# Operating System Process and Resource Management

*Memory Architecture*

SUPINFO

# *Course Objectives*

- ✓ Build your knowledge in computer architecture with a focus on memory
- ✓ Present the mechanisms involving segmentation and paging developments

# *Course Plan*

1. Memory

2. Memory Organization

3. Virtual Memory Mechanisms

# 1. Memory

# 1. Memory

## Memory hierarchy

- If a processor accesses a memory address, it is often possible that it will execute the instructions or variables of a neighboring address

- **Locality principle**: programs manipulate mainly **contiguous memory pages**

- The constructors introduce a hierarchy in the memories

- **First level of memory**: the fastest memory to access is the **register** in the processor

# 1. Memory

## Memory hierarchy

- **Registers** are used to limit transfers between the processor and memory, for example:

**i = j + k**
**i = i + t**

- We have 2 transfers between the processor and the memory, if the processor uses its registers we have:

**$reg = j + k**
**i = $reg + t**

- With **$reg** the value of **i** is obtained with a single transfer between processor and memory

# 1. Memory

## Memory hierarchy

- **Second level of memory**: the **caches** (L1, L2, L3, *etc*.) are directly attached to the processor
- For a cache loading request, the processor checks that the instruction or the data are not already present in one of the caches:
  - If so: transfer failure
  - Otherwise: transfer
- The preparation time of a transfer is long (greater than the transfer time itself), so during a transfer, there is the simultaneous copy of a whole block of instructions or data in the caches

# 1. Memory

## Memory hierarchy

- The capacity of the caches depends on the power of the processor (and the transfer time)

- A too small cache would not satisfy a fast processor (often interrupted by memory accesses)
  - Variations of 2 to 3 in the speed of a machine can be measured according to this parameter
  - Among other things, loops that are too long must be split into several so that each one fits in the cache, neighboring array elements must be manipulated and, more generally, the proximity principle must be respected

- To limit accesses and gain speed, the programmer must optimize his/her code and know the architecture and system of his/her computer

# 1. Memory

## Memory hierarchy

- **Third level of memory**: the central memories, the disks

- The I/O buffers are considered to be the disks’ cache memories

- Computers with large databases have disks considered as machines, allowing more speed with a hierarchy of data access
  - The memory of the latter acts as a cache in order to limit accesses to disks that are too slow
  - This is where we first look for the information
  - When it is not there, we read it on a disk, a whole block is transferred and placed in the memory of the storage machine

# 1. Memory

## Questions

# 2. Memory Organization

# 2. Memory Organization

## Introduction

- During the normal operation of a computer, the main memory is shared by all active processes
- If the memory sharing management mechanisms are not efficient or if a program does not respect the operating rules, then the computer will not perform well (no matter how powerful its processor is)
- The main memory can be split in three methods:
  - **Segmentation**: programs are cut into segments (of variable length)
  - **Paging**: the memory is divided into blocks, and the programs are divided into pages of fixed length
  - The **combination of segmentation and paging**: parts of the memory are segmented, and the others are paged

# 2. Memory Organization

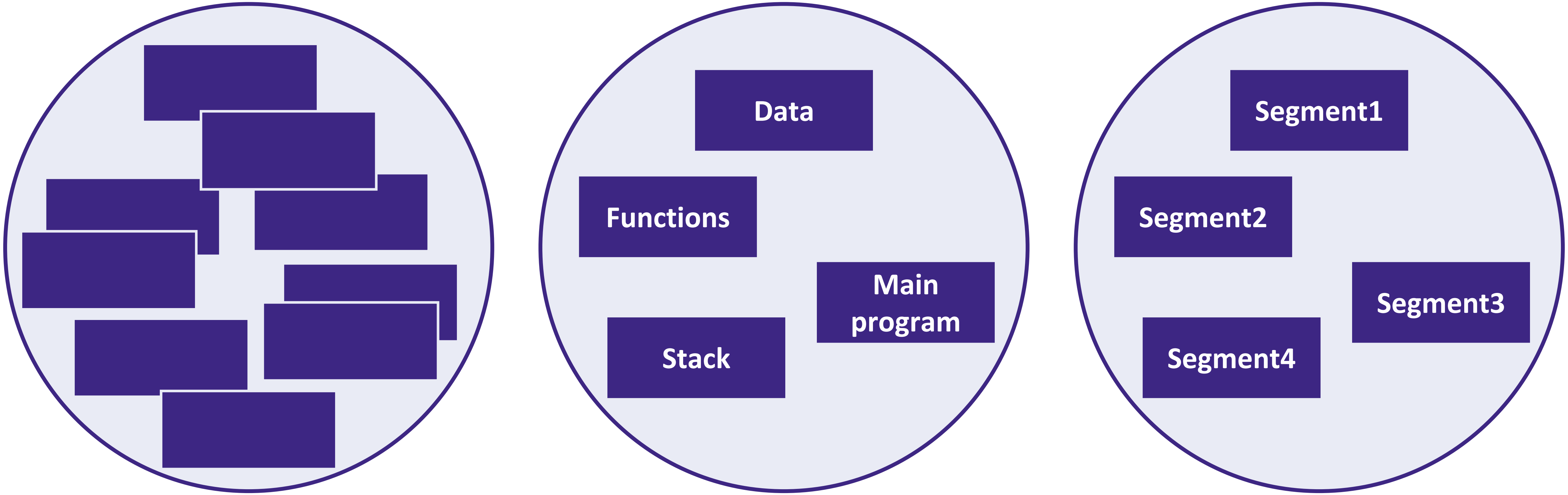
## Introduction

- **Paging** is a division of the process address space that does not correspond to the programmer’s image of his/her program
  - For the programmer, a program is generally made up of the data manipulated by this program, a main program, separate procedures and an execution stack

- **Segmentation** is a division of the address space that seeks to preserve this view of the programmer
  - Thus, during compilation, the compiler associates a segment with each piece of the compiled program
  - A segment is a set of consecutive non-breakable memory locations
  - Unlike pages, the segments of the same address space can be of different sizes

# 2. Memory Organization

## Segmentation



**Program address space**
**Set of pages**

**User view**

**Program address space**
**Set of segments**

# 2. Memory Organization

## Segmentation

- Segments are a series of consecutive locations

- In general, a segment corresponds to a logical division of the program (stack segment, data segment, code segment, *etc*.) which makes memory management difficult (use of garbage collectors)

- A segmented address is a segment number with an offset in the segment

- A segment is an entity containing specific information for the system and its management
  - **Validation**: indicates if the segment should remain in main memory
  - **Replacement**: proposes information to the replacement algorithm

| Segment name |
|---|
| Size |
| Rights |
| Validation |
| Replacement |
| Program index |

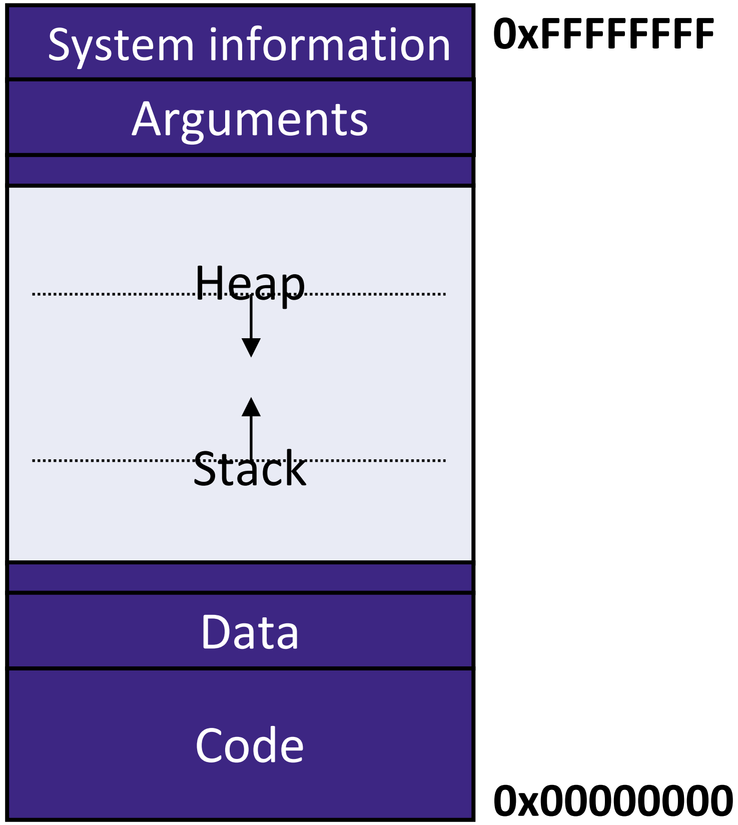
# 2. Memory Organization

## Process memory organization

- The memory space used by a process is divided into several parts

- We find especially the code segment (often called text segment), the data segment, the stack and the heap

