ESCI upf. School of International Studies

Memory management

THEORY OF INFORMATION, ARCHITECTURE OF COMPUTERS AND OPERATING SYSTEMS

Bioinformatics

Course 2022/23 T3

UPC Departament d'Arquitectura de Computadors

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Concepts

Physical memory vs. Logical memory

Process address space

Addresses assignment to processes

Operating system tasks

Hardware support

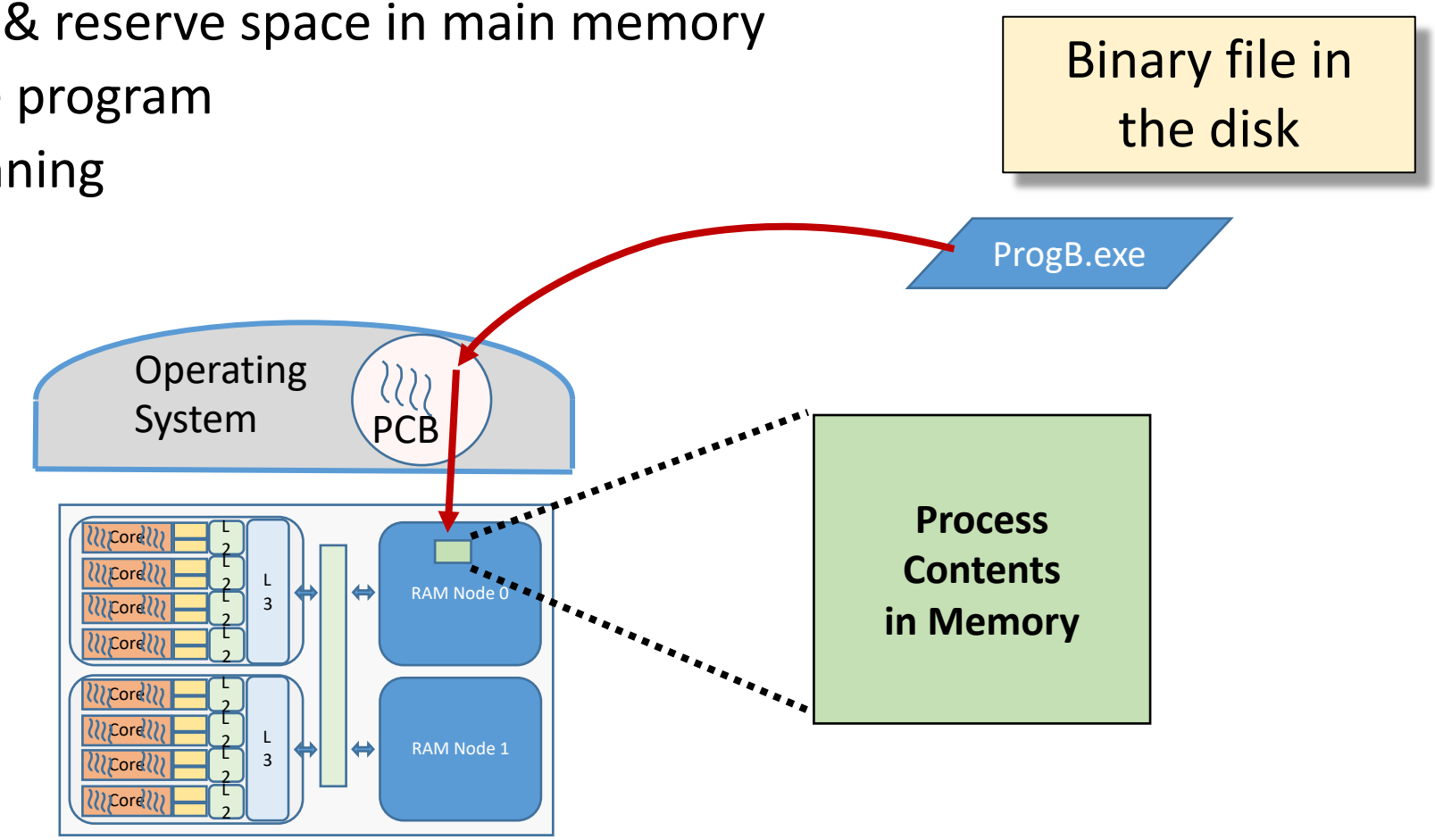
Memory Management

- The CPU can only access directly to memory and the register bank
  - Instructions and data must be located in main memory

- The CPU sends out **logical addresses (logical @s)**
- The requested instructions/data are located in **physical addresses**
- Logical @s **may not directly match** the correspondent physical @s
  - The OS in conjunction with the Hardware manages this translation
    - logical @ → physical @
- The process uses **virtual memory** to become larger than main memory size
  - Logical addresses point to virtual memory locations

