Memory

CISC 3310 — CUNY Brooklyn College
Lecture Notes

Computer Instructions

In this chapter, we will:

1. Remind ourselves of various memory-related concepts.
2. Introduce the memory hierarchy.
3. Learn the concept of memory locality and why it matters.
4. Talk about cache: mapping schemes, access time, and write policies.
5. Explain what paging and virtual memory are, and what problems they solve.
6. Calculate how much time it takes the CPU to access main memory.

Memory

We learned that the CPU is where computations and decisions are made, and that it is the heart of a computer. What is the purpose, then, of having main memory installed in the computer?

Main memory, which is also commonly called **RAM** (= Random Access Memory, due to it working like an array that you can access at any spot,) **primary memory**, or **read-write memory**, is used to store data, temporary configurations, and other information that the computer needs while running.

- Since the CPU has a small storage space, RAM also stores the instructions (code) of the currently running programs.

Before the computer shuts down, all the important data that is stored in main memory is saved to disk (for future access by the user or the operating system.) The rest is erased.

Main memory is organized as a linear (simple, one-dimensional) array of words, each of which is several bytes long.

Memory

Other memory concepts are:

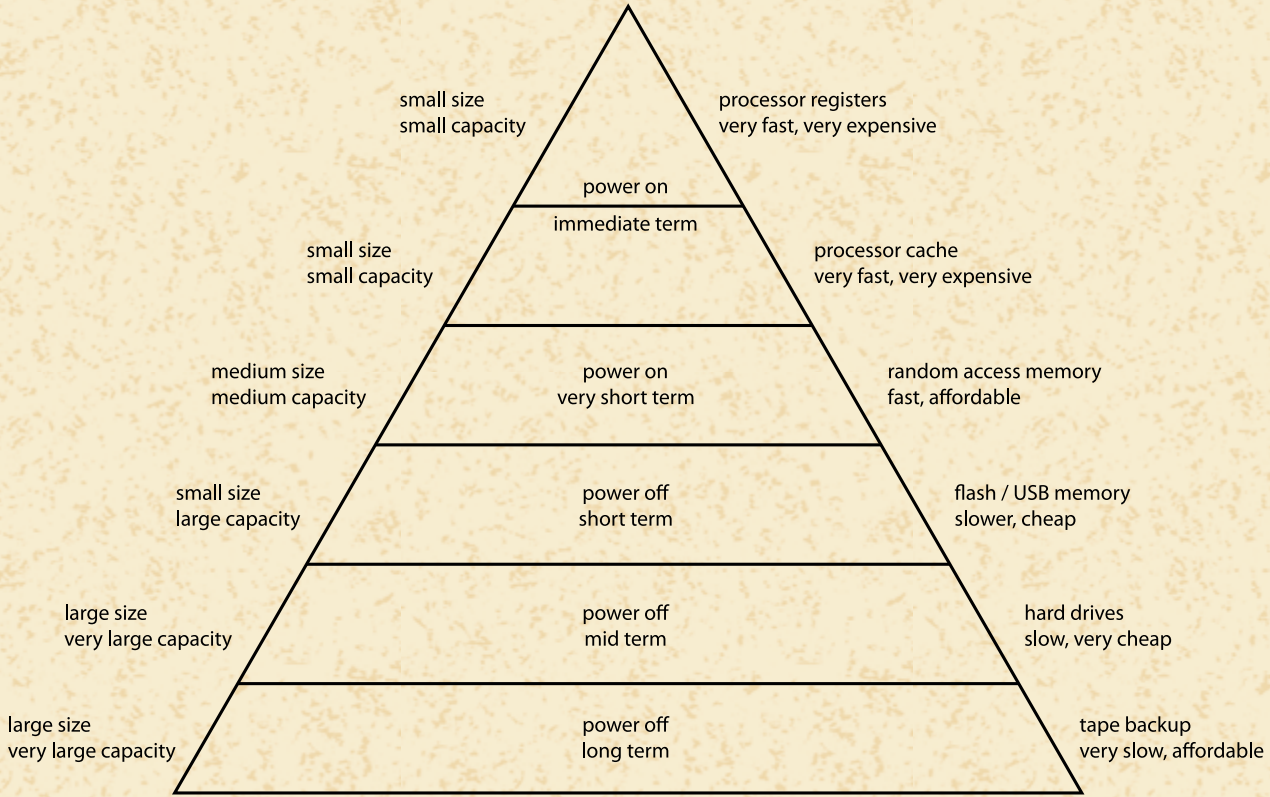
- **Cache** is a small, high-speed memory located close to the CPU that stores frequently accessed data and instructions. It reduces the time the CPU takes to retrieve data from slower main memory. Modern processors have multiple levels of cache (L1, L2, L3), where L1 is the smallest and fastest, and L3 is larger but slower. Cache is critical for improving performance in applications in which the same data is accessed repeatedly.
- **DRAM** (Dynamic Random-Access Memory) is a type of volatile (= erasable) memory used as the main memory in computers. It must be periodically fed with electricity to maintain the data inside it. DRAM is slower but less expensive than SRAM, making it suitable for large memory capacities required for running applications and the operating system.
- **SRAM** (Static Random-Access Memory) is a faster and more reliable type of volatile memory that does not require electric recharges since it uses flip-flops to store data. However, it is more expensive and larger per bit compared to DRAM. SRAM is commonly used in CPU caches and other small, high-speed applications where performance is critical.
- **ROM** (Read-Only Memory) is non-volatile memory that is typically used to store firmware or system-level code that does not change, such as the computer's BIOS or embedded system instructions, written into the ROM chip during manufacturing. ROM ensures the system can boot and function even without access to other storage.

Memory Hierarchy

We classify memory types according to their closure to the CPU:

1. **Registers** are located directly inside the CPU and used to store intermediate data. Registers are accessed in a single CPU clock cycle (= a couple nanoseconds,) making them the closest and quickest memory available to the processor.
2. **Cache** memory is located very close to the CPU, often integrated onto the CPU chip. It is designed to store frequently accessed data and instructions, significantly reducing the time the CPU spends fetching data from slower memory layers. Cache memory is slightly slower than registers and is accessed in several nanoseconds.
3. **Main memory** is separate from the CPU but connected to it via the system bus. While slower than cache, with access times in tens of nanoseconds, it provides the large capacity needed to store an operating system and application data.
4. **Secondary memory**, like HDDs and SSDs, is located much farther from the CPU and is accessed via storage interfaces like SATA. It is used for non-volatile storage, retaining data even when the system is powered off. Its access times ranges from microseconds to milliseconds, making it about 1000 times slower than main memory.
5. **External or disconnected storage devices** are typically used for archiving and long-term data storage. These devices are the farthest from the CPU, often requiring manual installation. As the slowest memory type, it is used for infrequent data access or backups. Examples include optical disks (CD/DVDs), tape drives, and external USB drives.

Computer Memory Hierarchy



ComputerMemoryHierarchy.png: User:Danlash at en.wikipedia.org, Public domain, via Wikimedia Commons.

Memory Hierarchy

Any questions?

Memory Locality

Memory locality is the principle that programs tend to access data and instructions in patterns that are predictable and optimized for performance.

- It explains how memory access tends to cluster, allowing systems to design caches more efficiently.
- Memory locality is critical for improving performance, as it enables the CPU to retrieve frequently accessed data faster.
- There are three main types of locality: temporal locality, spatial locality, and sequential locality.

