Operating Systems

Computadors

Grau en Ciència i Enginyeria de Dades

Xavier Verdú, Xavier Martorell

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

2019-2020 Q2

Creative Commons License

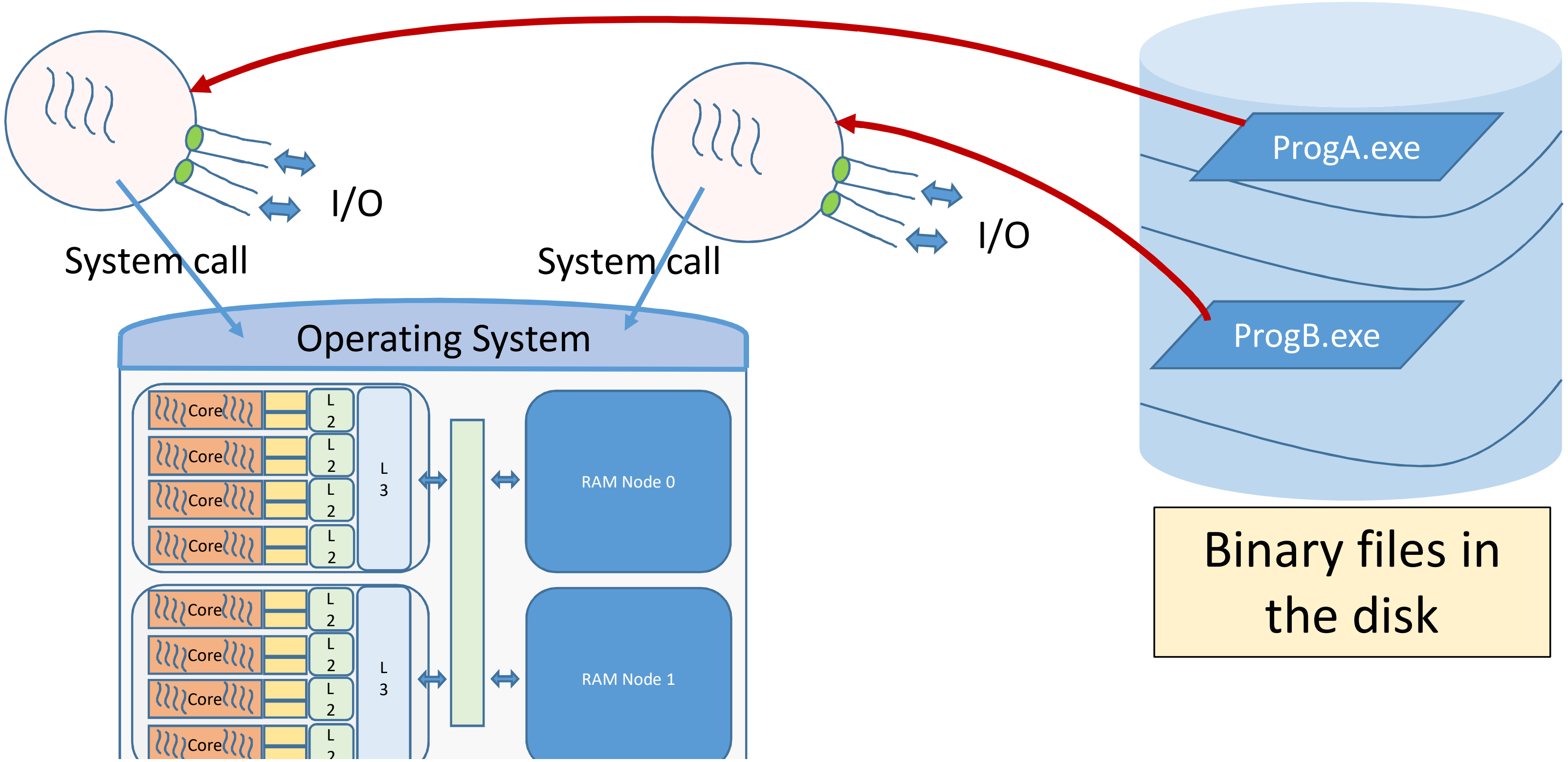
Table of Contents

- Operating System
- Basic Concepts
- Access to Kernel functions
- Process management
- Memory subsystem
- Other important OS concepts and tasks

- NOTE: I/O subsystem and filesystems will be studied in future lessons

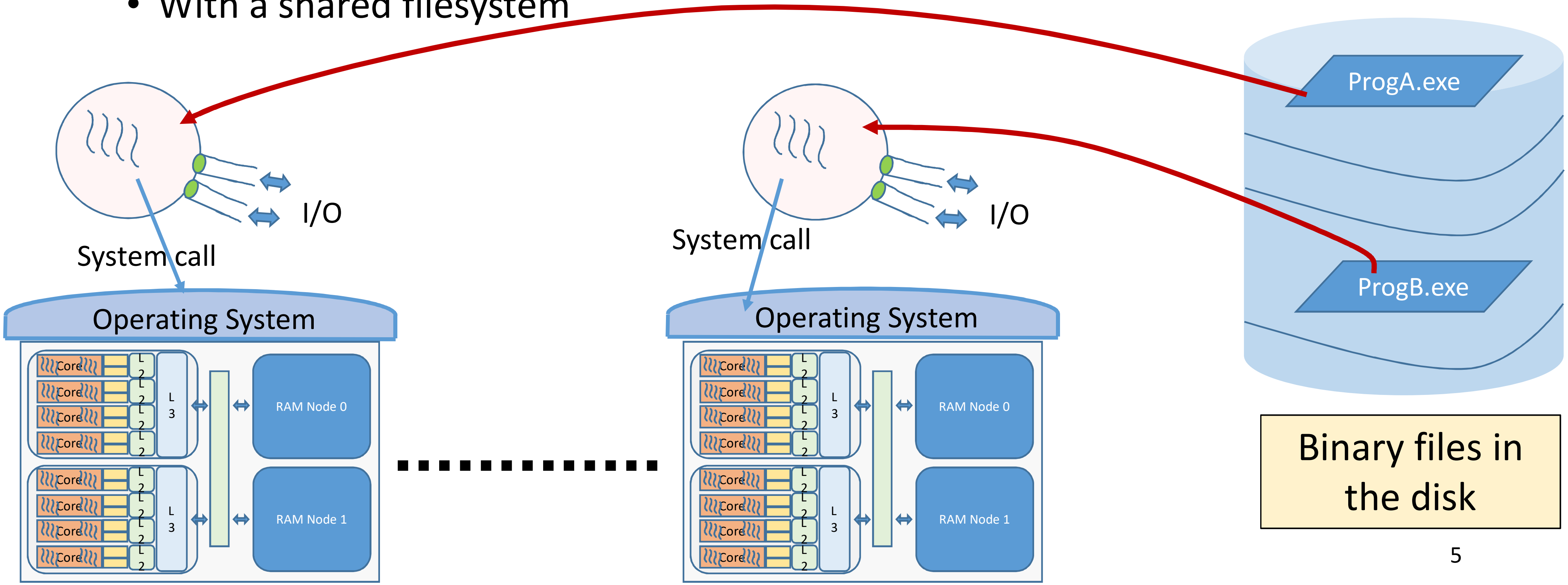
What is an Operating System?

- The OS is a software that manages hardware and software resources



Management of a distributed environment

- Multiple nodes have usually one copy of the OS each
  - With a shared filesystem



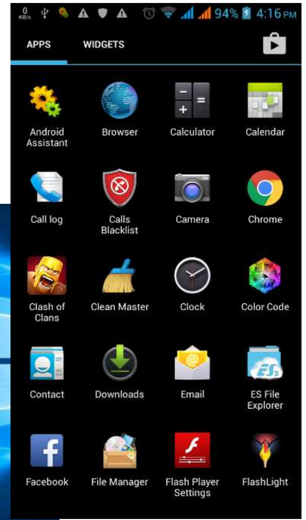
Design Principles

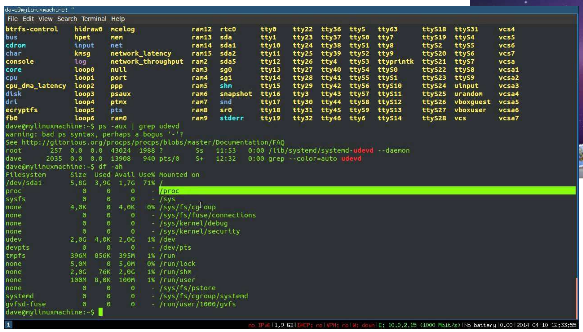
- The OS also plays a key role for user interaction

- It offers a **usable** environment
  - Abstracts the user from the different kind of "systems" and hardware
- It offers a **safe/robust/protected** execution environment
  - Safe from the point of view of accessing HW correctly and protected from the point of view of user's interaction
- It offers an **efficient** execution environment
  - Fine grain management of the HW
  - Allows many users/programs sharing resources, ensuring a good resource utilization

Dealing with the system

- Using a graphical environment / a command shell in text mode
  - Allow a high-level view of the system services and resources
    - Services
    - Storage
    - Networking
  - On top of the kernel system calls

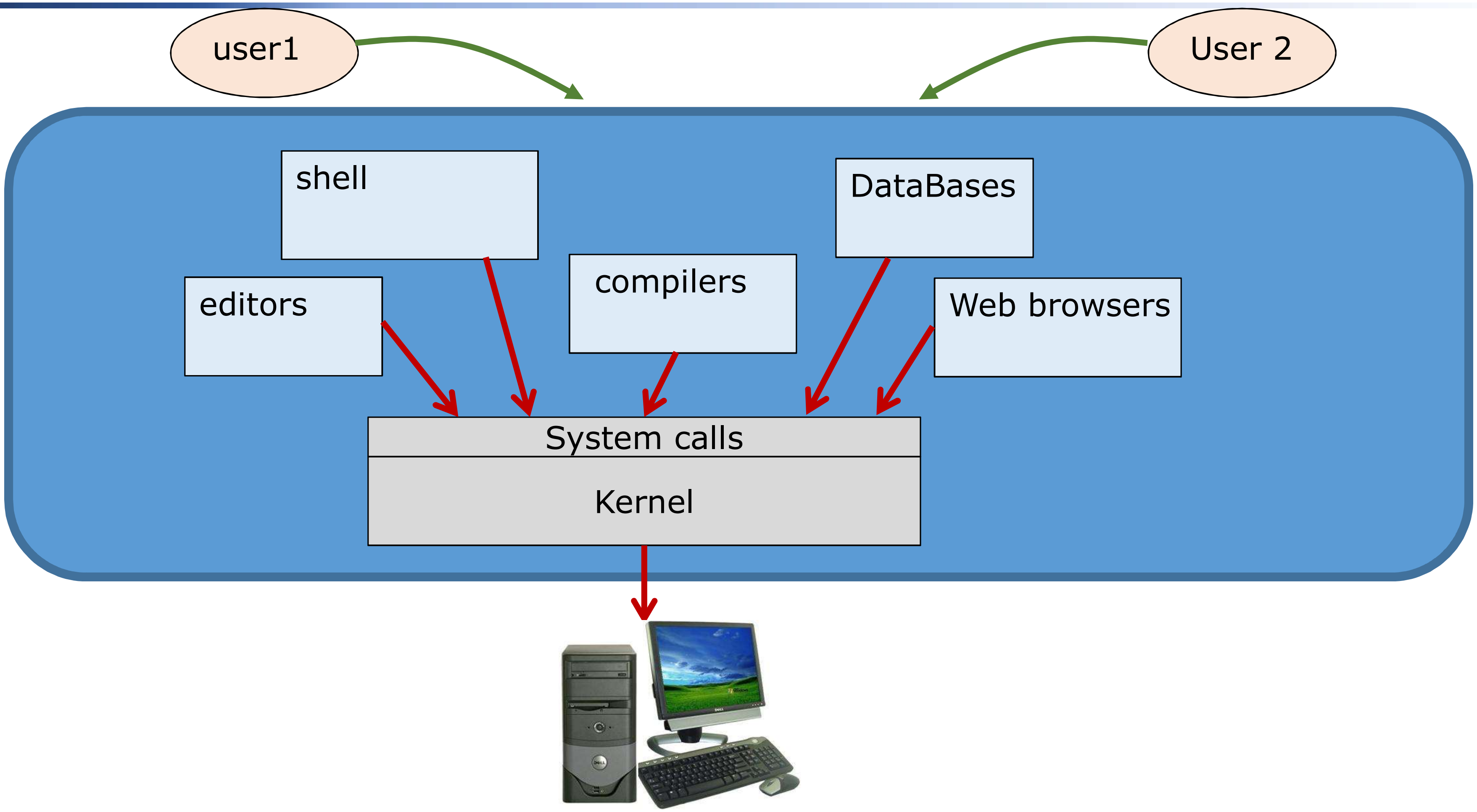




System Management

- Intermediary between applications and hardware
  - **Kernel internals:** define data structures to manage HW and algorithms to decide how to use it
  - **Kernel API:** offers a set of functions to ask for **system services**
    - **System calls**