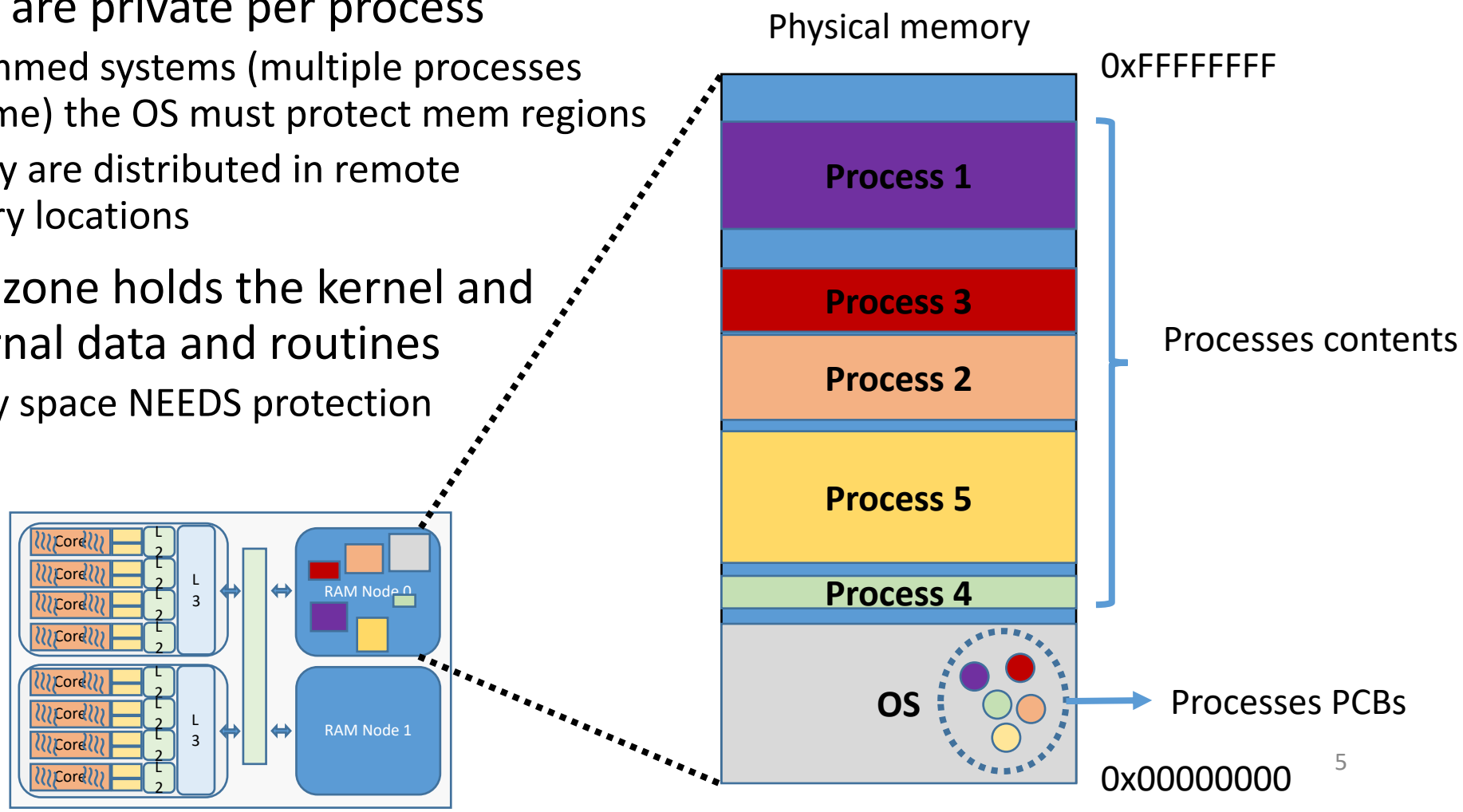
Program Loading

- The OS loads the program from the disk to Physical Memory
  1) Request & reserve space in main memory
  2) Load the program
  3) Start running



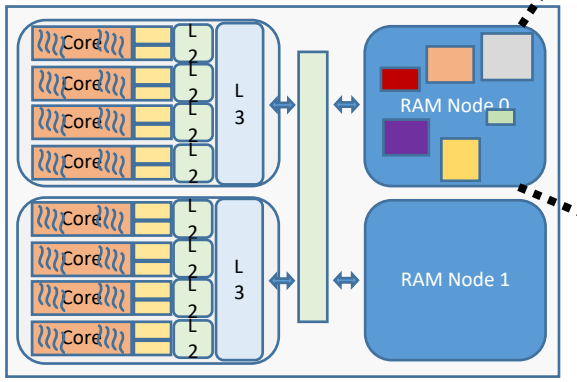
Multiprogrammed OS

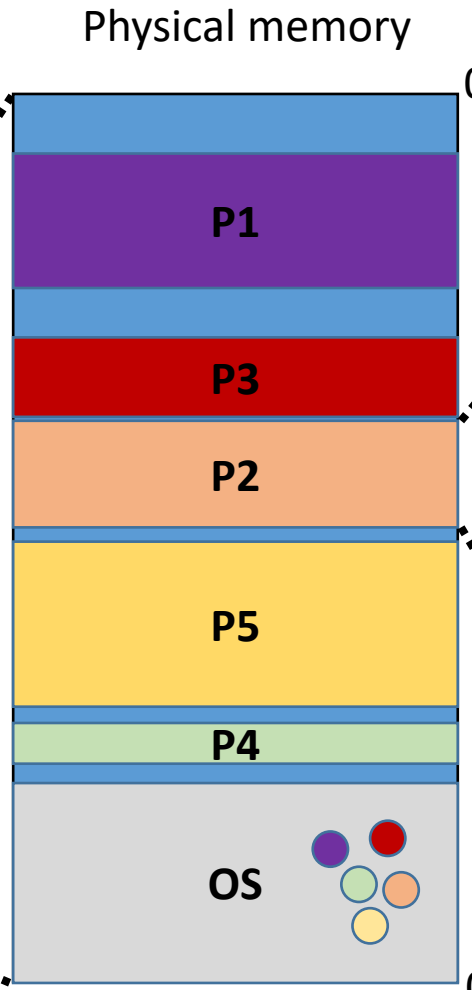
- Memory regions are private per process
  - In multiprogrammed systems (multiple processes are alive at a time) the OS must protect mem regions
  - Contents usually are distributed in remote physical memory locations
- The OS memory zone holds the kernel and all required internal data and routines
  - The OS memory space NEEDS protection

