Computer Fundamentals
Logical Address Space (2/2)

Grau en Intel·ligència Artificial

Xavier Martorell

Xavi Verdú

Facultat d’Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

2021-2022 Q1

Creative Commons License

This work is under a Creative Commons Attribution 4.0 Unported License

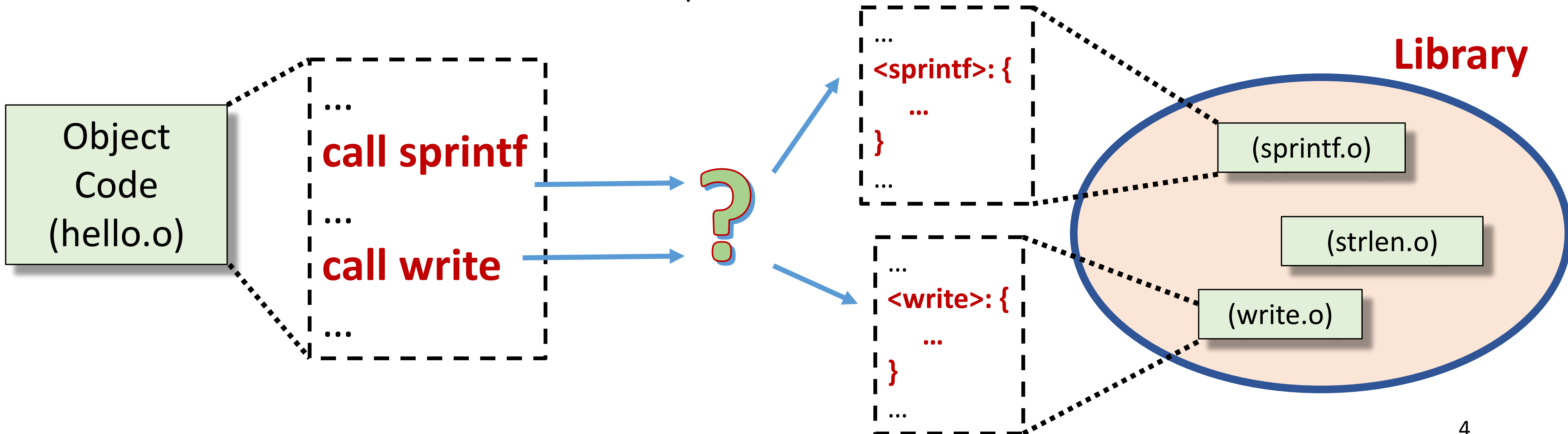
The details of this license are publicly available at
https://creativecommons.org/licenses/by-nc-nd/4.0

Table of Contents

- This lesson consists of three main parts
    1) Basic Concepts
    2) **Libraries and OS suport**

Libraries

- **Library:** a file that encapsulates a set of object files
  - Libraries grant **independency**
    - A high-level code is independent of **OS** and **Hw architecture**, but …
    - … executables are created for a particular **OS** and **Hw architecture**



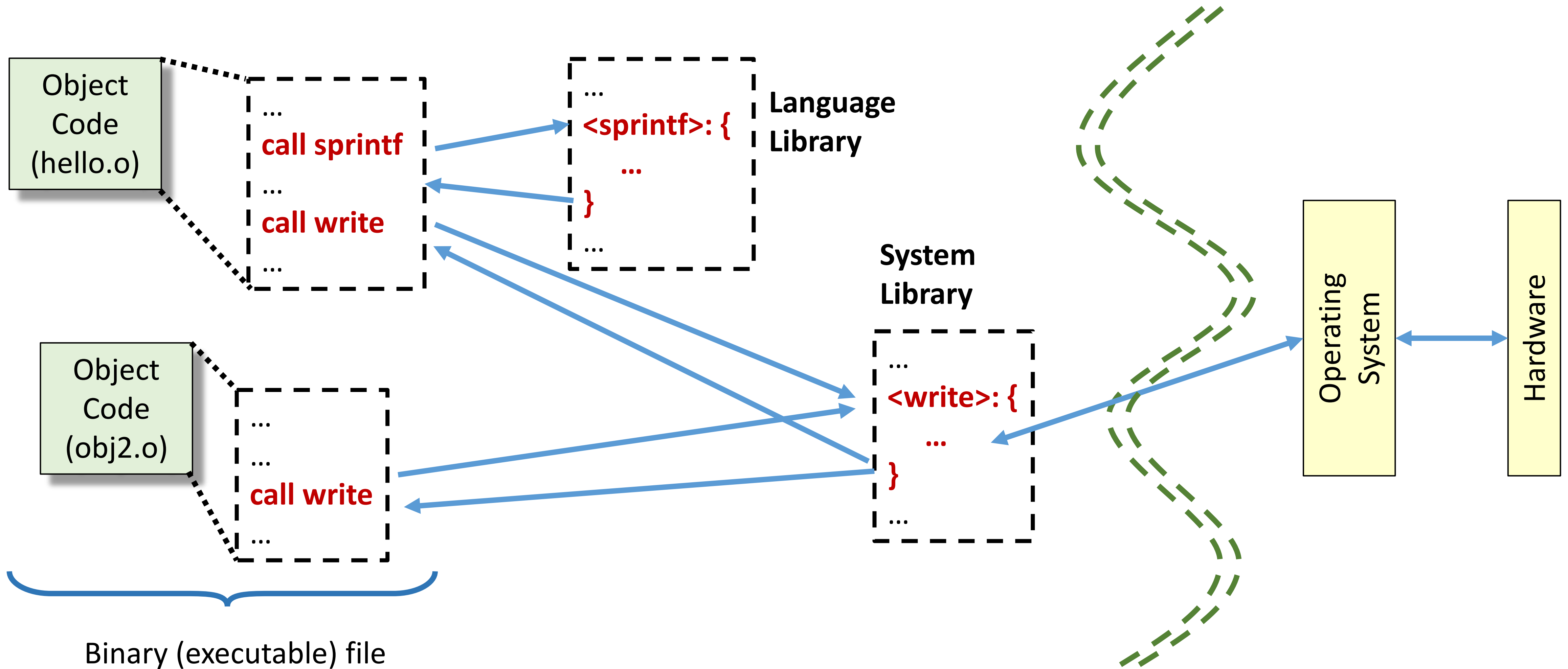
Types of Libraries

- Language Libraries
  - Bind a language to a particular OS, but independently of the hardware architecture
  - Provide functions of the programming language
  - Sometimes they are self-contained (e.g. **math**): independent of OS
  - Sometimes they request services to the OS (e.g. **sprintf**, cin, cout…)

