Systems and Networking I

Applied Computer Science and Artificial Intelligence

2025-2026

SAPIENZA
UNIVERSITÀ DI ROMA

Gabriele Tolomei

Computer Science Department

Sapienza Università di Roma

tolomei@di.uniroma1.it

Paging + Segmentation

- Paging (OS' view of memory)
  - Divide memory into fixed-size pages and map them to physical frames
- Segmentation (compiler's view of memory)
  - Divide process into logical segments (e.g., code, data, stack, heap)
- Combine paging with segmentation
  - Segmented Paging

Paging + Segmentation

- Paging (OS' view of memory)
  - Divide memory into fixed-size pages and map them to physical frames
- Segmentation (compiler's view of memory)
  - Divide process into logical segments (e.g., code, data, stack, heap)
- Combine paging with segmentation
  - Segmented Paging

So far, the entire virtual address space of a process was assumed to fit and be all in memory

Virtual Memory

- In practice, most real processes do not need all their pages loaded in memory, or at least not all at once, e.g.,:

Virtual Memory

- In practice, most real processes do not need all their pages loaded in memory, or at least not all at once, e.g.,:
  - Error handling code is not needed unless that specific error occurs, some of which are quite rare

Virtual Memory

- In practice, most real processes do not need all their pages loaded in memory, or at least not all at once, e.g.,:
  - Error handling code is not needed unless that specific error occurs, some of which are quite rare
  - Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays is actually used in practice

Virtual Memory

- In practice, most real processes do not need all their pages loaded in memory, or at least not all at once, e.g.,:
  - Error handling code is not needed unless that specific error occurs, some of which are quite rare
  - Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays is actually used in practice
  - Some features of certain programs are rarely used

Virtual Memory

- In practice, most real processes do not need all their pages loaded in memory, or at least not all at once, e.g.,:
  - Error handling code is not needed unless that specific error occurs, some of which are quite rare
  - Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays is actually used in practice
  - Some features of certain programs are rarely used

Virtual Memory uses backing storage (i.e., disk) to store unused pages and give the illusion of "infinite" space

Virtual Memory: Benefits

- The ability to load only the portions of processes that are actually needed (and only when needed) from disk has several benefits:

Virtual Memory: Benefits

- The ability to load only the portions of processes that are actually needed (and only when needed) from disk has several benefits:
  - Programs could be written for a much larger address space than physically exists on the computer

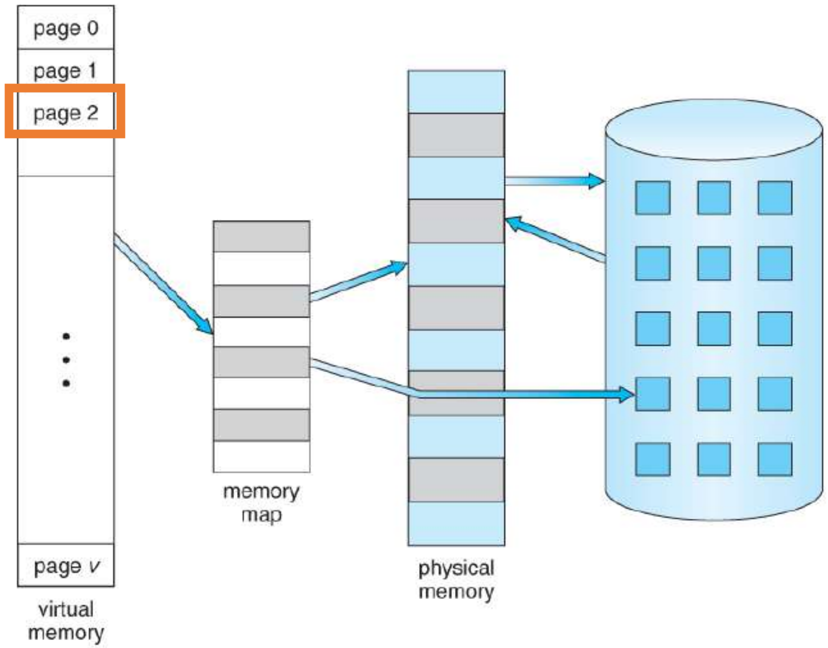
Virtual Memory: Benefits

- The ability to load only the portions of processes that are actually needed (and only when needed) from disk has several benefits:
  - Programs could be written for a much larger address space than physically exists on the computer
  - More memory is left for other programs, improving CPU utilization

Virtual Memory: Benefits

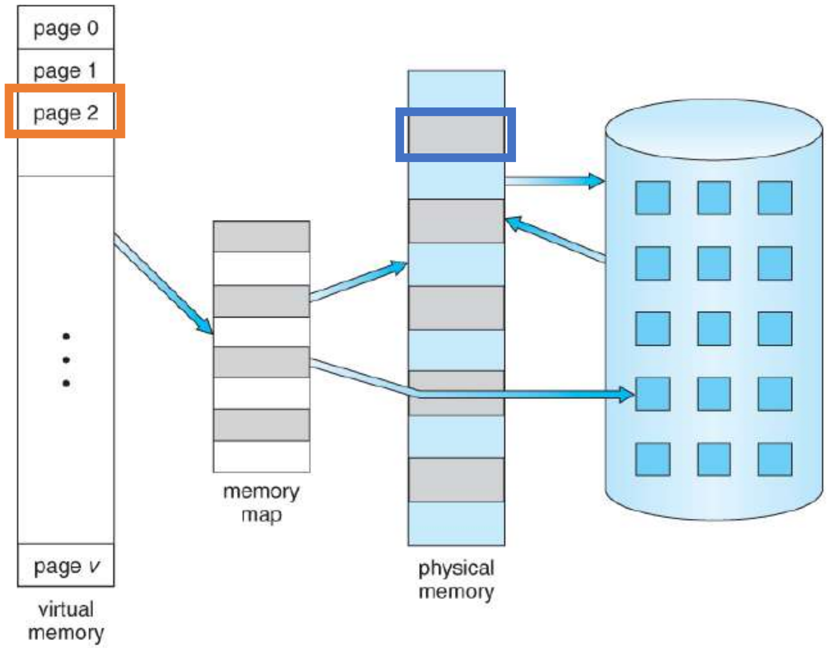
- The ability to load only the portions of processes that are actually needed (and only when needed) from disk has several benefits:
  - Programs could be written for a much larger address space than physically exists on the computer
  - More memory is left for other programs, improving CPU utilization
  - Less I/O is needed for swapping processes in and out of memory, speeding things up