- Each process believes that it is the only one to occupy the memory and thinks that it has a maximum size
  – Theoretically 4GB on a 32-bit system and 16EB on a 64-bit system
  – OS limits, *ex*: Windows 32-bit 2GB-4GB (LAA), Windows 64-bit 2GB-128TB (LAA)



# 2. Memory Organization

## Process memory organization

- The **code segment** is obtained by copying the code segment of the executable file directly into memory
  - During program execution, the next instruction to be executed is marked by an instruction pointer

- If the execution of an instruction in the code changes the arrangement of the process memory space and causes the code segment to move:
  - The instruction pointer will no longer be valid
  - The program will run incorrectly

- To avoid this problem, the code segment is always placed in fixed areas of (virtual) memory; as the system uses virtual memory, this area usually starts at address 0 and the resulting addresses are then translated into physical addresses

# 2. Memory Organization

## Process memory organization

- The addresses of the executable’s text segment also start at 0
- Placing the text segment at address 0 of the process memory avoids having to shift all the code addresses
- To avoid re-entrance problems, the code segment is generally only accessible in read mode
  - It is often shared by all the processes running the same program
- Above the code segment is the **data segment**
  - This segment is composed of a segment of initialized data, directly copied from the executable, and a segment of non-initialized data (**BSS** for Block Started by Symbol) created dynamically

# 2. Memory Organization

## Process memory organization

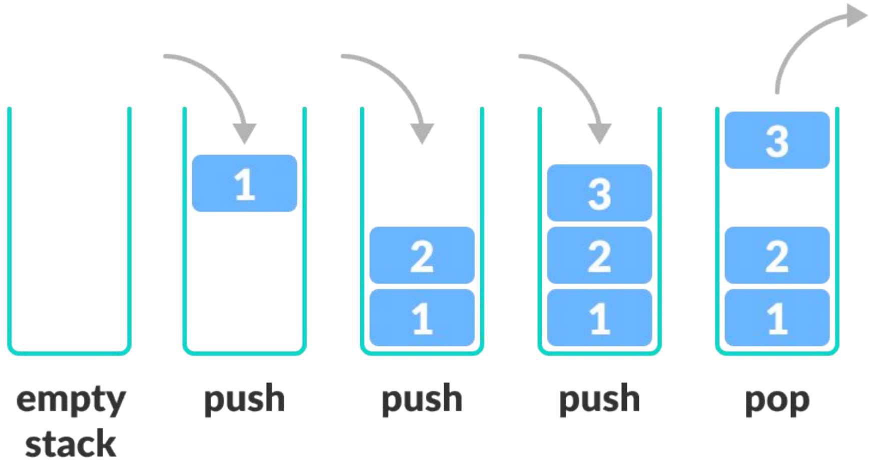
- The data correspond to all global and static (**un**)**initialized** variables of programs
- The data segment is extended or reduced during runtime to allow for the placement of data
- Nevertheless, it is usual to consider that this segment is fixed and corresponds to the one obtained before the execution of the first instruction: the **BSS** segment is then reduced to the local variables of the **main()** function

# 2. Memory Organization

## Process memory organization

- To store the data being executed, the system uses a segment called the **stack**

- Example: the stack (defined here from left to right) contains the data **ABCD**, and the current instruction calls a function
  - The system places the parameters (**Q** and **R**) of the function on the stack (**ABCDQR**), then places the various local variables (**X**, **Y** and **Z**) of the function on the stack (**ABCDQRXYZ**)
  - The function is executed normally, and the system only needs to unstack all the data placed for the function call (**ABCD**)
  - The system can directly access the data it had stored before the function was called and can execute the next instruction



# 2. Memory Organization

## Process memory organization

- The advantages of this stack are:
  - Stacking and unstacking data
  - Data is always stored contiguously
  - The memory space of the process constitutes a compact set that can be easily translated into physical addresses

- Unfortunately, this is no longer true in the case of dynamic allocation: the memory space allocated by a function like **malloc()** cannot be unstacked and the data is no longer stored contiguously
  - Consider a function called, which allocates memory bytes (**#** = free bytes, **+** = dynamically allocated bytes): before the function call the stack is **ABCD**, just before the end of the function the stack is **ABCDQRXYZ+++** and after the function call the stack is **ABCD#####+++**

# 2. Memory Organization

## Process memory organization

- To avoid keeping the location of free but non-contiguous areas in memory, operating systems use another segment for dynamic allocations: the **heap**

- The problem is solved only if the location of the heap does not vary according to the data stored in the stack

- For this reason, the heap is placed at the end of the process memory space and is extended from top to bottom, unlike the stack

- Since the operating system uses virtual memory, the virtual address of the heap start does not matter because the free memory areas between the heap and the stack will not be translated into physical areas

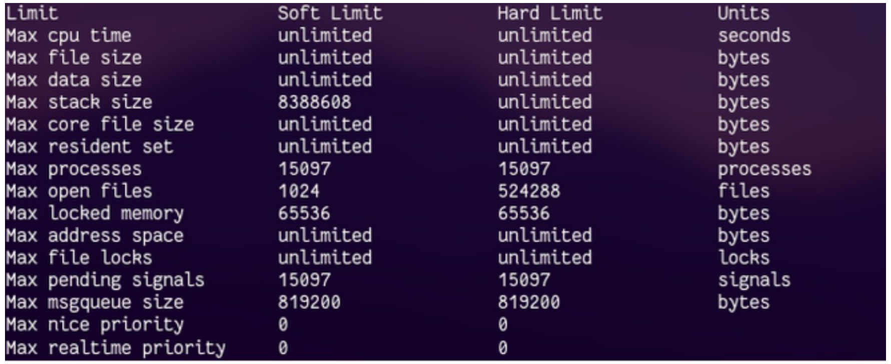
# 2. Memory Organization

## Process memory organization

- In practice, the stack and heap sizes are bounded, and these bounds are part of the (editable) limitations imposed by the operating system

- Other information is placed in the process memory space, such as:
  - The parameters passed when the program is executed
  - The mode in which the process should run

- This information is stored in a structure that is often called Process Control Bloc (**PCB**) or Task Control Bloc (**TCB**)



```
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        unlimited            unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             15097                15097                processes
Max open files            1024                 524288               files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       15097                15097                signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
```

# 2. Memory Organization

## Process memory organization

- All this other information is used by the system and must be easy to find: the system must not have to update its internal structures every time the stack or the heap is changed

