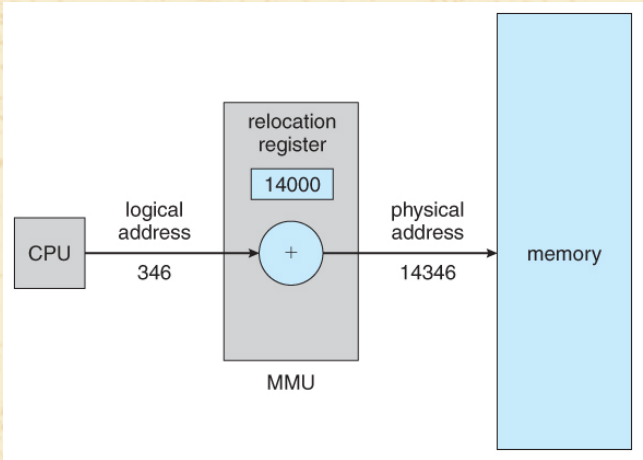
The set of all logical addresses used by a program composes the **logical address space**, and the set of all corresponding physical addresses composes the **physical address space**.

The run time mapping of logical to physical addresses is handled by a computer hardware component called the **memory-management unit (MMU)**.

Logical Address Space vs. Physical Address Space

One way in which the memory-management unit can perform the mapping is by using the `base` register, which we call the **relocation register**, as displayed in the diagram below. We simply add the value at the relocation register to the logical address to get the physical address.

User programs never see physical addresses and work entirely in logical address space.



Dynamic relocation using a relocation register. Derived from Bell, John T. "Main Memory." University of Illinois, Chicago.

Dynamic Loading and Linking

With **dynamic loading**, the operating system loads a function's code into memory from disk only when it is called. The advantage is that memory isn't filled up with functions that aren't used, which makes the program start faster.

Static linking is when library modules get fully included in executable files, wasting both disk space and main memory usage, because every program that included a certain function from the library would have to have their own copy of that function linked into their executable code.

Dynamic linking happens when only a **stub** is linked into the executable module, containing references to the actual library module linked in at run time, which saves disk space. Java uses dynamic linking.

With such **dynamically linked libraries (DLL)**, program maintenance is more convenient since, each time a change in the functions of the library (or class or interface) is made, there is no necessity in re-compiling a program that uses the library as long as the stub didn't change.

Swapping

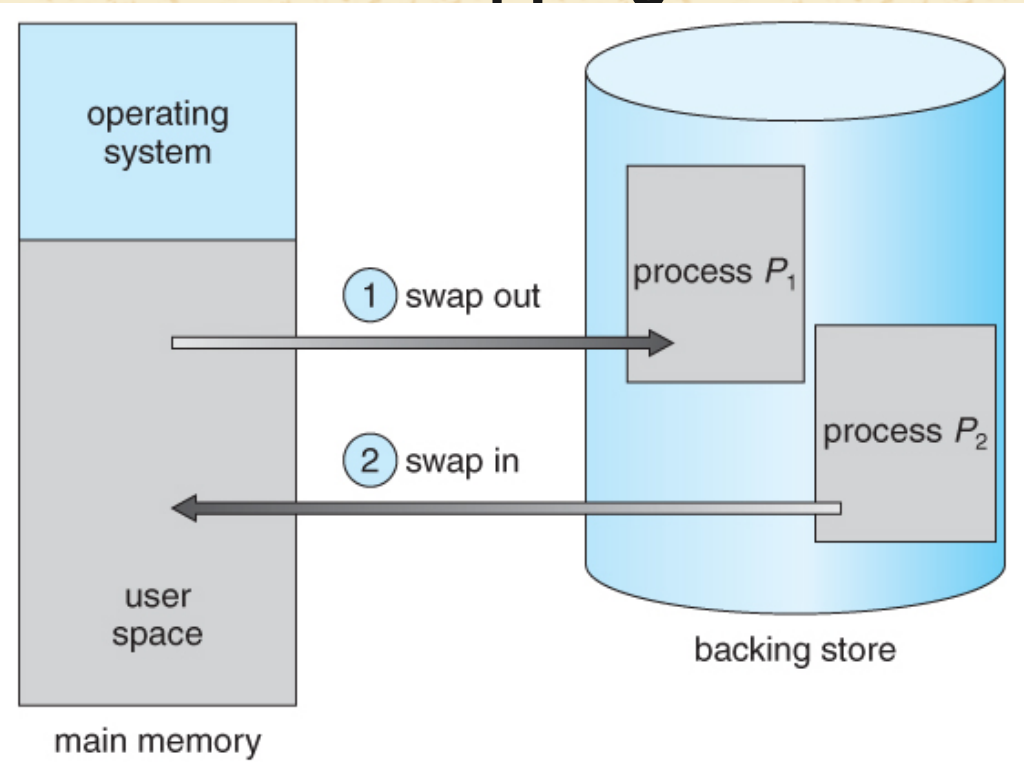
When a new process is about to be loaded to memory, but there is not enough space in memory, a currently-loaded program that is not using the CPU at the moment could be temporarily removed away from memory into a fast local disk called the **backing store**, to create enough space for the new program to load. This is called **memory swapping**.

If compile-time or load-time address binding is used, then processes must be swapped back into the same memory location from which they were swapped out. If execution time binding is used, then the processes can be swapped back into any available location.

It is important to swap processes out of memory only when they are idle, with no pending I/O operations.

Swapping is a very slow process compared to other operations. Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available (e.g., Paging, which we cover in these lecture notes as well.)

Swapping



Swapping of two processes using a disk as a backing store. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Swapping

Any questions?

Contiguous Memory Allocation

How does the operating system divide memory among programs? One approach is to load each process into a contiguous space, which is called **contiguous memory allocation**.