Virtual Memory: The Big Picture



At any given time, each page can be:

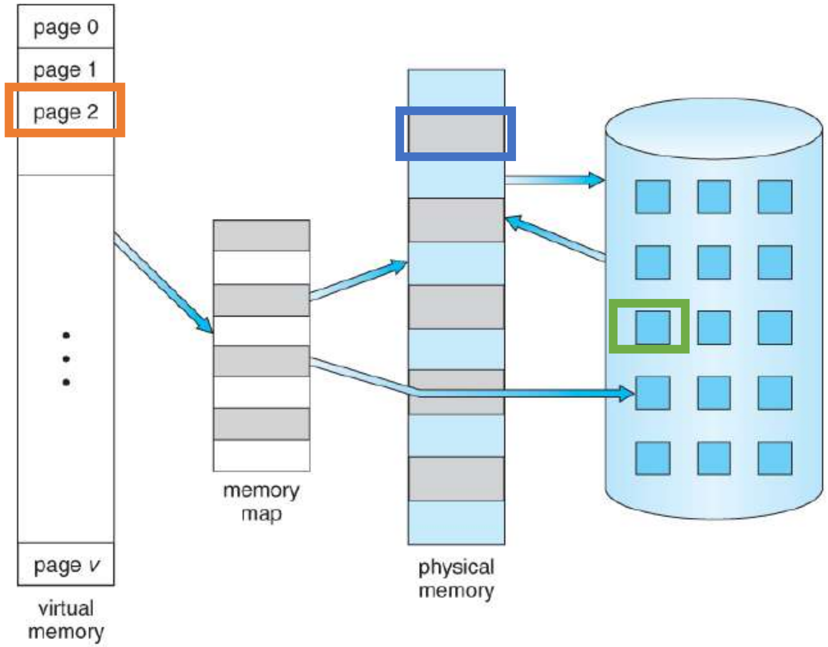
Virtual Memory: The Big Picture



At any given time, each page can be:

in memory (physical frame)

Virtual Memory: The Big Picture



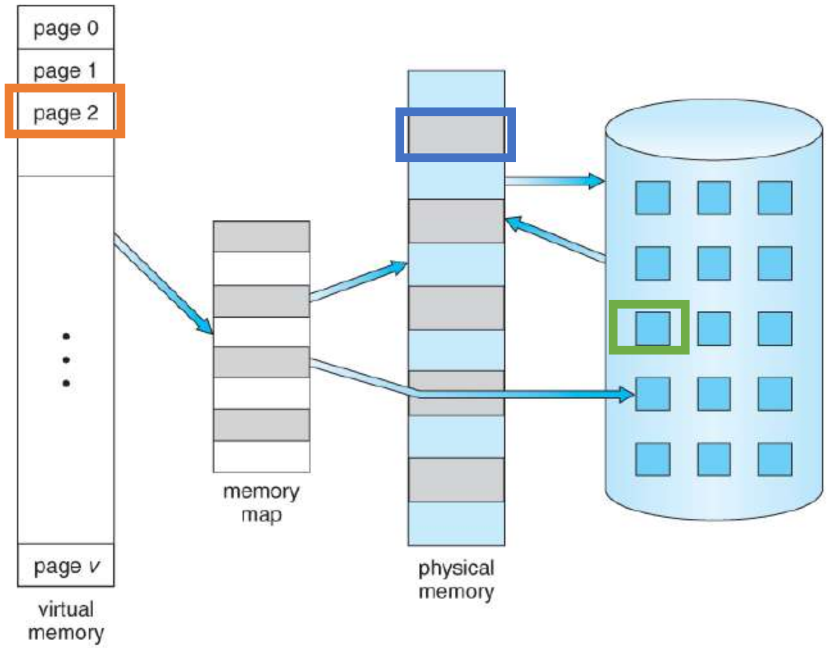
At any given time, each page can be:

in memory (physical frame)

on backing store (disk)

Virtual Memory: The Big Picture

virtual memory can be much larger than physical memory

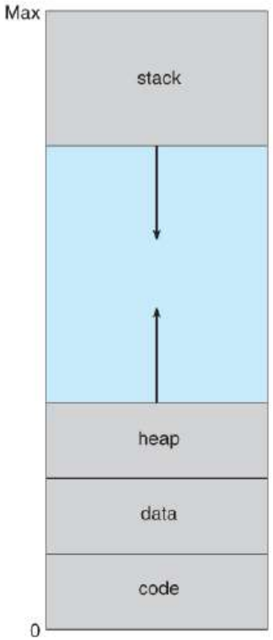


At any given time, each page can be:

in memory (physical frame)

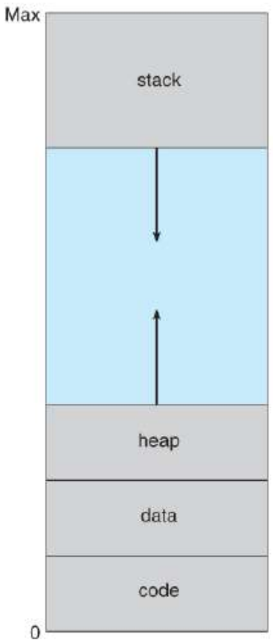
on backing store (disk)

The Sparseness of Virtual Address Space



Typically, virtual address space is highly sparse

The Sparseness of Virtual Address Space



Typically, virtual address space is highly sparse

A lot of virtual memory addresses remain unreferenced

```
int *arr = malloc(4 TB);
arr[0] = 1;
arr[500000000] = 2;
```

```
void f() {
    char big[4 * 1024 * 1024];    // 4 MB array
    big[0] = 1;                   // first page touched only
}
```

Virtual Memory: Idea

- Use the memory as a cache for the disk

Virtual Memory: Idea

- Use the memory as a cache for the disk
- The page table must also indicate if the page is on disk or in memory (just using a single invalid bit)

Virtual Memory: Idea

- Use the memory as a cache for the disk
- The page table must also indicate if the page is on disk or in memory (just using a single invalid bit)
- Once the page is loaded from disk to memory, the OS updates the corresponding entry of the page table along with the valid bit

Virtual Memory: Idea

- Remember: access to disk is extremely slower than access to memory

Virtual Memory: Idea

- Remember: access to disk is extremely slower than access to memory
- Therefore, memory accesses must reference pages that are in memory **with high probability**