- System Libraries
  - Bind an OS to a particular Hw architecture (independent of the high-level code)
  - Include functions that invoke services/routines of the OS (e.g. **write**, open, …)
  - Depends on the type of Operating System and the architecture of the CPU

- Examples of incompatibility:
  - Linux 32 vs 64 bits, CPU Intel vs ARM, Ubuntu vs OpenSUSE, …

Global Picture



Let’s assume an example

- From high-level code to program execution

```
#include <unistd.h>
#include <stdio.h>

int global = 4;

int main(int argc, char **argv){
        char buf[512];
        int num;

        num = sprintf(buf, "Hello World %d!\n", global);
        write(1, buf, num);

        return 0;
}
```

High-Level Code (hello.c)

High Level Language Code specified in a file that cannot be understood by the CPU to be executed right away

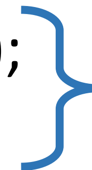
Let’s assume an example

```
#include <unistd.h>
#include <stdio.h>

int global = 4;

int main(int argc, char **argv){
        char buf[512];
        int num;

        num = sprintf(buf, "Hello World %d!\n", global);
        write(1, buf, num);

        return 0;
}
```

Including Header Files

Global Variable

Local Variables



Calling to external functions

Return the exit status of the program,
Indicating success(0) or some failure(1)

Header files contain required information for the compiler about external components used in this code, such as “sprintf” and “write” functions

Static versus Dinamically linking

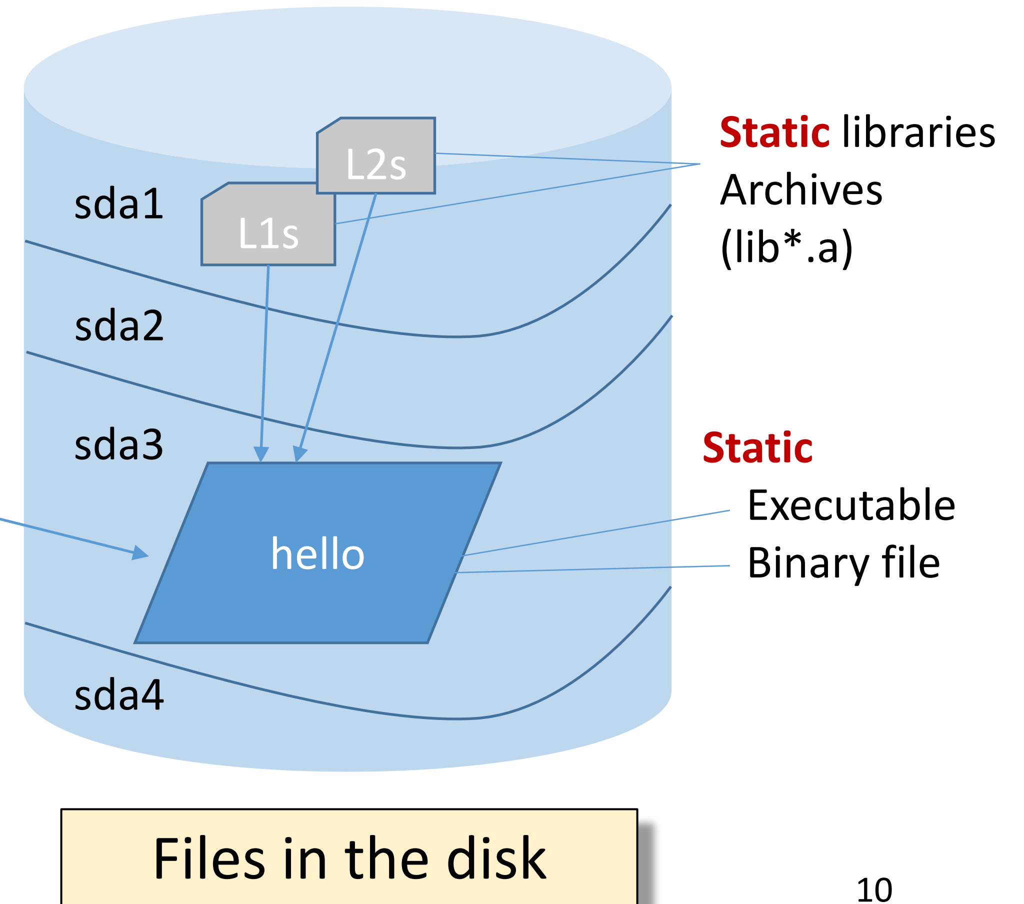
- Static linking
  - The codes of libraries are included INSIDE the executable
  - Waste of space (in disk when stored, and in memory when running)
    - Different programs may use the same libraries
  - Static libraries: "**.a**" in UNIX-like OS; "**.lib**" in Windows-like OS
- Dynamic linking
  - The linking stage is delayed to run-time. Thus, the code is not included in the program
  - Sharing libraries
  - Save space in both disk and memory
    - Different programs can access to a shared library loaded in memory
  - Dynamic libraries: "**.so**" in UNIX-like OS; "**.dll**" in Windows-like OS