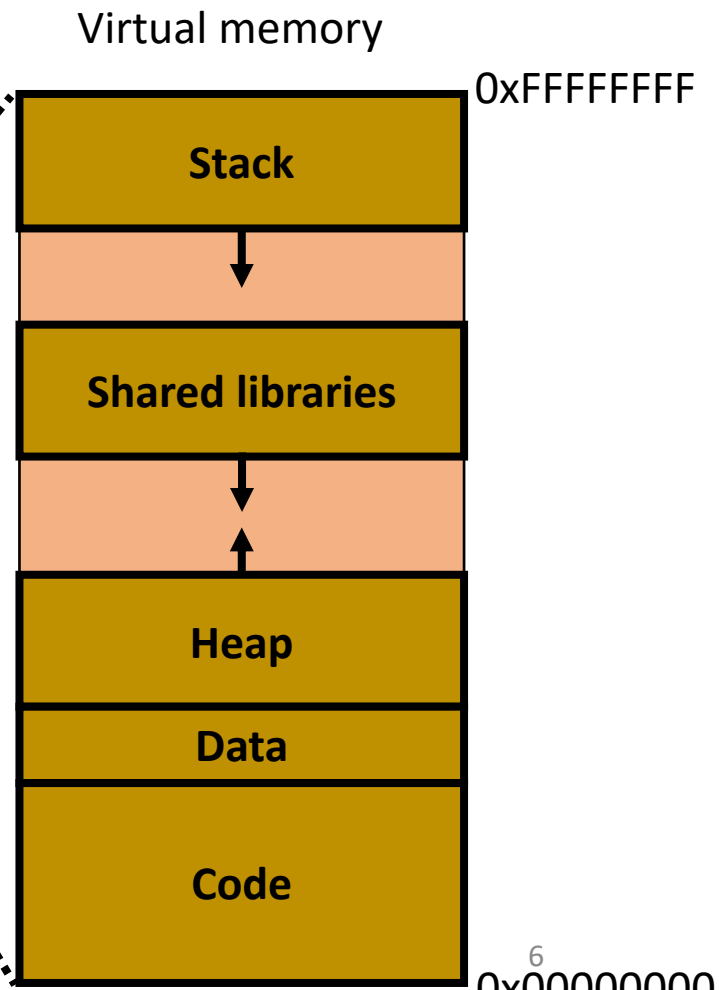


Process Contents in Memory

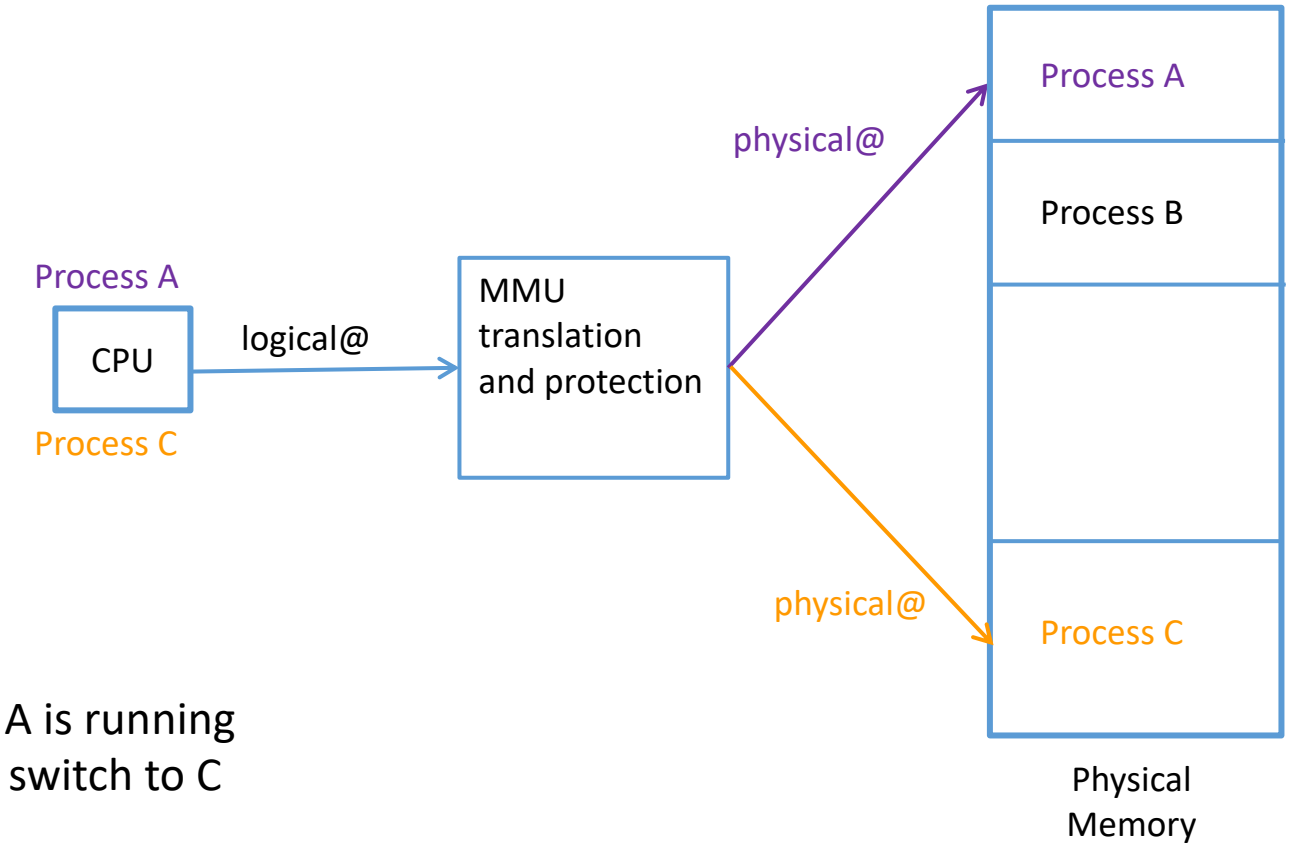
- **Stack**: dynamic mem
  - Function arguments
  - Local Variables
- **Shared libraries**: Code, data…
- **Heap**: dynamic mem
  - Mem allocated at runtime
- **Data**: .bss & .data
  - Global variables
- **Code**: .text
  - Instructions







Multiprogrammed systems



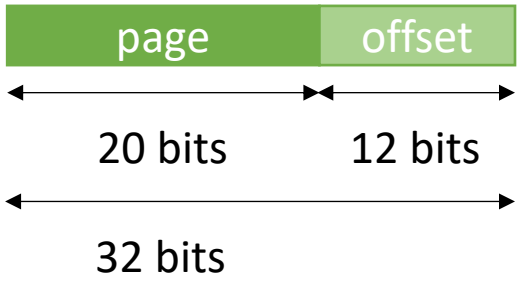
1-Process A is running
2-Context switch to C

Paging: main concepts

- Page
  - Fixed size amount of memory
    - Its size needs to be power of 2 to simplify the hardware
    - In the order of few Kbytes (4Kb, 8Kb, …)
- Logical memory is divided into pages (page)
- Physical memory is divided into pages (frame)
- The OS maps logical pages into frames (physical pages)
  - We need HW support
    - This support I called MMU (Memory Management Unit)

Memory addresses

- If we envision the memory as a vector of bytes
  - The address is “the index of the vector”
- Memory addresses are between 16 to 64 bits long
  - The exact length depends on the processor architecture
- Let’s assume a 32-bit memory address and 4K size pages
  - To address 4K bytes = 4096 bytes I need 12 bit