---

1. **Temporal locality** occurs when a program accesses the same memory location repeatedly within a short period.
    - This type of locality takes advantage of the tendency to reuse recently accessed data or instructions.
    - Example: In a program loop, a counter variable is repeatedly updated and accessed. Since the variable resides in memory, repeated access to it demonstrates temporal locality.

Memory Locality

2. **Spatial locality** happens when a program accesses memory locations that are close to each other within short time.
    - This occurs because data is often stored and accessed in contiguous blocks.
    - Example: When processing an array, elements are accessed sequentially. If the program accesses element `A[i]`, it is likely to access `A[i+1]` or `A[i-1]` next, as they are stored in adjacent memory locations.
3. **Sequential locality** is a specific case of spatial locality where the memory accesses follow a linear sequence.
    - This is particularly common in instruction fetching and certain data access patterns.
    - Example: The CPU fetching instructions from memory often reads them sequentially, executing one instruction and then moving to the next in memory order.

Understanding and leveraging these types of locality helps optimize system performance. For example, cache designs are structured to prioritize frequently accessed (temporal locality) and nearby (spatial and sequential locality) data to minimize costly access to slower memory levels.

Pages and Paging

As we already understand now, memory is among the most basic, yet essential resources of processes. **Memory management** includes responsibilities such as allocation, manipulation, and release of memory slots.

The **address space** of a program is all the memory slots that the operating system gave to that program to stores its variables in. The program, therefore, has legal access to these memory locations.

On modern computers, the CPU doesn't work with **physical addresses**, which are the actual addresses of memory slots in RAM. Instead, when looking at a program's code, the CPU sees this program's address space as the 'entire universe', without even being aware of the existence of other memory slots or other programs that run on that computer.

As such, the addresses the program uses (called **logical addresses** or **virtual addresses**) are different from physical memory addresses, so, to know where exactly we need to access main memory given a logical address, the CPU uses a software called the **memory management unit (MMU)** to covert each logical address to a physical address.

Pages and Paging

- The virtual address space of a program is divided into **pages**, which are contiguous groups of a fixed size of bytes.
- Machine architecture determines the page size of that computer. Typical sizes include 4KB on 32-bit systems and 8KB on 64-bit systems. (32-bit and 64-bit tell the length of a memory address: 32 vs. 64 bits long.)
- A page is called **valid** if the data (values of variables and objects, program's instructions, etc.) of this page is now located in RAM.
- An page is called **invalid** if its data is located on disk instead of in main memory. ***How could this happen?*** Sometimes, when there's not enough place in RAM to store data, the OS keeps some of the program's data on disk. This memory management technique is called **virtual memory**.
- A **page fault** happens when the program needs to use data that is located on an invalid page. That is, when the data is located on disk and not in main memory. The operating system will **page-in** (= copy) the data from the disk into RAM. The location in RAM that stores a page is called a **frame**.
- **Paging-out** is the process of moving data from RAM to secondary storage (to make room for new pages). Usually, such data are unlikely to be used soon, which is why we decide to them to disk instead of moving other, more crucial data.

Fragmentation: The Reason for Using Paging

Why do computers even need to use paging?

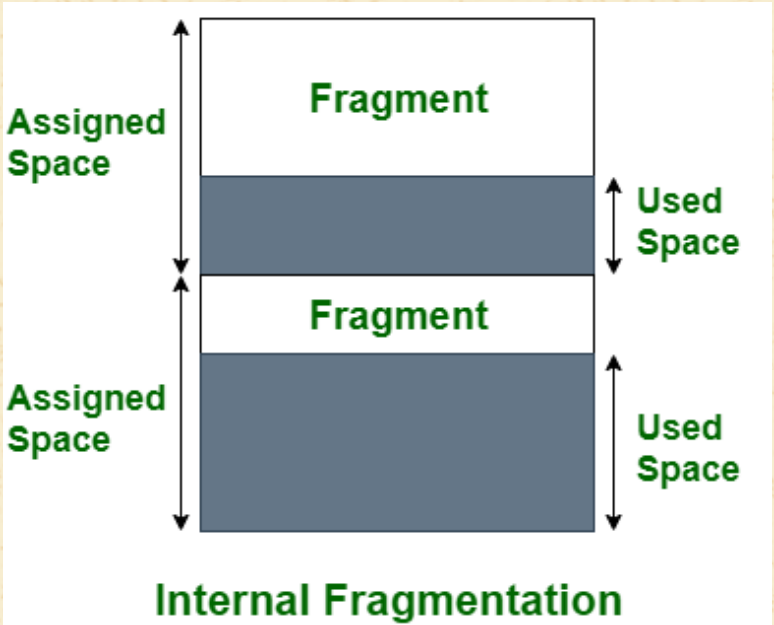
If we were to store the instructions and variables of a program in blocks of unequal sizes, we would encounter the following 2 nasty problems:

1. **External fragmentation** happens when the available memory is broken up into lots of little pieces, none of which is big enough store the instructions and variables of a program that we want to launch, although the combined holes' size could.
2. **Internal fragmentation** happens when a program is given a chunk of memory that is larger than what the program requested or uses. Since memory is allocated to processes in blocks, the portion of a block that the process doesn't use is an internal fragment.

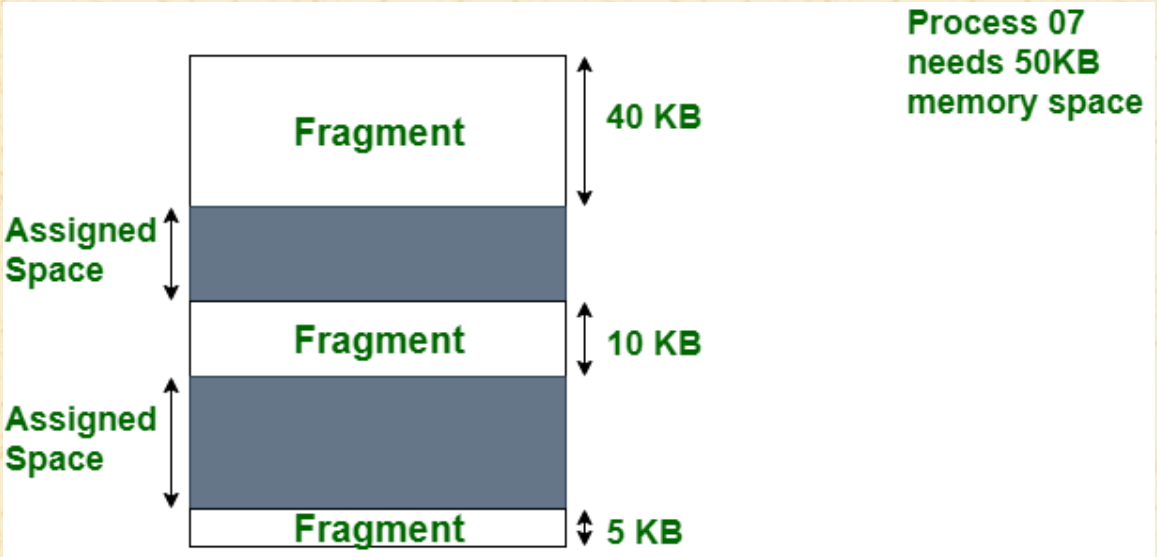
These problems affect not only main memory but also cache, disks, and other storage devices.

We can eliminate the external fragmentation problem and minimize the internal fragmentation problem by using pages!

Fragmentation: The Reason for Using Paging



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



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

Fragmentation: The Reason for Using Paging

Any questions?

Pages and Paging

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

An approach to this issue is called **paging**, in which the logical address space of a program 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 program) can be placed into any available frame, which means that a program can exists in memory at several, non-contiguous places.