The operating system is placed inside memory first, usually at either far low or far high memory addresses, and then the remaining available memory is allocated to processes as needed.

One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used in practice.

An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. Several strategies exist for allocating memory this way (see the next slide.)

Contiguous Memory Allocation

The 3 most used strategies are:

1. **First fit**: the OS searches the list of holes until one is found that is big enough to satisfy the request, and assigns a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole.
2. **Best fit**: the OS allocates the *smallest* hole that is big enough to satisfy the request. If the OS sorts the list of free memory chunks by the chunk size, the OS could speed up the process of finding the right hole.
3. **Worst fit**: the OS allocates the largest hole available, thereby increasing the likelihood that the remaining smaller hole will be usable for satisfying future requests.

Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

Fragmentation

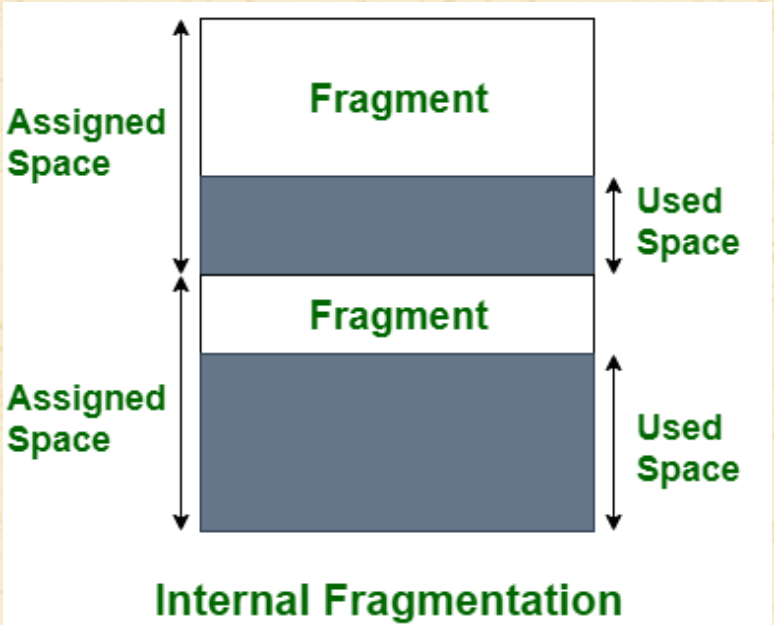
External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the combined holes' size could.

Internal fragmentation means that a process is given a chunk of memory that is larger than what the process requested. Since memory is allocated to processes in blocks, the portion of a block that the process doesn't use is an internal fragment.

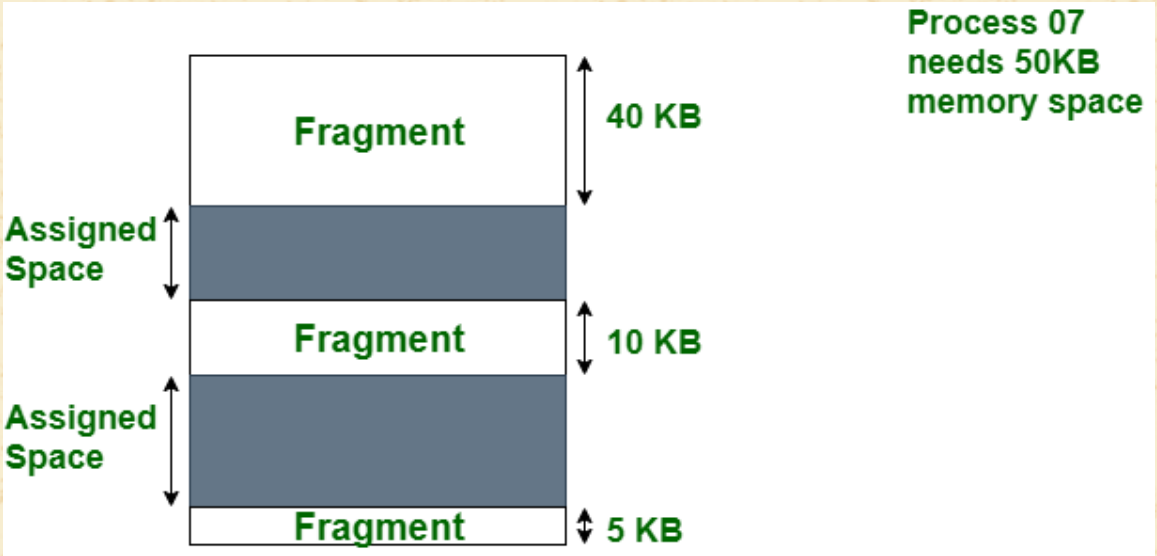
All the memory allocation strategies suffer from both external and internal fragmentation. Moreover, the same effect happens with hard drives.

Possible solutions: (1) use variable size blocks to minimize losses due to internal fragmentation, (2) if the programs in memory are relocatable, reduce external fragmentation with compaction, i.e. moving all processes down to one end of physical memory by only updating the relocation register for each process, and (3) allow processes to use non-contiguous blocks of physical memory, with a separate relocation register for each block (i.e., segmentation.)

Fragmentation



Intrenal fragmentation. Taken from "Difference between Internal and External fragmentation," geeksforgeeks.org.



Extrenal fragmentation. Taken from "Difference between Internal and External fragmentation," geeksforgeeks.org.