Virtual Memory: The Locality Principle

- The 90÷10 rule claims that on a particular time frame, most of the memory references made by a process is around a small "area"

Virtual Memory: The Locality Principle

- The 90÷10 rule claims that on a particular time frame, most of the memory references made by a process is around a small "area"
- We call this area as the **working set** of the process

Virtual Memory: The Locality Principle

- The 90÷10 rule claims that on a particular time frame, most of the memory references made by a process is around a small "area"
- We call this area as the **working set** of the process
- Since the working set is fairly small compared to the whole virtual address space, it will likely fit in memory

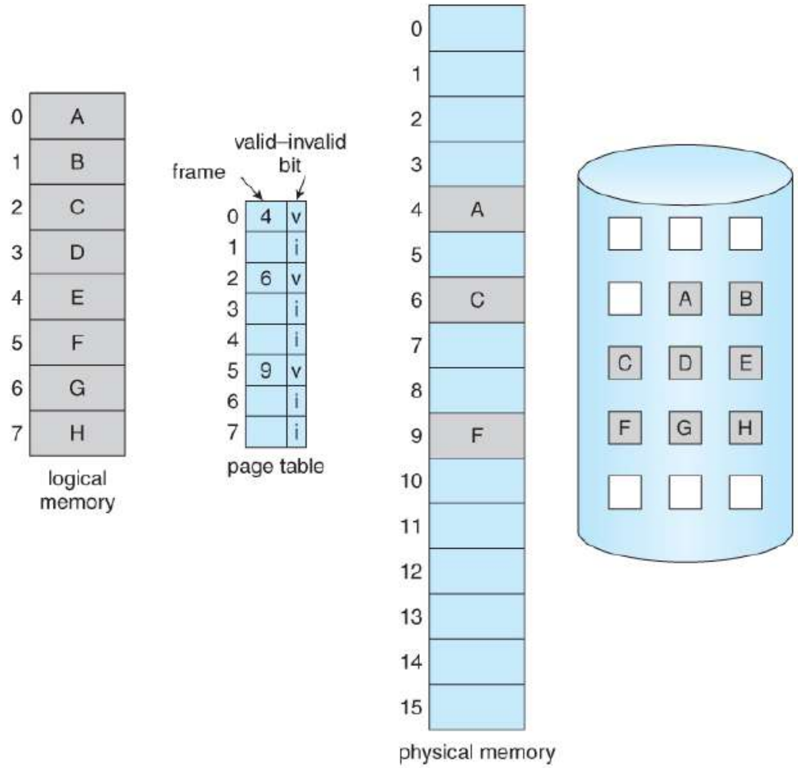
Virtual Memory: The Locality Principle

- Of course, during the lifetime of a process its working set may change (i.e., a process may eventually refer *all* of its virtual address space)

Virtual Memory: The Locality Principle

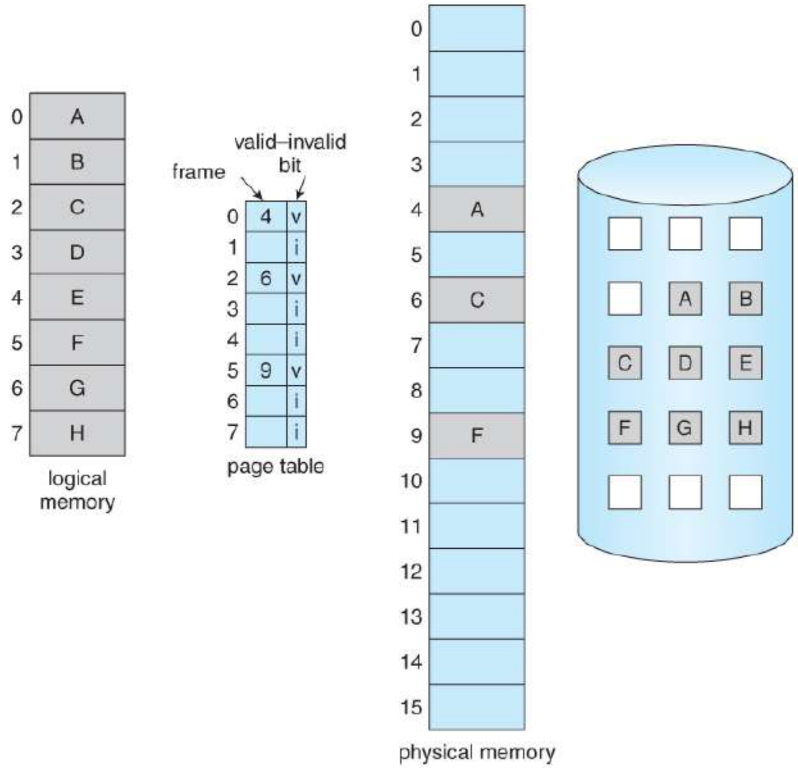
- Of course, during the lifetime of a process its working set may change (i.e., a process may eventually refer *all* of its virtual address space)
- But in a reasonably small time frame, the working set stays "the same"