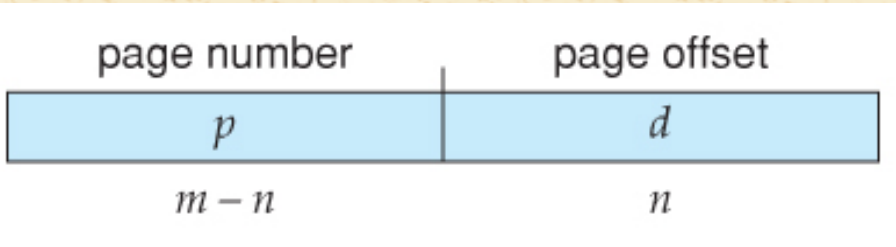
Paging is the predominant memory management technique used today.

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. 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.

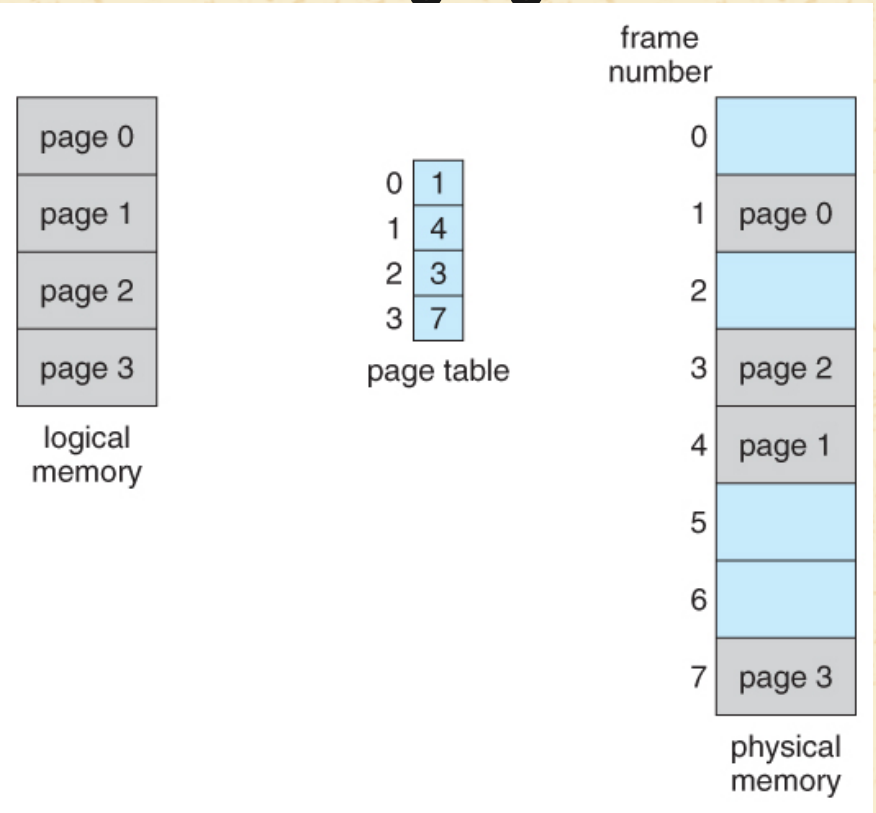
Paging

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.

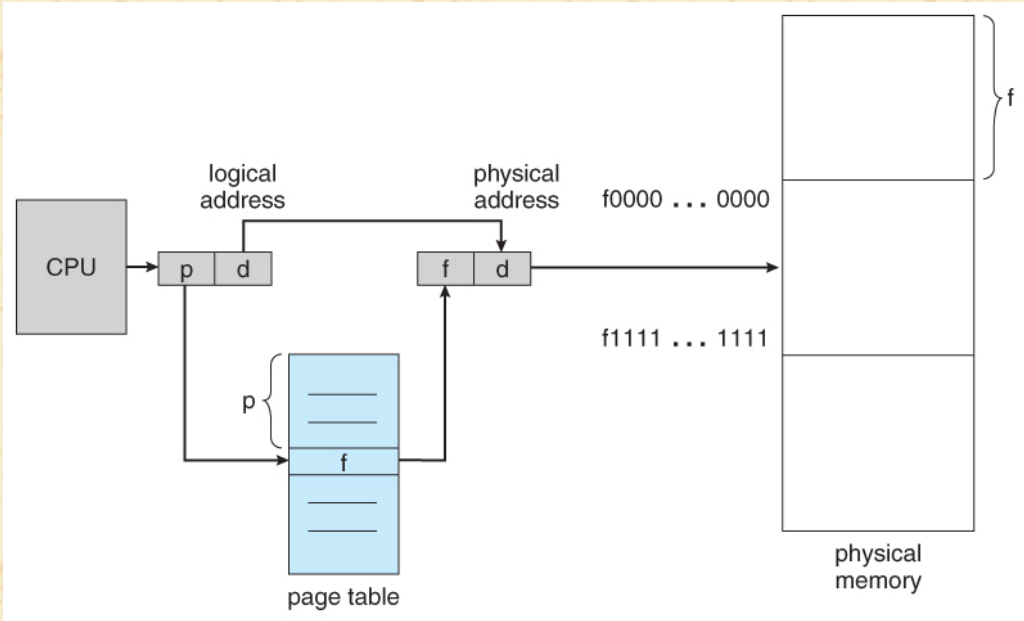
Paging



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

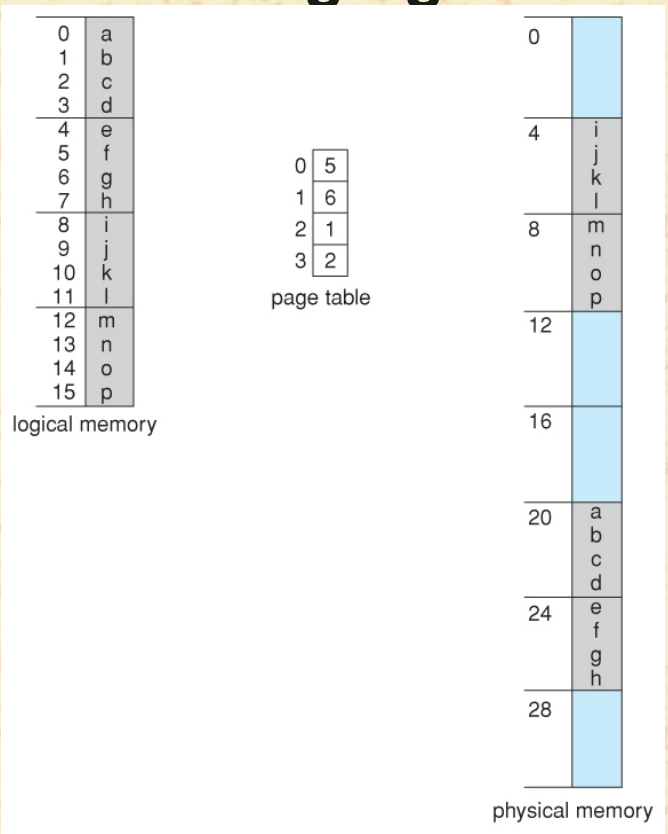
On the following slide is a small example in which each page and each frame stores 4 bytes. Logical memory has 16 addresses (0 to 15), and physical memory has 32 addresses (0 to 31.)

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.