Segmentation

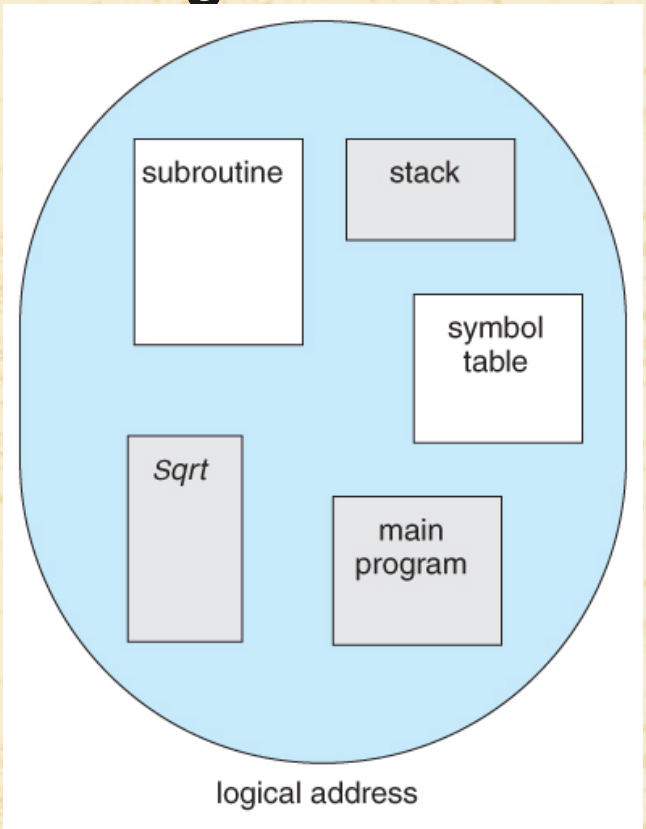
Most users (programmers) do not think of their programs as existing in one continuous linear address space.

Rather they tend to think of their memory in multiple segments, each dedicated to a particular use, such as the code [= text section], data, the stack, the heap, etc.

Memory segmentation supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment. That is, each segment will have its own base address.

For example, a C compiler might generate 5 segments for the user code, library code, global variables, the stack, and the heap, as shown in the figure on the following slide.

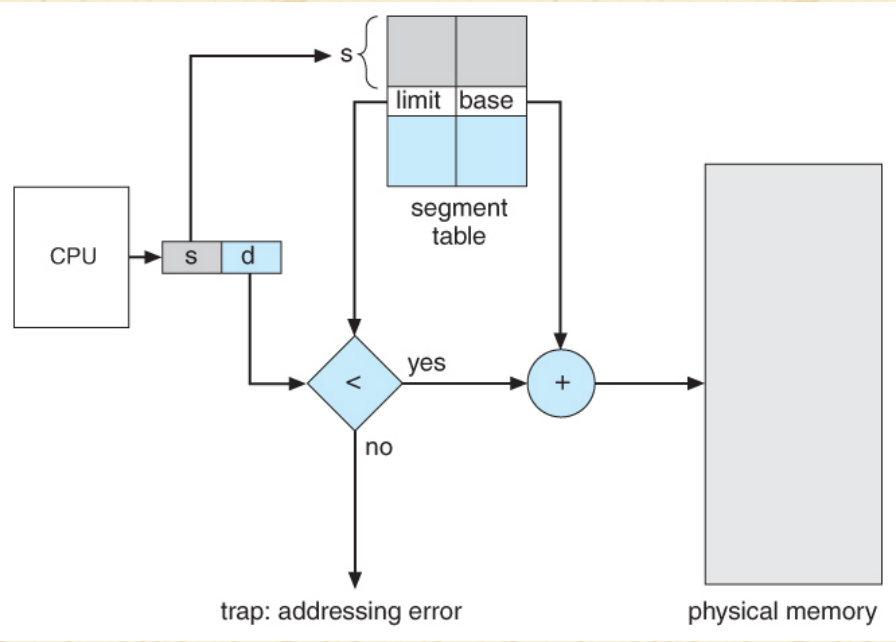
Segmentation



Programmer's view of a program. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

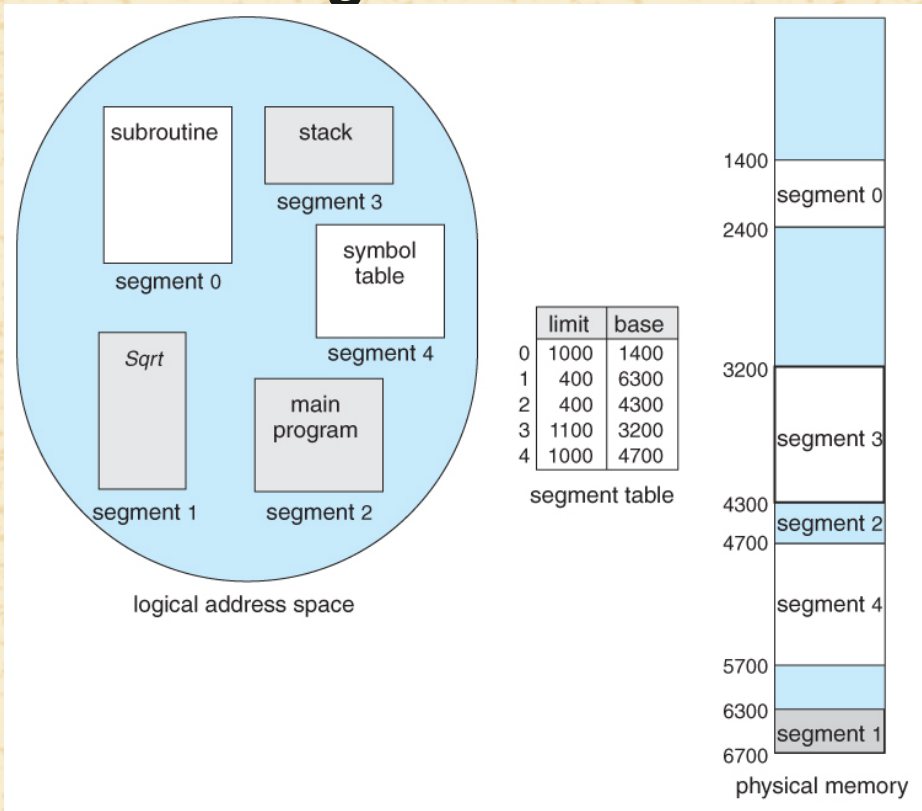
Segmentation

A **segment table** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to relocation base registers discussed previously on slide 7.