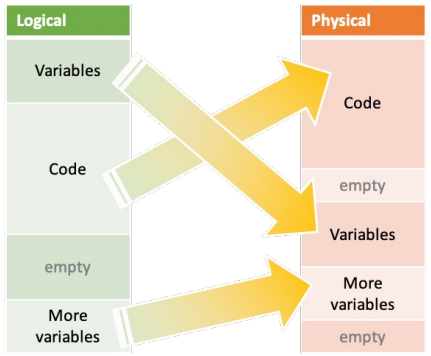


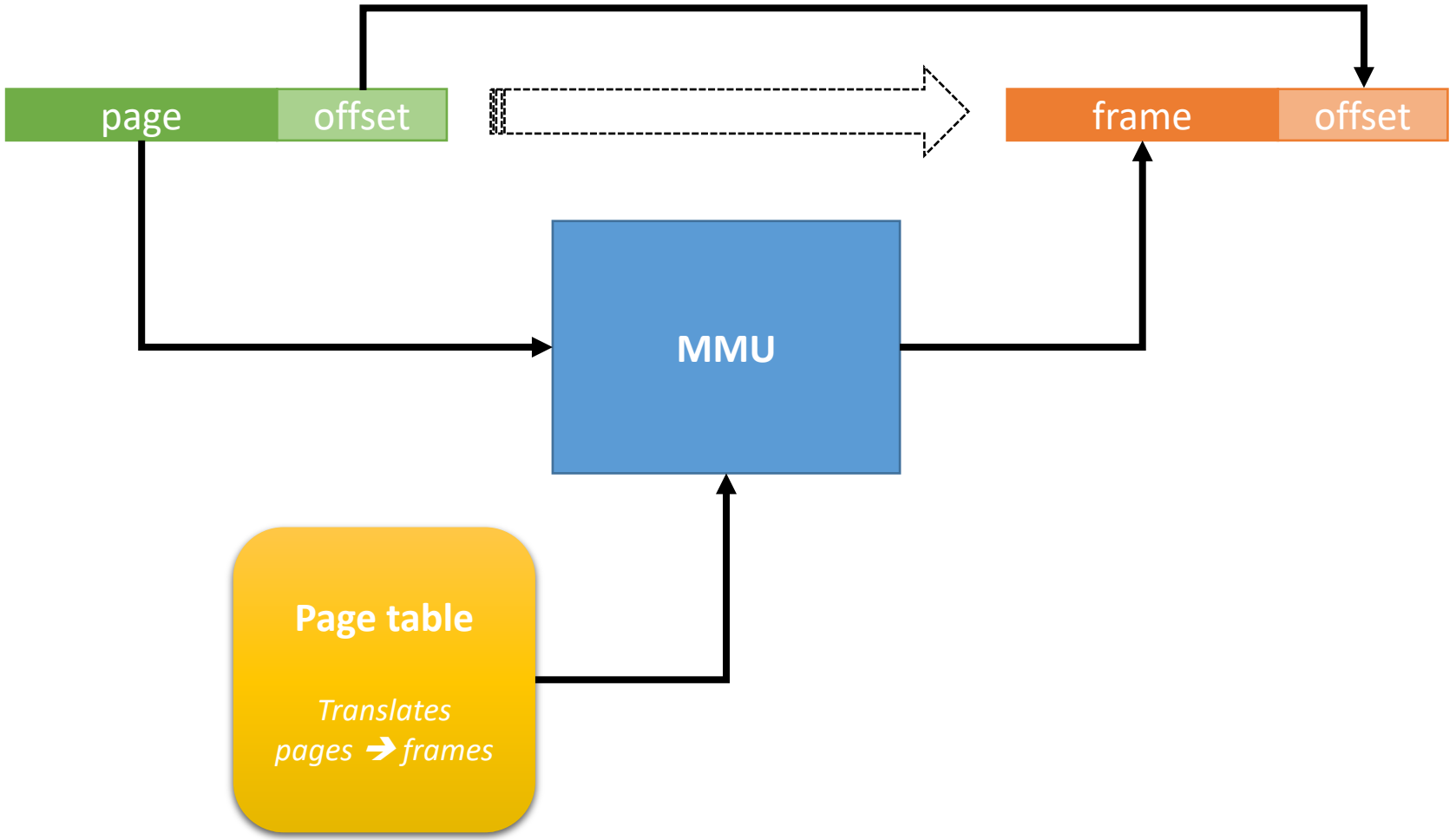
  - To get the physical address, we just need to change the page, not the offset

Page and offset: example

- Computing the page and object of an address
  - Logical addresses ➔ 32 bits
  - Page size ➔ 4Kbytes ➔12 bits
  - What page and what offset has address 12356?
- Using the bits
  - 12356 = 0000 0000 0000 0000 0011 0000 0100 0100
  - Page = 0000 0000 0000 0000 0011 = 3
  - Offset = 0000 0100 0100 = 68
- Dividing
  - Page = ⌊12356/4096⌋ = 3
  - Offset = 12356 module 4096 = 68

MMU simplified schema





Memory Management Unit

- the MMU is a hardware component which, at least, offers **address translation and memory access protection**. It can also support other management tasks.
- **OS is responsible for configuring the MMU with the correct address translation values for the current process in execution**
  - Which logical @ are valid and which are their corresponding physical @
  - Guarantees that each process gets assigned only its own physical @
- **HW support to translation and protection between processes**
  - MMU receives a logical @ and translates it to the corresponding physical address using its data structures
    - It throws an exception to the OS if the logical address is not marked as valid or if it has not associated a physical address
  - **OS manage the exception according to the situation**
    - For example, if the logical address is not valid it can kill the process (SISEGV signal)

Hardware support: translation

- **When** does the OS need to update the address translation information???
- Case 1: When assigning memory
  - Initialization when **assigning new memory (mutation, execlp)**
  - **Changes in the address space:** grows/diminishes. When allocating/deallocating dynamic memory
- Case 2: When switching contexts
  - For the process that leaves the CPU: if it is not finished, then keep in its data structures (PCB) the information to configure the MMU when it resumes the execution
  - For the process that resumes the execution: configure the MMU