The System: Programs + Kernel



Overall main steps of the OS

Boot

- Executed when the system is switched on, the kernel code is loaded in memory
- Interrupts and basic HW configurations are initialized
- It starts the system access mechanism: daemons, login, shell, etc

Usage

- User sessions / execution of applications, services
- Support development of new applications…

Shutdown

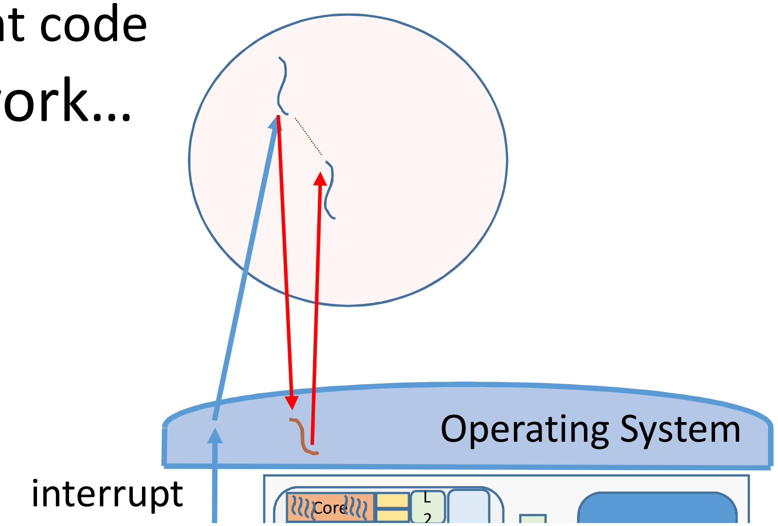
- Executed when system is switched off
- Saves persistent information, stop services and devices, etc

Access to Kernel functions

- Execution modes
  - Hardware support to guarantee security
    - The CPU must be able to differentiate when it is executing instructions coming from normal (non-privileged) user code or instructions coming from the kernel code
  - Two or more levels of privilege
    - User vs. kernel modes
    - Privileged levels: 0 (most-privileged) – 1 – 2 – 3 (user-level)

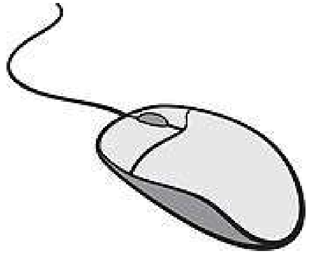
When is the kernel code executed? (I of III)

- When an **interrupt** occurs:
  - Interrupts are generated by HW devices
    - Interrupt: asynchronous and involuntary notification
  - Clock interrupts: there are several clocks in a system
    - **OS Tick**: configured during boot period by the OS (e.g. 10ms) to execute management code
  - Storage/network…



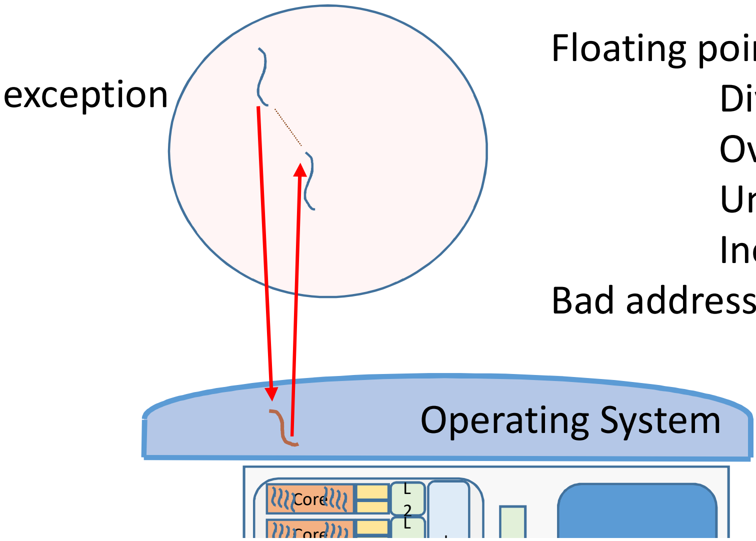
Example of Interrupts

IRQ	Usage
0	system timer (cannot be changed)
1	keyboard controller (cannot be changed)
2	cascaded signals from IRQs 8–15
3	second RS-232 serial port (COM2: in Windows)
4	first RS-232 serial port (COM1: in Windows)
5	parallel port 2 and 3 or sound card
6	floppy disk controller
7	first parallel port
8	real-time clock
9	open interrupt
10	open interrupt
11	open interrupt
12	PS/2 mouse
13	math coprocessor
14	primary ATA channel
15	secondary ATA channel



When is the kernel code executed? (II of III)

- When an **exception** occurs:
  - exceptions are generated by the CPU when some problem occurs during the execution of one instruction
    - Exception: synchronous, but involuntary notification



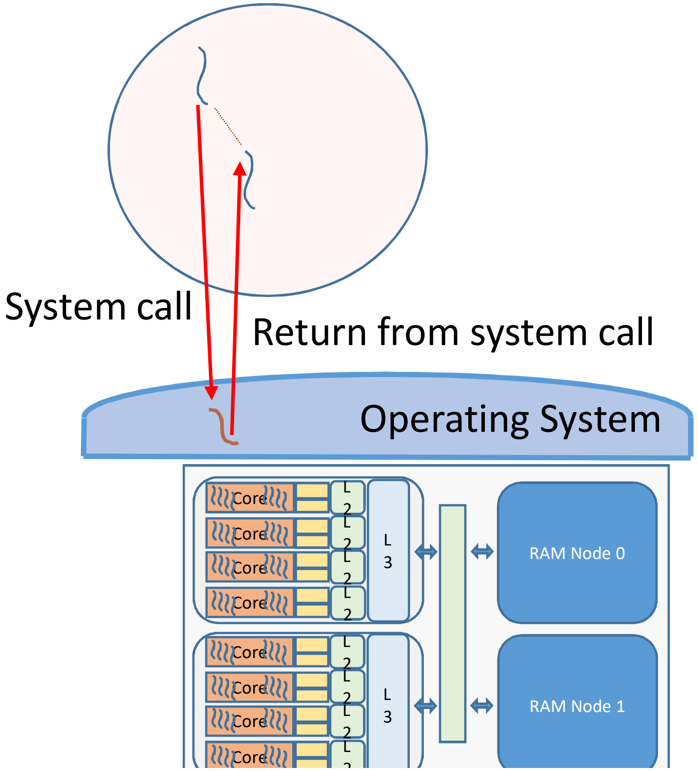
Floating point
- Division by 0
- Overflow
- Underflow
- Inexact

Bad address (segmentation fault/bus error)

Page fault
Breakpoint
Invalid instruction
Alignment check
...

When the kernel code is executed? (III of III)

- When a program requests a service from the OS through a **system call**



open, close, read, write, ioctl, fnctl, stat, fstat...
mount, umount...
fork, exec, exit, clone...
socket, bind, listen, accept, connect...
...

OpenFile, ReadFile, WriteFile...
CreateProcess, CreateProcessEx...
...

... up to tens of syscalls

System Calls

- When a program requests a service from the OS through a **system call**
- A system call has stronger requirements than a “simple” function call
  - Requirements
    - The kernel code MUST be executed in privileged mode
    - For security, the “jump” implicit in the call instruction and the execution mode change must be done with a single instruction
    - The memory address of a system call could change from one kernel version to another, and it must be compatible → we need an instruction different from a “call”
  - To take into account
    - Changes in the execution mode imply that some HW resources are not shared
      - for instance, the stack

System Calls

- The implementation depends on the architecture and the OS itself

	Function call	System call
Argument passing	stack / registers	**stack / registers**
Function invocation	call	**syscall (depend)**
At function start	save registers (sw)	save registers (sw)
Accessing arguments	stack	**stack / registers**
Before return	restore registers (lw)	restore registers (lw)
Return values	registers	**stack / registers**
Return function	ret	**sysexit (depend)**

Linux i386 (32bits) vs Linux x64 (64bits)

Linux i386	Linux x64

Linux i386

```
...
movl  $4, %eax      ; use the write syscall
movl  $1, %ebx      ; write to stdout
movl  $msg, %ecx ; use string "Hello World!\n"
movl  $14, %edx   ; write 14 characters
int     $0x80         ; make syscall
...
```