Static linking

- Static compilation and linking phases

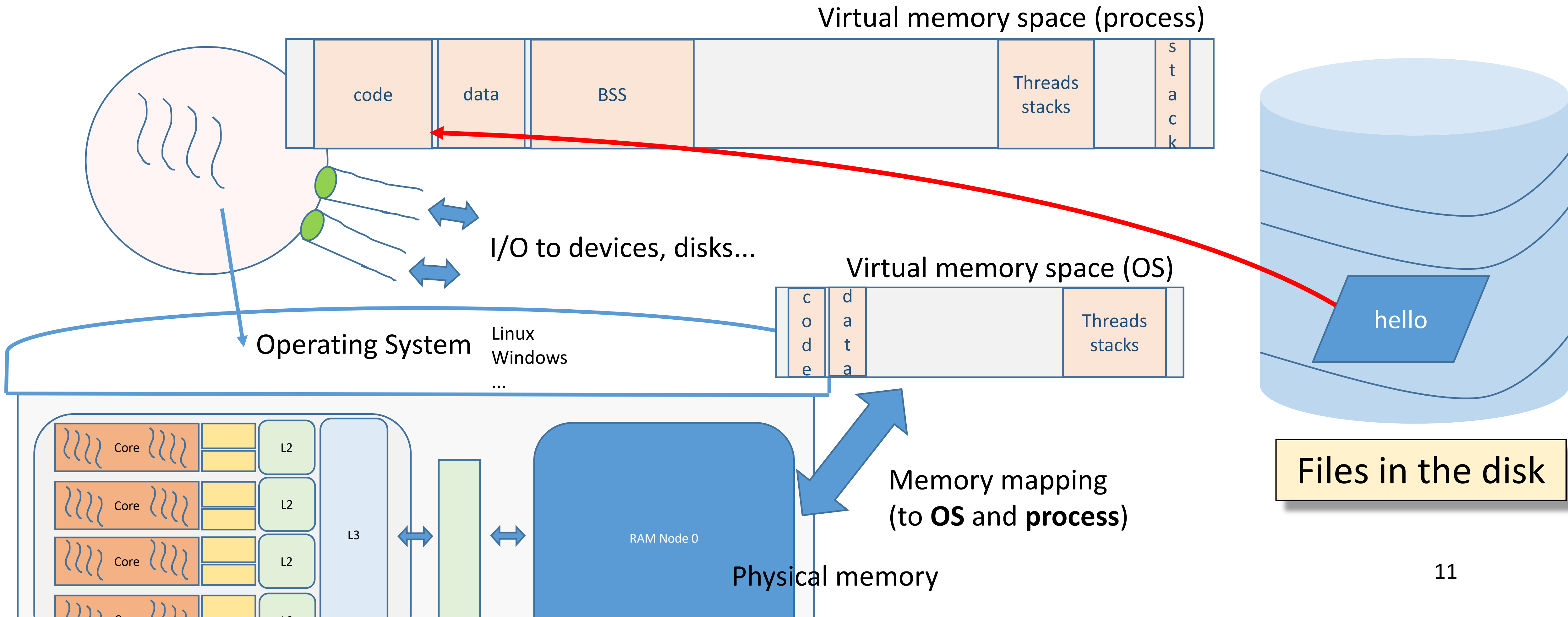
```
#include <unistd.h>
#include <stdio.h>

int global = 4;

int main(int argc, char **argv){
        char buf[512];
        int num;

        num = sprintf(buf, "Hello World %d!", global);
        write(1, buf, num);

        return 0;
}
```

gcc **-static** -o hello hello.o
(compile & link)