Virtual Memory: Basic Concepts



At each logical memory reference, a page table lookup is performed as usual

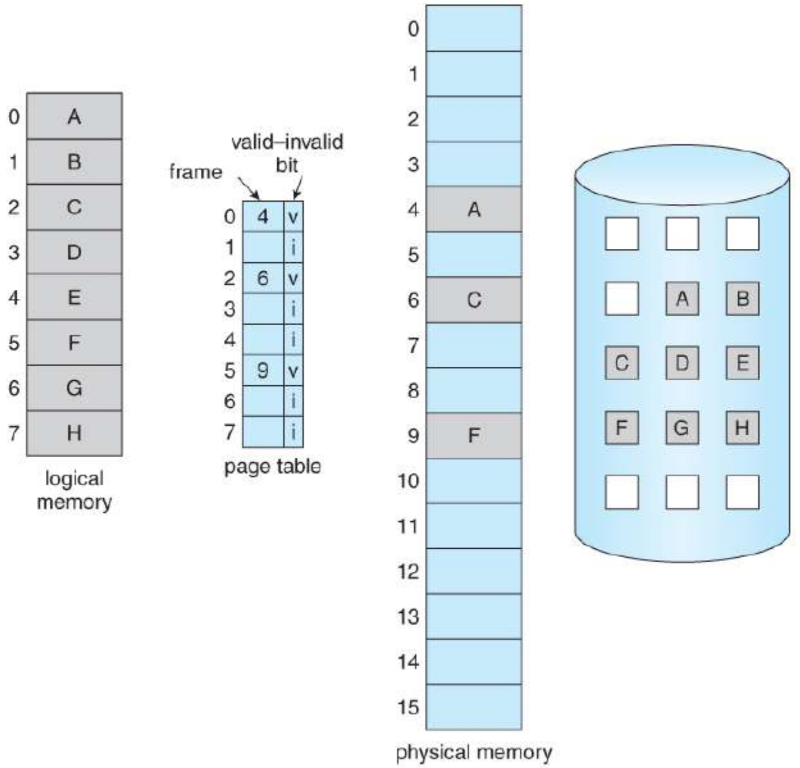
Virtual Memory: Basic Concepts



At each logical memory reference, a page table lookup is performed as usual

In the page table, the valid-invalid bit is checked for the corresponding entry

Virtual Memory: Basic Concepts

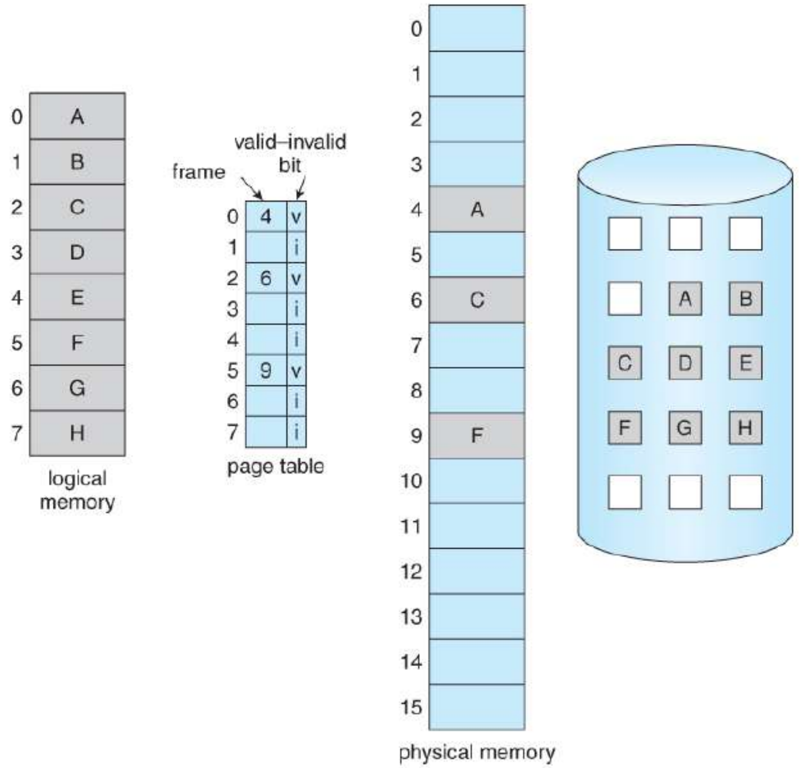


At each logical memory reference, a page table lookup is performed as usual

In the page table, the valid-invalid bit is checked for the corresponding entry

If the bit is set to 1 it means the page entry is valid (i.e., the requested page is in memory)

Virtual Memory: Basic Concepts



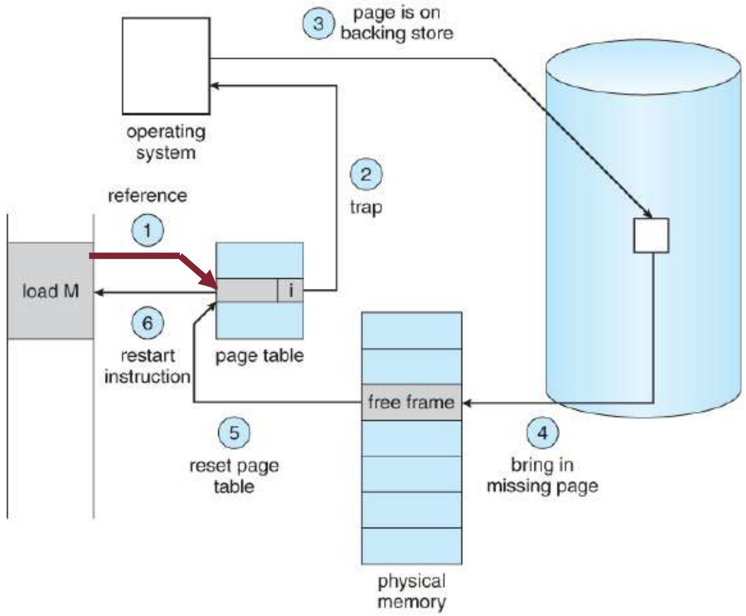
At each logical memory reference, a page table lookup is performed as usual

In the page table, the valid-invalid bit is checked for the corresponding entry

If the bit is set to 1 it means the page entry is valid (i.e., the requested page is in memory)

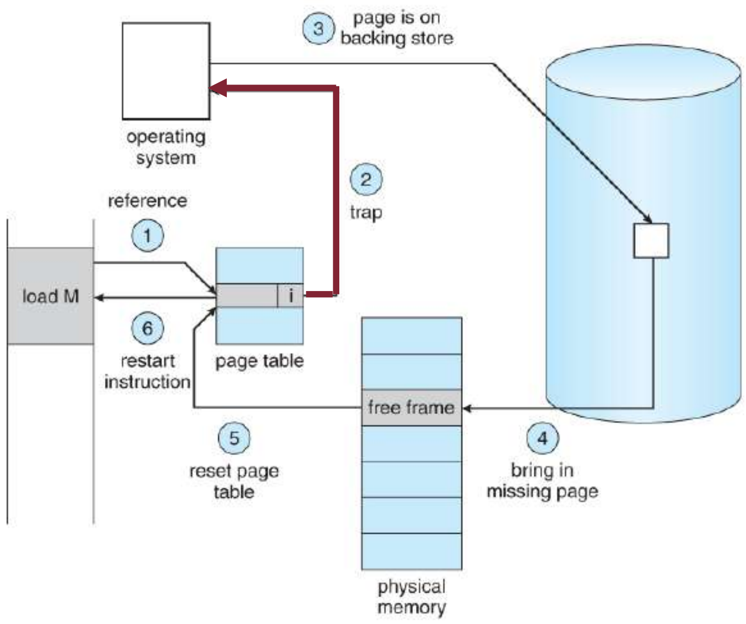
Otherwise, a page fault trap occurs, and the page has to be loaded (i.e., fetched) from disk