Linux x64

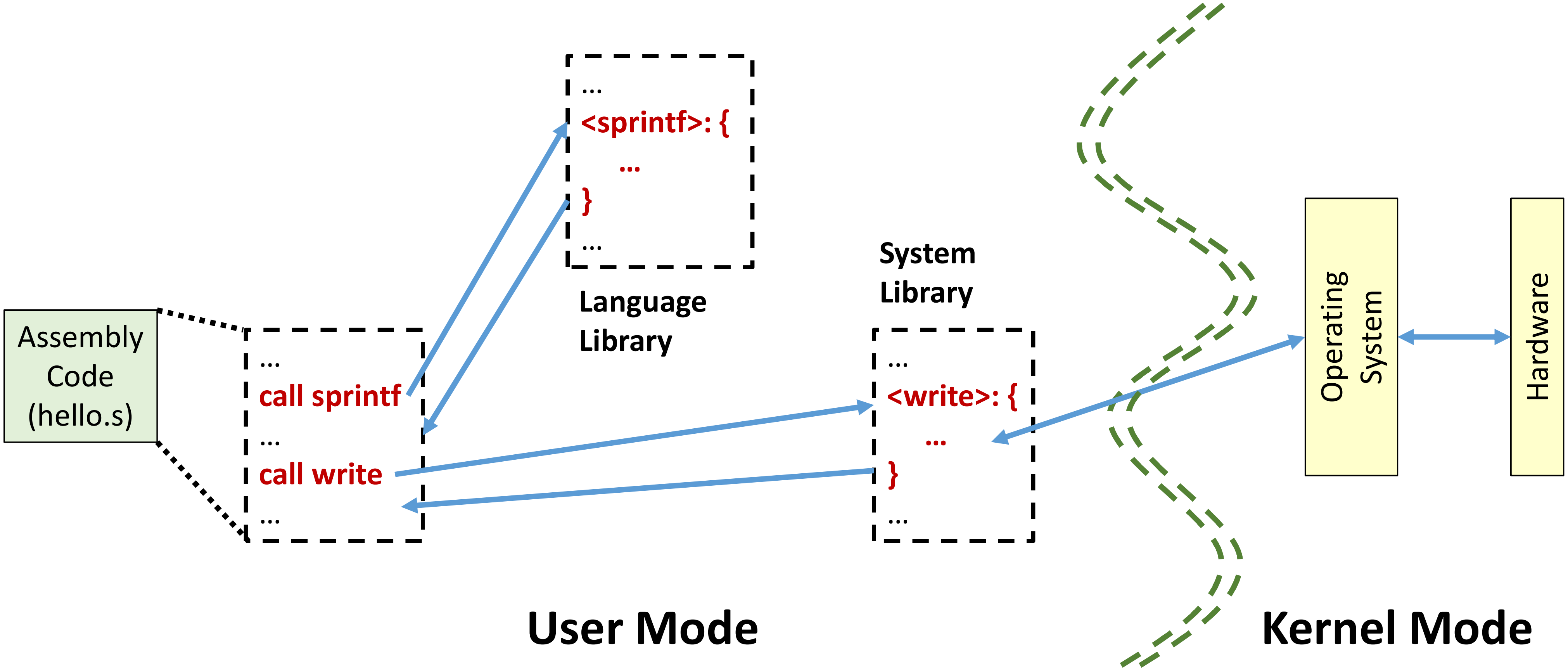
```
...
movq $1, %rax     ; use the write syscall
movq $1, %rdi      ; write to stdout
movq $msg, %rsi ; use string "Hello World!\n“
movq $14, %rdx   ; write 14 characters
syscall               ; make syscall
...
```

System Library

- To hide all these details to the user, the system provides a library to be linked with user codes
    - This is automatically done by the compile (gcc / g++ for instance)

- It is called the system library, and translates from the high level system call API for the specific language (C, C++, etc.) to the assembler code where all the requirements are taken into account
    - For C and C++, system calls are included in the C support library (libc)

Non-privileged vs privileged execution mode



Summary: Step-by-Step kernel code invocation

- Save user context

→**Change from user mode to kernel mode**

  - Restore system context
  - Retrieve user parameters
  - Identify service
  - Execute service
  - Return result

←**Change from kernel mode to user mode**

- Restore user context

This procedure involves an **overhead**

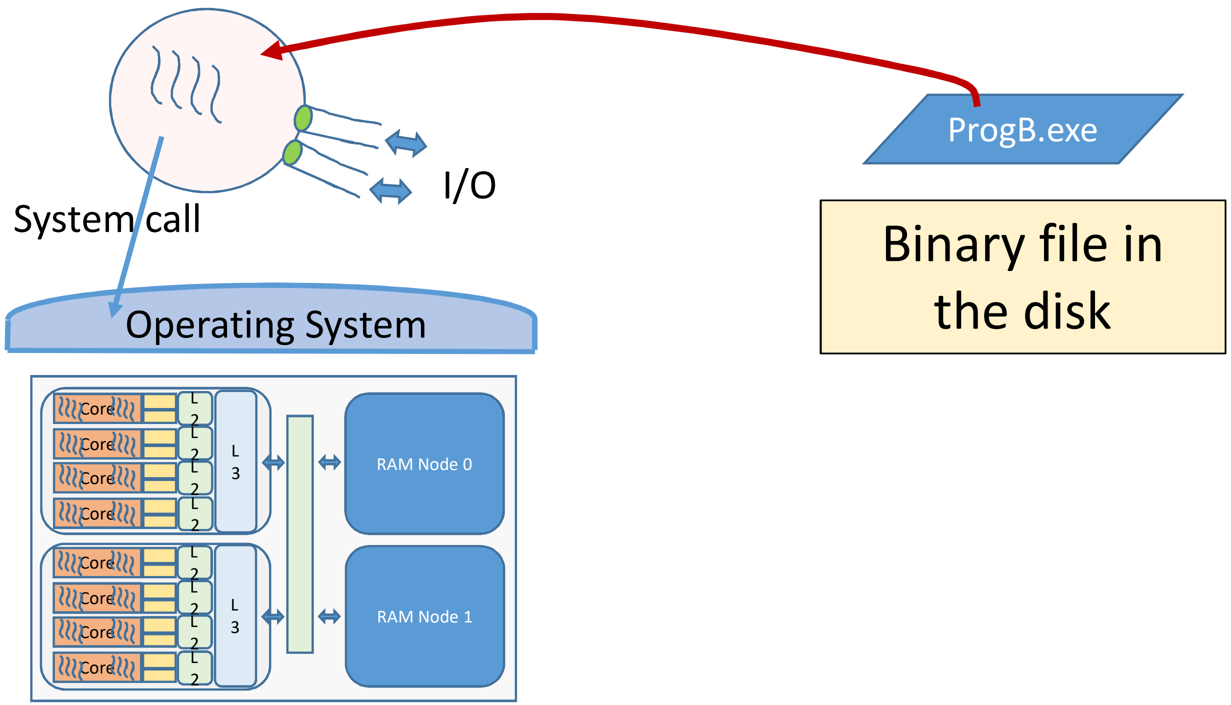
Table of Contents

- Operating System
- Basic Concepts
- Access to Kernel functions
- Process management
- Memory subsystem
- Other important OS concepts and tasks

- NOTE: I/O subsystem and filesystems will be studied in future lessons

Program vs Process

- A process is the OS representation of a program during its execution



Program vs Process

- A process is the OS representation of a program during its execution
- The user **program** is static: it is just a sequence of bytes stored in a "disk"
- The user **process** is dynamic, and it consists of…
  - What regions of physical memory is using
  - What files is accessing
  - Which user is executing it (owner, group)
  - What time it was started
  - How much CPU time it has consumed
  - …

Processes

- Assuming a general purpose system, multi-user
  - each time a user starts a program, a new (unique) process is created
  - The kernel assigns resources to it: physical memory, some slot of CPU time and allows file accesses

- In a general purpose system, we have a multiprogrammed environment
  - **Multiprogrammed System**: a system with multiple programs running at a time
- **Process creation**
  - The kernel reserves and initializes a new process data structure with dynamic information (the number of total processes is limited)
    - Each OS uses a name for that data structure, in general, we will refer to it as **PCB (Process Control Block)**
    - Each new process has a unique identifier (in Linux it is a number). It is called **PID (Process Identifier)**

Process Control Block (PCB)

- The PCB holds the information the system needs to manage a process
- The information stored on the PCB depends on the operating system and on the HW
  - Address space
    - Description of the memory regions of the process: code, data, stack,…
  - Execution context
    - SW: PID, scheduling information, information about the devices, accounting,…
    - HW: page table, program counter, …

Process Control Block (PCB)

- Typical attributes are:
  - The process identifier (PID) and the parent process identifier (PPID)
  - Credentials: user and group
  - Environment variables, input arguments
  - CPU context (to save cpu registers when entering the kernel)
  - Process state: running, ready to run, blocked, stopped…
  - Data for I/O management
  - Data for memory management
  - Scheduling information
  - Resource accounting
  - …
- **We will deal with some of these attributes in the Lab**

Multi-Process environment

- Usually there are many processes alive at a given time in a common OS
- Processes usually alternate using the CPU with other resource usage
  - In a multi-programmed environment the OS manages how resources are shared among processes
- In a general purpose system, the kernel alternates processes in the CPU
  - We have to alternate processes without losing the execution state
    - We will need a place to save/restore the processes execution state
    - We will need a mechanism to change from one process to another
  - We have to alternate processes being as much fair as possible
    - We will need a scheduling policy
- If the kernel makes this CPU sharing efficiently, users will have the feeling that a CPU is constantly assigned to the process

Parallelism vs Concurrency

- Parallelism: N processes run at a given time in N CPUs

Time(each CPU executes one process)

CPU0	Proc. 0
CPU1	Proc. 1
CPU2	Proc 2

- Concurrency: N processes could be potentially executed in parallel, but there are not enough resources to do it
  - The OS selects what process can run and what process has to wait

Time (CPU is shared among processes)

CPU0	**Proc. 0**	**Proc. 1**	**Proc. 2**	**Proc. 0**	**Proc. 1**	**Proc. 2**	**Proc. 0**	**Proc. 1**	**Proc. 2**	**...**

Execution Flows (Threads)

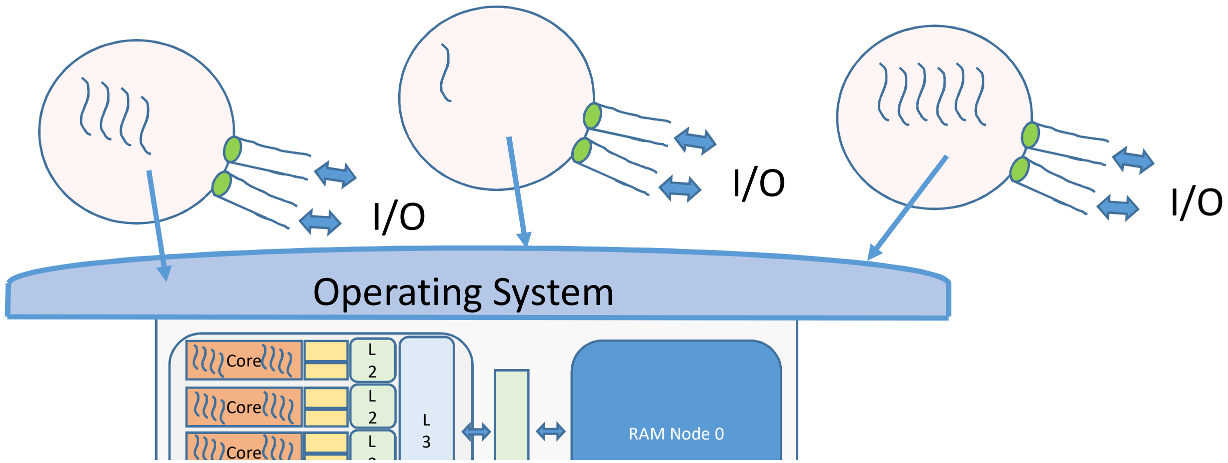
- Analyzing the concept of Process…
  - the OS representation of a program during its execution

…we can state a **Process** is the resource allocation entity of a executing program (memory, I/O devices, threads)

- Among other resources, we can find the **execution flow/s (thread/s)** of a process
  - The execution flow is the basic scheduling entity the OS manages (CPU time allocation)
    - Every piece of code that can be independently executed can be bound to a thread
  - Threads have the required context to execute instruction flows
    - Identifier (Thread ID: TID)
    - Stack Pointer
    - Pointer to the next instruction to be executed (Program Counter),
    - Registers (Register File)
    - Errno variable
  - Threads **share** resources of the same process (PCB, memory, I/O devices)

Multi-threaded processes

- A process has a single thread when it is launched
- A process can create a number of additional threads
  - E.g.: current high-performance videogames comprise >50 threads; Firefox/Chrome show >80 threads
- The management of multi-threaded processes depends on the OS support
  - User Level Threads vs Kernel Level Threads



Execution Flows (Threads)

- When and what are threads used for…
  - Parallelism exploitation (code and hardware resources)
  - Task encapsulation (modular programming)
  - I/O efficiency (specific threads for I/O)
  - Service request pipelining (keep required QoS)