Process with static libraries

Logical addresses refer to virtual memory addresses
due to the OS support

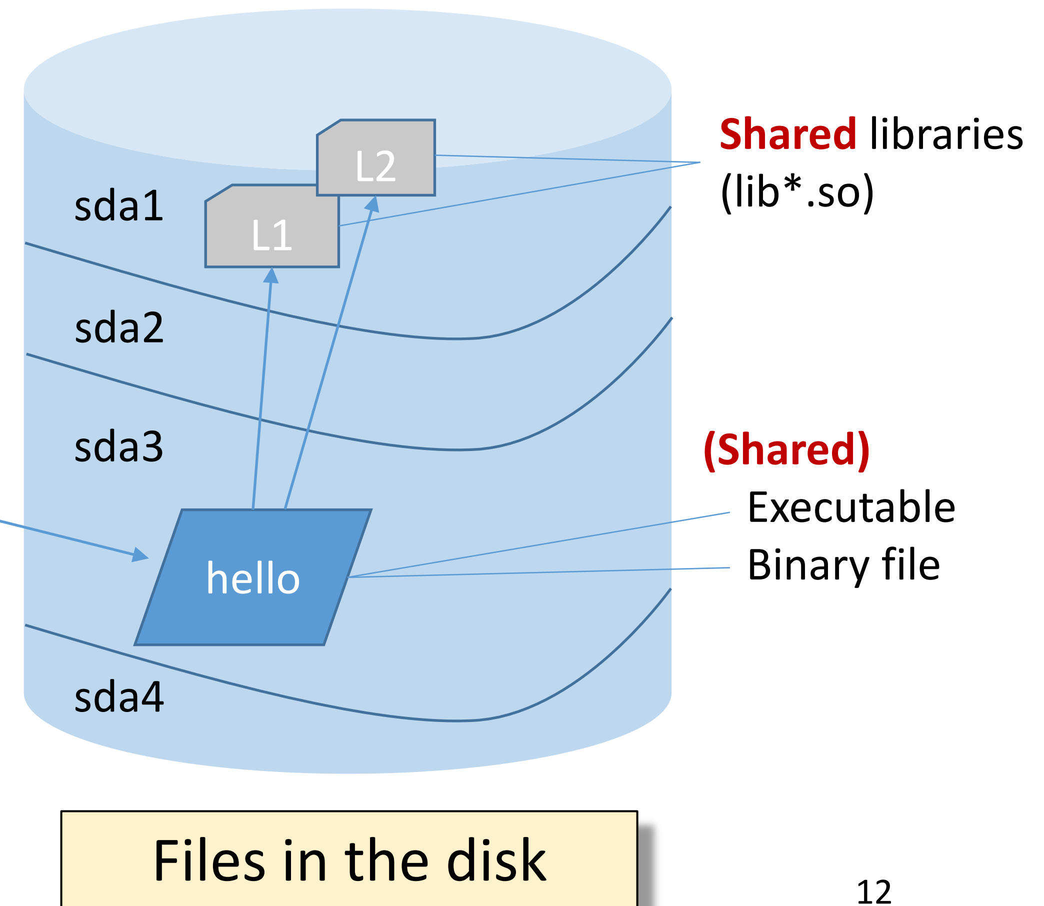


Dinamically linking

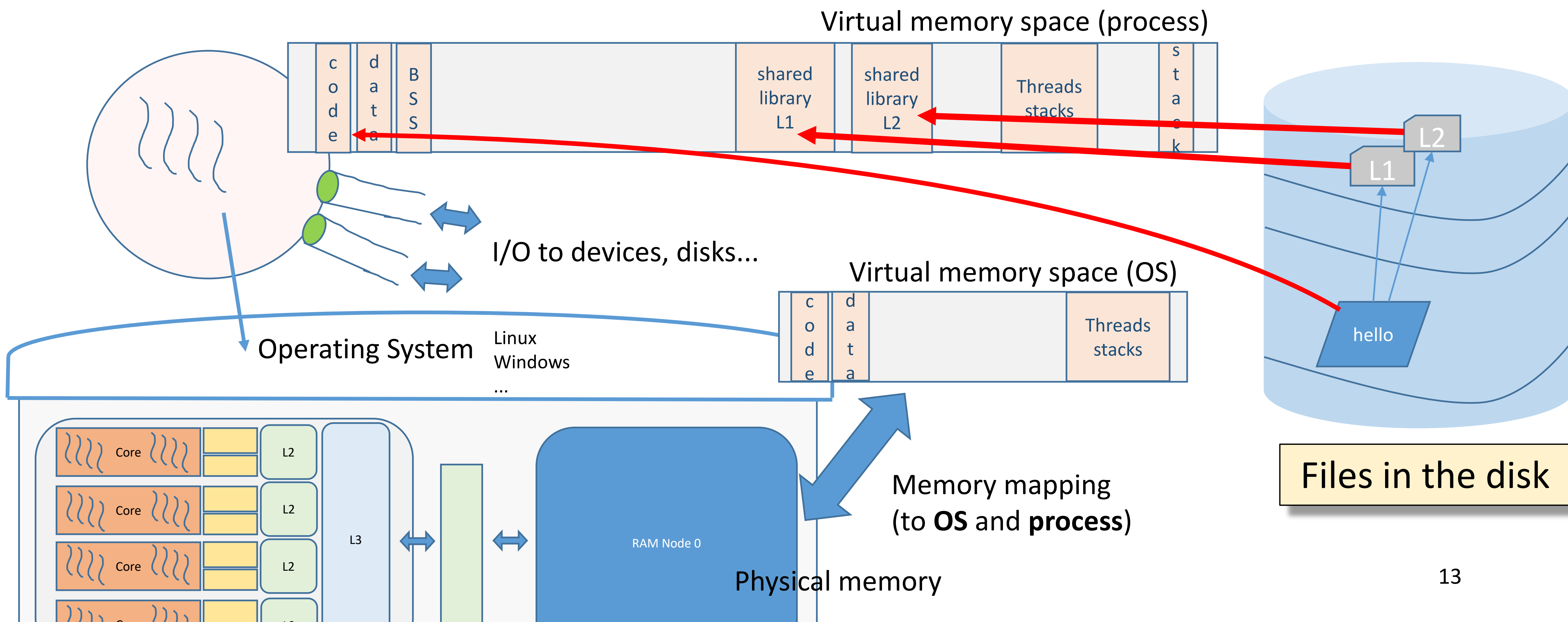
- Dynamic compilation and linking phases

```
#include <unistd.h>
#include <stdio.h>

int global = 0;

int main(int argc, char **argv){
        char buf[512];
        int num;

        num = sprintf(buf, "Hello World %d!", global);
        write(1, buf, num);

        return 0;
}
```

gcc -o hello hello.o
(compile & link)