Page Fault Handling



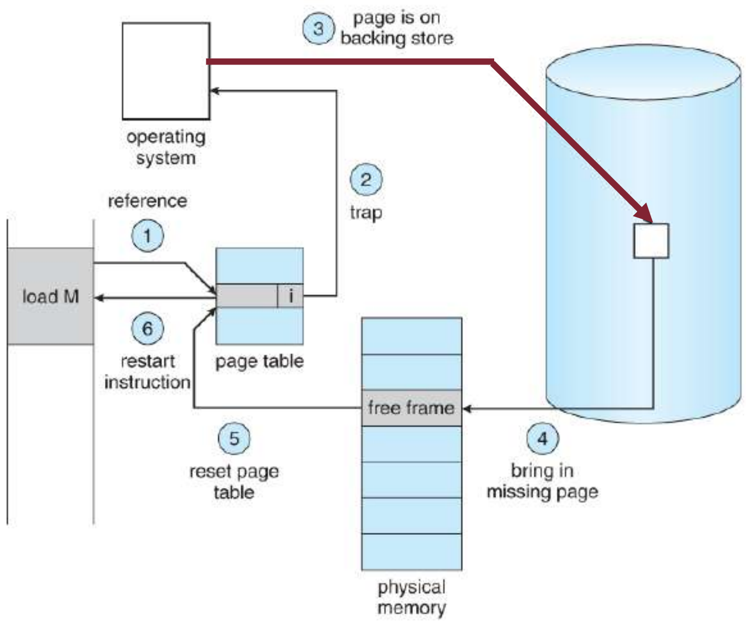
1. The memory address is first checked, to see if it is legitimate

Page Fault Handling



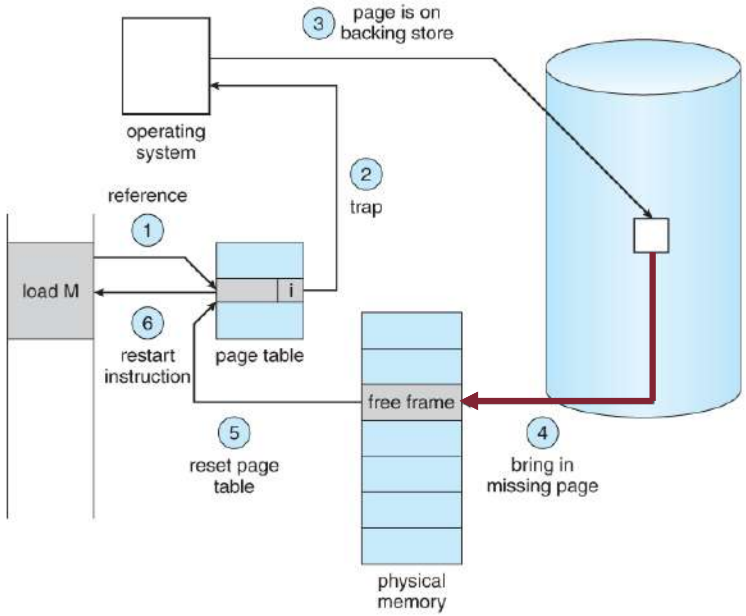
1. The memory address is first checked, to see if it is legitimate
2. If the address is legitimate the page must be fetched from disk

Page Fault Handling



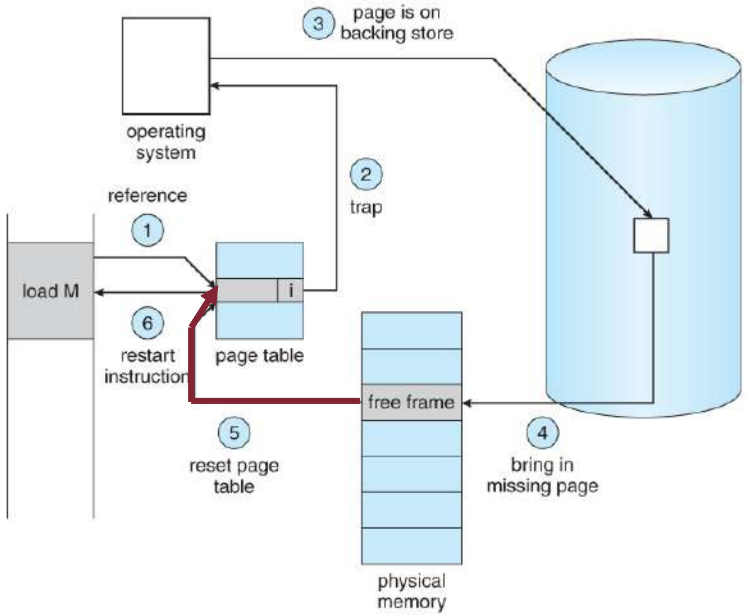
1. The memory address is first checked, to see if it is legitimate
2. If the address is legitimate the page must be fetched from disk
3. A free frame is located, possibly from a free-frame list (the OS might need to pick a frame to unload if all memory is full)

Page Fault Handling



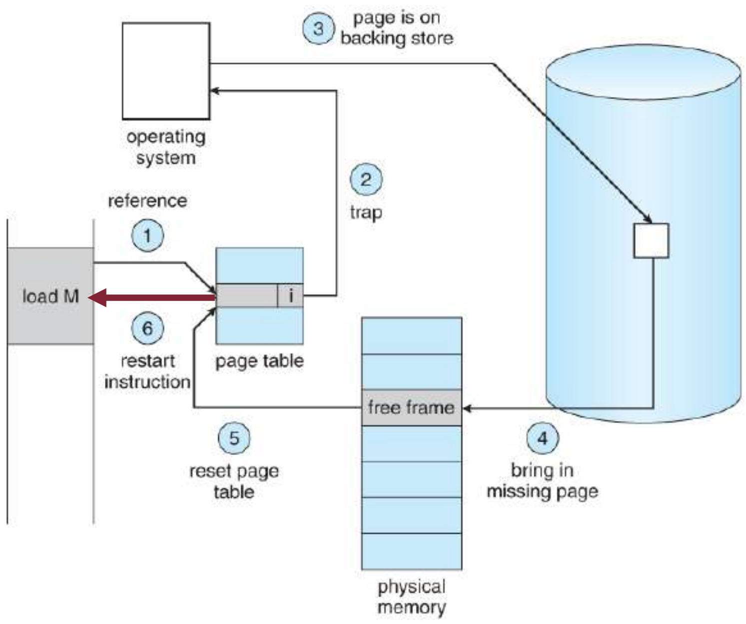
1. The memory address is first checked, to see if it is legitimate
2. If the address is legitimate the page must be fetched from disk
3. A free frame is located, possibly from a free-frame list (the OS might need to pick a frame to unload if all memory is full)
4. A disk operation is scheduled to bring in the necessary page from disk (the user process blocks on an I/O wait, allowing some other process to run)