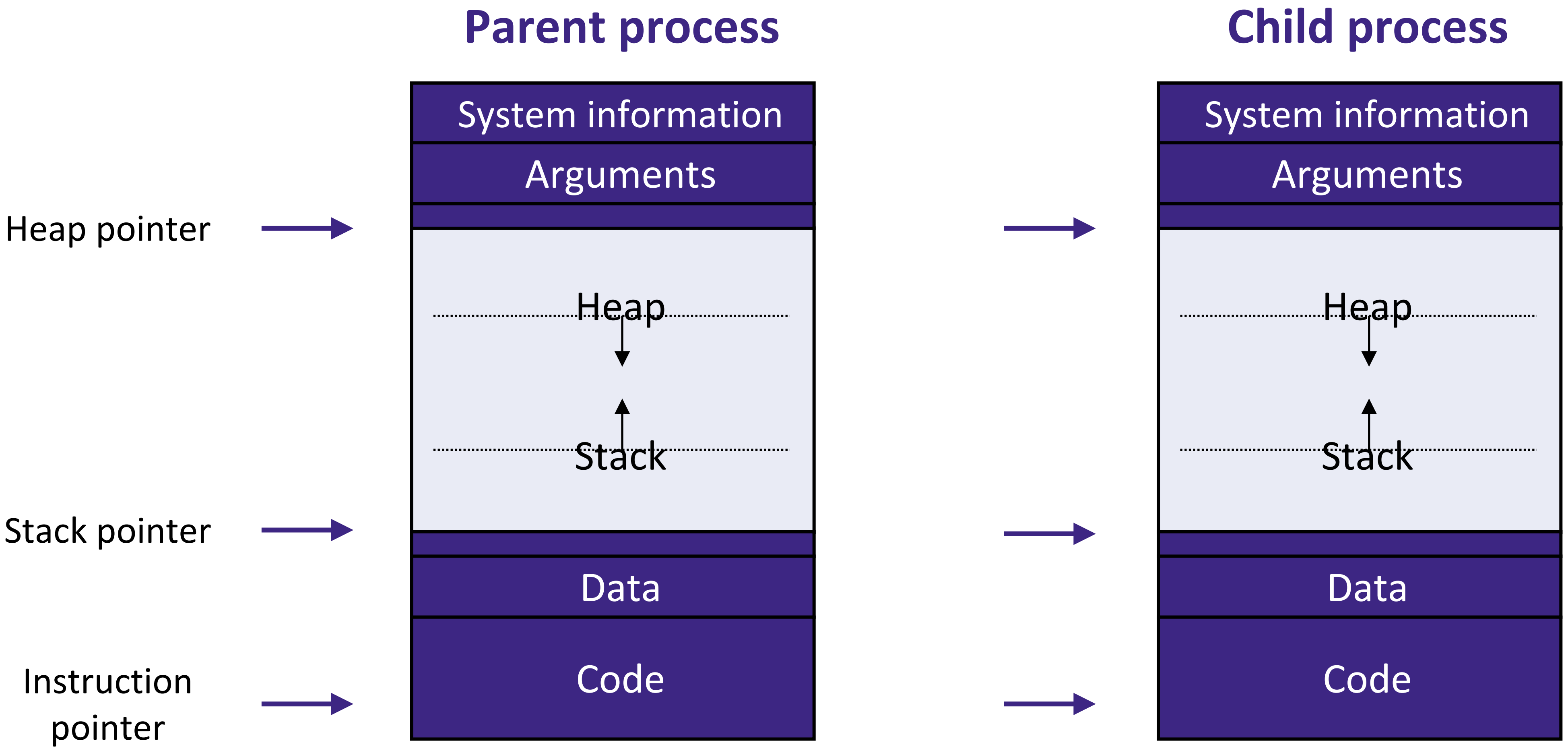
- For this reason, this information is usually stored at the end of the process memory space, after the heap

- Since their size is fixed, this does not interfere with the operation of the heap

- The process memory space is thus divided into 2 variable parts, and themselves divided into several segments of fixed size (ends), or of variable size (heap/stack)

# 2. Memory Organization

## Process memory organization



# 2. Memory Organization

## Paging

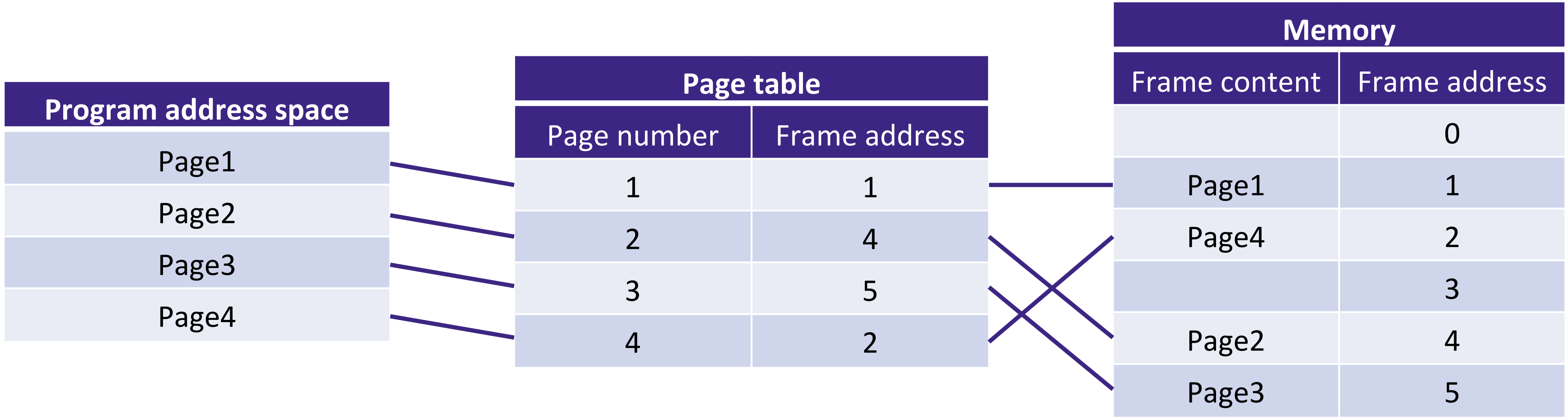
- **Paging** is an approach to reduce memory fragmentation

- The logical address space of a process is paged
  - The program address space is cut into linear pieces of the same size: the page
  - The physical memory space is itself cut into linear frames of the same size
  - The size of a frame is equal to the size of a page
  - Loading a program in main memory consists in placing the pages in any available frame

- Pages are independent in terms of addressing within memory and are not necessarily contiguous; the operating system allocates physical memory whenever the need to accommodate a page arises

# 2. Memory Organization

## Paging



| Program address space |
|---|
| Page1 |
| Page2 |
| Page3 |
| Page4 |

| Page table | |
|---|---|
| Page number | Frame address |
| 1 | 1 |
| 2 | 4 |
| 3 | 5 |
| 4 | 2 |

| Memory | |
|---|---|
| Frame content | Frame address |
| | 0 |
| Page1 | 1 |
| Page4 | 2 |
| | 3 |
| Page2 | 4 |
| Page3 | 5 |

- The paged address has its equivalent in **physical address**
  - Physical address is the frame location address containing the page and the offset (guaranteed by the page table)

# 2. Memory Organization

## Paging

- The page table is a table containing as many entries as there are pages in the process address space
  - Each process has its own page table
  - Each entry in the table is a couple **<page number, physical frame number>**

- In the previous example, the process has 4 pages in its address space, so the page table has 4 entries
  - Each entry establishes the equivalence relative to the main memory

- The process address space being divided into pages, the addresses generated in this address space are **paged addresses**: a byte is marked by its location relative to the beginning of the page to which it belongs (byte address is therefore constituted by the couple **<byte page number, page start offset>**)

# 2. Memory Organization

## Paging

- Bytes in physical memory can only be addressed at the hardware level by their physical address
  - For any operation concerning the memory, it is thus necessary to convert the paged address generated at the CPU level into an equivalent physical address
  - The byte physical address is obtained from its virtual address by replacing the page number of the virtual address by the physical address of the frame containing the page and by adding to this physical address the byte offset in the page
  - It is the **MMU** (Memory Management Unit) that is responsible for converting the physical address to the virtual address; this microprocessor unit generally uses its own cache memory for the pages (**TLB**, Translation Lookaside Buffer)
  - It is thus necessary to know for any page, in which frame of the main memory this one was placed (page table)

# 2. Memory Organization

## Paging

- The implementation of the page table is a set of hardware registers with process switching (loading all registers with the process page addresses), page table cannot be very large (like 256 records)

- In main memory, marked by a **PTBR** (Page Table Base Register):
  - Switching processes means loading the PTBR register with the address of the process page table
  - No size limit to the page table
  - Heavy access time to a program memory location

- Since each process has its own page table, each context switch operation also results in a change of page table, so that the active table matches that of the elected process