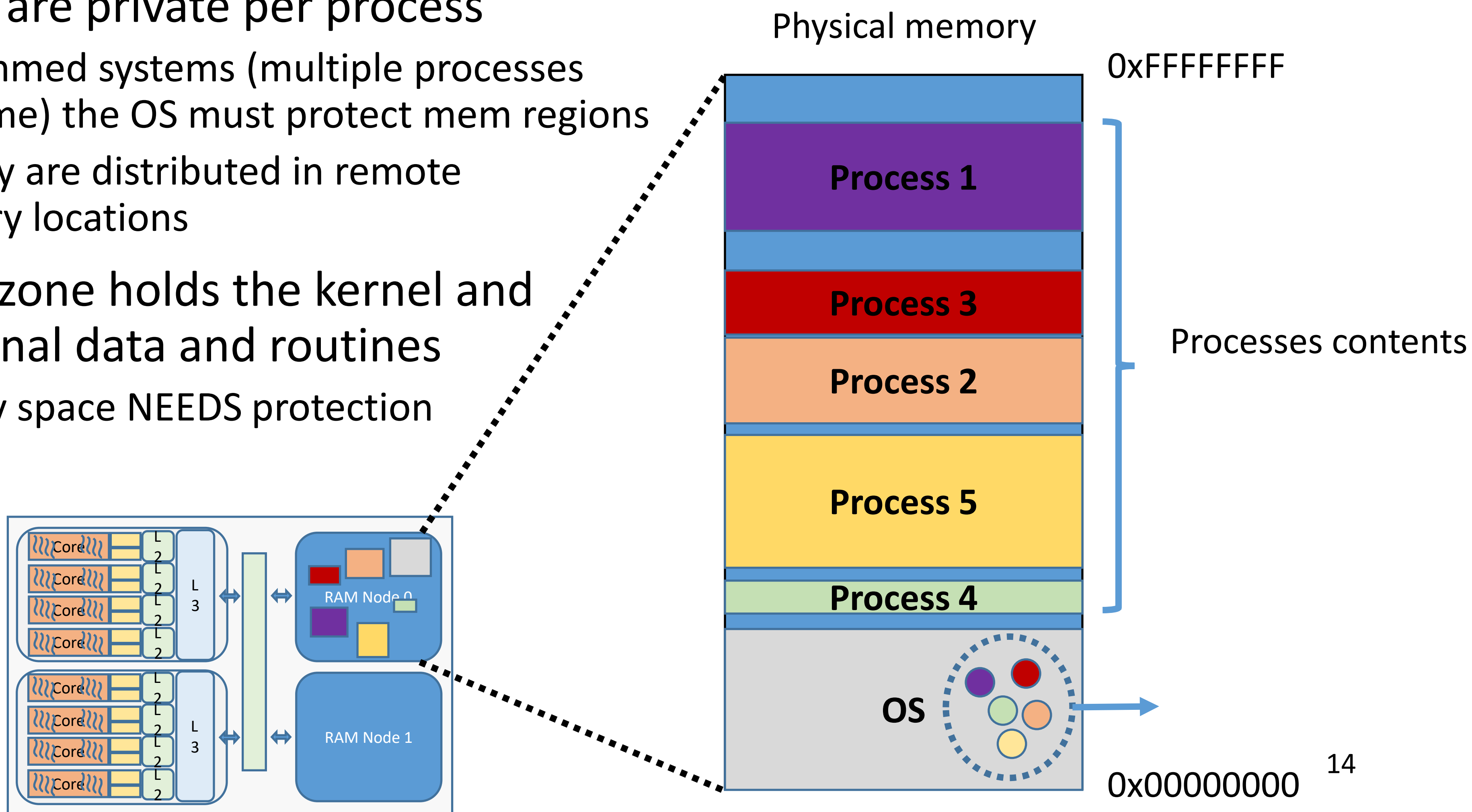


Process with shared libraries (Dynamic linking)



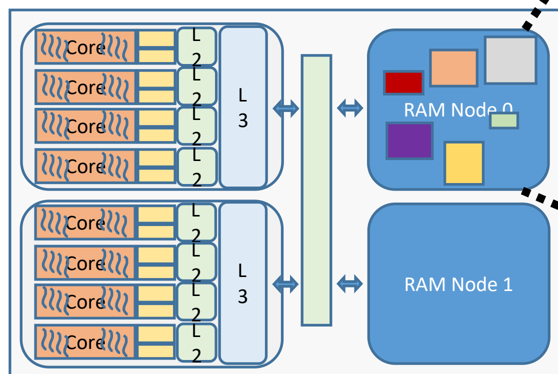
OS Support

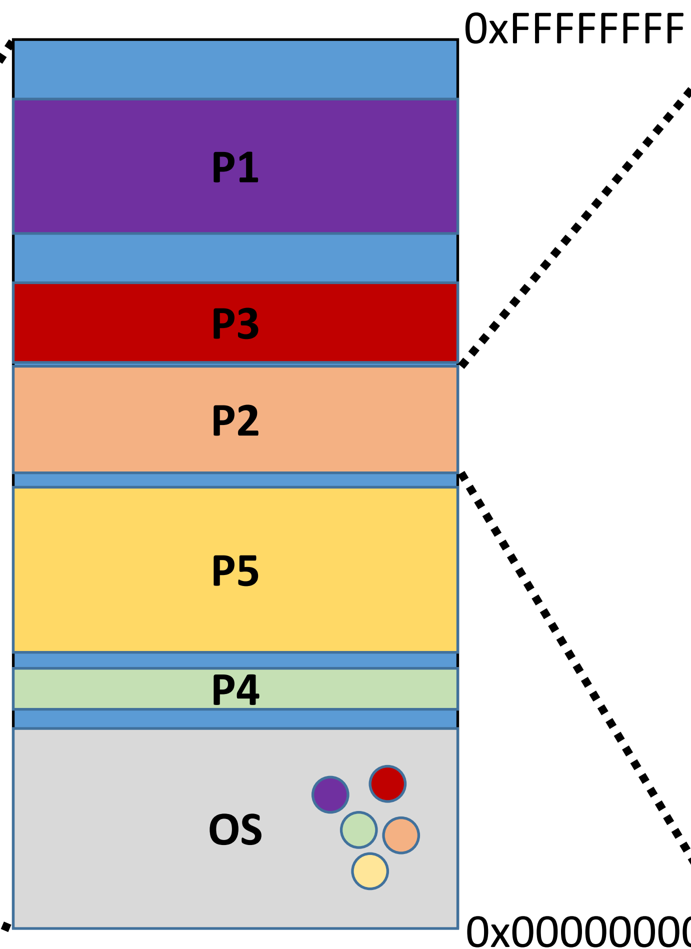
- Memory regions are private per process
  - In multiprogrammed systems (multiple processes are alive at a time) the OS must protect mem regions
  - Contents usually are distributed in remote physical memory locations
- The OS memory zone holds the kernel and all required internal data and routines
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