Page Fault Handling



1. The memory address is first checked, to see if it is legitimate
2. If the address is legitimate the page must be fetched from disk
3. A free frame is located, possibly from a free-frame list (the OS might need to pick a frame to unload if all memory is full)
4. A disk operation is scheduled to bring in the necessary page from disk (the user process blocks on an I/O wait, allowing some other process to run)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the bit is set to valid

Page Fault Handling



1. The memory address is first checked, to see if it is legitimate
2. If the address is legitimate the page must be fetched from disk
3. A free frame is located, possibly from a free-frame list (the OS might need to pick a frame to unload if all memory is full)
4. A disk operation is scheduled to bring in the necessary page from disk (the user process blocks on an I/O wait, allowing some other process to run)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the bit is set to valid
6. The current process gets interrupted and the instruction that caused the page fault must be restarted from the beginning

Page Fault Handling: TLB Hit

- The TLB also uses the valid bit to indicate the fact that the page is in main memory

Page Fault Handling: TLB Hit

- The TLB also uses the valid bit to indicate the fact that the page is in main memory
- If we get a TLB hit but the frame is not actually in main memory, we have to go fetch the page from disk anyway!

Page Fault Handling: TLB Hit

- The TLB also uses the valid bit to indicate the fact that the page is in main memory
- If we get a TLB hit but the frame is not actually in main memory, we have to go fetch the page from disk anyway!
- TLB hit means the requested page entry is in the cache **and** the referenced frame is also in memory

Page Fault Handling: TLB Miss

- If the requested page is not in the cache (TLB miss) **but it is in memory**:

Page Fault Handling: TLB Miss

- If the requested page is not in the cache (TLB miss) **but it is in memory**:
  - The OS picks a TLB entry to replace and fills it with the new entry

Page Fault Handling: TLB Miss

- If the requested page is not in the cache (TLB miss) **and it is not even in memory** (i.e., it is on disk):

Page Fault Handling: TLB Miss

- If the requested page is not in the cache (TLB miss) **and it is not even in memory** (i.e., it is on disk):
  - The OS picks a TLB entry to replace and fills it with the new entry as follows
    - invalidates the TLB entry
    - performs page fault trap operations
    - updates the TLB entry
    - restarts the faulting instruction

Page Fault Handling: Faulty Address

- How does the OS figure out which page generated the fault?

Page Fault Handling: Faulty Address

- How does the OS figure out which page generated the fault?
- Architecture-dependent:
  - x86: hardware saves the virtual address that caused the fault (CR2 register)
  - On some platforms, OS gets only address of faulting instruction, must simulate the instruction and try every address to find the one that generated the fault

Page Fault Handling: Transparency

- Transparently restarting process execution after a page fault is tricky, since the fault may have occurred in the middle of an instruction

Page Fault Handling: Transparency

- Transparently restarting process execution after a page fault is tricky, since the fault may have occurred in the middle of an instruction
- To restart (from scratch) a faulty instruction the OS needs hardware support for saving:
  - The faulting instruction
  - The CPU state

Page Fault Handling: Transparency

- idempotent vs. non-idempotent instructions

Page Fault Handling: Transparency

- idempotent vs. non-idempotent instructions
- idempotent → just restart the faulting instruction (hardware saves instruction address during page fault)

Page Fault Handling: Transparency

- idempotent vs. non-idempotent instructions
- idempotent → just restart the faulting instruction (hardware saves instruction address during page fault)
- non-idempotent → much more difficult to restart
  - MOV [%R1], +(%R2) → increment the value of R2 and store it to memory address in R1
  - What if memory address [%R1] causes the page fault?
  - Cannot naively redo the instruction from scratch, otherwise R2 gets incremented twice

Page Fault Handling: Transparency

- Even harder when using instructions that are not easily undoable
  - E.g., instructions that are used to move a block of memory at once
  - The block may span multiple pages: some of them can be in memory while some others not
  - Pages that are in memory can be changed meanwhile a page fault occurs

Page Fault Handling: Transparency

- Even harder when using instructions that are not easily undoable
  - E.g., instructions that are used to move a block of memory at once
  - The block may span multiple pages: some of them can be in memory while some others not
  - Pages that are in memory can be changed meanwhile a page fault occurs

How to unwind those complicated side-effects?

Page Fault Handling: Transparency

- Even harder when using instructions that are not easily undoable
  - E.g., instructions that are used to move a block of memory at once
  - The block may span multiple pages: some of them can be in memory while some others not
  - Pages that are in memory can be changed meanwhile a page fault occurs

Ensure all the addresses within the block to be moved are in memory before executing the instruction

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction
  - A process may reference addresses belonging to different page at each step

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction
  - A process may reference addresses belonging to different page at each step
- Luckily, processes usually exhibit so-called **locality of reference**
  - **temporal** → if a process accesses an item in memory, it will tend to reference the same item again soon

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction
  - A process may reference addresses belonging to different page at each step
- Luckily, processes usually exhibit so-called **locality of reference**
  - **temporal** → if a process accesses an item in memory, it will tend to reference the same item again soon
  - **spatial** → if a process accesses an item in memory, it will tend to reference a close item again soon

Virtual Memory: Performance

$$t_{MA} = \text{physical memory access time}$$
$$t_{FAULT} = \text{time to handle a page fault}$$
$$p \in [0, 1] = \text{probability of page fault}$$
$$t_{ACCESS} = \text{effective time for each memory reference}$$

$$t_{ACCESS} = (1 - p) * t_{MA} + p * t_{FAULT}$$

Let's assume: $t_{MA} = 100$ nsec and $t_{FAULT} = 20$ msec $= 20,000,000$ nsec

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

Virtual Memory: Performance

$$t_{MA} = \text{physical memory access time}$$
$$t_{FAULT} = \text{time to handle a page fault}$$
$$p \in [0, 1] = \text{probability of page fault}$$
$$t_{ACCESS} = \text{effective time for each memory reference}$$

$$\boxed{t_{ACCESS} = (1 - p) * t_{MA} + p * t_{FAULT}}$$

Let's assume: $t_{MA} = 100$ nsec and $t_{FAULT} = 20$ msec $= 20,000,000$ nsec

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

This heavily depends on p!