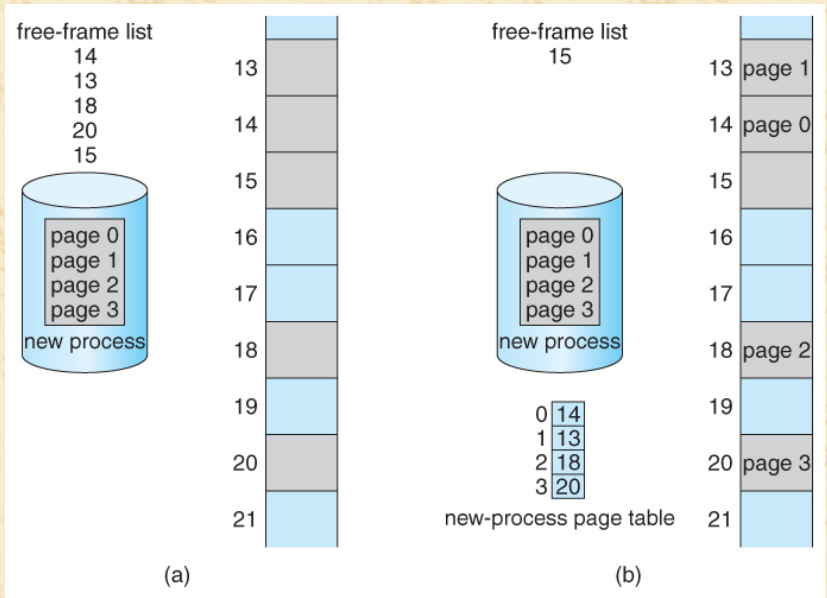
Paging



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

Paging

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

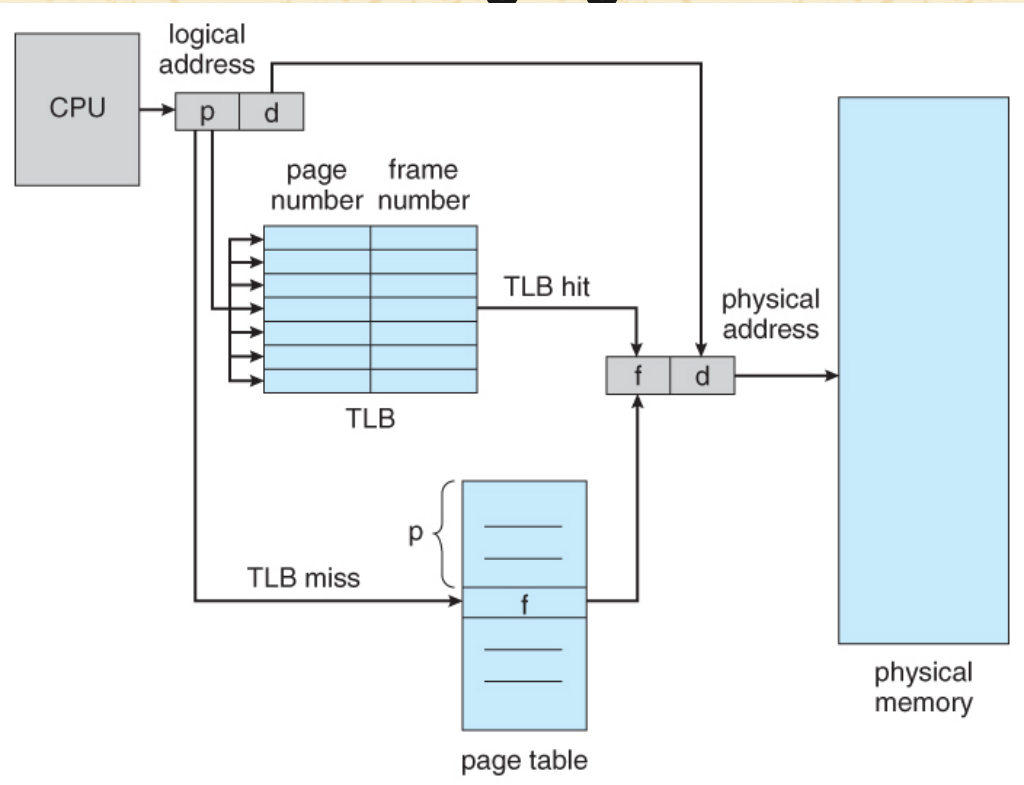
1. Store the page table entries inside a set of registers on the CPU. This will limit the amount of entries that we can store at a time because the CPU has several available registers only.
2. Store the page table in main memory, and use a single register (**page-table base register (PTBR)**) to indicate 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: $0.95 \cdot 100 + (1 - 0.95) \cdot 2 \cdot 100 = 0.95 \cdot 100 + 0.05 \cdot 200 = 105$ 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: $0.80 \cdot 85 + (1 - 0.80) \cdot 2 \cdot 85 = 0.80 \cdot 85 + 0.20 \cdot 170 = 102$ ns.

Paging

Any questions?

Virtual Memory

Suppose that you wrote a program that needs to use more memory space than main memory can offer. If the operating system attempts to fully store the program in memory, the program will crash due to lack of memory.

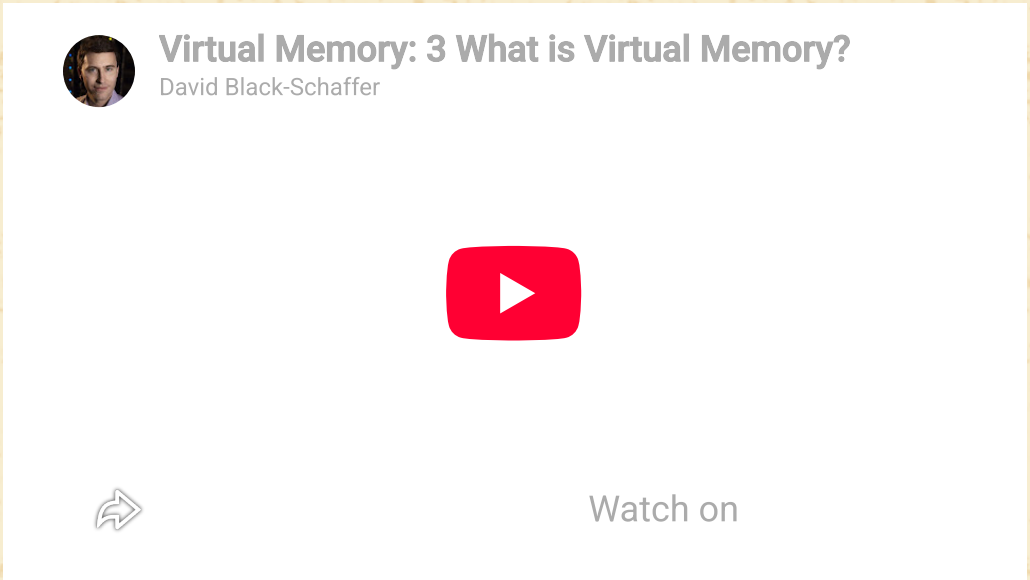
A solution to this situation would be to treat the program as virtual memory.

Virtual memory is the logical view of the program, and can be much larger than the existing physical memory. With virtual memory, we can choose to store only the necessary program pages in main memory, while storing the unused ones on disk until we need them.

When a page that is currently stored on disk is needed to be used, the operating system will copy some currently loaded page from memory back to disk (to provide enough space for the new page) and load the needed page to memory.

Virtual Memory

The following video defines and illustrates the concept of virtual memory in a nutshell:



https://www.youtube.com/watch?v=qlH4-oHnBb8

Virtual Memory

Why is it possible to load some page to memory while keeping others on disk?