- Pros
  - Threads management (among threads of the same process) has less cost than process management
  - Threads can exchange data without syscalls, since they share memory

- Cons
  - Hard to code and debug due to shared memory

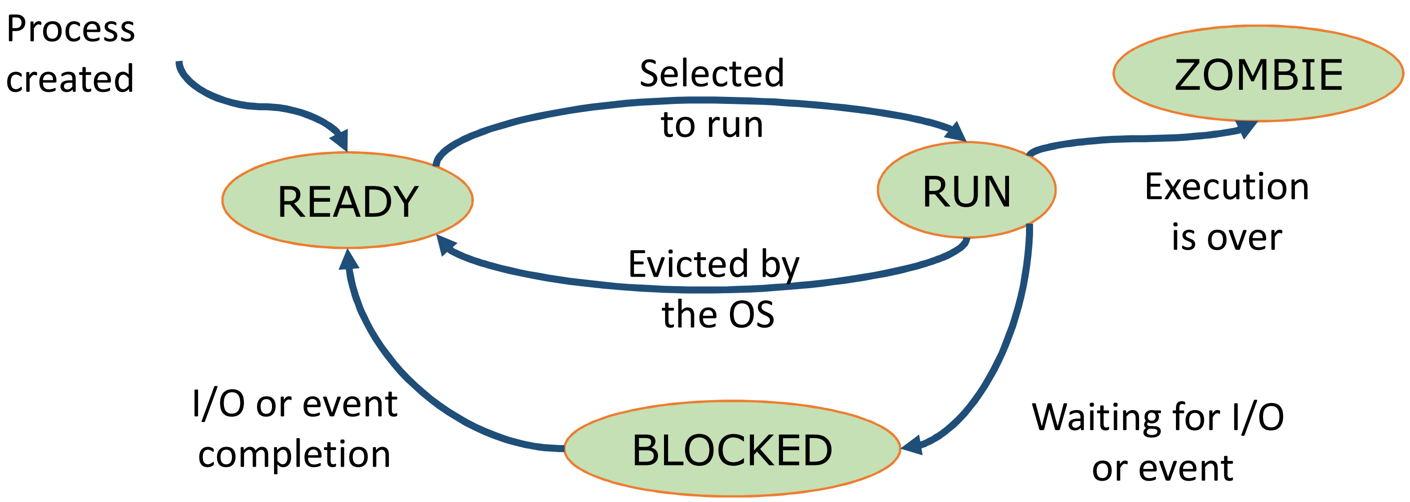
Process State

- The PCB holds the information required to exactly know the current status of the process execution

- Processes do not always use the CPU
  - E.g.: Waiting for data coming from a slow device, waiting for an event…

- The OS classifies processes based on what their are doing, this is called the **process state**
  - It is internally managed like a PCB attribute or grouping processes in different lists (or queues)

Process State Graph



- This is a generic process state graph approach mostly used by kernels, but…
  - …every kernel defines its own process state graph with slight modifications

Kernel Internals for Process Management

- Data structures to keep per-process information and resource allocation
- Data structures to manage PCB's, usually based on their state
  - In a general purpose system, such as Linux, Data structures are typically queues, multi-level queues, lists, hast tables, etc.
- Scheduling algorithms to select the next process to run in the CPU

Schedulers

- Schedulers are critical for the proper performance of the system

- **Short** term: every OS Tick
  - What is the next process to run in the CPU

- **Medium** term: when the OS detects it is running out of resources (E.g. Memory)
  - What processes are candidate to **temporally** release resources to let other processes use them

- **Long** term (*optional*): every start/end of a process
  - What is the maximum number of processes suitable to run in the system
    - It controls the multiprogrammed level of the system

Short Term Scheduler

- Every OS tick the scheduler checks whether another process has to run in the CPU

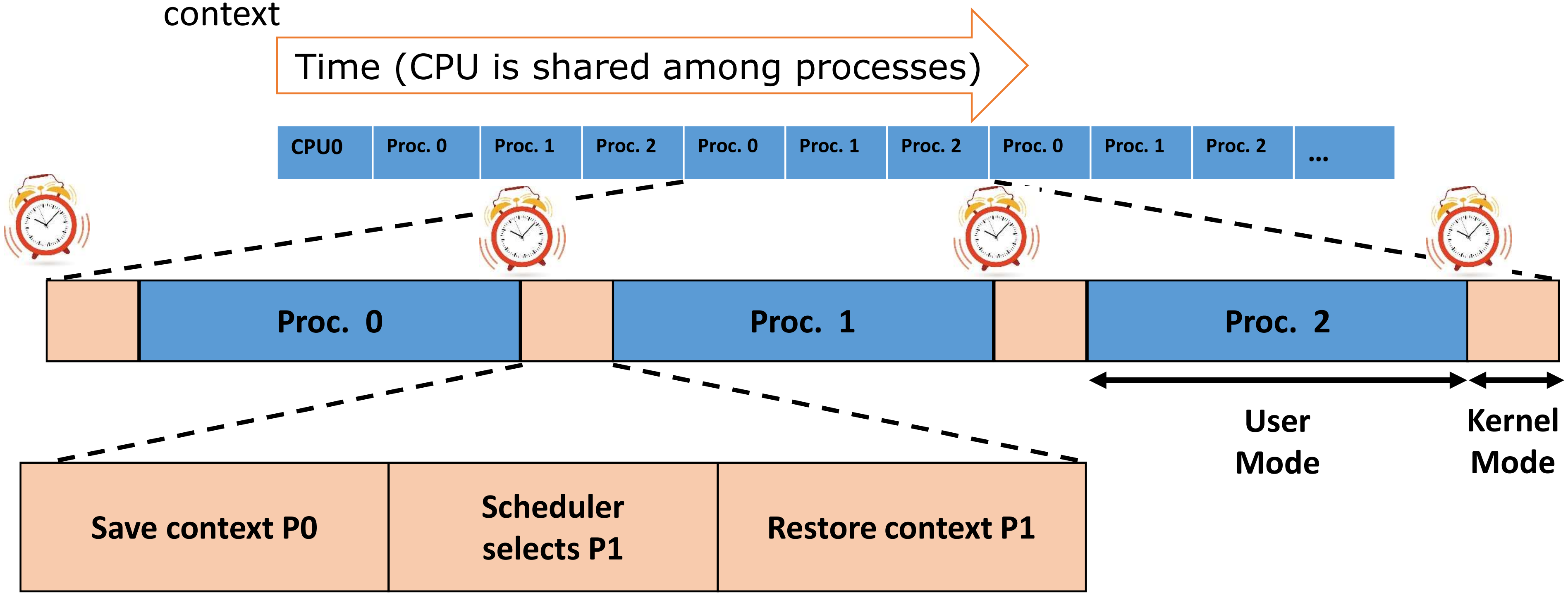
- **Non-Preemptive Policies**: the scheduler cannot put a process out of the running state
  - Only the process itself can decide to release the CPU (e.g. blocking I/O call)

- **Preemptive Policies**: the scheduler can put a process out of the running state in order to enable another process run instructions in the CPU
  - **Quantum**: period of time the scheduler grants a process to run in a row in the CPU
  - Priority/non-priority based policies
    - E.g. Round-Robin

- Schedulers of current general purpose OSes are based on complex approaches
  - Multiple policies using multiple queues

Impact of context switch on performance

- Context Switch: changing the process that is running in the CPU
  - It involves an overhead due to kernel code execution and manage the save/restore of a process context



Performance/Efficiency of a Scheduling Policy

- What is the main goal of the system?
  - Real-time systems versus High Performance Computing
    - It is not the same the device that manage the ABS of a car than a node in a Supercomputer

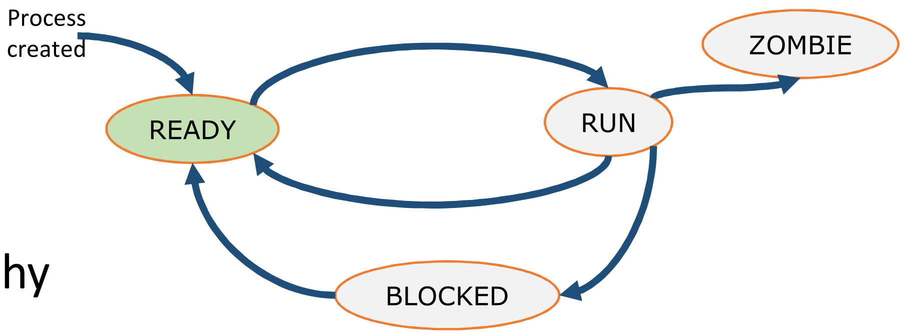
- The definition of “optimal” scheduling policy depends on the purpose of the system
  - Different metrics to find out whether a scheduling policy is well chosen
    - E.g. Response time, throughput, efficiency, turnaround time

Syscalls related to Process Management

- Process creation
  - A process creates a new child process
- End of process execution
  - A process notifies to the kernel that it finishes its execution
- Wait for a child process to finish
  - And allow the system to release its data structures (PCB, kernel stack…)
- Get process identifiers
  - Get the Process ID ("**getpid();**") and the Parent Process ID ("**getppid();**")
- Execute a new program
  - The process changes the program that is executing

Process Creation

```
int  fork();
```



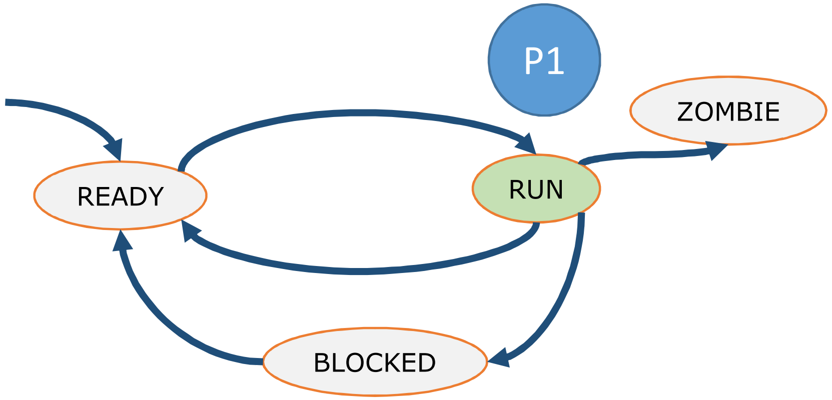
- The current process creates a child process
  - It is the base of the whole system process hierarchy
  - The child process is a clone of its parent
    - Most of the content is inherited
      - Such as memory regions, I/O devices, register file values
    - Some characteristics are not inherited
      - Such as PID, PPID (Parent's PID), stats (use of CPU…)

- Both processes keep executing from the very next instruction
- But both receive different return values
  - The parent receives the PID of the child process
  - The child receives 0

Example: Fork

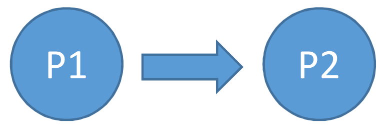
```
main () {
   int ret, count = 0;
   count = 1;
   ret = fork();
   if (ret == 0){
      count = 2;
      printf("Child with counter = %d\n", count);
      ...
   } else {
      count = 3;
      printf("Parent with counter = %d\n", count);
   }
   ...
}
```