Segmentation hardware. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Segmentation



Example of segmentation. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Segmentation

Any questions?

Paging

Recall that **external fragmentation** is the issue of memory having holes (available memory sections) that are too small to fit in new processes.

An approach to this issue is called **paging**, in which the logical address space of a process is divided into equally-sized chunks called **pages**. Physical memory (main memory) is also divided into chunks of that same size called **frames**, and we map (copy) pages into frames.

Any page (from any running process) can be placed into any available frame, which means that a process can exists in memory at several, non-contiguous places.

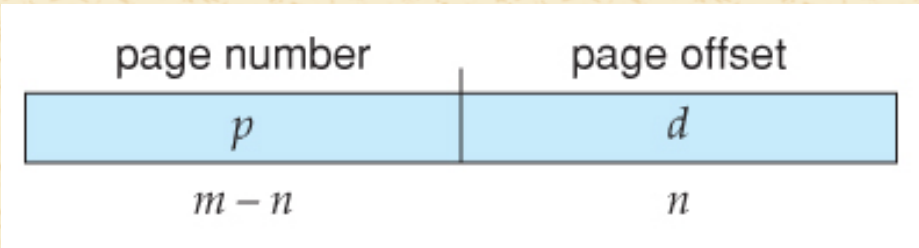
Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.

Paging

To know which page of logical memory is related to which frame in main memory, the operating system uses a data structure called **page table** that it stores in memory (inside the kernel address space.) The page table behaves like a dictionary: the keys are page numbers, and the output values are frame numbers.

Finding the physical address from the logical address goes as follows:

1. The logical address, which consists of *m* bits, is divided into 2 parts: the page offset (*n* rightmost bits), and the page number (*m* - *n* leftmost bits). The **page number** tells on which page of logical memory this address is located, and the **page offset** tells how deep into this page the address can be found.



The structure of the logical address. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

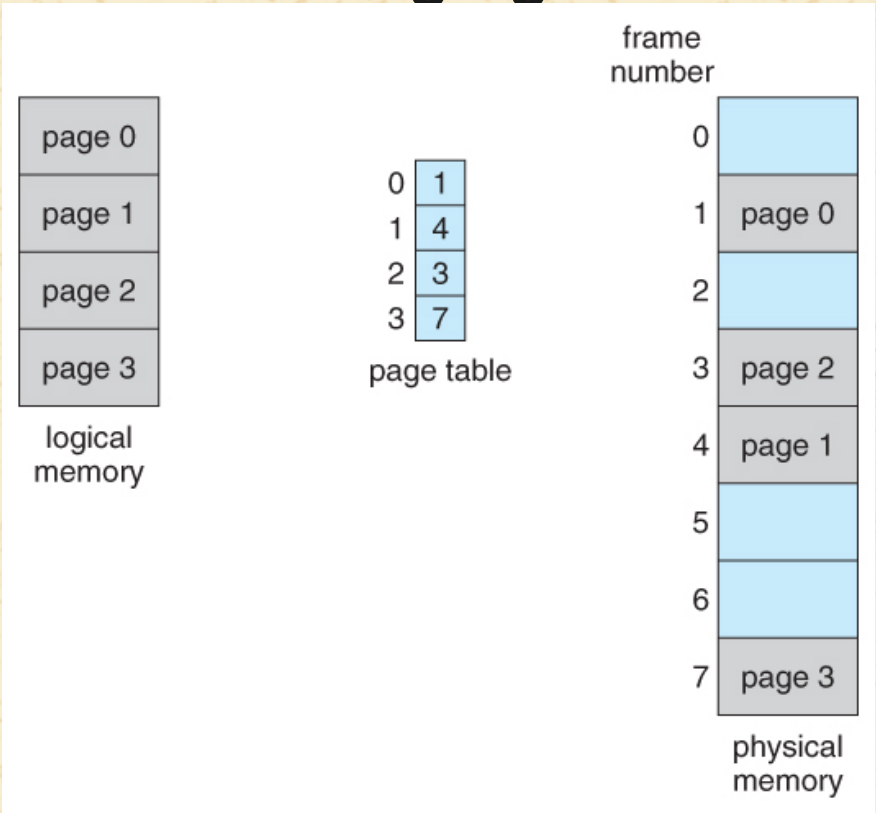
Paging

2. The operating system extracts the page number, and indexes into the page table to find the frame number.
3. The physical address will consist of the bits of the frame number followed by the bits of the page offset. That is, since all the pages and all the frames have the same size, the depth at which the address is found on the page is the same depth at which the address is located on the frame.
4. The operating system uses this newly-constructed address to access main memory.

Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is *2^m* (*m* bits in each address) and the page size is *2^n* (*n* bits given to the page number,) then the high-order (*m - n*) bits of a logical address designate the page number and the remaining *n* bits represent the offset, just as the image on the previous slide illustrates.

The following slide shows an example of using the page table to map pages to frames.