1. Programs often contain code for handling errors (e.g., using `try-catch` blocks) that occur rarely. During the execution of a program, if such a rare error doesn't occur, it is not necessary to keep the error-handling instructions in memory until the error happens.
2. When an array (or another data structure that serves like a container) is created, many times a very large size is allocated for the array, just in case the worst case occurs for a certain algorithm. As long as the worst case doesn't occur, we can store only the used-up potion of the array in memory.
3. Many functions/methods in a program are rarely used, so it is not necessary to keep the instructions of such functions in memory until the program calls the function(s).

In other words, there are many instances in which we could choose to keep a certain program page on disk rather than loading it to memory.

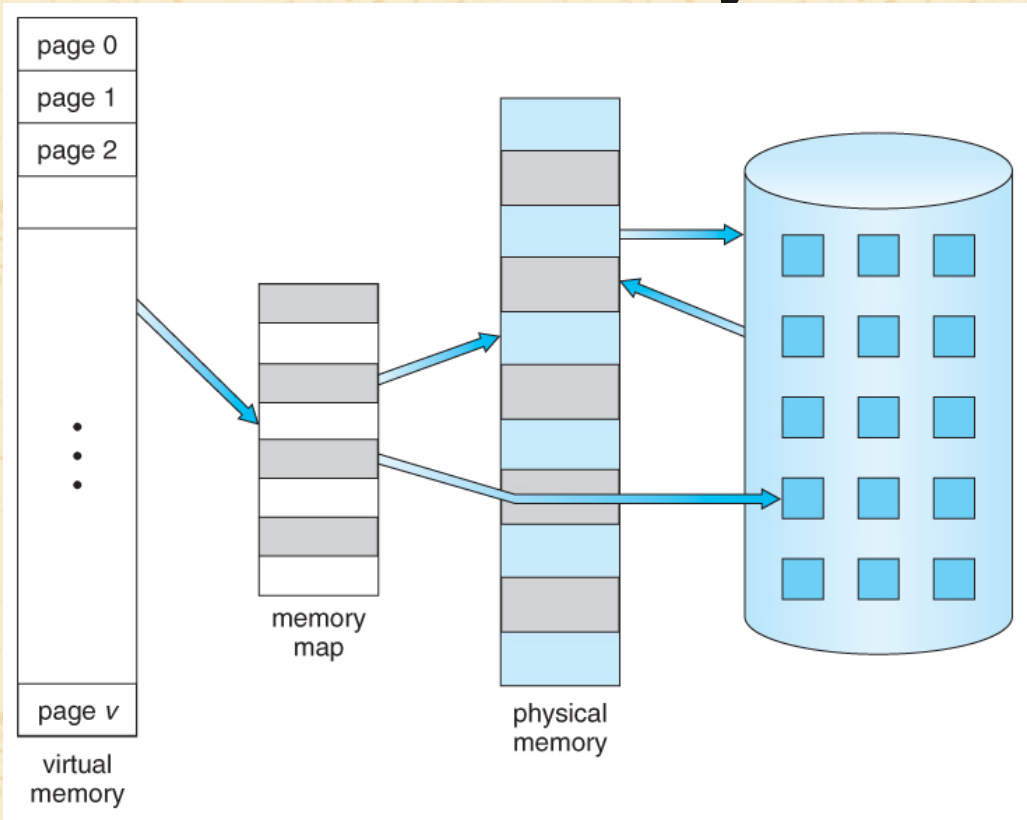
Virtual Memory

Benefits of loading only a portion of a program's pages into memory:

1. Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer. Loading all the page will crash the program, so if we decide to load only some pages at a time, the program could successfully execute.
2. Because each program is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
3. Less time is needed for swapping programs in and out of RAM during context switches.

The figure on the following slide shows the general layout of virtual memory.

Virtual Memory



General layout of virtual memory. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Virtual Memory

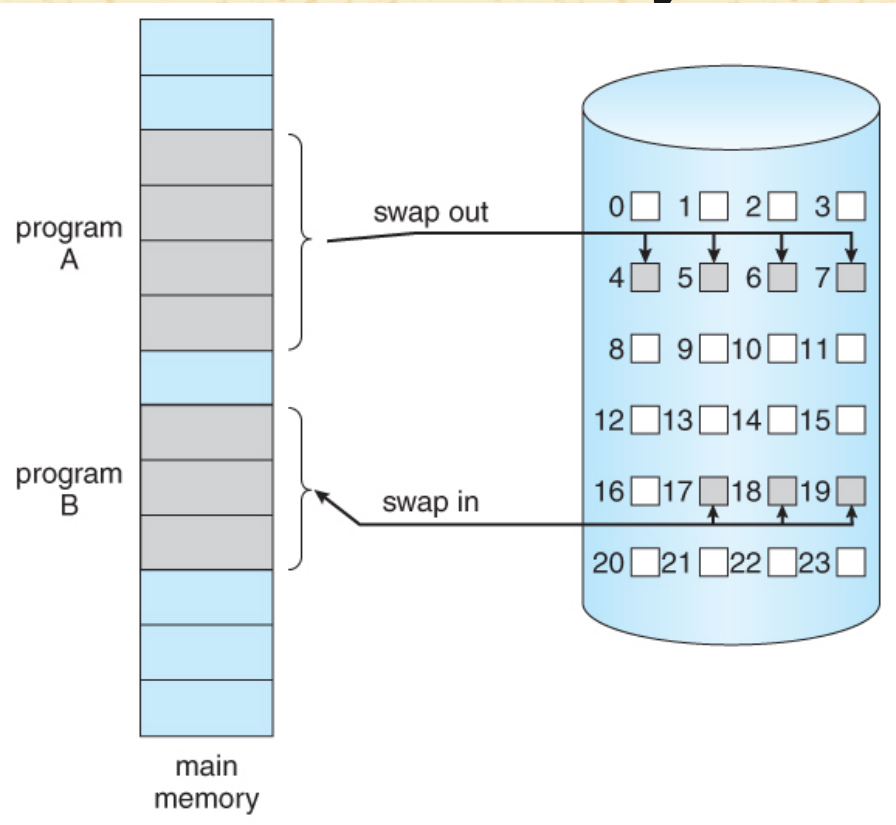
When a program is loaded to memory, but its pages are not loaded all at once (but only on demand) is called **Demand Paging**. That is, only when the program needs the data/code on a specific page, this page will be copied from disk to memory.

We call the mechanism behind such paging a **lazy pager** (or **lazy swapper**.)

To tell what pages are currently loaded to memory and which ones aren't, we add another column to the page table called the **invalid bit**. If a page is loaded onto a main memory frame, the bit will be 1 (valid); if it isn't loaded, the bit will be 0 (invalid). The OS will use this column to learn which pages are present in memory at the current moment (= **memory resident** pages).