# 2. Memory Organization

## Paging

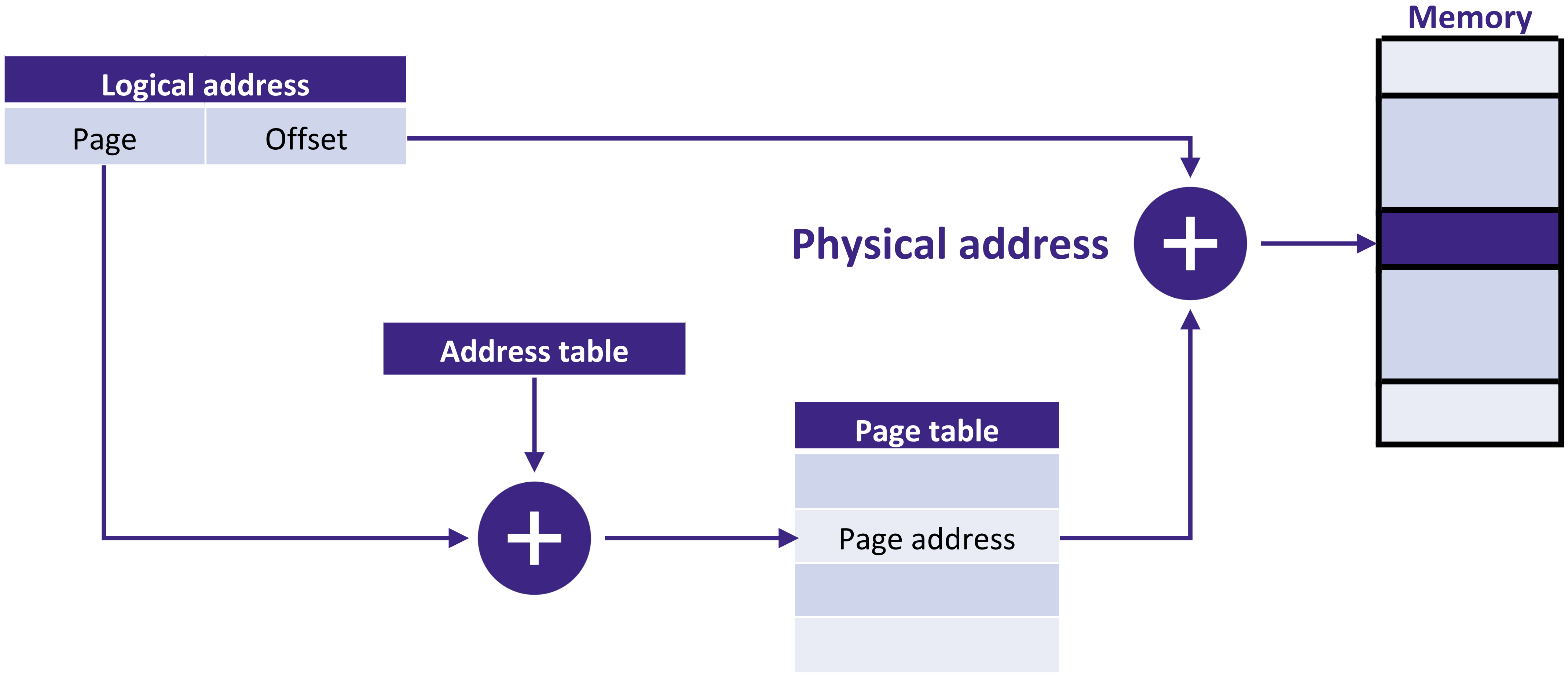
- 2 approaches exist for the realization of the page table:
  - Using CPU registers, the page table is saved with the CPU context in the PCB
  - Placing the page tables in main memory, the active table is marked by a special register of the processor (PTBR): each process saves in its PCB the PTBR value corresponding to its table

- In the first approach, accessing a memory location requires only one access to the memory: the one necessary to read or write the searched byte since the page table is stored in registers of the processor

- In the second approach, accessing a memory location requires instead 2 accesses to the memory: a first access allows to read the page table entry corresponding to the searched page and a second access to read or write the searched byte

# 2. Memory Organization

## Paging



Memory

Logical address

Page

Offset

Physical address

Address table

Page table

Page address

# 2. Memory Organization

**Questions**

# 3. Virtual Memory Mechanisms

# 3. Virtual Memory Mechanisms

## Comparing paging and segmentation

- Segmentation and paging are 2 different concepts that are used together on some machines (*ex*: iAPX386 or Multics)

- **Segmentation** should be seen as structuring the process address space
  - **Segments** should be allocated in linear memory using area allocation techniques (contiguous memory portion of given size)
  - With segmentation, the process has a two-dimensional address space that the processor transforms into an address in a linear memory (without, the process has this linear memory directly at its disposal)

# 3. Virtual Memory Mechanisms

## Comparing paging and segmentation

- **Paging** should be seen as a means of adapting virtual memory to real memory
  - The pages of the linear memory must be allocated in the physical memory
  - With paging, we have a function for dynamically transforming linear memory addresses into physical memory addresses, which allows linear memory pages to be placed in any physical memory frames (without, the linear memory pages must be placed in the physical memory frames of the same number)

- Data sharing between 2 processes can be achieved in different ways, for example it can be achieved at the level of:
  - **Pages** of their linear memory, by allocating the same physical memory frame to both processes, at the same time
  - **Hyper pages** of their linear memory, by allocating the same page table to 2 hyper pages of these processes

# 3. Virtual Memory Mechanisms

## Comparing paging and segmentation

- Without paging, it can be obtained at the segment level by allocating the same physical memory area to 2 segments of each of the 2 processes

- It may seem complex to have to implement both concepts; however, they are complementary, and paging simplifies the allocation by zone of the segmentation:
  - On the one hand, it can then have free pages in the middle of occupied pages in linear memory, without losing physical memory space
  - On the other hand, moving a page from linear memory to another page can be obtained without moving it in physical memory, since it is enough to modify the corresponding entries in the page table

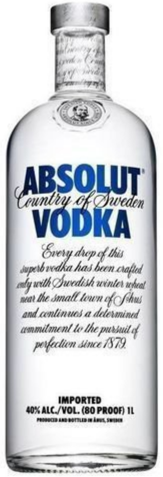
# 3. Virtual Memory Mechanisms

## Consequences

- The code of a program is not absolute, which means you cannot move it in the virtual memory (it has a fixed address)

- Having a dynamic coupling function and a translation vector should solve the issue, but it is expensive in terms of performance

- To avoid conflicts of implementation in virtual memory and fragmentation, we associate a virtual memory to each user


ABSOLUT
Country of Sweden
VODKA
Every drop of this superb vodka has been crafted only with Swedish winter wheat near the small town of Åhus and continues a determined commitment to the pursuit of perfection since 1879
IMPORTED
40% ALC./VOL. (80 PROOF) 1L