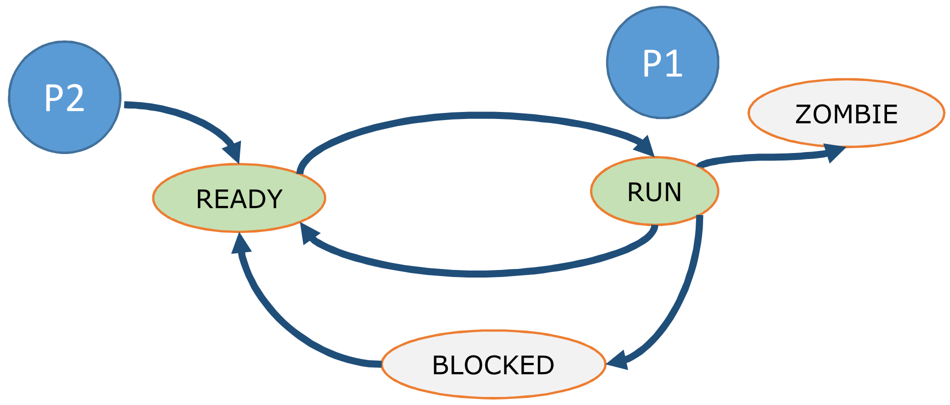
P1



Example: Fork

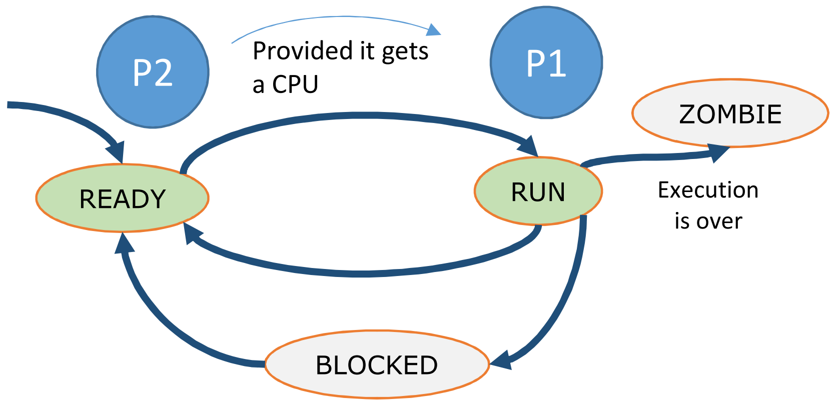
```
main () {
    int ret, count = 0;
    count = 1;
    ret = fork();
    if (ret == 0){
        count = 2;
        printf("Child with counter = %d\n", count);
        …
    } else {
        count = 3;
        printf("Parent with counter = %d\n", count);
    }
    …
}
```





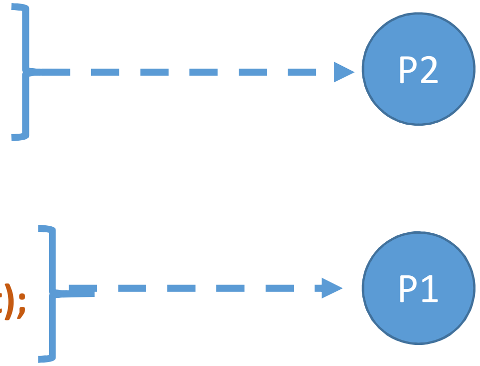
Example: Fork

```
main () {
    int ret, count = 0;
    count = 1;
    ret = fork();
    if (ret == 0){
        count = 2;
        printf("Child with counter = %d\n", count);
        ...
    } else {
        count = 3;
        printf("Parent with counter = %d\n", count);
    }
    ...
}
```



Example: Fork

```
main () {
    int ret, count = 0;
    count = 1;
    ret = fork();
    if (ret == 0){
        count = 2;
        printf("Child with counter = %d\n", count);
        …
    } else {
        count = 3;
        printf("Parent with counter = %d\n", count);
    }
    …
}
```



Output

Child with count = 2

Parent with counter = 3

Notes:
- Memory regions are not shared between the processes
- Concurrent/parallel executions are possible

Concurrent vs Sequential Process Creation

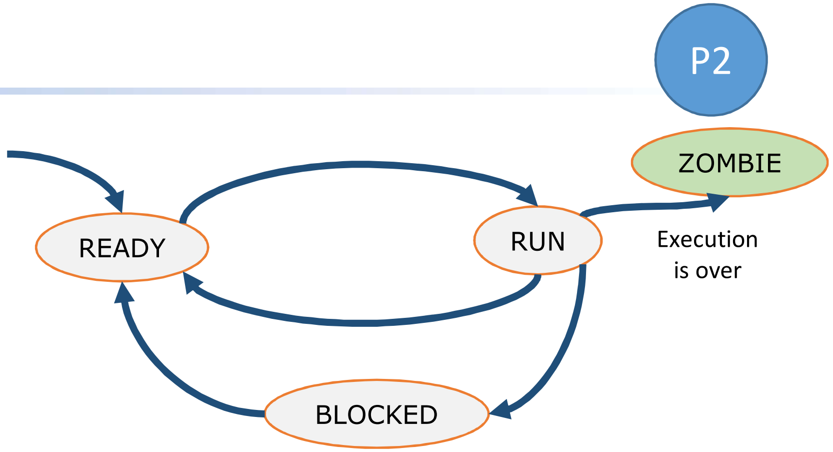
- A parent process can create multiple child processes

- Management of concurrent child process creations
  - The parent process does not wait to the death of a given child process to create the next one
  - Multiple child processes are alive at a time
    - More processes to be handled by the short term scheduler

- Management of sequential child process creations
  - The parent process waits to the death of a given child process before creating the next one
  - Only one child process is alive at a time
    - Only one additional child process to be handled by the short term scheduler

End of process execution

```
void exit (int);
```



- The process ends the execution
  - It turns to Zombie status
    - All resources are released (e.g. memory),
    - but the PCB (PID and return value) is preserved
  - Parameters:
    - An integer value that is the return value of the process execution (it is truncated to 1 Byte)

- The parent process has to release the *zombie* child process
  - Until that time, the PCB still exists and thus its PID

Wait for a child process to finish

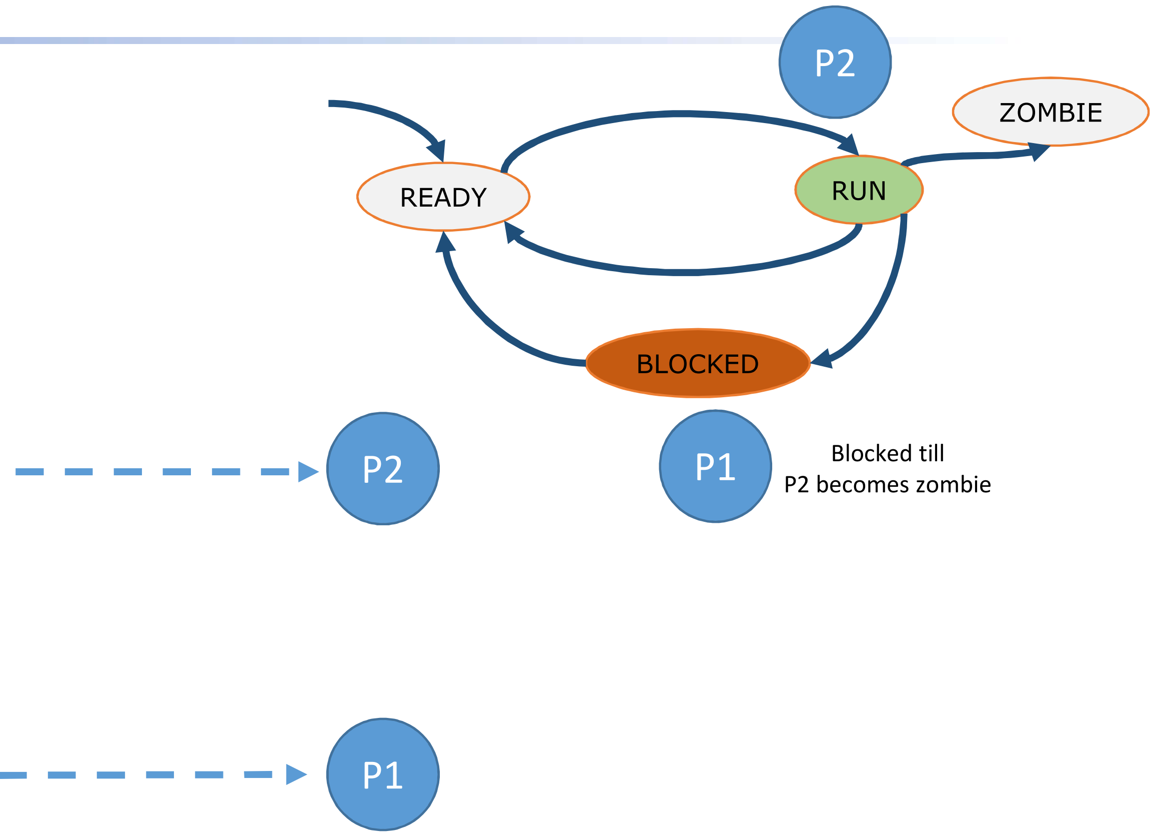
```
int  waitpid(int pid, int *status, int flags);
```

- The parent process (caller) releases a zombie child process
  - Returns the PID of the released child process
    - **Release the zombie child process** means to release the PCB, PID and related structures
  - Parameters:
    - Pid: pid of the child process to be released. The value “-1” indicates any child process
    - Status: is updated to hold the return value of the child process (or event that involved its finalization)
    - Flags: modify the behavior of the syscall. The value “0” indicates default behaviour

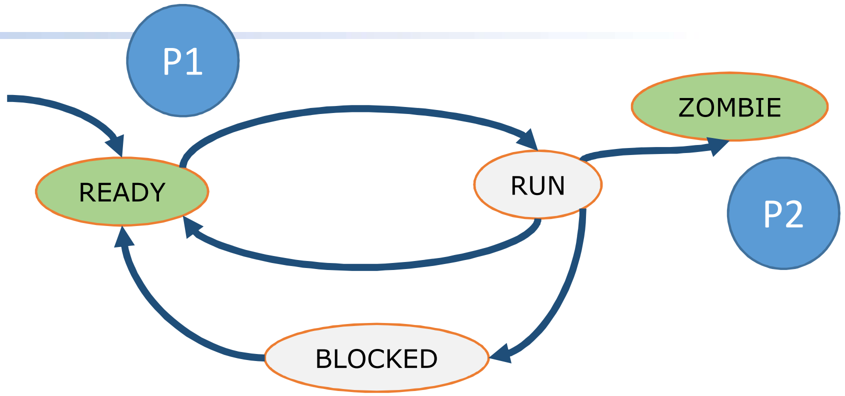
- The behavior
  - If there are child processes
    - If there is a zombie child process that matches with the “pid” parameter, it is released
    - Otherwise the parent process (caller) is blocked → this is a **blocking system call**
  - If there are no child processes, returns “-1”

Example: exit & waitpid

```
main () {
   int ret, count = 0, status = 0;
   count = 1;
   ret = fork();
   if (ret == 0){
      count = 2;
      printf("Child with counter = %d\n", count);
      for (count = 0; count < 100000; count++) {…}
      exit(count);
   } else {
      count = 3;
      printf("Parent with counter = %d\n", count);
   }
   ret = waitpid(-1, &status, 0);
   …
}
```