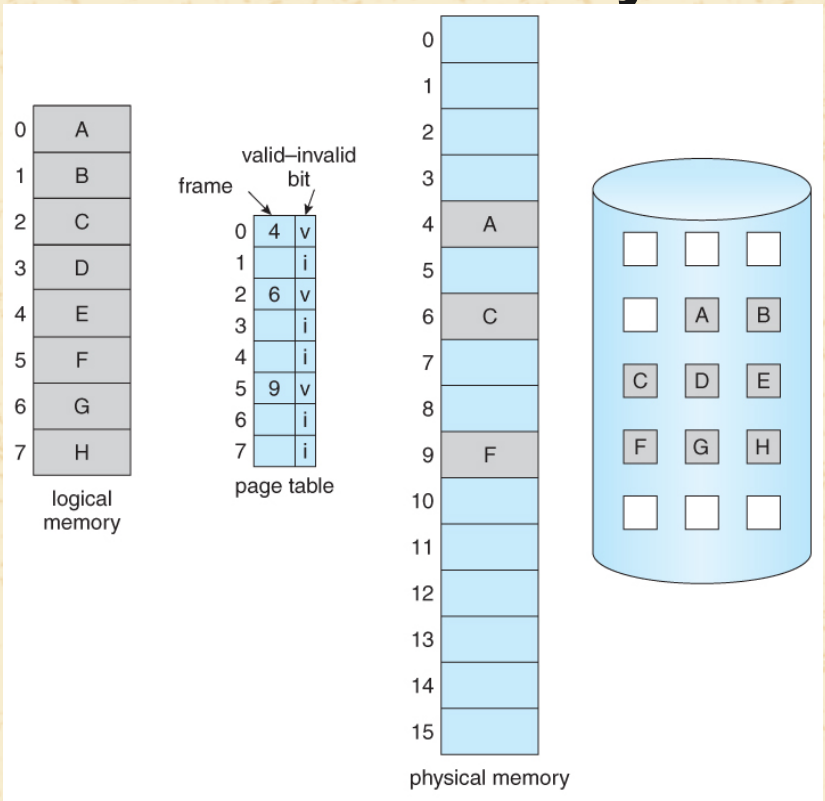
In case a program needs to use a page that is not currently in memory, a **page fault** trap (= software interrupt) will happen. To handle this interrupt, the OS will copy the content of the missing page from disk to memory.

Virtual Memory



Swapping pages in and out. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Virtual Memory



Some pages are out of memory. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Virtual Memory

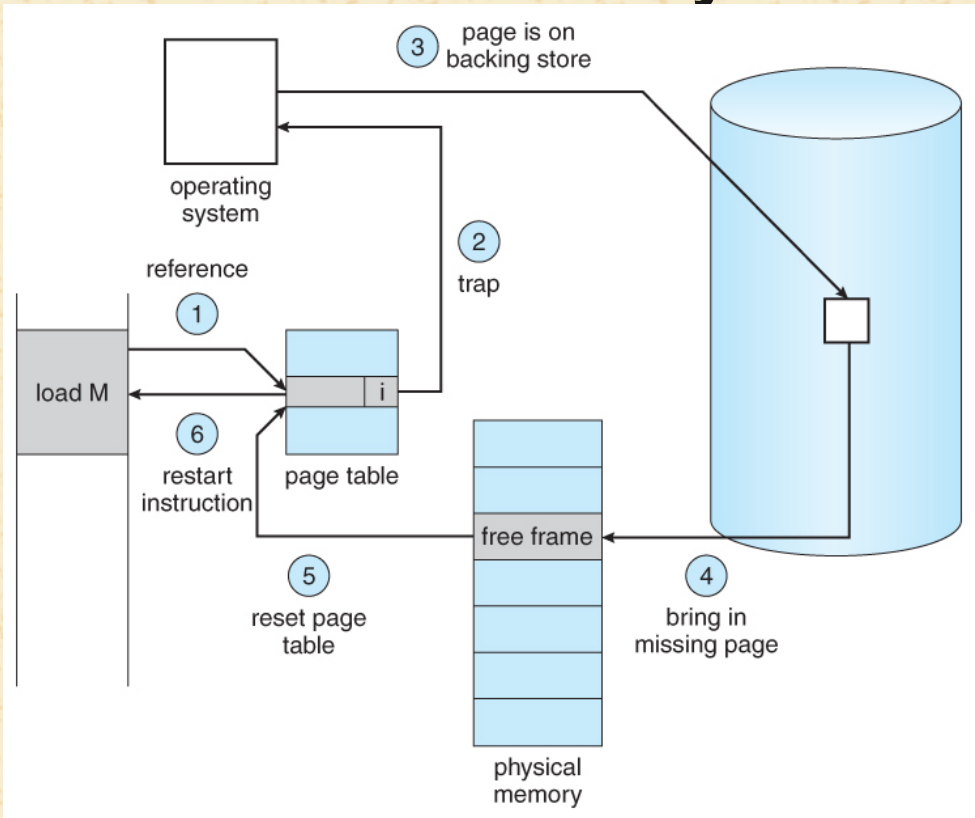
Any questions?

Virtual Memory

An OS handles a page fault trap as follows:

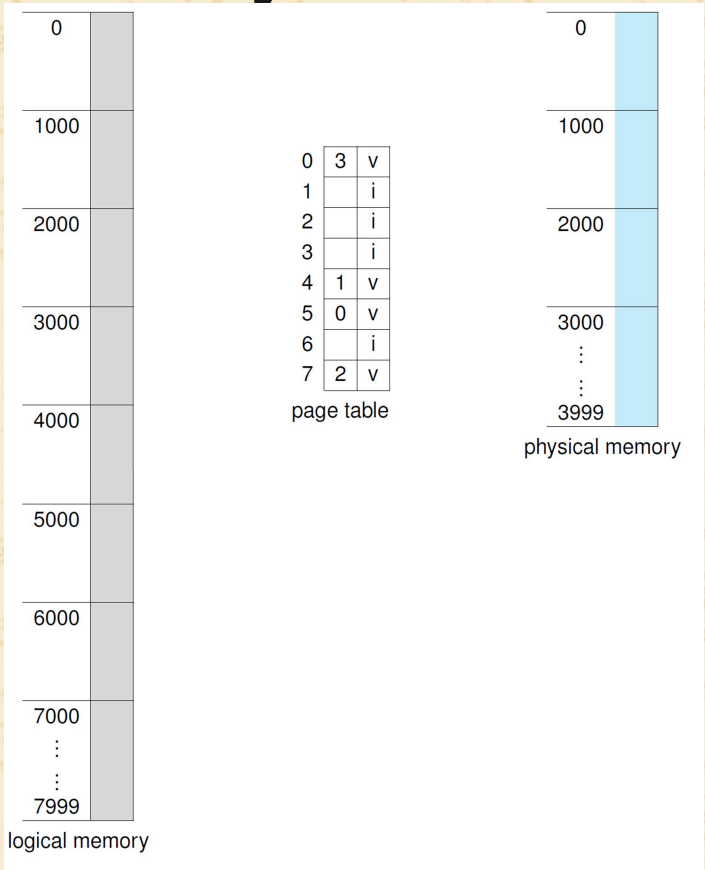
1. The memory address requested is first checked, to make sure it was a valid memory request (that is, that this address doesn't belong to another program.)
2. If the reference was invalid, the program is terminated. Otherwise, the page must be **paged in** (= brought from disk to memory.)
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk, during which the program's execution is paused.
5. When the copy from disk finishes, the program's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted *from the beginning*, and this program will continue executing on the CPU.

Virtual Memory



Steps in handling a page fault. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Virtual Memory: Practical Example



Virtual Memory: Practical Example (click to open in a new tab.) Miriam Briskman, CC BY-NC 4.0

Virtual Memory: Practical Example

The figure on the previous slide shows a logical memory space of 8 pages, a physical memory space of 4 frames, and the page table that maps the former to the latter. The size of each page and each frame is 1000 (1000 addresses/bytes).

Let's look at that diagram and answer the following questions:

a. The CPU tries to access the physical address that relates to the logical address 7485. Will a page fault happen? Explain how the computer will check this.
b. If there is no page fault, compute the physical address to which the logical address 7485 is mapped. Otherwise, if a page fault happens, briefly explain what the operating system will do in such a case using at least 1 sentence.
c. Answer the questions in parts (a) and (b) for the logical address 159.
d. Answer the questions in parts (a) and (b) for the logical address 3389.