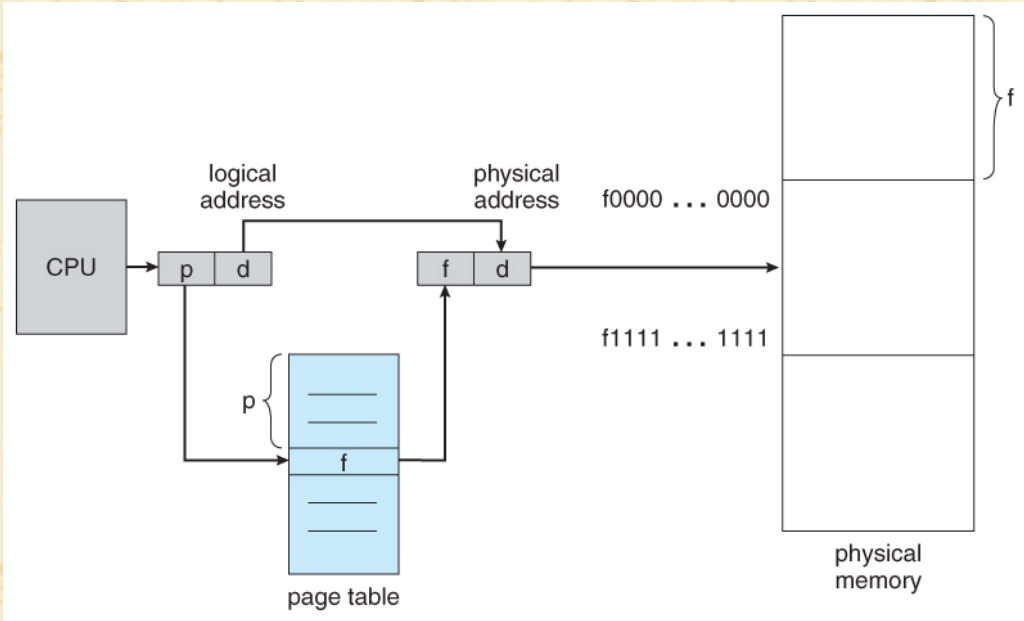
Paging



Paging model of logical and physical memory. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Paging

The following flow chart shows how physical address is derived from logical address via the page table:



Paging hardware. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Paging

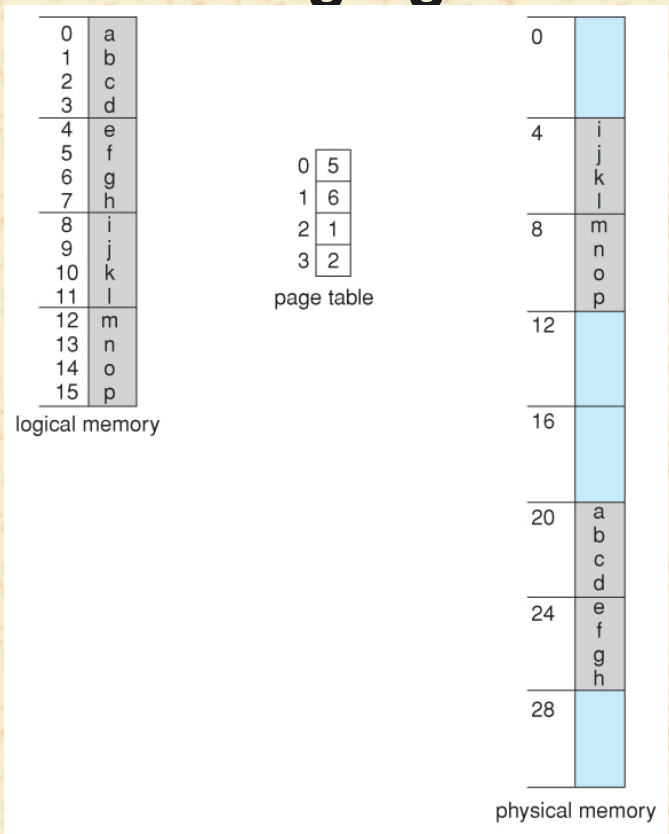
On the following slide is a small example in which each page and each frame stores 4 bytes. Logical memory has 16 addresses (0 to 15), and physical memory has 32 addresses (0 to 31.)

Mapping examples:

1. Logical address 11 in that example will map to physical address 7 since 11 is on page 2, which maps to frame **1**, and the offset of 11 is **3** (since 11 is address 3 on page 2 because 8 is address 0 on this page,) so the physical address will be 1*4 + 3 = 7 (**4** is the size of the pages/frames = 4 bytes.)
2. Logical address 13 maps to physical address 9 since 13 is on page 3, which maps to frame **2**, and the offset is **1** (since 13 is address 1 on page 3 because 12 is address 0 there,) so the physical address will be 2*4 + 1 = 9.

This is a miniature example; in real-world operating systems, pages are typically 512 to 8192 bytes in size, with 4096 (= 4K) being a typical value.

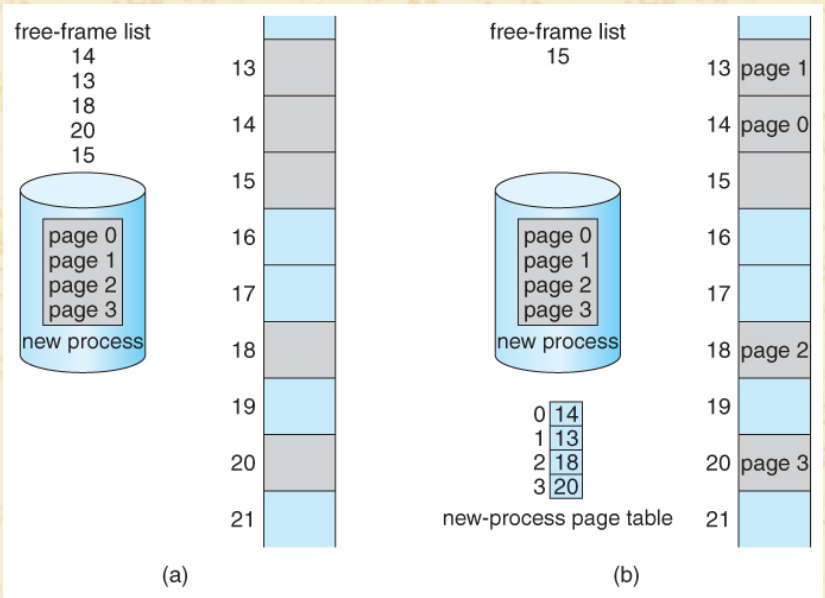
Paging



Paging example for a 32-byte memory with 4-byte pages. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Paging

The OS will maintain a page table for every process. When a new process needs memory, free frames are allocated from a data structure called the **free-frame list**, and inserted into that process's page table.