Example: exit & waitpid



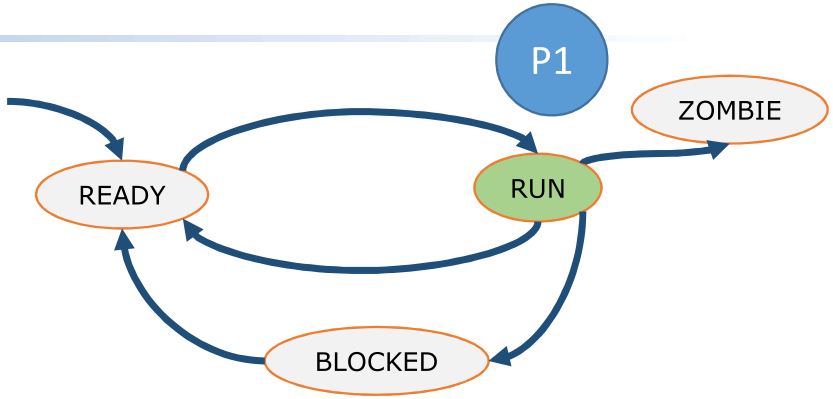

```
main () {
   int ret, count = 0, status = 0;
   count = 1;
   ret = fork();
   if (ret == 0){
      count = 2;
      printf("Child with counter = %d\n", count);
      for (count = 0; count < 100000; count++) {…}
      exit(count);
   } else {
      count = 3;
      printf("Parent with counter = %d\n", count);
   }
   ret = waitpid(-1, &status, 0);
   …
}
```

P2

count is 100001 (0x186**a1**)
exit (100001);

P1

Example: exit & waitpid




```
main () {
   int ret, count = 0, status = 0;
   count = 1;
   ret = fork();
   if (ret == 0){
      count = 2;
      printf("Child with counter = %d\n", count);
      for (count = 0; count < 100000; count++) {…}
      exit(count);
   } else {
      count = 3;
      printf("Parent with counter = %d\n", count);
   }
   ret = waitpid(-1, &status, 0);
   …
}
```

status is 161 (0x**a1**)

Example: exit conventions

- Error codes in exit(…) follow some common conventions
  - Code 0: program exited successfully
  - Code 1
    - Minor issues, e.g., grep returns 1 if no matching lines are found in any files
    - Errors occurred, e.g., find
  - Code 2 and above
    - Errors occurred, e.g. grep could not open at least one of the files provided
  - Usually, no negative numbers are returned

Execute a new program

```
int  execlp(const char *filename, const char *argv0, const char *argv1, const char *argv2, …, NULL);
```

```
int  execvp(const char *filename, char * const argv[]);
```

- Current process replaces the program (file) that is executing
  - A whole new memory contents and register values are loaded from “filename”
  - It performs dynamic linking, if necessary, and starts the program from its entry point
  - Parameters:
    - filename: indicates the name of the program to be loaded and executed (PATH is used to find it)
    - argvX: hold the command line arguments for the program to be executed
      - As the number of input parameters is variable, a NULL is required to indicate there are no more parameters
    - argv[]: same as argvX, but in array format. Last entry in the array must be a NULL pointer

- The behavior
  - If the new program can be found, loaded and started, it never returns
    - **Once it is mutated, the previous memory contents (e.g. code, data) are not there any more**
  - If there is any problem DURING the mutation, it returns “-1” to indicate an error
    - E.g. the “filename” is wrong, the user has no permission to execute the “filename”, etc.

To sum up: Example of Shell Behavior

- For every command line (e.g. “#> ls –l –a”), the shell uses these syscalls

```
int ret, status;
char InArgs[10][256];

while (InArgs = Read Command Line()){
    ret = fork();
    if (ret == 0){
        /* we want to execute  “ls   -l   -a”. That is, InArgs[0] = “ls”, InArgs[1] = “-l”, InArgs[2] = “-a” */
        execvp(InArgs[0], InArgs);
    }
    waitpid(-1, &status, 0);
}
```

Error control

- It is extremely important to check for errors
  - On system calls
  - On library calls
- Manual pages describe the way system/library routines return errors

```
RETURN VALUE    -- exec
       The  exec() functions return only if an error has occurred.  The return
       value is -1, and errno is set to indicate the error.

RETURN VALUE    -- fork
       On success, the PID of the child process is returned in the parent, and
       0 is returned in the child.  On failure, -1 is returned in the  parent,
       no child process is created, and errno is set appropriately.

RETURN VALUE    -- exit
       These functions do not return.
```

Error control

- System calls usually return -1 on error and
  - Set the **errno** variable to contain the code of the specific error

```
RETURN VALUE    -- waitpid

       waitpid(): on success, returns the process ID of the child whose  state
       has changed; if WNOHANG was specified and one or more child(ren) speci-
       fied by pid exist, but have not yet changed state, then 0 is  returned.
       On error, -1 is returned.

ERRORS
       ECHILD The process specified by pid does not exist or  is  not  a
              child  of  the  calling process.

       EINVAL The options argument was invalid.
```

Error control

- Sample code to manage errors (test-for-children.cpp)

```
#include <sys/wait.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
   int status;
   pid_t pid, mychild;

 ... mychild = ...

   pid = waitpid(mychild, &status, 0);
   if (pid < 0) {
      perror ("waitpid");
      exit(1);      // optional?
   }
...
```

If the pid returned is -1 ...

perror formats the error message:

`waitpid: No child processes`

- If the application cannot continue, issue the exit(...)
- If the application can continue, the user will just get the error message