# 3. Virtual Memory Mechanisms

## Virtual memory

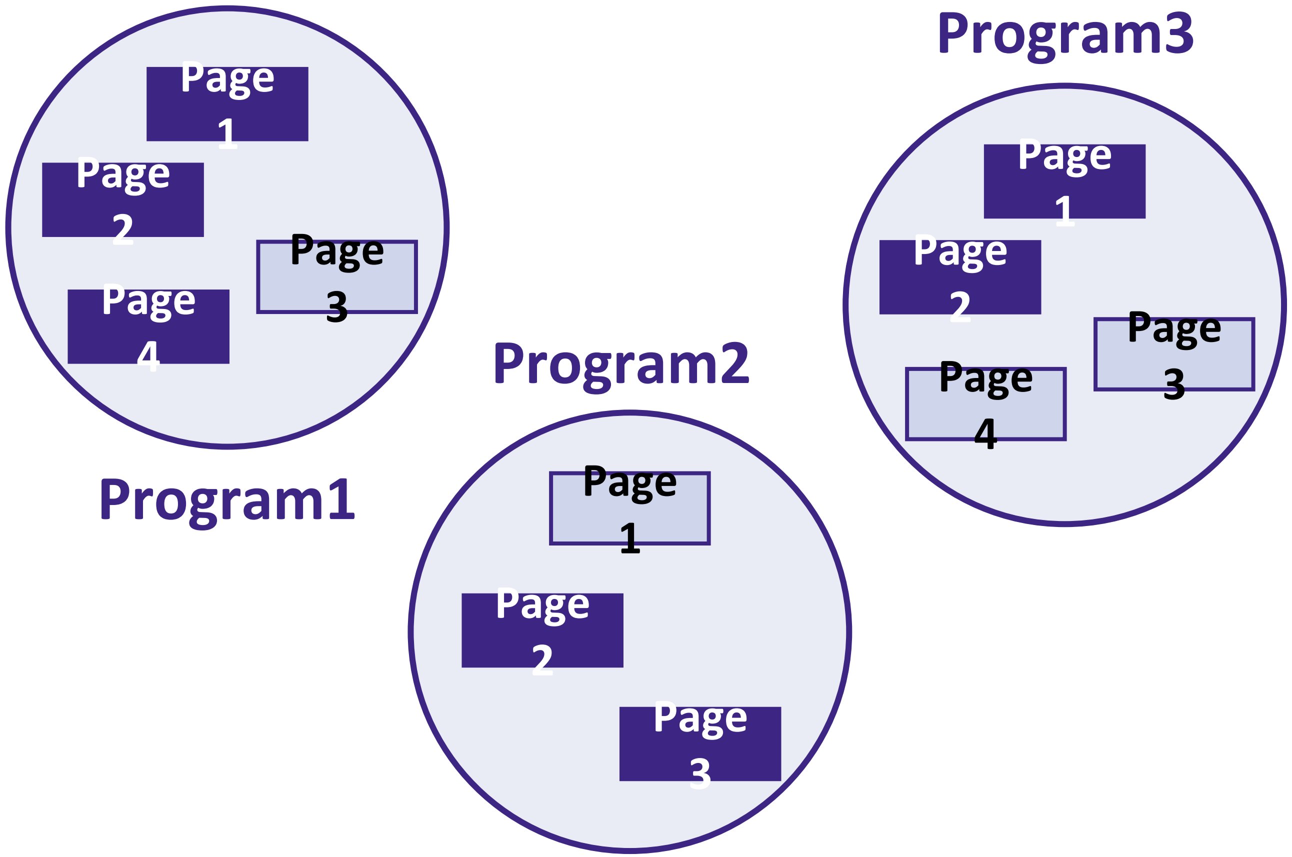
- **Virtual memory** is a set of segments
- A **virtual address** is a segment number and an offset
- The **main** (physical) **memory** is the set of locations in the RAM
- We have a **segment** table, and the replacement algorithm substitutes one (or more) whole segments with the referenced segment
- A **shift in the segment** is a virtual page number with an offset in the page

# 3. Virtual Memory Mechanisms

## Virtual memory

- The memory is often too small to place all the pages of the programs; the loading is done according to the pages useful for the execution of the programs (at a given moment)

| Memory | | |
|---|---|---|
| Frame content | | Frame address |
| Program number | Page number | |
| 1 | 1 | 1 |
| 1 | 2 | 2 |
| 1 | 4 | 3 |
| 2 | 3 | 4 |
| 2 | 2 | 5 |
| 3 | 1 | 6 |
| 3 | 2 | 7 |



# 3. Virtual Memory Mechanisms

## Virtual memory

- When we look at a process execution, we see that at a given time the process accesses only a part of its address space (for example the code page currently executed by the processor and the corresponding data page)

- The other pages of the address space are not accessed and are therefore useless in main memory

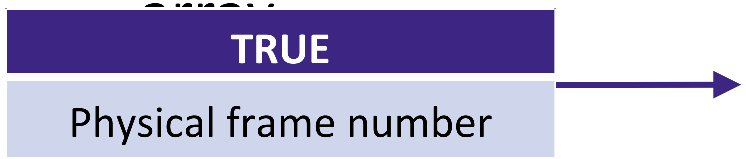
- A solution to be able to load more programs in the main memory is therefore for each program to load only the pages currently used

- In the previous example, only pages 1, 2 and 4 of program 1 are loaded as well as pages 2 and 3 of program 2, and pages 1 and 2 of program 3

# 3. Virtual Memory Mechanisms

## Virtual memory

- In order to load only the pages that are useful at a given time, a manager is set up to control the presence of the page in memory; the data structure used is an array



**Validation bit is true if the page is present in main memory**

- Since not all pages in a process address space are loaded into main memory, the processor must be able to detect their possible absence when it seeks to perform a paged address conversion to the physical address

- Each entry in the page table has an additional field, the **validation bit**, which is true if the page is available in main memory

# 3. Virtual Memory Mechanisms

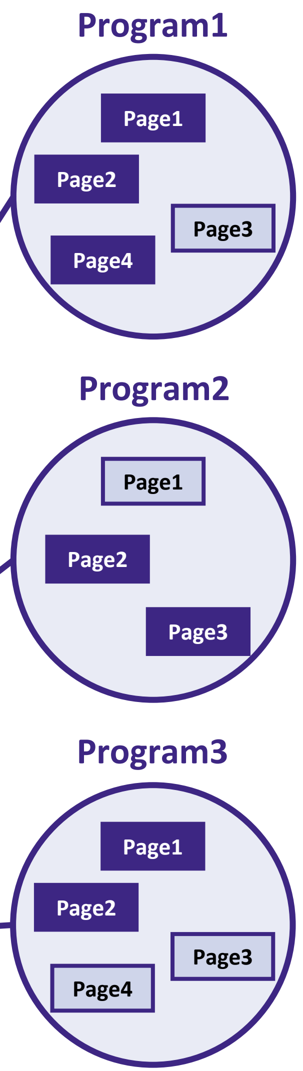
## Virtual memory and page fault

- The values of the validation bits for the page tables of the 3 programs, considering the loading of their pages in main memory

| Memory | | |
|---|---|---|
| Frame content | | Frame address |
| Program number | Page number | |
| 1 | 1 | 1 |
| 1 | 2 | 2 |
| 1 | 4 | 3 |
| 2 | 3 | 4 |
| 2 | 2 | 5 |
| 3 | 1 | 6 |
| 3 | 2 | 7 |

| Page | Validation | Frame |
|---|---|---|
| 1 | TRUE | 1 |
| 2 | TRUE | 2 |
| 3 | FALSE | X |
| 4 | TRUE | 3 |
| 1 | FALSE | X |
| 2 | TRUE | 5 |
| 3 | TRUE | 4 |
| 1 | TRUE | 6 |
| 2 | TRUE | 7 |
| 3 | FALSE | X |
| 4 | FALSE | X |