Free frames (a) before allocation and (b) after allocation. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Paging

Whenever we perform a page lookup, we access memory, which could hinder the activity of the OS. In addition, page tables must be copied during context switches, making context switches slower.

How can paging be done faster? Possible solutions:

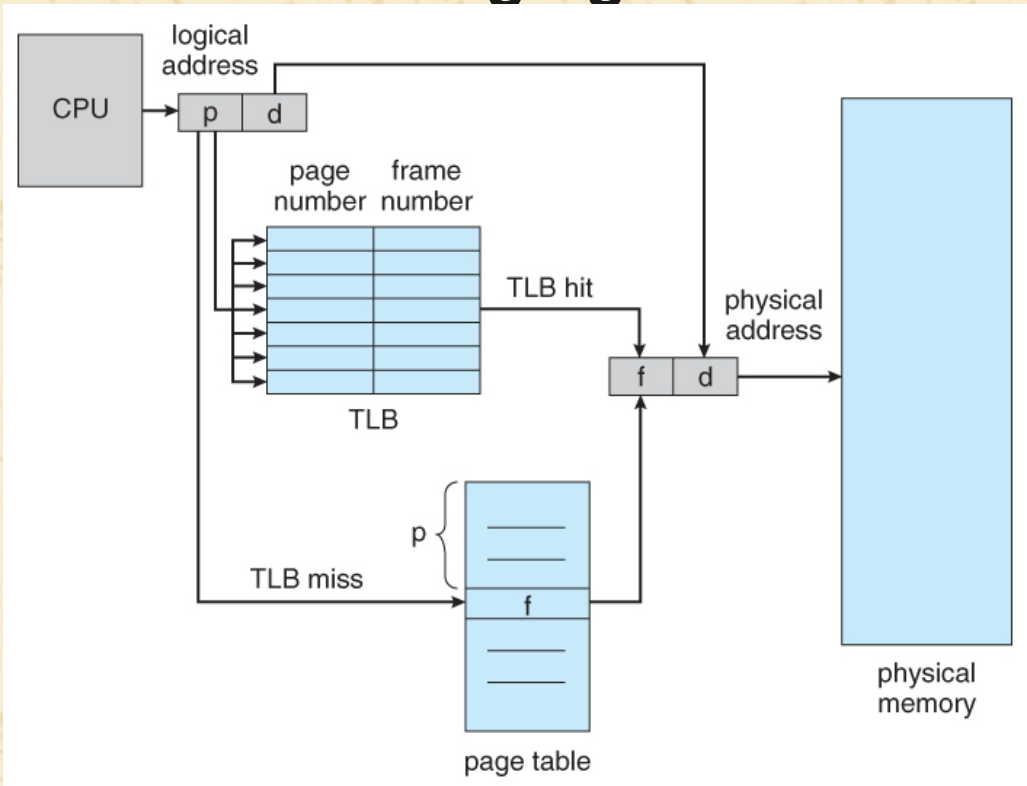
1. Store the page table entries inside a set of registers on the CPU. This will limit the amount of entries that we can store at a time because the CPU has several available registers only.
2. Store the page table in main memory, and use a single register (**page-table base register (PTBR)**) to indicate the location of the page table in memory. Context switching is fast, because only this single register needs to be changed. However, we will need to access memory *twice* for every access: to find the frame number and then to use the resulting physical address to access memory.

Paging

3. [Best solution:] Use a very special high-speed memory device called the **translation look-aside buffer (TLB)** to store a part of the page table.
   - This cache-like device is very fast and doesn't incure performance penalty.
   - The TLB must be kept small in size (typically from 32 to 1024 entries).
   - This means that the TLB can store only a part of the entries of the page table at a time.

   The usage of the TLB goes as follows: whenever the CPU is looking up an address, it will first look into the TLB first to check if the page entry is in there.
   - If yes, the CPU won't need to access main memory to search the full page table, and will instead extract the frame number from the TLB. Afterwards, it will make only a single access to memory: to access the physical address.
   - If no, the CPU will make 2 memory accesses: (1) to fetch the frame number from the page table, and (2) to access the memory location at the calculated physical address based on the found frame number.

Paging



Paging hardware with TLB. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Paging

The percentage of time that the desired information is found in the TLB is termed the **hit ratio**. The **miss ratio** is equal to (1 - hit ratio).

We can compute the **average memory access time** by knowing the (1) hit ratio and (2) the speed of accessing a physical memory address. This is possible because we know that we access memory only **once** if the entry is in the TLB and **twice** if it is not in the TLB.