Virtual Memory: Practical Example

Any questions?

Virtual Memory

A page fault incurs slowdown in the activity of the OS. We can find the new **average memory access time** that takes such a slowdown into account. To calculate it, we need to know (1) the normal average memory access time (which we calculated on slide 26), (2) the time it takes to handle a page fault, and (3) the **page fault rate** on a scale from 0 to 1 (the fraction of memory accesses in which page faults happen).

Examples:

- If the normal average memory access time is 200 ns, the page fault handling time is 8000000 ns, and the page fault rate is 0.001, the updated average memory access time would be $0.001 \cdot 8000000 + (1 - 0.001) \cdot 200 = 8000 + 199.8 = 8199.8$ ns (about 8.2 microseconds), which is about 40 times greater than 200 ns!
- If the normal average memory access time is 50 ns, the page fault handling time is 8000000 ns, and the page fault rate is 0.002, the updated average memory access time would be $0.002 \cdot 8000000 + (1 - 0.002) \cdot 50 = 16,000 + 49.9 = 16,049.9$ ns (about 16 microseconds), which is about 320 times greater than 50 ns!

Virtual Memory

Any questions?

Cache Schemes

Recall that **cache** is a small RAM-like chip that is placed between the CPU and RAM and serves as a faster memory storage than RAM.

We already learned a few **memory locality** principles, which tell that, if we accessed a certain variable in main memory, there's a high chance that we'll access another near-by data.

We will now learn 3 different cache data organization methods called **cache schemes**, which instruct the CPU where to find data in cache.

Depending on which one of these schemes is used on a certain computer, the CPU will know how to check whether a particular variable or program instruction is stored in cache or not. If it is found in cache - hurray! we don't need to spend time to search RAM (which takes longer time to access than cache.) Otherwise, if that data isn't currently present in cache, we have no choice but to get it from RAM.

Cache Schemes: Direct-Mapped Cache

If the cache on a computer is **direct-mapped**, it means that every block (chunk) of data in main memory will be mapped to exactly one particular block in cache. The size of every main memory block is equal to the size of every cache block.

Since the cache device is smaller than RAM, it means that cache contains fewer blocks than RAM does. For example, RAM might contain $2^{10} = 1024$ blocks of data, while cache contains only $2^4 = 16$ blocks of data.

This means that multiple of RAM blocks will map to the same cache block: only one of these RAM blocks can be located in the cache block at any given time. Moreover, to know to which cache block a specific memory block is mapped, we need to do the following calculation: $n \mod c$, where n is the RAM block for which we are checking, and c is the total number of cache blocks. That is, the remainder of the division $\frac{n}{c}$ is the number of the target cache block.

Example: If RAM has 16 blocks, while cache has 4 blocks, then RAM blocks $0, 4, 8, \text{ and } 12$ are mapped to cache block number 0; RAM blocks $1, 5, 9, \text{ and } 13$ are mapped to cache block number 1; RAM blocks $2, 6, 10, \text{ and } 14$ are mapped to cache block number 2; and RAM blocks $3, 7, 11, \text{ and } 15$ are mapped to cache block number 3.

Cache Schemes: Direct-Mapped Cache


```
RAM Block #   Cache Block # | RAM Block #   Cache Block #
    0 ------------> 0       |     8 ------------> 0
    1 ------------> 1       |     9 ------------> 1
    2 ------------> 2       |    10 ------------> 2
    3 ------------> 3       |    11 ------------> 3
    4 ------------> 0       |    12 ------------> 0
    5 ------------> 1       |    13 ------------> 1
    6 ------------> 2       |    14 ------------> 2
    7 ------------> 3       |    15 ------------> 3
```

So, to find to which cache block the RAM block 13 is mapped, for example, we'll divide 13 by 4 (the total number of cache blocks,) which is equal to 3 with a remainder of 1, so 1 will be that cache block: $13 = 4 \cdot 3 + 1$.

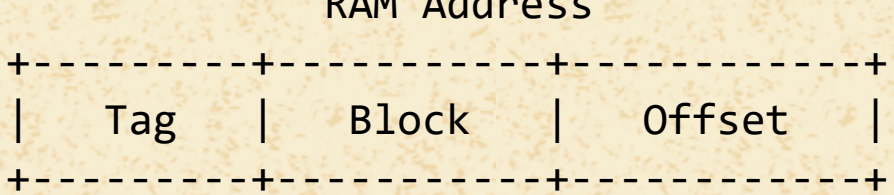
As we said before, only one of the RAM blocks that correspond to a cache block will be found in that cache block. For example, only one of the RAM blocks $1, 5, 9, \text{ and } 13$, for instance, RAM block 5, will be located in cache block number 1.

Cache Schemes: Direct-Mapped Cache

To find whether a certain variable is located in cache or not, the CPU needs to look into the main memory address of the variable that it is searching for. Then, based on the tag bits in that address (see more below,) the CPU will check inside cache if the variable is present in cache or not.

For this purpose, the main memory address is divided into sections:

1. The **offset** number, which represents the depth of a byte inside a block,
2. The **block** number, which is the cache block to which the RAM block where the variable is stored is mapped to, and
3. The **tag** number, which, combined with the block number, represents the RAM block where the variable is stored.



Cache Schemes: Direct-Mapped Cache

The CPU does the following check on a given RAM address:

1. The CPU extracts the **block** bits from the RAM address.
2. Using these **block** bits, the CPU accesses the cache block whose number is given by the block bits.
3. Each block in cache stores a tag number. The CPU compares the tag of the RAM address with the tag of the cache block:
    - If both tags are the same, it means that the variable is inside cache: hurray! We have a **cache hit** scenario, so the CPU can access it by plugging the offset number into cache.
    - If the tags are different, this means that a different RAM block is currently in cache (not the one that we are looking for.) We have a **cache miss** scenario, so we have to access main memory and copy the *entire* wanted RAM block into cache so that the variable we are looking for is successfully put in cache.

The reason we copy entire RAM blocks into cache rather than individual variables is because of the locality principle: if we use a certain variable, chances are that we'll need to use this variable again (so we'd better put it in cache.) Moreover, chances are that we'll also need to use variables located nearby the current variable (in the same block,) so we copy the entire block of data.

Cache Schemes: Direct-Mapped Cache

Example of finding the format of a RAM address (the numbers of bits in each of the tag, block, and offset fields:)

Suppose that a computer using direct mapped cache has 2^{10} bytes of main memory and a cache of 32 blocks, where each cache block is of the size of 8 bytes.

Now, let's find the format of a RAM address on this computer:

- Since each block is of the size of 8 bytes, the **offset** needs to be of the size of 3 bits since $2^3 = 8$ bytes can be uniquely represented using 3 bits.
- Since there are 32 cache blocks, the **block** field is of size 5 bits since $2^5 = 32$ blocks can be uniquely represented by a binary number of 5 bits.
- Finally, since a RAM address contains 10 bits (because the RAM size is 2^{10} bytes,) to find the **tag** size, simply subtract 3 and 5 from 10: $10 - 3 - 5 = 2$, so the tag consists of 2 bits.

Cache Schemes: Direct-Mapped Cache

Here's a quick diagram of the format of the RAM address number:

```
    RAM Address
 Tag   Block  Offset
2 bits 5 bits 3 bits
+----+-------+-----+
| 00 | 00000 | 000 |
+----+-------+-----+
```

Using the details we are given in the example, we can also find into how many blocks our main memory is divided: because the size of a main memory block is always equal the size of any cache block, we divide the total number of bytes in main memory by the size of a cache block, thus receiving: $\frac{2^{10}}{8} = \frac{2^{10}}{2^3} = 2^{10-3} = 2^7 = 128$ total RAM blocks.