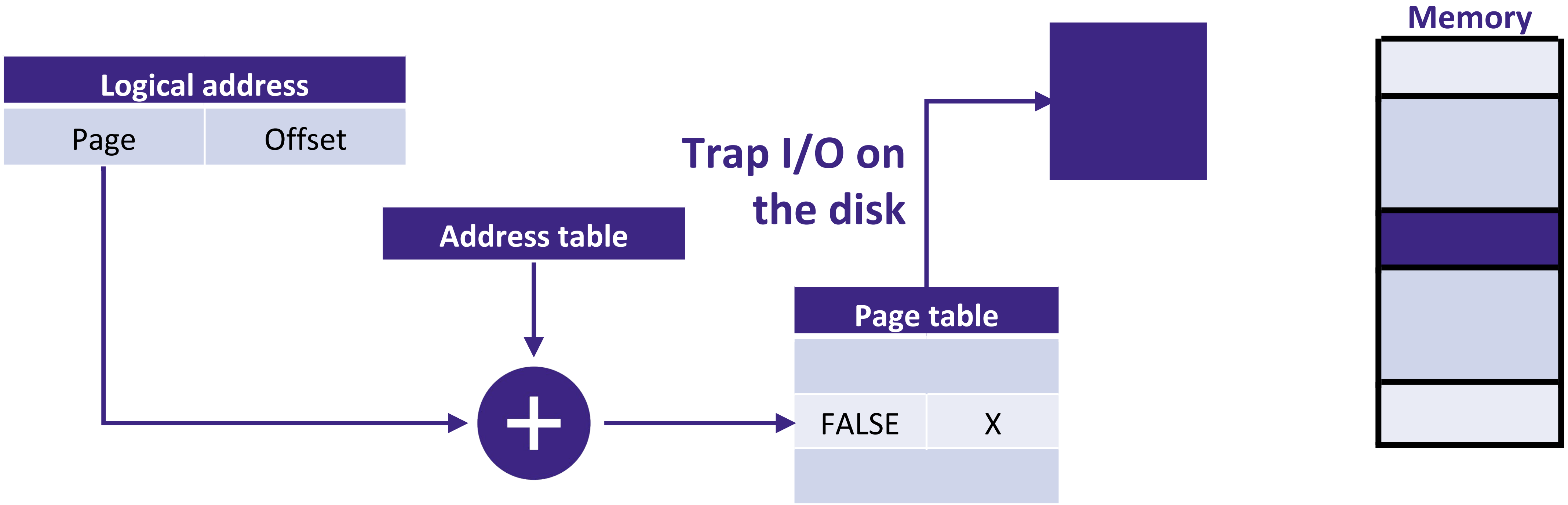
🟥 = page fault



# 3. Virtual Memory Mechanisms

## Page fault

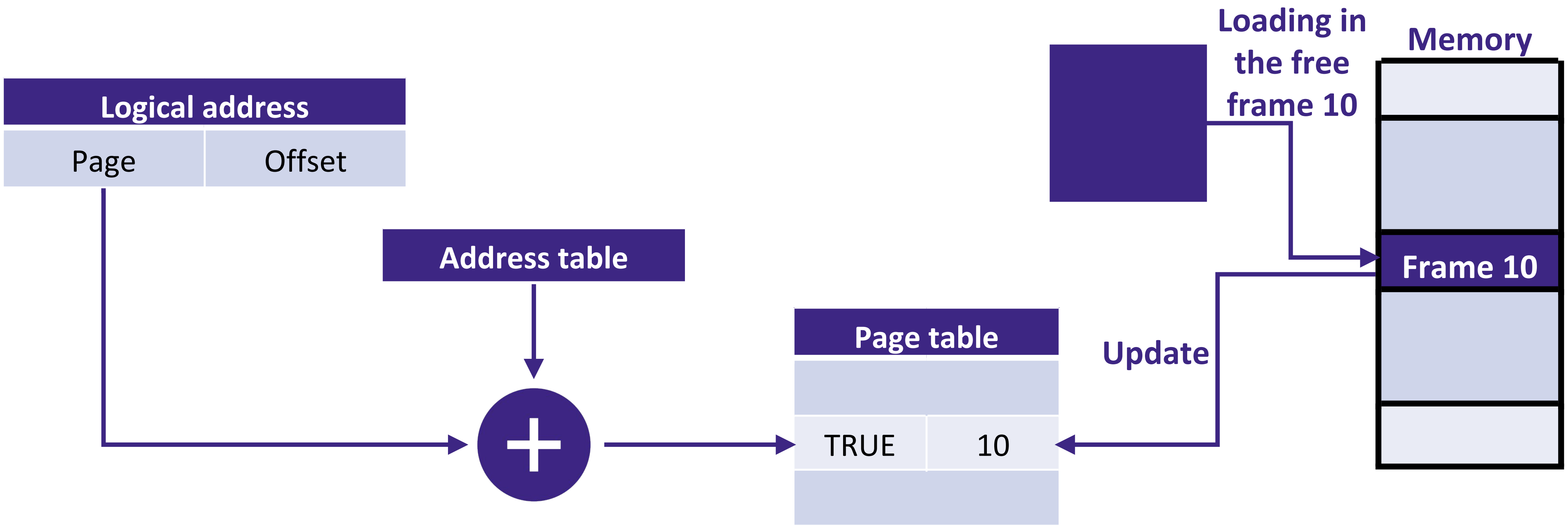
- When a process wants to access a page not present in main memory, a **page fault** occurs
  - This is a trap, and the operating system loads the missing page into an empty frame or must first clear a frame and then replace the value with the new page



# 3. Virtual Memory Mechanisms

## Page fault

- When a process wants to access a page not present in main memory, a **page fault** occurs
  - This is a trap, and the operating system loads the missing page into an empty frame or must first clear a frame and then replace the value with the new page



# 3. Virtual Memory Mechanisms

## Page fault

- When a page fault occurs, the missing page is loaded into a free frame, but all the frames in the main memory may be occupied

- In this case, a frame must be freed globally (among all the frames) or locally (among the frames occupied by the pages of the faulty process)

- The operating system uses an algorithm for the choice of the frame to release and replace:
  - First In, First Out (**FIFO**)
  - Least Recently Used (**LRU**)

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| **9** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **9** | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| **F** | | | | | | | | | | |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page

- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | | | | | | | | | |
| | 8 | | | | | | | | | |
| | | | | | | | | | | |
| F | F | | | | | | | | | |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | | | | | | | | |
| | 8 | 8 | | | | | | | | |
| | | 1 | | | | | | | | |
| F | F | F | | | | | | | | |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 |  |  |  |  |  |  |  |
|  | 8 | 8 | 8 |  |  |  |  |  |  |  |
|  |  | 1 | 1 |  |  |  |  |  |  |  |
| F | F | F | F |  |  |  |  |  |  |  |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | | | | | | |
| | 8 | 8 | 8 | 8 | | | | | | |
| | | 1 | 1 | 1 | | | | | | |
| F | F | F | F | | | | | | | |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | | | | | |
| | 8 | 8 | 8 | 8 | 3 | | | | | |
| | | 1 | 1 | 1 | 1 | | | | | |
| F | F | F | F | | F | | | | | |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | | | | |
| | 8 | 8 | 8 | 8 | 3 | 3 | | | | |
| | | 1 | 1 | 1 | 1 | 8 | | | | |
| F | F | F | F | | F | F | | | | |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | 4 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | 4 | | | |
| | 8 | 8 | 8 | 8 | 3 | 3 | 3 | | | |
| | | 1 | 1 | 1 | 1 | 8 | 8 | | | |
| F | F | F | F | | F | F | F | | | |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page

- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | 4 | 2 |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | 4 | 4 |  |  |
|  | 8 | 8 | 8 | 8 | 3 | 3 | 3 | 2 |  |  |
|  |  | 1 | 1 | 1 | 1 | 8 | 8 | 8 |  |  |
| F | F | F | F |  | F | F | F | F |  |  |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page

- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | 4 | 2 | 3 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | |
| | 8 | 8 | 8 | 8 | 3 | 3 | 3 | 2 | 2 | |
| | | 1 | 1 | 1 | 1 | 8 | 8 | 8 | 3 | |
| F | F | F | F | | F | F | F | F | F | |

# 3. Virtual Memory Mechanisms

## FIFO page replacement

- **FIFO**: the oldest loaded page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | 4 | 2 | 3 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 8 |
|  | 8 | 8 | 8 | 8 | 3 | 3 | 3 | 2 | 2 | 2 |
|  |  | 1 | 1 | 1 | 1 | 8 | 8 | 8 | 3 | 3 |
| F | F | F | F |  | F | F | F | F | F | F |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| F | | | | | | | | | | |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | | | | | | | | | |
| | 8 | | | | | | | | | |
| | | | | | | | | | | |
| F | F | | | | | | | | | |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | | | | | | | | |
| | 8 | 8 | | | | | | | | |
| | | 1 | | | | | | | | |
| F | F | F | | | | | | | | |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | | | | | | | |
| | 8 | 8 | 8 | | | | | | | |
| | | 1 | 1 | | | | | | | |
| F | F | F | F | | | | | | | |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page

- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 |  |  |  |  |  |  |
|  | 8 | 8 | 8 | 8 |  |  |  |  |  |  |
|  |  | 1 | 1 | 1 |  |  |  |  |  |  |
| F | F | F | F |  |  |  |  |  |  |  |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | | | | | |
| | 8 | 8 | 8 | 8 | 8 | | | | | |
| | | 1 | 1 | 1 | 3 | | | | | |
| F | F | F | F | | F | | | | | |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page

- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | | | | |
| | 8 | 8 | 8 | 8 | 8 | 8 | | | | |
| | | 1 | 1 | 1 | 3 | 3 | | | | |
| F | F | F | F | | F | | | | | |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | 4 |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | 4 |  |  |  |
|  | 8 | 8 | 8 | 8 | 8 | 8 | 8 |  |  |  |
|  |  | 1 | 1 | 1 | 3 | 3 | 3 |  |  |  |
| F | F | F | F |  | F |  | F |  |  |  |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page

- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| **9** | **8** | **1** | **2** | **8** | **3** | **8** | **4** | **2** |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| **9** | **9** | **9** | **2** | **2** | **2** | **2** | **4** | **4** |  |  |
|  | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** |  |  |
|  |  | **1** | **1** | **1** | **3** | **3** | **3** | **2** |  |  |
| **F** | **F** | **F** | **F** |  | **F** |  | **F** | **F** |  |  |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page

- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | 4 | 2 | 3 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | |
| | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 3 | |
| | | 1 | 1 | 1 | 3 | 3 | 3 | 2 | 2 | |
| F | F | F | F | | F | | F | F | F | |

# 3. Virtual Memory Mechanisms

## LRU page replacement

- **LRU**: the least recently accessed page is the replaced page
- Reference sequence for a 3-frame memory (initially empty): **9 8 1 2 8 3 8 4 2 3 8**
  - It indicates the pages referenced over time
  - **F** is noted when there is a page fault

| 9 | 8 | 1 | 2 | 8 | 3 | 8 | 4 | 2 | 3 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 9 | 9 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 8 |
|  | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 3 | 3 |
|  |  | 1 | 1 | 1 | 3 | 3 | 3 | 2 | 2 | 2 |
| F | F | F | F |  | F |  | F | F | F | F |

# 3. Virtual Memory Mechanisms

## Address conversion algorithm

- Different fields are added in the page table, to manage page replacements:
  - Access field (FIFO loading date, LRU last access date; allows to store useful information to choose the victim page when a replacement is necessary)
  - Update bit (TRUE if the page has been modified in main memory; when replacing it, it will be necessary to rewrite this page on the disk)
  - Validation bit (TRUE if the page is present in main memory)
  - Frame number

If you can't hide a crime scene, just pretend you are a victim.

# 3. Virtual Memory Mechanisms

## Address conversion algorithm

```
function conversion(I: virtualAddress, O: physicalAddress)
begin
    input = virtualAddress.page + addressTable(process)
    If (input.V = false) { #Page fault
        loadPage(virtualAddress.page, addressFrame)
        input.V = true
        input.frame = addressFrame
    }
    physicalAddress = addressFrame + virtualAddress.offset
    return physicalAddress
end
```

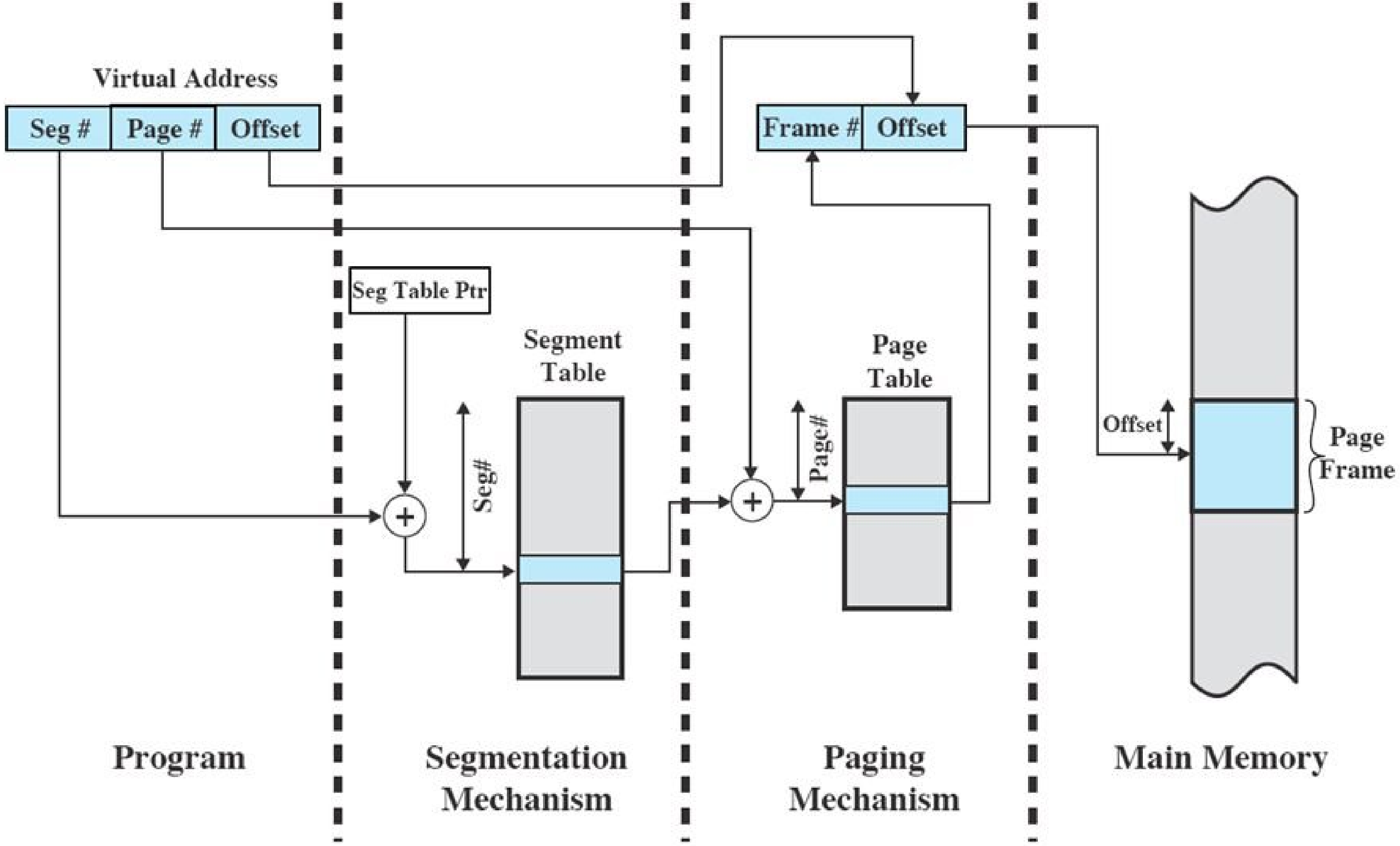
# 3. Virtual Memory Mechanisms

## Address conversion algorithm

```
function loadPage(I: page, O: frame)
begin
    If (findFreeFrame() = false) {
        chooseFrameToFree(frameToFree, selectedPage)
        If (selectedPage.U = true) {
            writeDisk(selectedPage)
        }
    }
    readDisk(frameToFree, page)
    return frameToFree
end
```

# 3. Virtual Memory Mechanisms

## Segmented paged memory



# 3. Virtual Memory Mechanisms

## Segmented paged memory

- External fragmentation is still a problem, because the segments are of variable size and the memory allocated to the segment must be large enough to hold it

- We end up in a situation where all the remaining available segments are smaller than the segments that need to be allocated: compaction, swapping, *etc*.

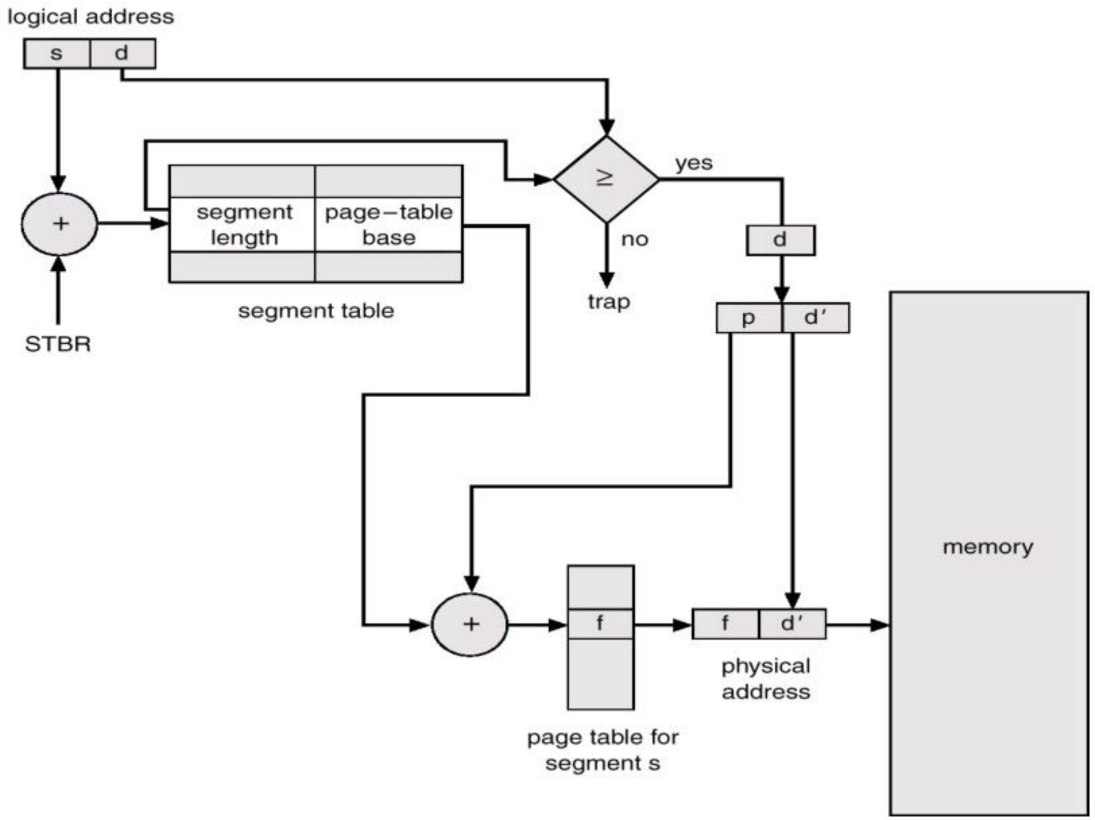
- **MULTICS** has solved problems related to external fragmentation and search times by paging the segments