Examples:

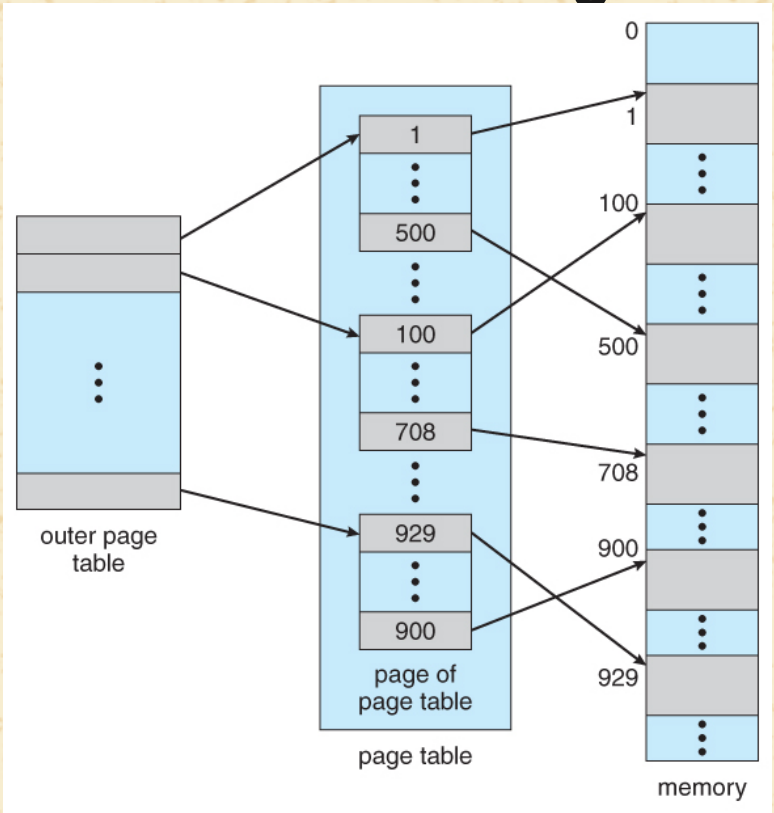
- If the hit ratio is standing at 95%, and the speed of accessing a physical memory is 100 nanoseconds (ns), the average memory access time is: 0.95 * 100 + (1 - 0.95) * 2 * 100 = 0.95 * 100 + 0.05 * 200 = 105 ns.
- If the hit ratio is standing at 80%, and the speed of accessing a physical memory is 85 nanoseconds (ns), the average memory access time is: 0.80 * 85 + (1 - 0.80) * 2 * 85 = 0.80 * 85 + 0.20 * 170 = 102 ns.

Structures of the Page Table

Because we use the page table to find in which frame some data resides in memory, it is imperative that the structure of the page table allows fast and efficient usage. Each of the following structures addresses a different efficiency issue/aspect:

1. **Hierarchical Paging:** when the page table consists of many entries (which is very common in OSs with large main memory chips,) we can break the page table into smaller, non-contiguous chunks, each of which could be stored at a different location inside memory. This way, we make our storage of the page table more flexible.
   - One solution is to create an outer page table for the page table! (known as a **two-level paging scheme**)
   - Here is a cool analogy: think of a huge book that has an index section at the end to let the reader know on which pages some mentioned terms and concepts can be found. Because the book is very large, the index itself is large, so a solution is to break the index into further sections, such as alphabetically, etc.
   - Furthermore, in systems with a huge address space, even with a two-level paging, the outer table will still be too large, so those systems use three-level tables: 2nd outer page table → 1st outer table → inner page table!

Structures of the Page Table

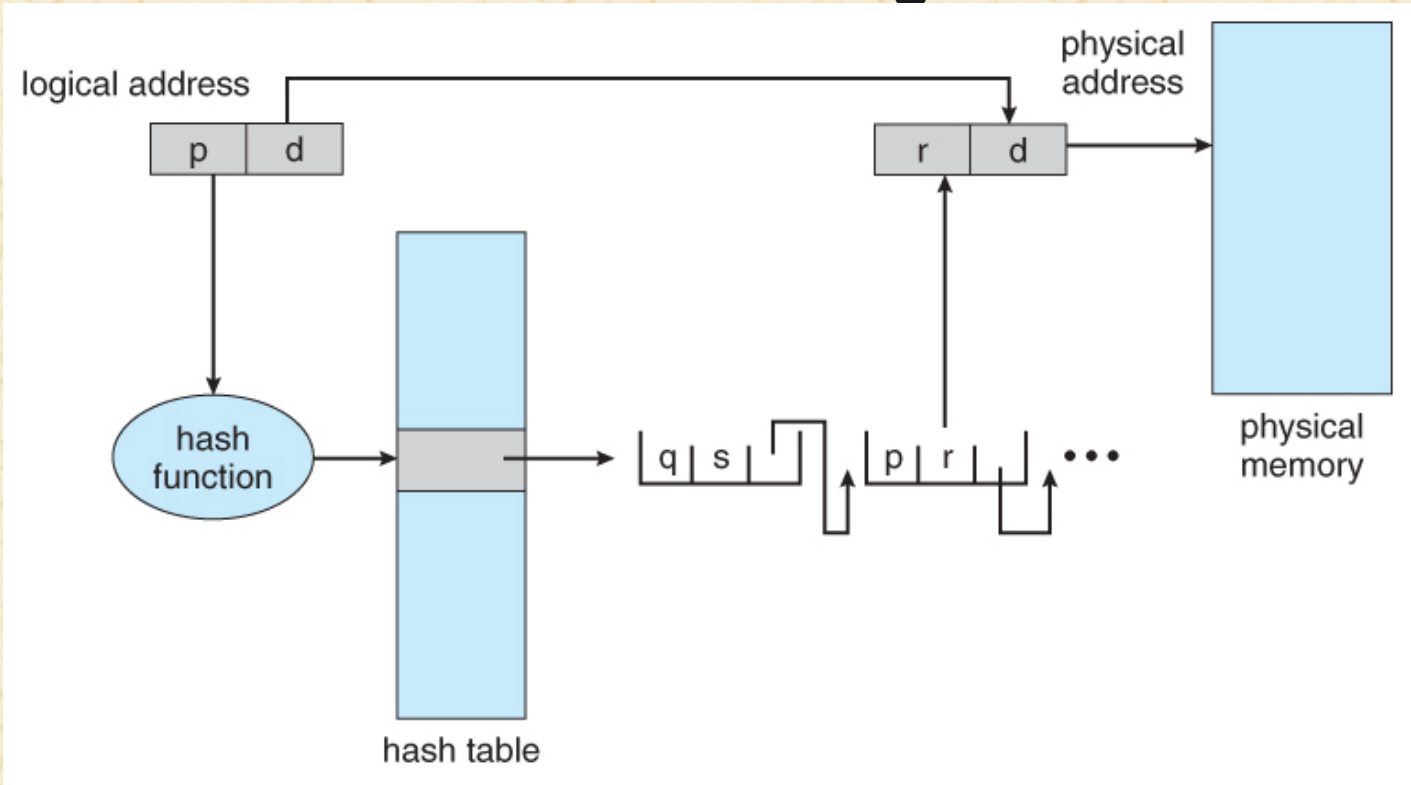


An example of a two-level page-table scheme. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Structures of the Page Table