HW Support : Protection

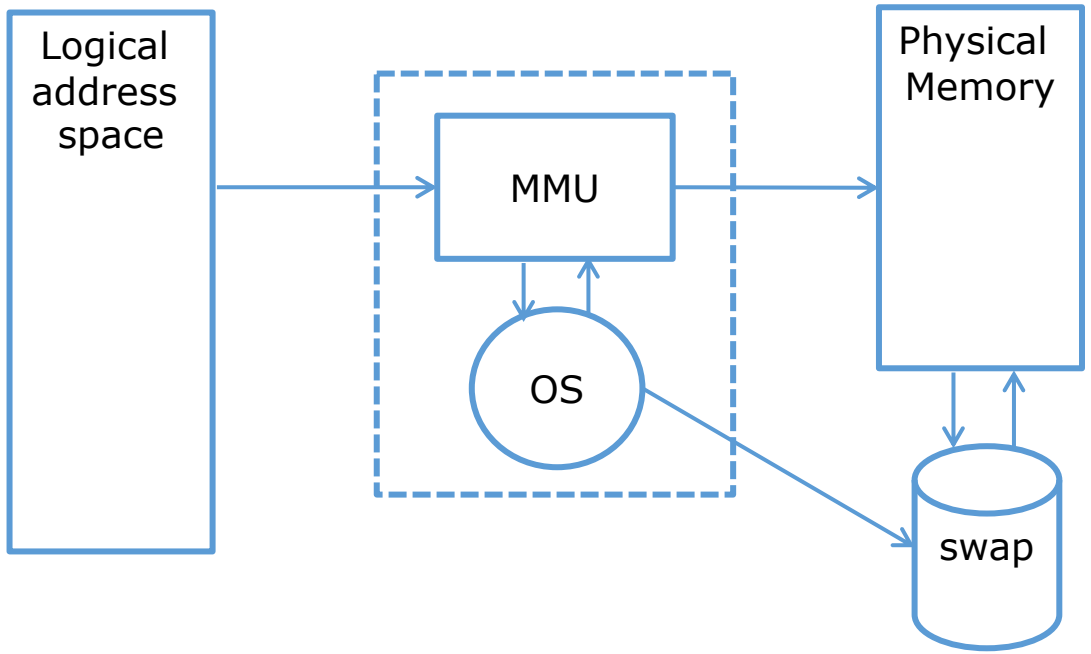
- It is performed in the same cases than the memory assignment
- It also enables to implement protection against undesirable accesses/type of accesses
  - Invalid logical addresses
  - Valid logical addresses but wrong type of access (writing on a read-only region)
  - Valid logical address and apparently “wrong” type of access due to some optimization implemented by the OS
    - For example, COW (we will explain it later)
  - For all cases → exception captured by the CPU and managed by the OS
  - OS has all the information about the description of the process address space and can check if the exception is really due to wrong access or not

Optimizations: Virtual memory

- Virtual memory
  - Extension for the on-demand loading optimization
  - In addition to load pages on-demand, it enables the system to take out pages that are not needed at a given time
  - Goal
    - To reduce amount of physical memory assigned to a process
      - **A process only needs physical memory to hold the current instruction and the data that this instruction references**
    - To increase potential multiprogramming grade
      - Amount of concurrent processes

Optimizations: Virtual memory (III)

- Virtual memory based on paging
  - **Logical address space of a process is distributed across physical memory (present pages) and swap area (non-present pages)**



Virtual memory: concept

- Let's examine an address
  - 32-bit memory addresses → memory size can be 4.294.967.296 bytes
    - 4G per process
  - 64-bit memory addresses → 16 Exabytes x process
  - No system has so much physical memory
- How about putting in disk what does not fit in memory?
  - This would allow to have much larger "memories"
  - What we need
    - Presence bit → Extra bit in each page-table entry to know whether a page is in memory
    - Dirty bit → Extra bit in each page-table entry to know whether a page has been modified
    - Page fault mechanism
    - Mechanism to find a page into the disk (we can use the page table)
    - Replacing algorithm

Virtual memory: how it works

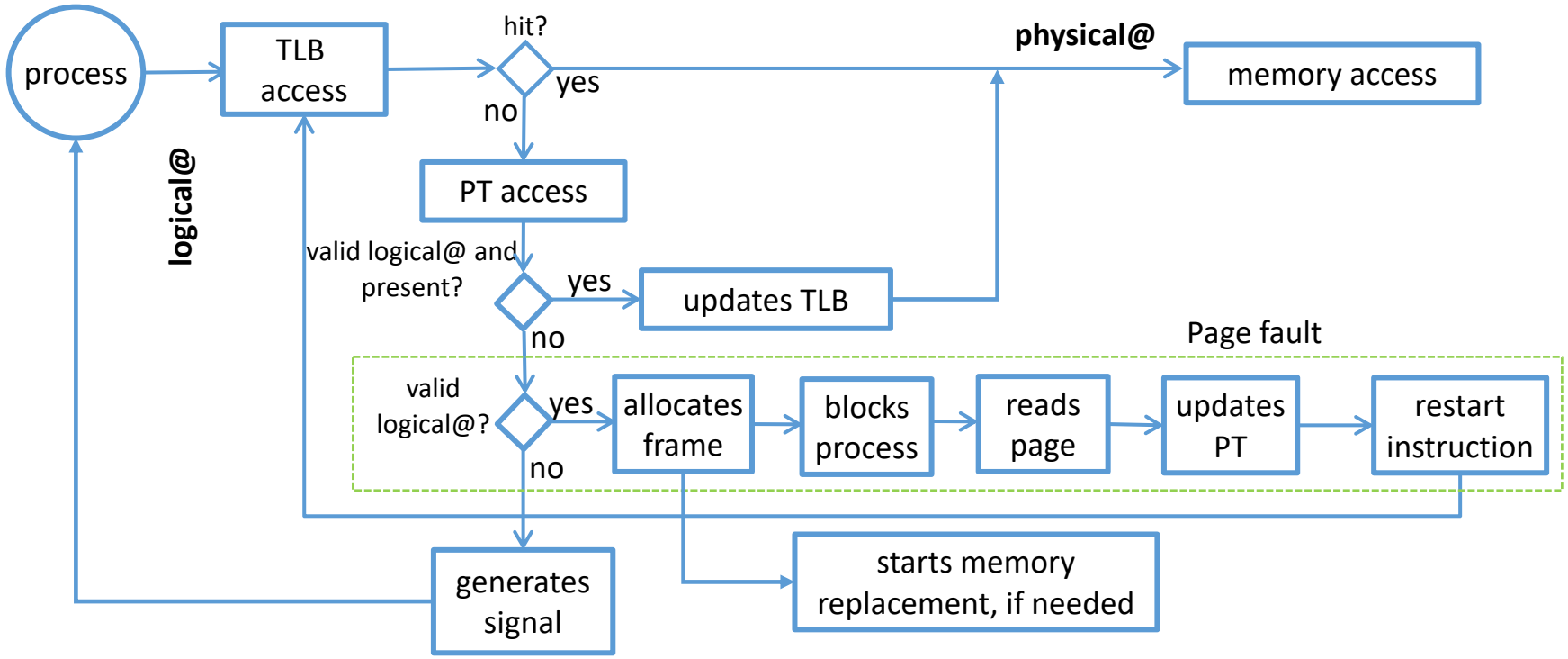
- When the MMU translates a page it can
  - Find the presence bit to 1 = the page is in memory
    ➔ same as we have already explained
  - Find the presence bit to 0 = the page is NOT in memory
    ➔ Raise an interruption (call the OS to solve the ”problem”)
- Page fault
  - Reaction of the OS to a missing page
  - Decide which page to move out of memory (if no empty memory available)
    - FIFO
    - Not Recently Used (needs hardware support)
  - Move the page to disk if it has been modified
  - Bring the page from disk and update page table
  - Restart memory access