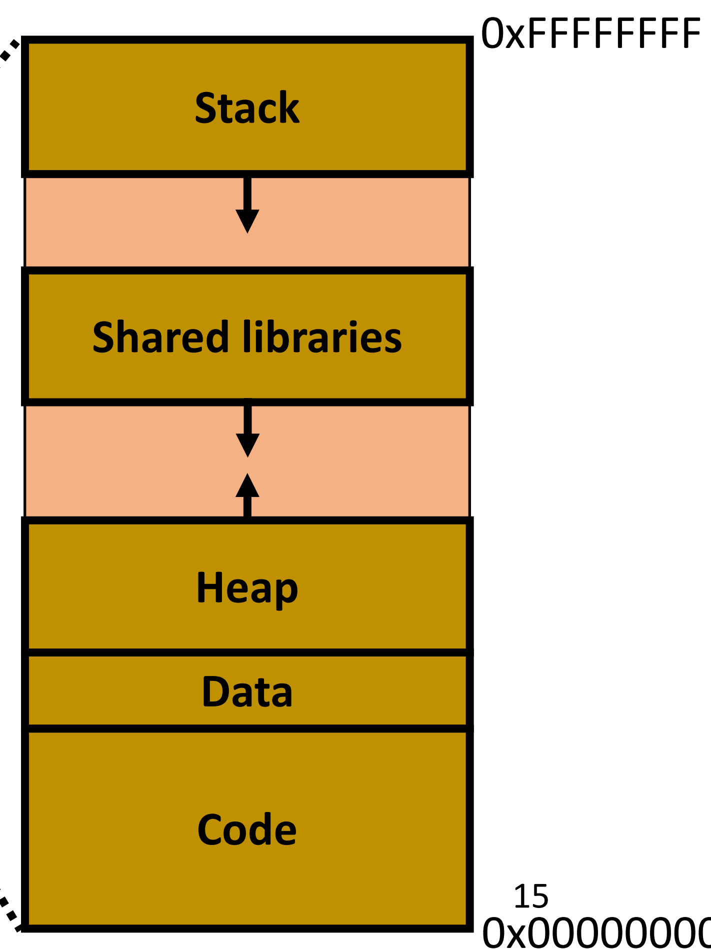


Process Contents in Memory

- **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







Syscalls related to Memory Management

- The Heap is mainly employed for dynamic data structures
  - E.g. Structures used only for a period of time, unknown memory size requirements, allocated and deallocated often

- Syscalls related to heap management
  - Memory allocation
    - malloc (C library)
    - new (C++ library)
  - Memory deallocation
    - free (C library)
    - delete (C++ library)
  - All above calls invoke a system call to modify the limit of the Heap Mem Zone (brk)

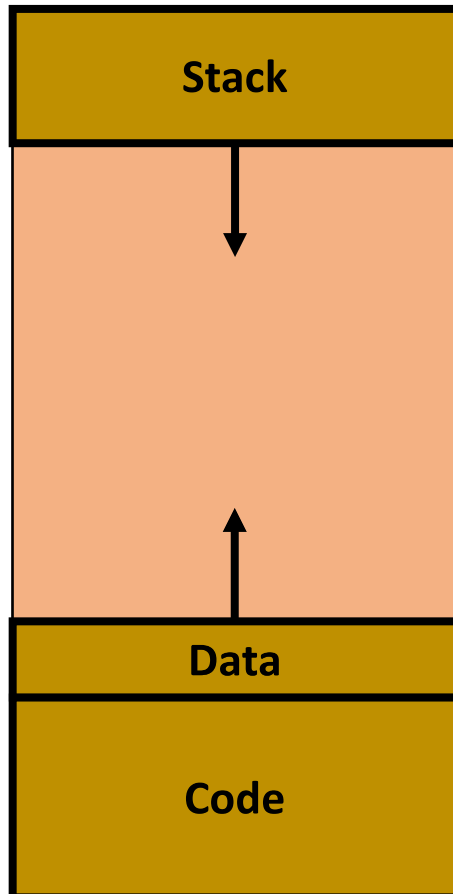
Memory Allocation

- (type) **malloc**(int size); // C language
  - Returns the starting @ of the newly allocated size Bytes in the HEAP
- **new** *type*[size];   // C++ language
  - Returns the starting @ of the sizeof(type)*size Bytes in the HEAP

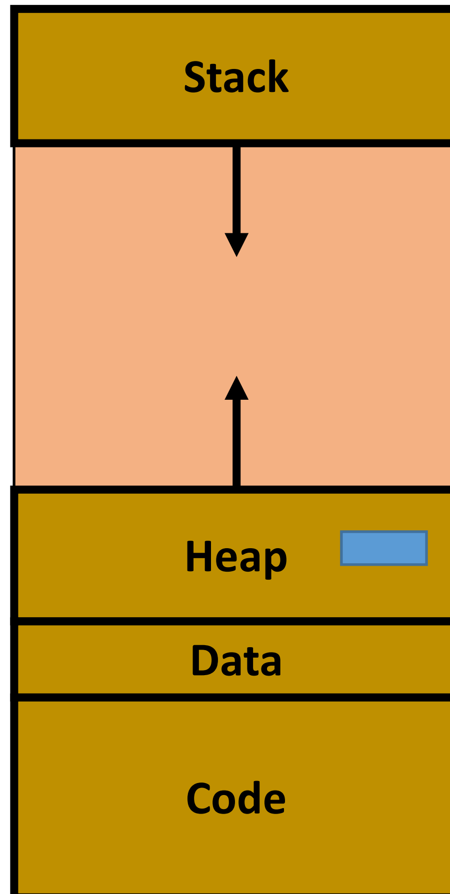
- Behavior
  - Before allocating @, it checks whether the HEAP has space enough to hold size bytes
    - If not, the OS increases the limit of the HEAP (sbrk)
    - The HEAP size is increased by a configurable number of bytes to reduce the number of times it has to be increased
  - The object memory zone is allocated in the HEAP following a placement algorithm
    - Can group objects by their size…