2. **Hashed Page Tables:** Remember that we introduced the idea of a hash map in Chapter 2? A page table can have a hash map structure!
   - When we want to find the frame that corresponds to a page number, we take the following 3 steps:
     1. We plug the page number into the hash function and get an integer as a result,
     2. We then plug this integer into the page table (just as you would access an array,)
     3. We extract the frame number from that spot in the page table.
   - Because the hash function might return the same result for two different pages (this rare case is called 'a collision'), each spot in the hash map is not a single integer but a list of nodes. Each node in a list contains a page number, a frame number, and a link to the next node in that list. We simply check against the nodes in the list at that page table slot to see which one contains our page number, and then extract the frame number from that same node.
   - A hashed page table is very fast: it has O(1) [constant] access time! Yay!

Structures of the Page Table

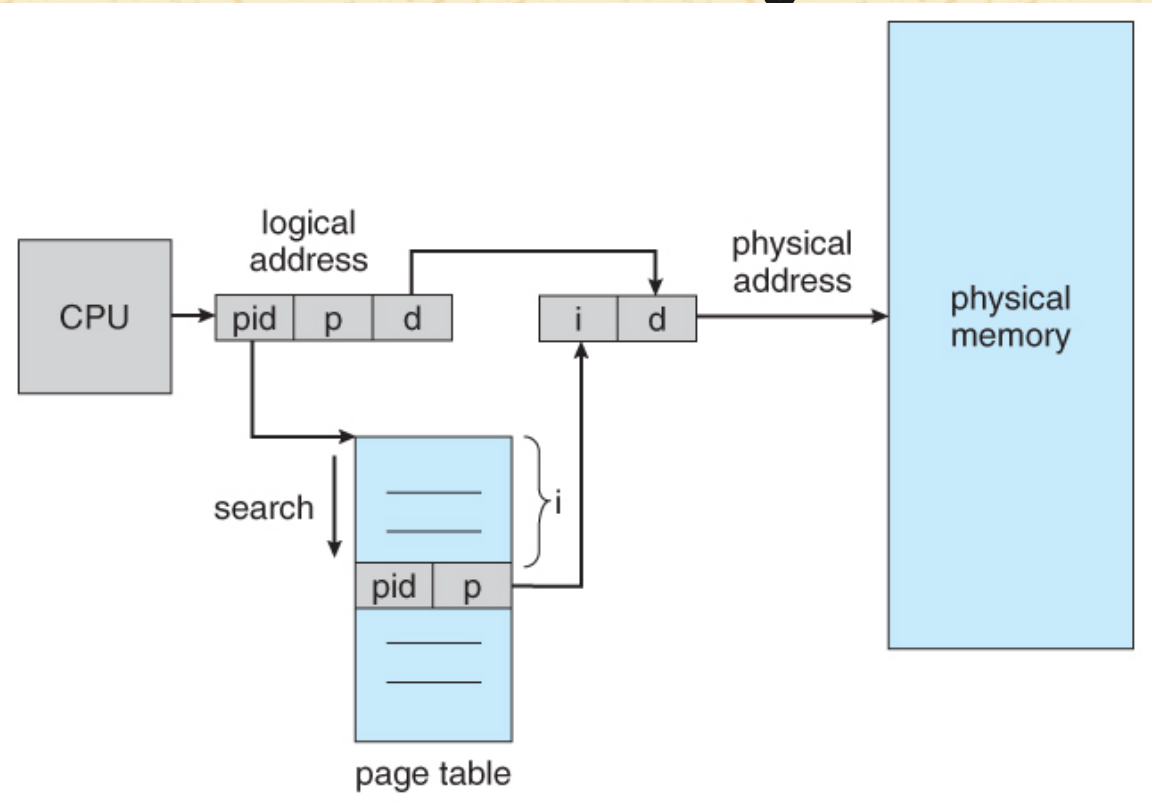


How a hashed page table is accessed. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Structures of the Page Table

3. **Inverted Page Tables:** As we learned earlier, page tables can get huge. One way to decrease their size is to only include entries for pages whose frames are currently inside main memory.
   - This is called an inverted page table because we focus more on the frames, not the pages.
   - Each entry of that table contains (1) the process id number, (2) the page number, and (3) the frame in memory.
   - If a page of a certain process isn't currently inside main memory, it won't appear in such a page table at all.
   - This way, we keep the page table small.
   - A possible disadvantage of this method is that the frame search time becomes longer (we might traverse the entire table just to find out that the page isn't there.) However, if the table is turned into a hashed page table (see slides 39-40 here,) this issue is resolved.
   - One issue that we can't prevent with inverted page tables is that each frame is now associated with only one process. We can't share memory between two processes anymore because the page table won't allow more than one entry for the same frame, so only 1 process can access the data in each frame.

Structures of the Page Table



How an inverted page table is accessed. Taken from Bell, John T. "Main Memory." University of Illinois, Chicago.

Paging

Any questions?

Memory Management: Sources

Bell, John T. "Main Memory." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/8_MainMemory.html