TLB and page table

- Page table can be very big
  - Where do I put it? → memory
- Putting the page table in memory is very slow
  - Every memory access needs the page table
  - Every access would need to access the memory twice
    - PANIC!!!
- Solution → TLB (Table lookup buffer)
  - Keep a portion of the page table in the MMU
  - As programs have locality this works very well
  - TLB misses → similar to page faults

Virtual memory

- Memory access steps:

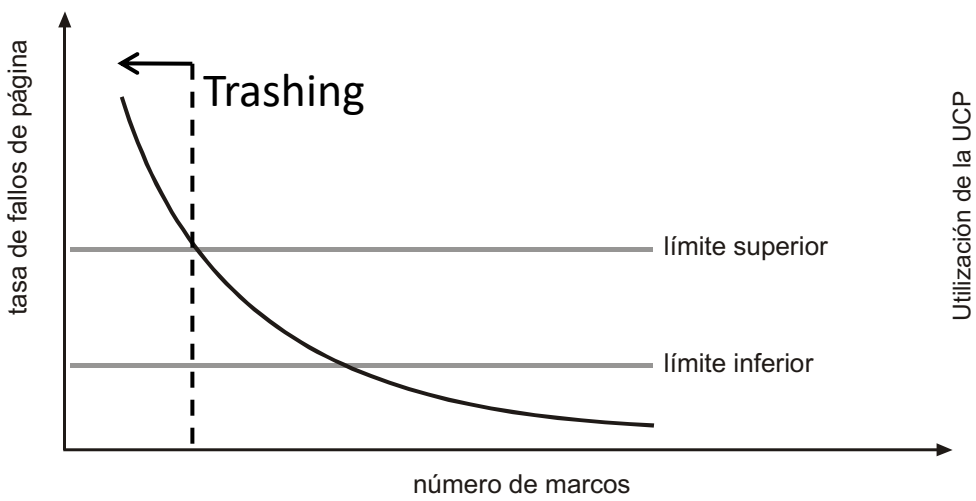


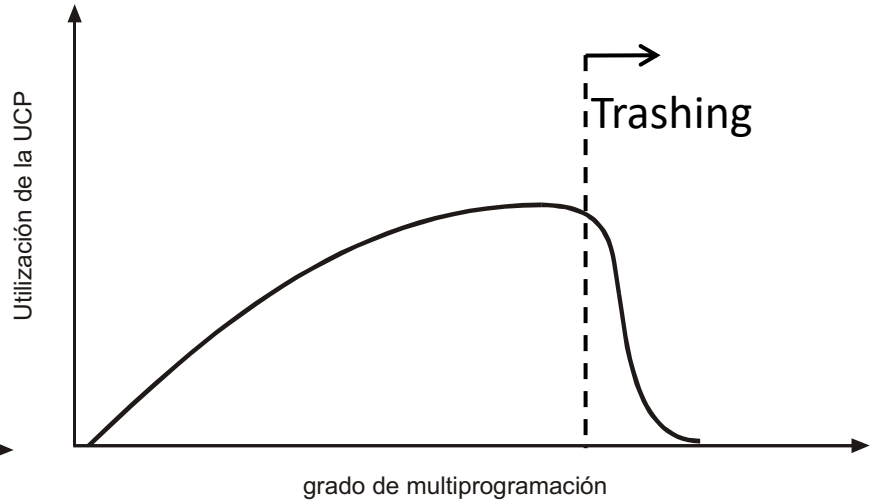
Virtual memory

- Effects of using virtual memory:
  - Physical memory can be smaller than the sum of the address spaces of the loaded processes
  - Physical memory can be smaller than the logical address space of a single process
  - Accessing to non-present pages is slower than accessing to present pages
    - Exception + page loading
    - It is important to reduce the number of page faults

Virtual memory

- Thrashing
  - Process is in thrashing when
    - **It spends more time performing page swapping than executing program code**
    - It is not able to keep simultaneously in memory the minimum number of pages required to advance with the execution.
  - Memory system is overloaded
    - Detection: to control page fault rate per process
    - Management: to control multiprogramming grade and to swap out processes





Optimizations: COW (Copy on Write)

- Goal: to delay allocation/initialization of physical memory until it is really necessary
  - If a new zone is never accessed → it is not necessary to assign physical memory to it
  - If a copied zone is never written → it is not necessary to replicate it
  - Save time and memory space
- Example: fork
  - **Delays copy of each region (code, data, etc.) until it is accessed for writing**
  - Avoids physical copy for those regions that are only read (for example, code region)
  - It is usually implemented at page-level: frames are allocated/copied when pages are accessed to write
- It can be applied
  - In the address space of one process: dynamic memory case
  - Between processes: fork case

COW: Implementation

- **Overview: OS needs a mechanism to detect writes and to perform the physical memory allocation and the copy**
- When the logical memory region is allocated:
  - OS registers in the MMU the new region with the same physical memory than the source region
  - OS registers the new region in the data structure describing the address space of the process (in the PCB), indicating which are the real permissions of access
  - OS marks in the MMU both new region and source region as read-only regions
- When a process tries to write on the new region or on the source region:
  - MMU throws an exception to the OS. OS management code performs the actual allocation and copy, updates MMU with the real permission for both regions and resets the instruction