Virtual Memory: Performance Example

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

What if only 1 every 1,000 memory references causes a page fault (i.e., p = 0.001)

Virtual Memory: Performance Example

$$t_{ACCESS} = (1-p) * 100 + p * 20,000,000$$

What if only 1 every 1,000 memory references causes a page fault (i.e., p = 0.001)
The access time increases from just 100 nsec up to ~20.1 microsec

200 times slowdown factor

Virtual Memory: Performance Example

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

What if we want the time access to be at most 10% slower than basic memory access?

Virtual Memory: Performance Example

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

What if we want the time access to be at most 10% slower than basic memory access?

We have to solve for p the following equation:

$$1.1 * 100 = (1 - p) * 100 + p * 20,000,000$$

Virtual Memory: Performance Example

$$t_{ACCESS} = (1-p) * 100 + p * 20,000,000$$

What if we want the time access to be at most 10% slower than basic memory access?

We have to solve for p the following equation:

$$1.1 * 100 = (1-p) * 100 + p * 20,000,000$$

$$1.1 * 100 = 100 - 100p + 20,000,000p =$$
$$19,999,900p = 110 - 100 =$$

To achieve that goal, we can tolerate at most 1 page fault every about 2 million accesses!

$$p = \frac{10}{19,999,900} = \frac{1}{1,999,990} \boxed{\approx 0,0000005 = 5 * 10^{-7}}$$

Virtual Memory: Performance Example

More generally, given t_{MA}, t_{FAULT}, and a threshold $\varepsilon > 0$ if we want to find p s.t.:

$$t_{ACCESS} = (1 + \epsilon) * t_{MA}$$

We substitute t_{ACCESS} and solve for p the resulting equation:

$$(1 - p) * t_{MA} + p * t_{FAULT} = (1 + \epsilon) * t_{MA} =$$
$$t_{MA} - p * t_{MA} + p * t_{FAULT} = t_{MA} + \epsilon * t_{MA}$$
$$p(t_{FAULT} - t_{MA}) = \epsilon * t_{MA} =$$

$$\boxed{p = \frac{\epsilon * t_{MA}}{t_{FAULT} - t_{MA}}}$$

Virtual Memory: Considerations

- So far, we have described how the OS (with the support of HW) manages page faults

Virtual Memory: Considerations

- So far, we have described how the OS (with the support of HW) manages page faults
- Still, the OS has to answer 2 fundamental questions:
  - When to load process' pages into main memory (**page fetching**)
  - Which page to remove from memory if this gets filled (**page replacement**)

Page Fetching Goals

- The overall goal is still to make physical memory look larger than it is
- Exploiting the locality reference of programs
- Keep in memory only those pages that is being used
- Keep on disk those pages that are unused
- Ideally, producing a memory system with the performance of main memory and the cost/capacity of disk!

Page Fetching Strategies

$\boxed{3 \texttt{ page fetching strategies}}$

Page Fetching Strategies

3 page fetching strategies

Startup

This is a special case where all the pages of the process are loaded at once

virtual address space cannot be larger than physical memory

Page Fetching Strategies

3 page fetching strategies

Startup

This is a special case where all the pages of the process are loaded at once

virtual address space cannot be larger than physical memory

Overlays

Let the programmer says when pages are loaded/removed

virtual address space can be larger than physical memory but hard and error-prone

Page Fetching Strategies

3 page fetching strategies

Startup

This is a special case where all the pages of the process are loaded at once

virtual address space cannot be larger than physical memory

Overlays

Let the programmer says when pages are loaded/removed

virtual address space can be larger than physical memory but hard and error-prone

Demand

Process tells the OS when it needs a page

The OS manages page requests

Page Fetching Strategies

3 page fetching strategies

Startup

This is a special case where all the pages of the process are loaded at once

virtual address space cannot be larger than physical memory

Overlays

Let the programmer says when pages are loaded/removed

virtual address space can be larger than physical memory but hard and error-prone

Demand

Process tells the OS when it needs a page

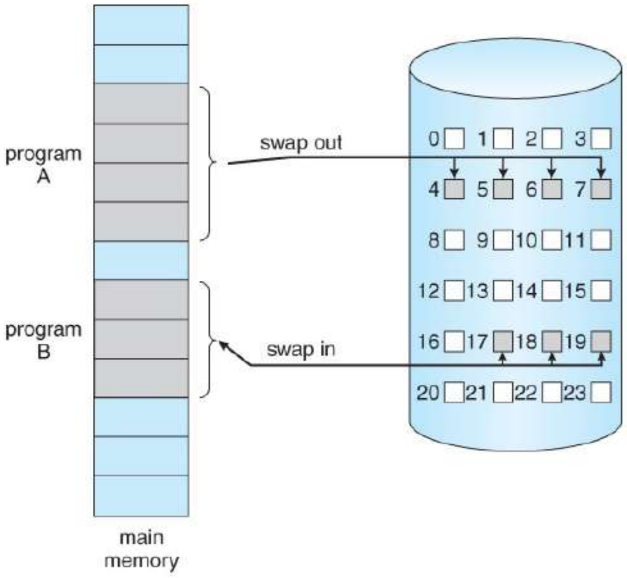
The OS manages page requests

Most modern OSs use demand fetching

(Pure) Demand Paging

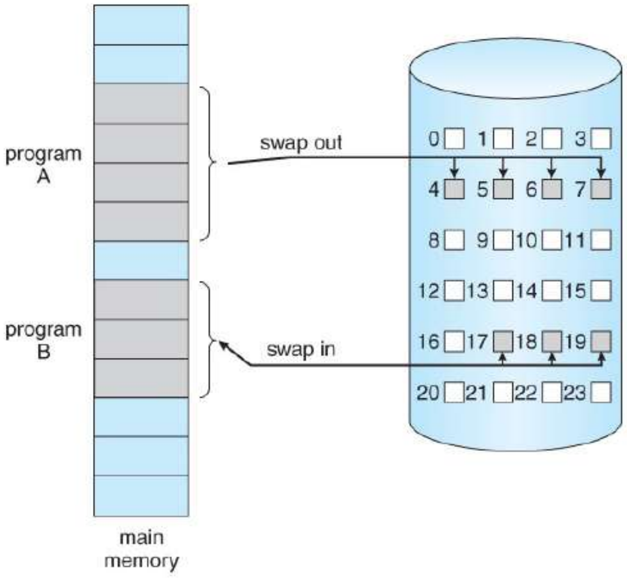
- When a process starts up, **none** of its pages are loaded
- Rather, a page is swapped in only when the process references it (upon a page fault)
- This is termed a **lazy swapper** or **pager**
- Opposite of loading all the pages at process startup!