Error control

- Common UNIX/Linux error codes

/usr/include/asm-generic/errno-base.h

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD          10      /* No child processes */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
#define EEXIST          17      /* File exists */
...
#define EHWPOISON       133     /* Memory page has hardware error */
```

Table of Contents

- Operating System
- Basic Concepts
- Access to Kernel functions
- Process management
- Memory subsystem
- Other important OS concepts and tasks

- NOTE: I/O subsystem and filesystems will be studied in future lessons

Memory Management

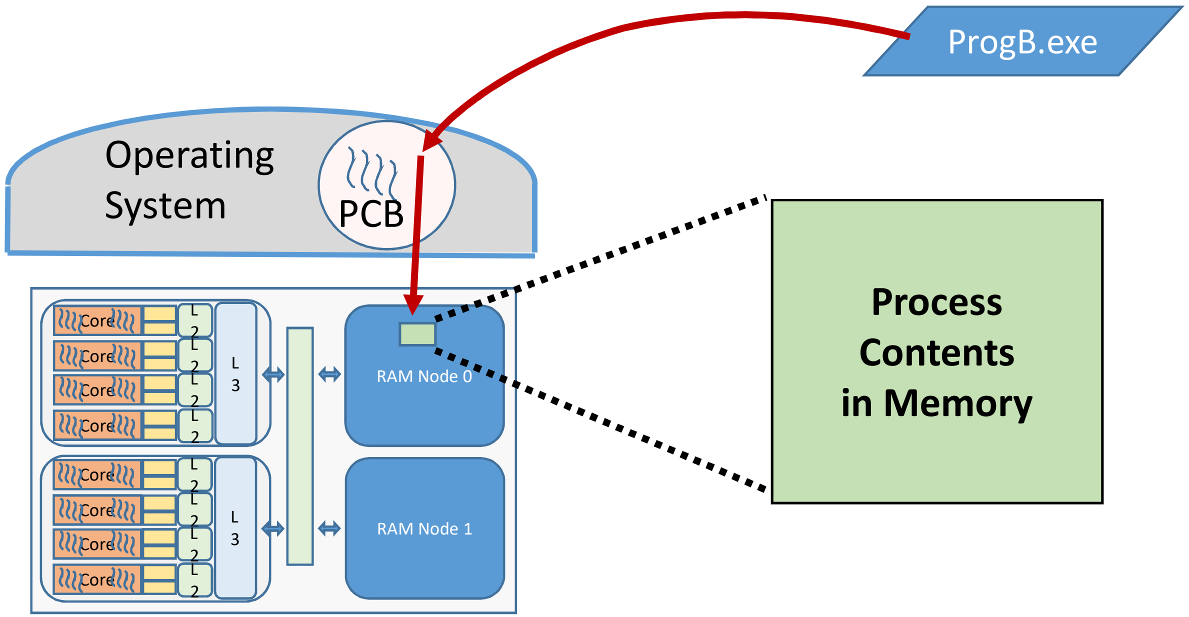
- The CPU can only access directly to memory and the register bank
  - Instructions and data must be located in main memory

- 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

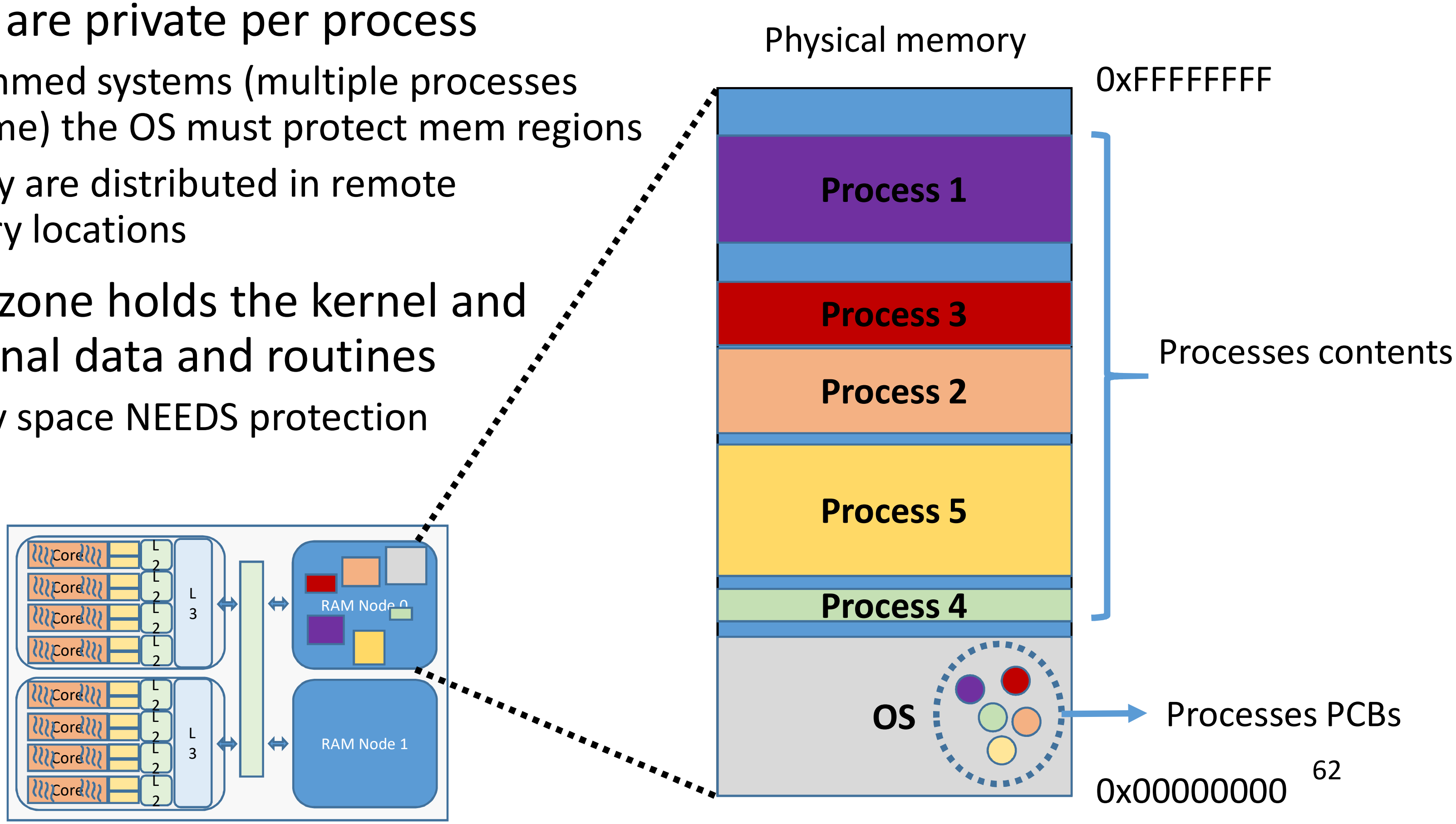
- The OS loads the program from the disk to Physical Memory
  1) Request & reserve space in main memory
  2) Load the program
  3) Start running

Binary file in the disk



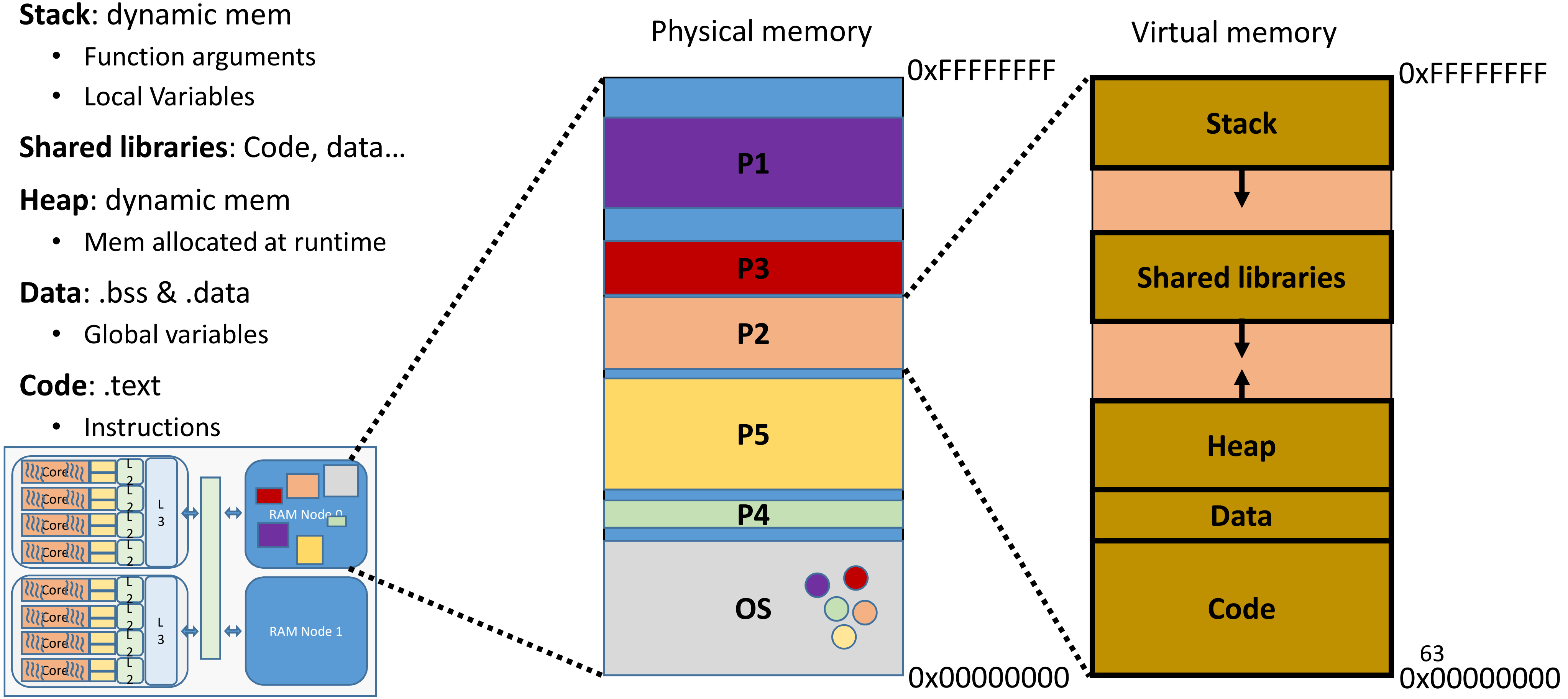
Multiprogrammed OS

- 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



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

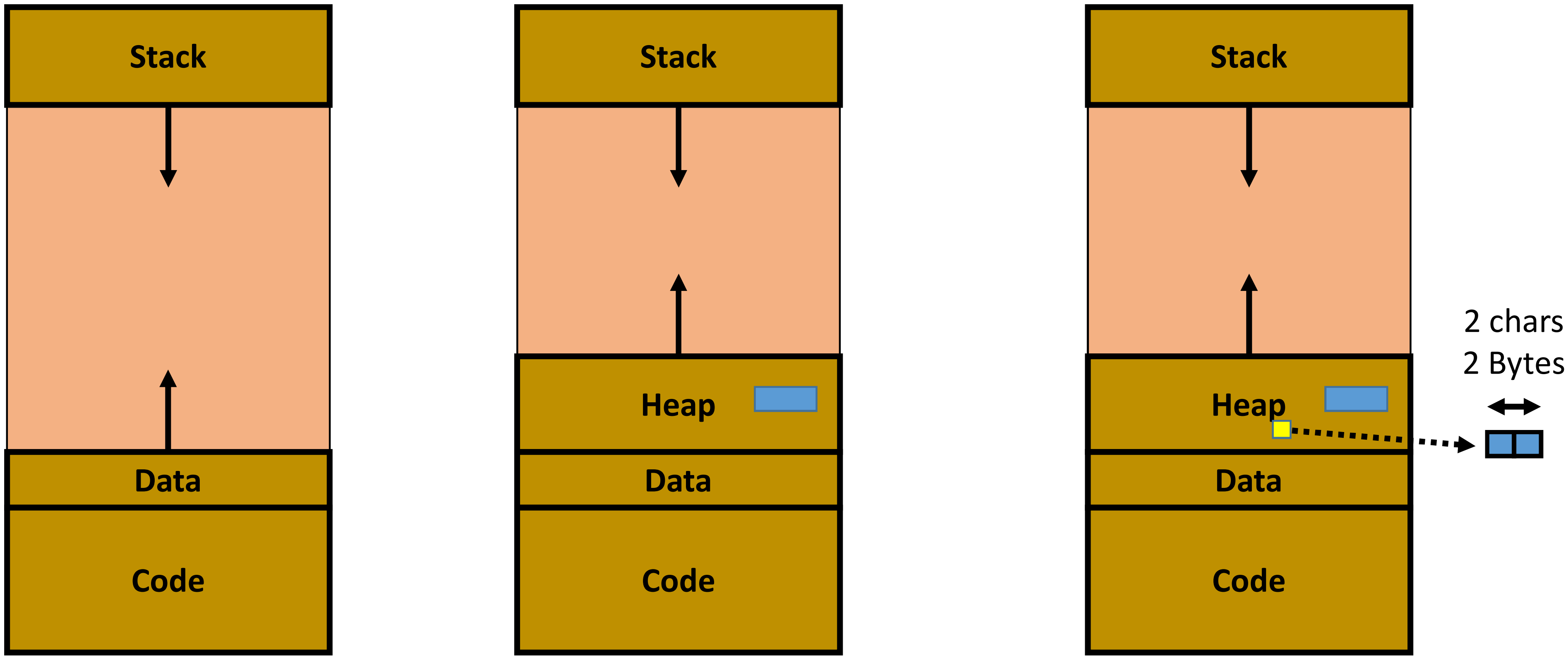
- Behavior
  - Before allocating @, it checks whether the HEAP has space enough to hold size bytes
    - If not, the OS increases the limit of the HEAP (sbrk)
    - The HEAP size is increased by a configurable number of bytes to reduce the number of times it has to be increased
  - The object memory 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

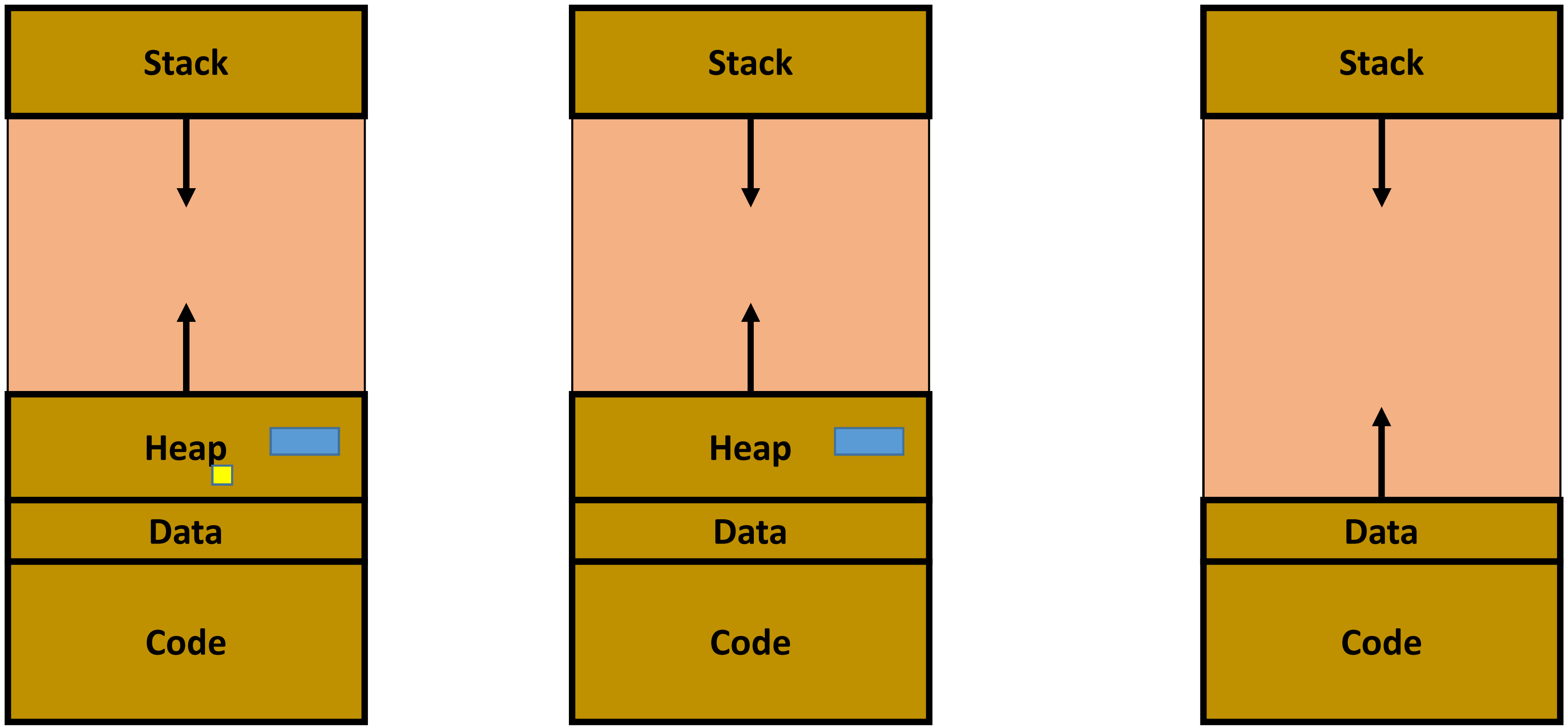
- Behavior
  - Deallocation of the memory zone pointed by the input parameter
  - The OS will consider to reduce or not the HEAP memory space
  - A pair of lists are maintained by malloc/new/free/delete
    - List of objects allocated
    - List of objects deallocated (may be merged)

Memory Deallocation

Starting status

After calling:
free(string);

After calling:
free(ptr);



Error control

- As with process management routines, memory management calls can return error codes in the **errno** variable

```
RETURN VALUE
       The malloc() and calloc() functions return a pointer to  the  allocated
       memory,  which  is  suitably  aligned for any built-in type.  On error,
       these functions return NULL.  NULL may also be returned by a successful
       call  to  malloc() with a size of zero, or by a successful call to cal-
       loc() with nmemb or size equal to zero.