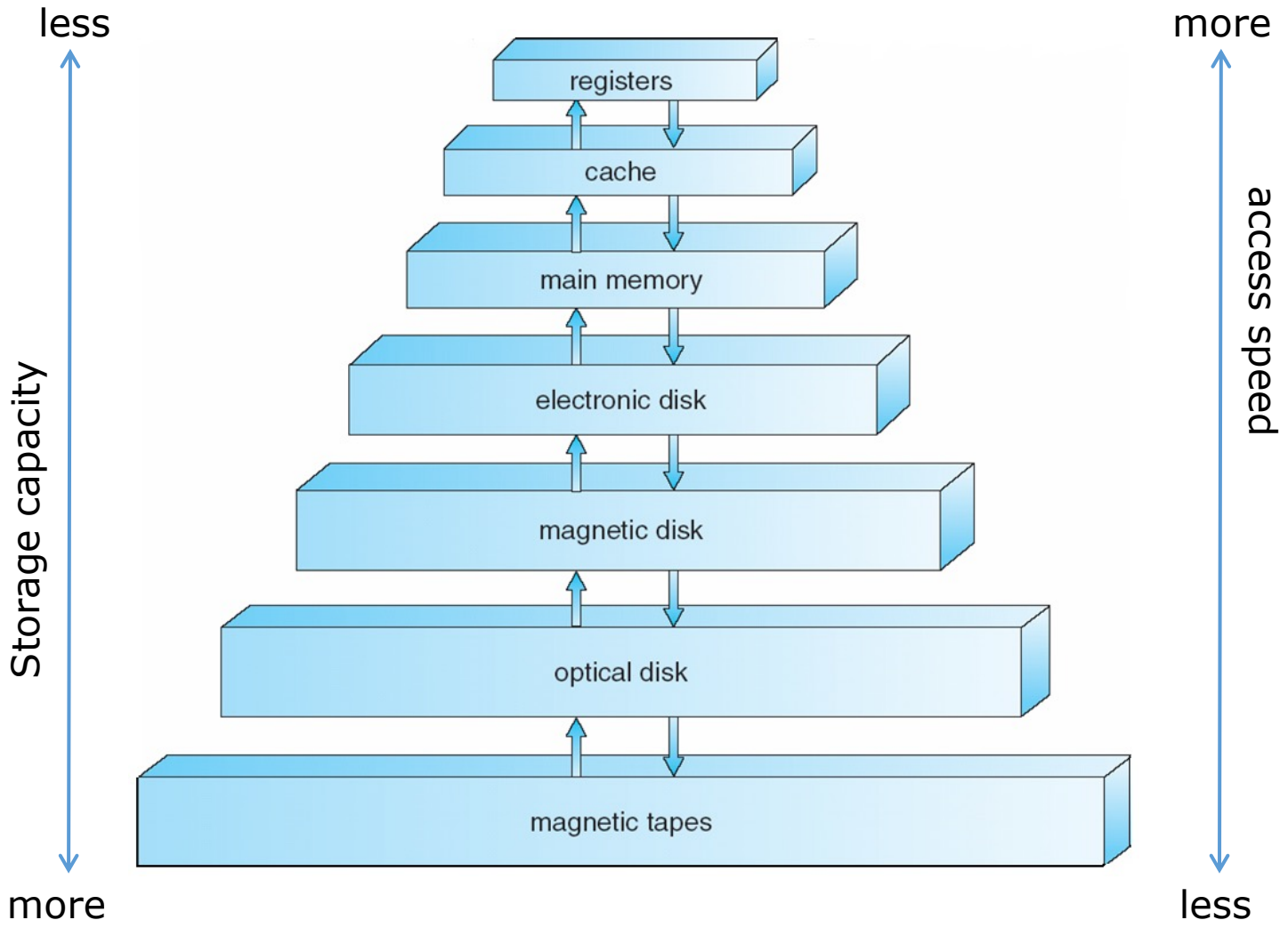
COW: example

- Process A physical memory assignment:
  - Code: 3 pages, Data: 2 pages, Stack: 1 page, Heap: 1 page
- Let's consider that process A executes a fork system call. Just after fork:
  - Total physical memory:
    - Without COW: process A= 7 pages + child = 7 pages = 14 pages
    - With COW: process A= 7 pages + child =0 pages = 7 pages
- Later on the execution… depends on the code executed by the processes, for example:
  - If child executes an exec (and its new address space uses 10 pages):
    - Without COW: process A= 7 pages+ child = 10 pages= 17 pages
    - With COW: process A= 7 pages+ child A=10 pages= 17 pages
  - If child does not execute an exec, at least code will be always shared between both processes and the rest of the address space depends on the code. If only the code is shared:
    - Without COW: process A= 7 pages+ child A= 7 pages= 14 pages
    - With COW: process A= 7 pages+ child A=4 pages= 11 pages
- **In general, in order to compute the amount of required physical memory it is necessary to compute how many pages are modified (and thus cannot be shared) and how many pages are read-only (and thus can be shared)**

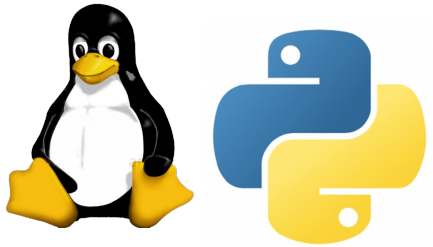
Optimizations: Prefetch

- Goal: to minimize number of page faults
- Overview: to predict which pages will need a process and load them in advance
- Parameters to consider:
  - Prefetch distance: time between the page loading and the page reference
  - Number of pages to load in advance
- Some simple prediction algorithms:
  - Sequential
  - Strided

Storage hierarchy



System calls



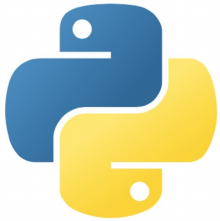
- Allocation/deallocation memory
  - Not used in python
    - Internal to create/destroy objects
- Mmap
  - Maps a file into memory
    - Allows the program to access a file as if it were a vector in memory
  - mmap.mmap(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE|PROT_READ*, *access=ACCESS_DEFAULT*[, *offset*])
    - `fileID` file descriptor of an open file
    - `length` (if 0 then size of the file)
    - `Flags`
      - `MAP_SHARED`
      - `MAP_PRIVATE`
    - `offset` must be a multiple of PAGESIZE

Mmap: example

```
import mmap
import os

fd = os.open("FILE",
             os.O_RDWR)

i= 0
while i < 10:
    buff = os.read(fd,1)
    print (buff)
    i = i + 1

os.close(fd)
```

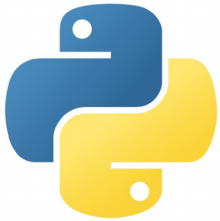
```
import mmap
import os

fd = os.open("FILE",
             os.O_RDWR)

i = 0
v = mmap.mmap(fd,0)
while i < 10:
    print (v[i]))
    i = i + 1

v.close()

os.close(fd)
```