Prefetching



The pager guesses when pages will be needed and load them ahead of time

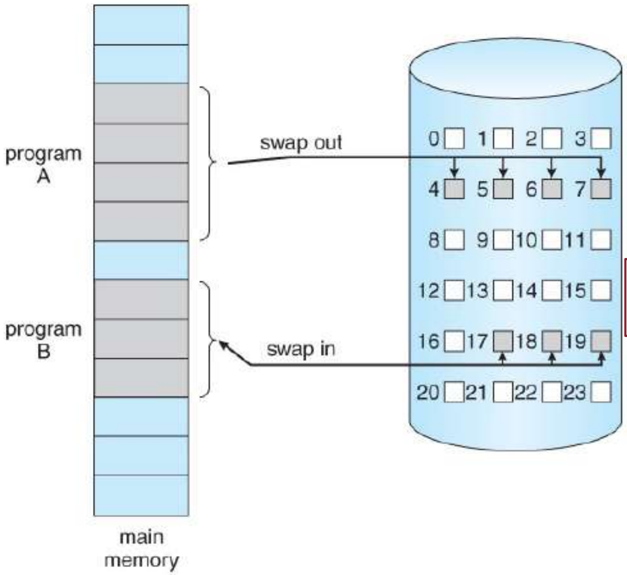
Prefetching



The pager guesses when pages will be needed and load them ahead of time

Trying to avoid page faults

Prefetching

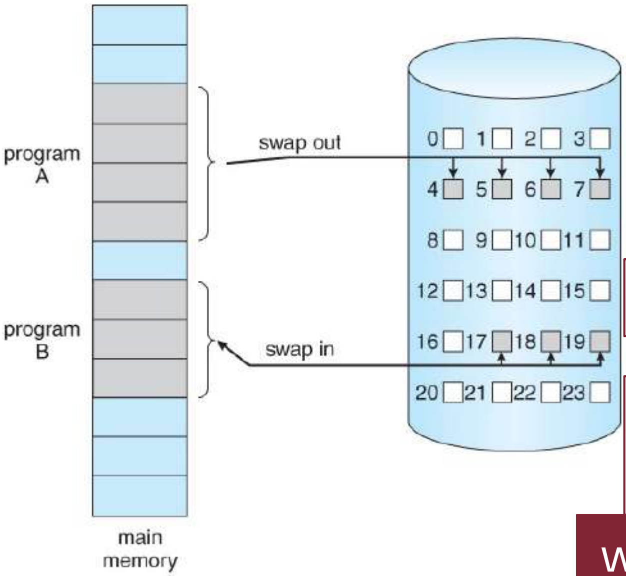


The pager guesses when pages will be needed and load them ahead of time

Trying to avoid page faults

Requires predicting the future → very hard!

Prefetching



The pager guesses when pages will be needed and load them ahead of time

Trying to avoid page faults

Requires predicting the future → very hard!

Possible approach: upon page fault, load many pages instead of only the faulty one

works if program accesses memory sequentially

Swap Space

- A portion of the disk reserved for storing pages that are evicted from memory

Swap Space

- A portion of the disk reserved for storing pages that are evicted from memory
- May be not part of the actual disk file system

Swap Space

- A portion of the disk reserved for storing pages that are evicted from memory
- May be not part of the actual disk file system
- On Linux there exists a dedicated **contiguous swap partition** (on disk)
    - no actual files are stored in that partition

Swap Space

- A portion of the disk reserved for storing pages that are evicted from memory
- May be not part of the actual disk file system
- On Linux there exists a dedicated **contiguous swap partition** (on disk)
  - no actual files are stored in that partition
- On Mac, instead, swap space is part of the file system (**swap files**) yet subject to fragmentation

Swap Out

- When a page needs to be swapped out, it will be generally copied to disk

Swap Out

- When a page needs to be swapped out, it will be generally copied to disk
- The pages for a process are divided into **2 groups**:
    - Code (read-only)
    - Data (initialized/uninitialized)

Swap Out

- When a page needs to be swapped out, it will be generally copied to disk
- The pages for a process are divided into **2 groups**:
  - Code (read-only)
  - Data (initialized/uninitialized)
- Based on which kind of page is removed, different optimizations may apply upon page swap-out

Swap Out Optimizations

- Code page (read-only):
  - Code content does not change!
  - Just remove and load it back from executable file stored on disk

Swap Out Optimizations

- Code page (read-only):
  - Code content does not change!
  - Just remove and load it back from executable file stored on disk
- Data page:
  - Data content does actually change!
  - Save it to the swap area/swap file, so that no changes are lost when it will be loaded in the future

Summary

- Virtual Memory allows processes to extend their memory footprint beyond the limit of the physical RAM

Summary

- Virtual Memory allows processes to extend their memory footprint beyond the limit of the physical RAM
- Combined to paging, uses secondary storage (i.e., disks) as backup for unallocated frames

Summary

- Virtual Memory allows processes to extend their memory footprint beyond the limit of the physical RAM
- Combined to paging, uses secondary storage (i.e., disks) as backup for unallocated frames
- Whenever a process requests a page, this could either be in main memory or on disk (**page fault**)

Summary

- Virtual Memory allows processes to extend their memory footprint beyond the limit of the physical RAM
- Combined to paging, uses secondary storage (i.e., disks) as backup for unallocated frames
- Whenever a process requests a page, this could either be in main memory or on disk (**page fault**)
- Ideally, the OS should keep in main memory each process' **working set** to lower the chance of a page fault