Memory Allocation

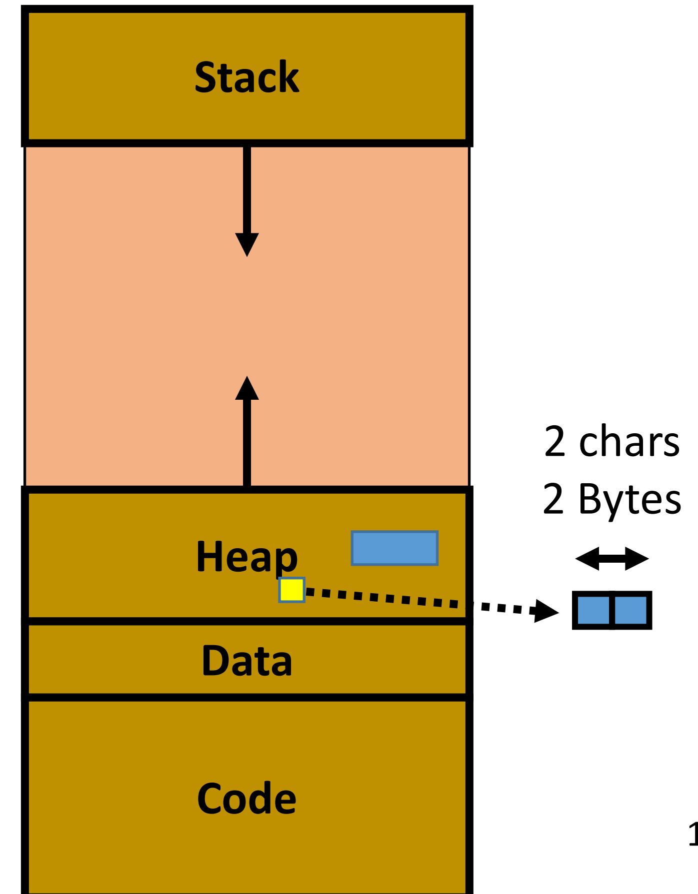
Starting status



After calling:
int *ptr = (int) malloc(4*sizeof(int));



After calling:
char *string = (char) malloc(2*sizeof(char));

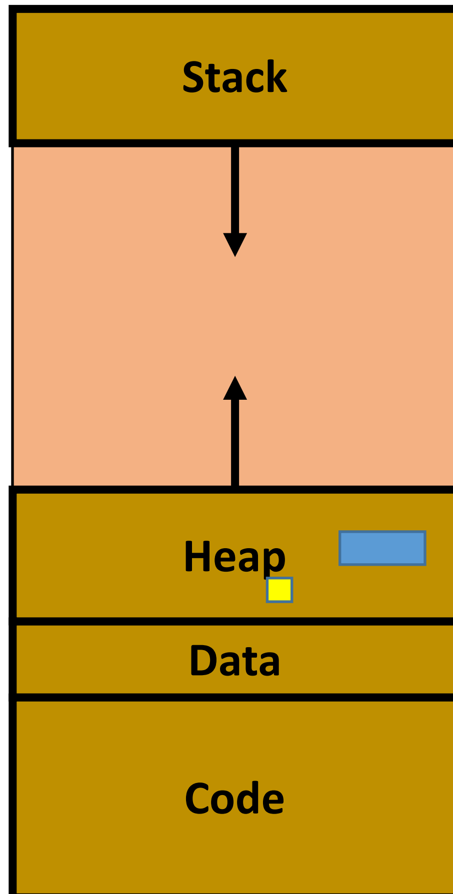


Memory Deallocation

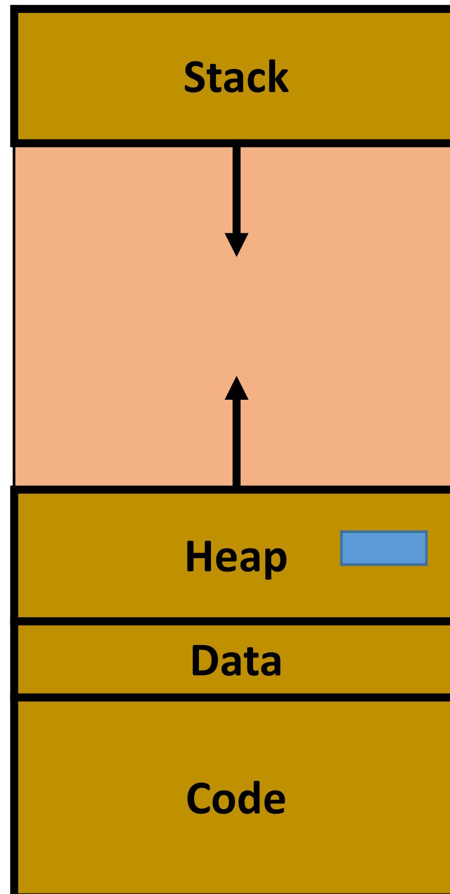
- **free**(pointer); // C language
- **delete** [] pointer; // C++ language
  - In both cases the memory zone pointed by the pointer is released
- Behavior
  - Deallocation of the memory zone pointed by the input parameter
  - The OS will consider to reduce or not the HEAP memory space
  - A pair of lists are maintained by malloc/new/free/delete
    - List of objects allocated
    - List of objects deallocated (may be merged)

Memory Deallocation

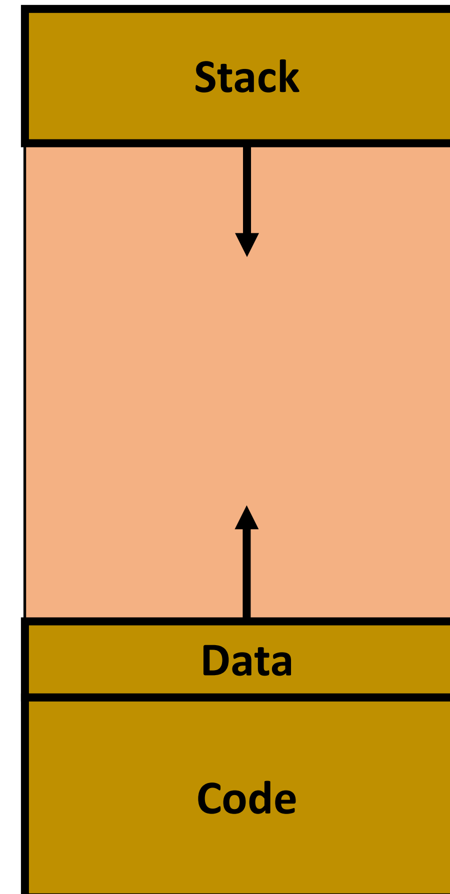
Starting status



After calling:
free(string);