Mmap: example of reading a file



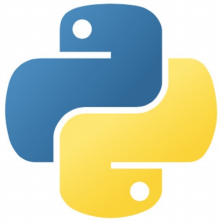
```
import mmap
import os
import sys

fd = os.open("SRR000049.fastq",os.O_RDONLY)
mm = mmap.mmap(fd,0, access=mmap.ACCESS_READ)
header = mm.readline()
print(header)
mm.seek(0)
i = mm.find(b'=')
print("Lenghth of the sequence = ",mm[i+1:i+4].decode())
mm.close()
os.close(fd)
```

Mmap: example of scratchpad between parent and child



```
import mmap
import os

mm = mmap.mmap(-1, 12)
mm.write(b"Hello world!")
pid = os.fork()
if pid == 0:
    mm.seek(0)
    os.write(1,mm.readline())
    os.write(1,b"\n")
    mm.close()
else:
    mm.close()
    os.wait()
```

Addresses and values in Python



- Everything in Python is an object
- Every object has a type, an address, and a value
- Type and address never change
- A **mutable** object can change its value
  - list, dict, set, bytearray
- An **immutable** object can't
  - int, float, string, …

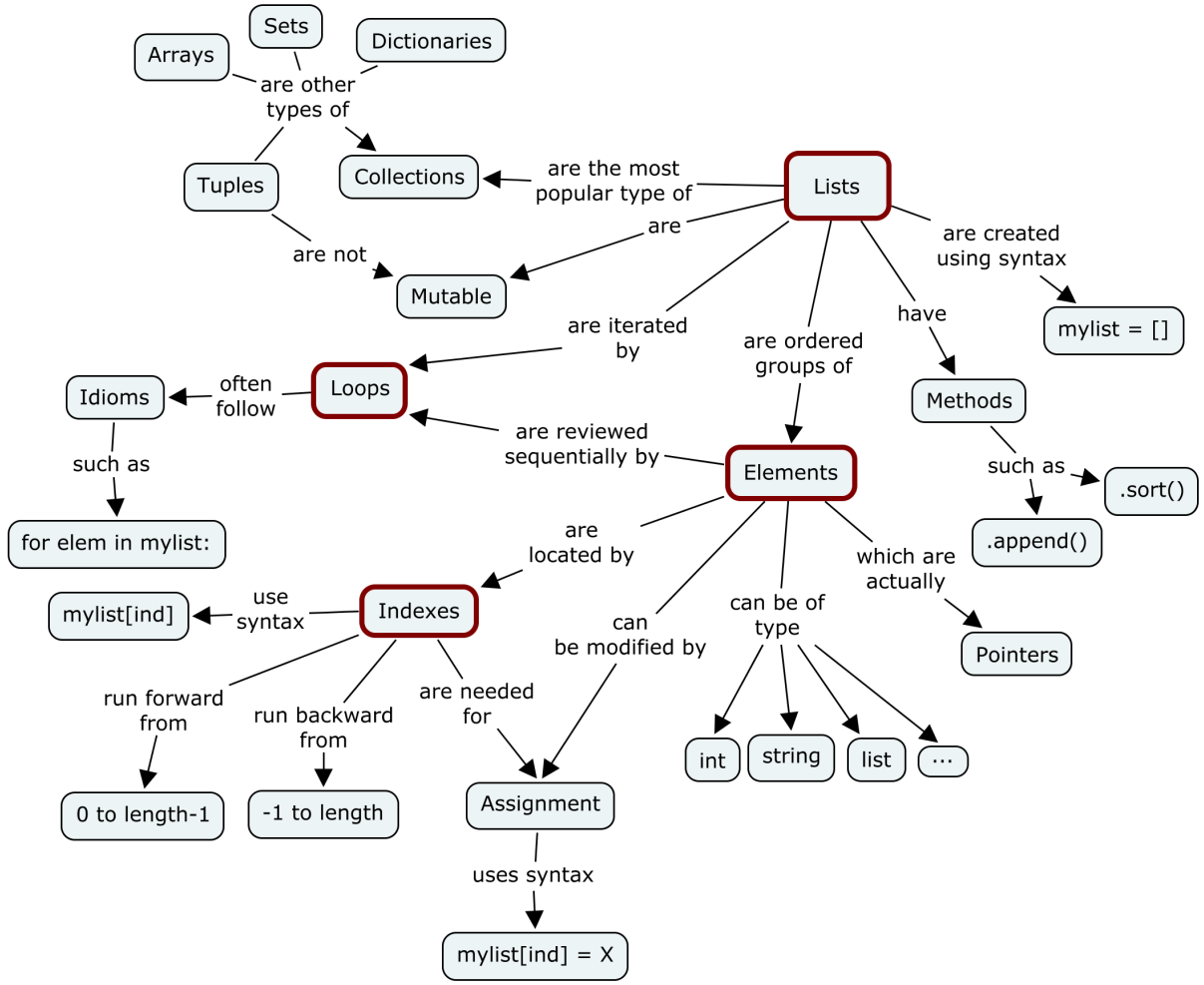
@	value
0x0000:	04
0x0001:	D2

Addresses and values in Python

```
 jfornes@tiacos:~/iolab$ python3
Python 3.8.5 (default, Sep  4 2020,
07:30:14)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or
"license" for more information.
>>> x = 3
>>> y = 3
>>> id(y)
94436516330880
>>> id(x)
94436516330880
>>> id(3)
94436516330880
>>> x += 2
>>> id(x)
94436516330944
```

```
>>> id(5)
94436516330944
>>> type(x)
<class 'int'>
>>> type(y)
<class 'int’>
>>> l = ['A','B','C']
>>> l
['A', 'B', 'C']
>>> p = l
>>> id(p)
140681145616832
>>> id(l)
140681145616832
>>> p.append('D')
>>> l
['A', 'B', 'C', 'D']
```

Mutable vs Immutable Objects in Python



https://medium.com/@meghamohan/mutable-and-immutable-side-of-python-c2145cf72747

Summary

- Physical address space
  - Memory addresses
- Logical address space
  - What the program sees
- MMU
  - Translates logical addresses to physical addresses
- Virtual memory
  - Enables to have more logical memory than physical
- TLB
  - Cache fo the page table
- Using too much memory can delay your program significantly

Bibliography

- Operating System
  - Silberschatz, A; Galvin, P. B; Gagne, G. 2019. Chapters (11-15)
  - https://cataleg.upf.edu/record=b1498664~S11*cat
- Python documentation
  - https://docs.python.org/3/library/os.html
  - https://docs.python.org/3/library/mmap.html
- Computer Systems. A programmers perspective
  - Randal E. Bryant, David R. O'Hallaron 2015. Chapter 10.
  - https://upfinder.upf.edu/iii/encore/record/C__Rb1318766