Cache Schemes: Direct-Mapped Cache

Now, if the CPU needs to check whether the variable with RAM address: 1011011001 is currently located in cache or not, it will divide the bits of this address according to the format above.

```
    RAM Address
 Tag   Block  Offset
+----+-------+-----+
| 10 | 11011 | 001 |
+----+-------+-----+
```

Then, the CPU will check if the tag number 10 is the same as the tag of the cache block 11011:

- If yes, the CPU will access the variable at depth 001 inside cache.
- If not, the CPU will need to access the variable inside RAM at address 1011011001. Afterwards, it will copy the entire corresponding RAM block of data (block number 1011011) into cache block number 11011: this is one of the $2^7 = 128$ RAM blocks, and it is of the size of 8 bytes (just like the size of any cache block.)

Cache Schemes: Direct-Mapped Cache

Any questions?

Cache Schemes: Fully Associative Cache

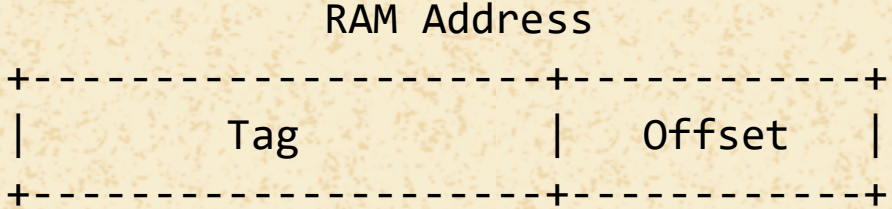
In a **fully associative** cache, every block of RAM can be mapped inside any cache block, without restriction. For example, now we might find a certain RAM block in cache block 0; then, at some future point, we will find it inside cache block 3, etc.

This concept is pretty similar to paging, in which we can place any page of any program inside any frame in main memory. Note, however, that there is usually no connection between RAM pages and RAM blocks: blocks are small chunks of data that we want to fit into cache, while pages are larger chunks of data that we sometimes keep on the disk.

So, to find whether a certain variable searched by the CPU is present in cache or not, we'll need to search the entire cache for that RAM block, which will take a lot of electric energy. If we want to make this search go in parallel, the computer needs to be installed with some complex hardware that includes plenty of **comparator** circuits, one for each block of cache, which are used to compare numbers (specifically, tags) and a **multiplexer** circuit to extract the variable that we are looking for if the RAM block for which we are searching is indeed located in cache. This makes a fully associative cache expensive not only in terms of how much electricity it needs to use but also the building materials and time needed to build such a cache device.

Cache Schemes: Fully Associative Cache

Because each RAM block can be copied anywhere in cache, we don't need to keep track of cache block numbers as part of a given RAM address, so the format of a RAM address will consist of a **tag** field and an **offset** (depth) field:



If the CPU, for example, wants to check if the variable with the RAM address 1011011001 is currently located in cache or not, it will check if any of the cache blocks has the tag of 1011011. If yes, the CPU will get the variable at depth 001 in that cache block. If not, the CPU will have to get the variable from RAM at address 1011011001.

Cache Schemes: Set-Associative Cache

In a **set-associative** cache, we divide cache blocks into **sets** of blocks, and every block of RAM can be mapped to one of the blocks inside a particular set of cache blocks.

Because a specific RAM block could only be found inside a limited set of cache blocks, we don't need to search the entire cache: we just need to search that one set of cache blocks. The resulting cache is both cheaper because we don't need to use as many comparator circuits (we just need a number of comparator circuits equal to the number of sets, not the number of all cache blocks) and uses less electricity. In other words, a set-associative cache is more efficient than a fully associative cache.

One reason for deciding to use a set-associative cache over a direct-mapped cache is that, in a direct-mapped cache, if we need to access a variable x from a RAM block that maps to cache block #2, for example, and then need to access a different variable y from another RAM block that maps to the same cache block #2, we will need to remove the 1st memory block and replace it with the 2nd one. If, at some point, we need to access x, then y, then x again, then y again, etc., we will find ourselves replacing the memory block each time, which takes time and slows down the work of the computer. However, with a set-associative cache, we can include both memory blocks in cache blocks that belong to the same set.

Cache Schemes: Set-Associative Cache

Because each RAM block can be present in one of the blocks that belong to the set, the format of a RAM address will consist of a **tag**, **set**, and **offset** (depth) fields:

```
              RAM Address
+---------+-----------+------------+
|   Tag   |    Set    |   Offset   |
+---------+-----------+------------+
```

If cache is divided into 8 sets, and if the CPU, for example, wants to check if the variable with the RAM address 1011011001 is currently located in cache or not, it will check if any of the cache blocks in set 011 has the tag of 1011. If yes, the CPU will get the variable at depth 001 in that cache block. If not, the CPU will have to get the variable from RAM at address 1011011001.

```
    Tag         Set         Offset
+---------+-----------+------------+
|  1011   |    011    |     001    |
+---------+-----------+------------+
```

Cache Schemes: Set-Associative Cache

Any questions?

Cache: Average Access Time

Since we discussed cache, we can now talk about finding another type of **average memory time** that takes the existence of the cache into account. To calculate it, we need to know (1) the normal average memory access time (which we calculated on slide 26), (2) the time it takes to access cache, and (3) the **cache hit rate** on a scale from 0 to 1 (the fraction of cache accesses in which we succeed in finding a variable.)

Examples:

- If the normal average memory access time is 200 ns, the cache access time is 5 ns, and the cache hit rate is 99%, then the average access time would be $0.99 \cdot 5 + (1 - 0.99) \cdot (5 + 200) = 4.95 + 2.05 = 7$ ns.
- If the normal average memory access time is 50 ns, the cache access time is 4 ns, and the cache hit rate is 90%, then the average access time would be $0.90 \cdot 4 + (1 - 0.90) \cdot (4 + 50) = 3.6 + 5.4 = 9$ ns.

Cache: Average Access Time

Any questions?

Cache Writing Methods

Often, we don't only read the value of a variable but also change it. A **dirty cache block** is a cache block that has been modified (e.g., by having one of its variables changed) in the cache but not yet updated in the main memory.

There are two general approaches as to when we should update RAM with those changes. They are called **cache write policies** or **methods**:

- **Write-Through:** Every change in cache is immediately done to main memory, too, ensuring consistency between cache and memory. This method increases memory traffic but simplifies data management.
- **Write-Back:** Modified data is written to main memory only when the cache block is evicted (= replaced by another RAM block,) which reduces memory traffic. A **dirty bit** is used to track whether a block needs to be written back, or if it's up to date in RAM.

Different computer architectures will benefit from different policies: the builders of an architecture need to run experiments to see which works faster on their particular computer.

Cache Writing Methods

Any questions?

Memory: Sources

Bergmann, Seth D. (2018). Chapter 8. *Computer Organization with MIPS*. License: CC BY-NC-ND: Attribution-NonCommercial-NoDerivs.

CS301: Computer Architecture. Unit 6.1, Unit 6.2, and Unit 6.3. *Saylor Academy*, 2010. License: CC BY: Attribution.

Viswanath, Divakar. Chapter 4. *Scientific Programming and Computer Architecture*. The MIT Press, 2017. License: CC BY-NC-ND: Attribution-NonCommercial-NoDerivs.

Wienand, Ian. (2019). Section 3.2. *Computer Science from the Bottom Up*. License: CC BY-SA: Attribution-ShareAlike.