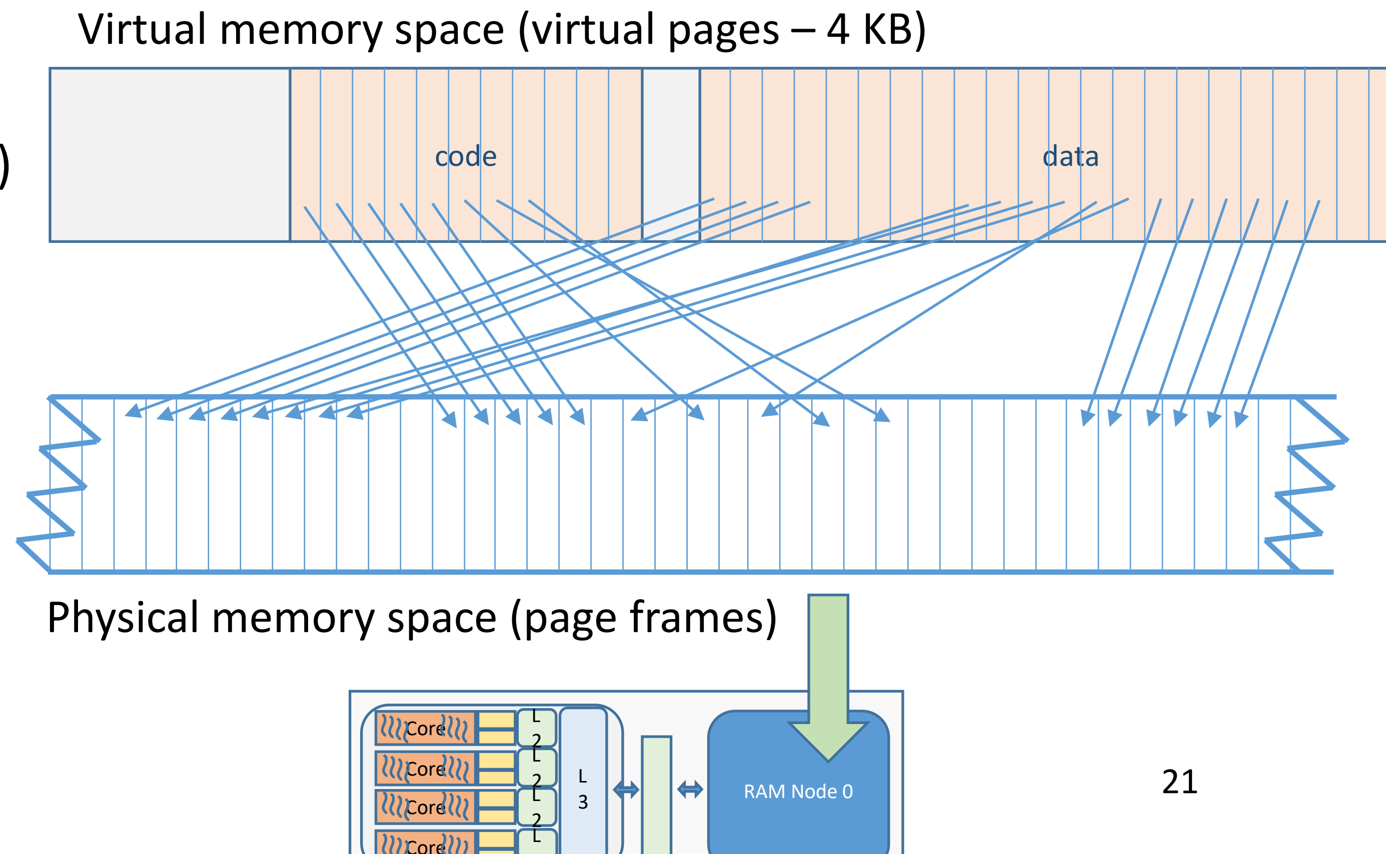


After calling:
free(ptr);



Final relation with physical memory

- Virtual memory
  - Split in pages (4KB usually, can be 2-4 MBytes)
  - A page can be
    - Valid and present (in memory)
    - Valid and not present (in memory)
    - Invalid
- Physical memory
  - Split in page frames
    - Same capacity as virtual pages
  - OS keeps information about
    - Available frames
    - Busy frames



When Memory is exhausted…

- OS constantly checks memory availability (used vs free) of the system
- If the memory becomes exhausted (less than a given availability threshold) OS starts an alternative suport approach:
  - Difference between page replacement (paging) and process replacement (swapping)
  - Pages are selected->Written to the **swap area** [if modified]->Reallocate other virtual pages
- Swap area: special memory area, usually in a storage device (e.g. disk) to temporary hold contents of the main memory that have not space enough
  - It is considered as additional space to the physical memory

Bibliography

- Computer Systems – A Programmer’s perspective (3rd Edition)
  - Randal E. Bryant, David R. O’Hallaron, Person Education Limited, 2016
  - https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004062589706711
    - Chapter 9

- Computer Organization and Design (5th Edition)
  - D. Patterson, J. Hennessy, and P. Alexander
  - http://cataleg.upc.edu/record=b1431482~S1*cat
    - Several Chapters