- The difference with *pure* segmentation is that the segment table does not point to the segment base address but to the base address of a page table for the segment

# 3. Virtual Memory Mechanisms

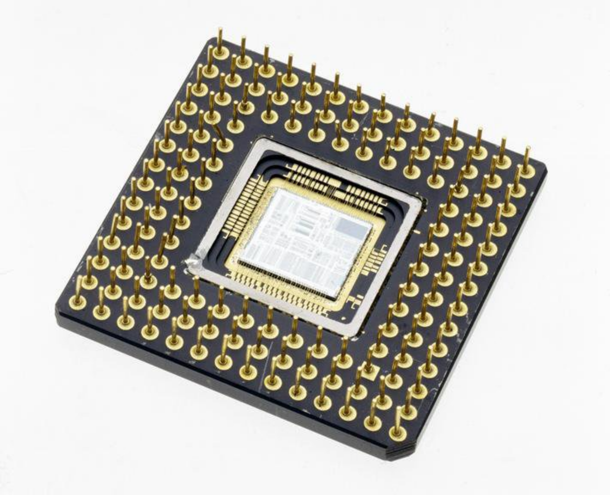
## Segmented paged memory

### MULTICS Address Translation Scheme



# 3. Virtual Memory Mechanisms

## Segmented paged memory



- **Intel 386**
  - Number of segments/process: 16KB
  - Maximum segment size: 4 GB
  - Page size: 4KB
  - The process address space is divided into 2 partitions:
    - 8KB of segments for the private space of the process, described by the **LDT** (Local Descriptor Table)
    - 8KB of segments for the shareable space, described by the **GDT** (Global Descriptor Table)
  - The entries in the 2 tables have 8 bytes and contain detailed information about the segments they describe

# 3. Virtual Memory Mechanisms

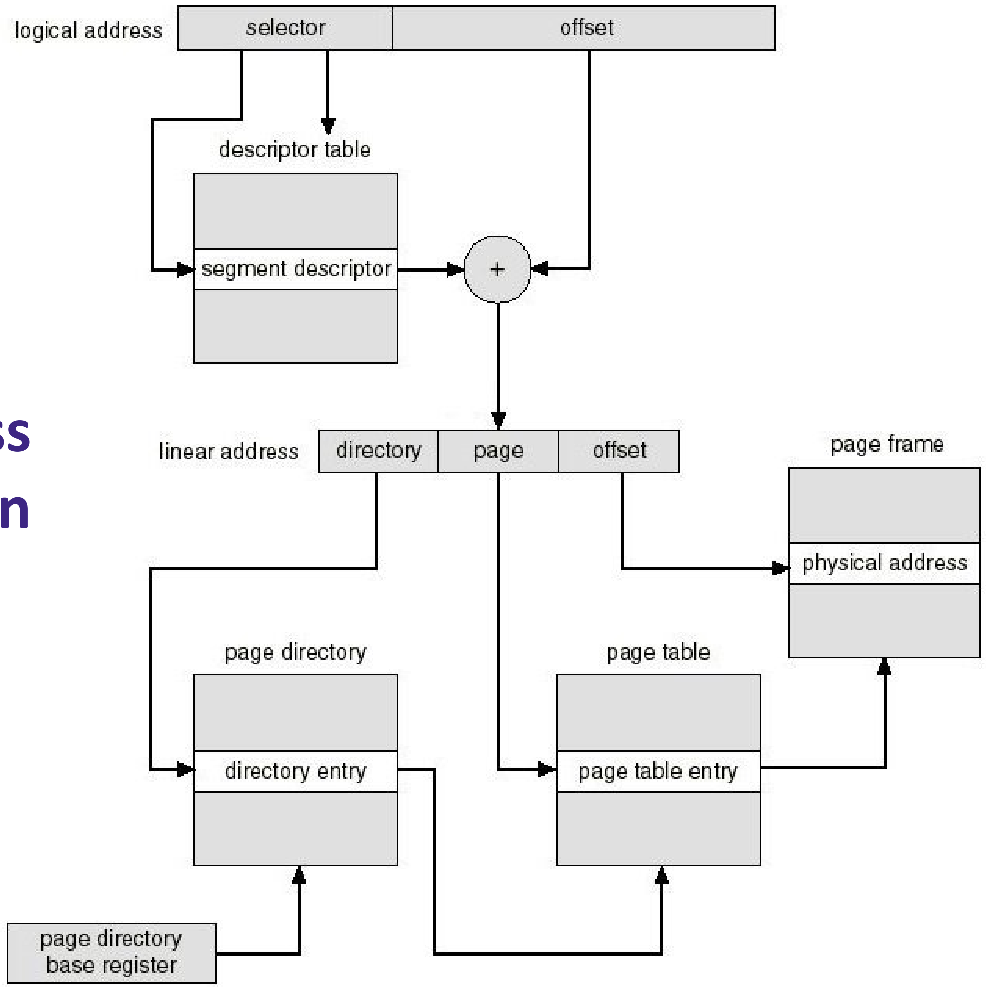
## Segmented paged memory

- **Intel 386**
  - The logical address is a couple composed of the segment selector (16 bits) and the offset (32 bits)
  - The segment scheduler specifies the segment number, the table (LDT or GDT) and the protection level
  - The segments are paginated, the pages having 4KB: the page table must therefore have up to 106 entries
  - Each entry of the page having 4 bytes, a process can require up to 4MB of physical memory, only for the page table
  - To place the process page table in memory, we adopt a 2-level addressing system: 20 bits for the page and 12 bits for the shift in the page; the page table being paginated, the 20 bits are divided into 2 fields: the page directory address (10 bits) and the directory page pointer (10 bits)

# 3. Virtual Memory Mechanisms

## Segmented paged memory

**Intel 386 Address Translation**



# 3. Virtual Memory Mechanisms

## Conclusion

- Paging advantages:
  - Better use of physical memory (programs implemented in fragments, in non-consecutive pages)
  - Possibility to load pages only when they are referenced (loading on demand)
  - Possibility of emptying only modified pages to disk
  - Independence of the virtual space and the physical memory (virtual memory generally larger)

- Paging drawbacks:
  - Internal fragmentation (not all pages are filled)
  - Impossibility to link 2 (or more) procedures related to the same addresses in the virtual space

# 3. Virtual Memory Mechanisms

**Exercise**

- In the case of paging and segmentation, answer the following questions:
  - Does the programmer need to know that this technique is used?
  - How many linear address spaces are there?
  - Can the total address space exceed the size of the physical memory?
  - Can procedures and data be separated and protected separately?
  - Can arrays of various sizes be easily accommodated?
  - Is it easy to share procedures between users?
  - Why was this technique invented?

# 3. Virtual Memory Mechanisms

**Questions**

# Operating System Process and Resource Management

## Memory Architecture

Thank you for your attention