       The free() function returns no value.

ERRORS
       calloc(), malloc(), and realloc() can fail with the following error:

       ENOMEM Out  of  memory.  Possibly, the application hit the RLIMIT_AS or
              RLIMIT_DATA limit described in getrlimit(2).
```

Error control

- C library routines can return the NULL pointer to indicate an error

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

#define N (1024*1024*1024*15L)

int main()
{
   int * p = (int *) malloc(N);

   if (p==NULL) {
      fprintf(stderr, "malloc returns NULL pointer:\n %s\n", strerror(errno));
      exit(1);        // optional?
   }
   printf ("pointer %p\n", p);
   //...
}
```

If the pointer returned is NULL...

fprintf formats the error message, similarly to “perror”:

```
malloc returns NULL pointer:
 Cannot allocate memory
```

- If the application cannot continue, issue the exit(...)
- If the application can continue, the user will just get the error message

Error control

- C++ gets also the possibility to use exceptions

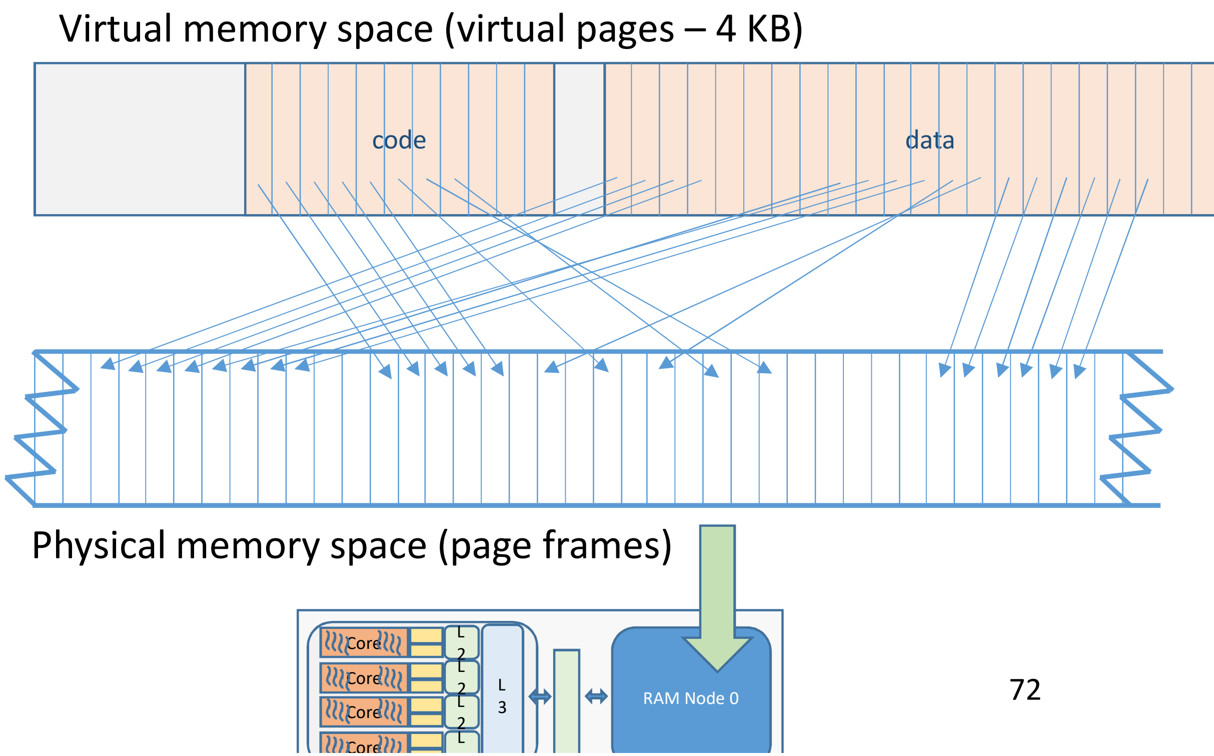
```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <vector>

#define N 1024*1024*512

int main()
{
 try {
   std::vector<float> *v = new(std::vector<float> [N]);

   // work with v

 } catch(std::exception & e) {
     std::cerr << "Exception catched: " << e.what() << std::endl;
     std::perror("std::vector allocation");
   std::exit(1);   // equivalent to return (1);
 }
...
```

If an exception is raised, the program can access to the type of the exception an the <u>errno</u> code:

```
std::vector allocation: Cannot allocate memory
```

Memory Management

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



Memory management

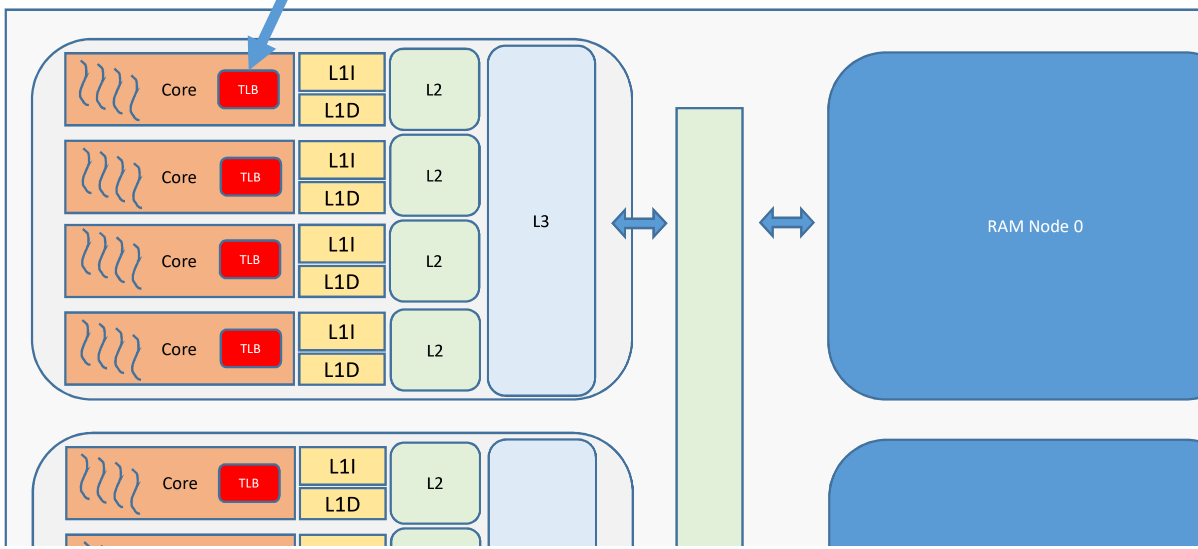
- Hardware support: Memory Management Unit **(MMU)**
  - Combination of technics (e.g. paged segmentation in Linux)
  - Protection checking
- Per-process **page tables** provide the translation
  - Virtual address to physical frame
  - Pages are validated when allocated. Pages are brought (Present) when accessed
- When memory is exhausted...
  - A physical frame is selected->Written to the **swap area** [if modified]->Reallocated to another virtual page
  - Difference between page replacement and swapping
- Memory leakage: **Fragmentation** (internal vs external)

Memory management

- Address translation
  - Address space is defined through the page table (per process)

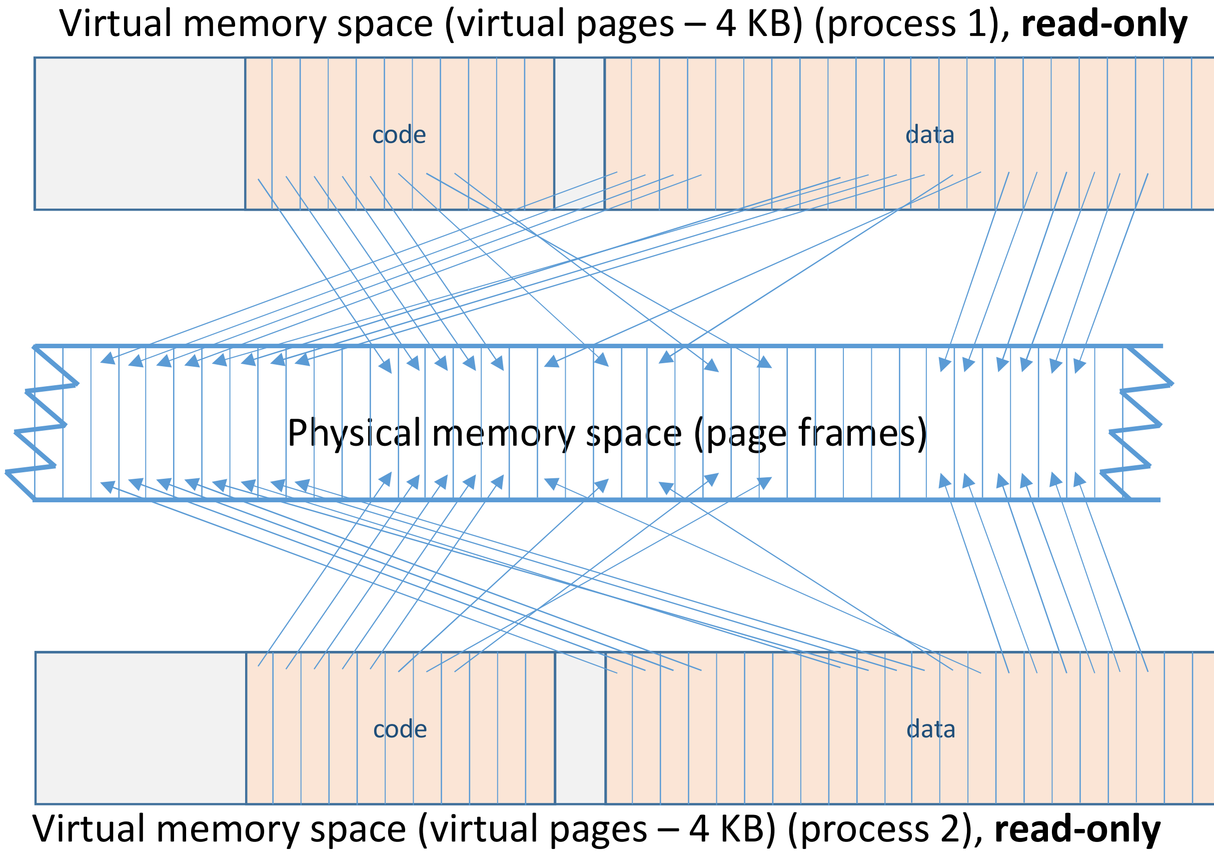
Virtual address	Physical frame	Protection	Valid/Present
			I –
0x400000	0x053000	r – x	V P
0x401000	0x048000	r – x	V nP
…			I –
0x601000	0x02F000	r – –	V P
0x602000	0x147000	rw –	V P
0x603000	0x148000	rw –	V P
0x604000	0x149000	rw –	V nP
…			I –
0xFF0000	0x15F000	rw –	V P
0xFF1000	0x160000	rw –	V P
0xFF2000	0x044000	rw –	V nP
0xFF3000	0x059000	rw –	V P

- Optimization
  - Part of the page table is kept inside the processor
    - Translation Lookaside Buffer (TLB)



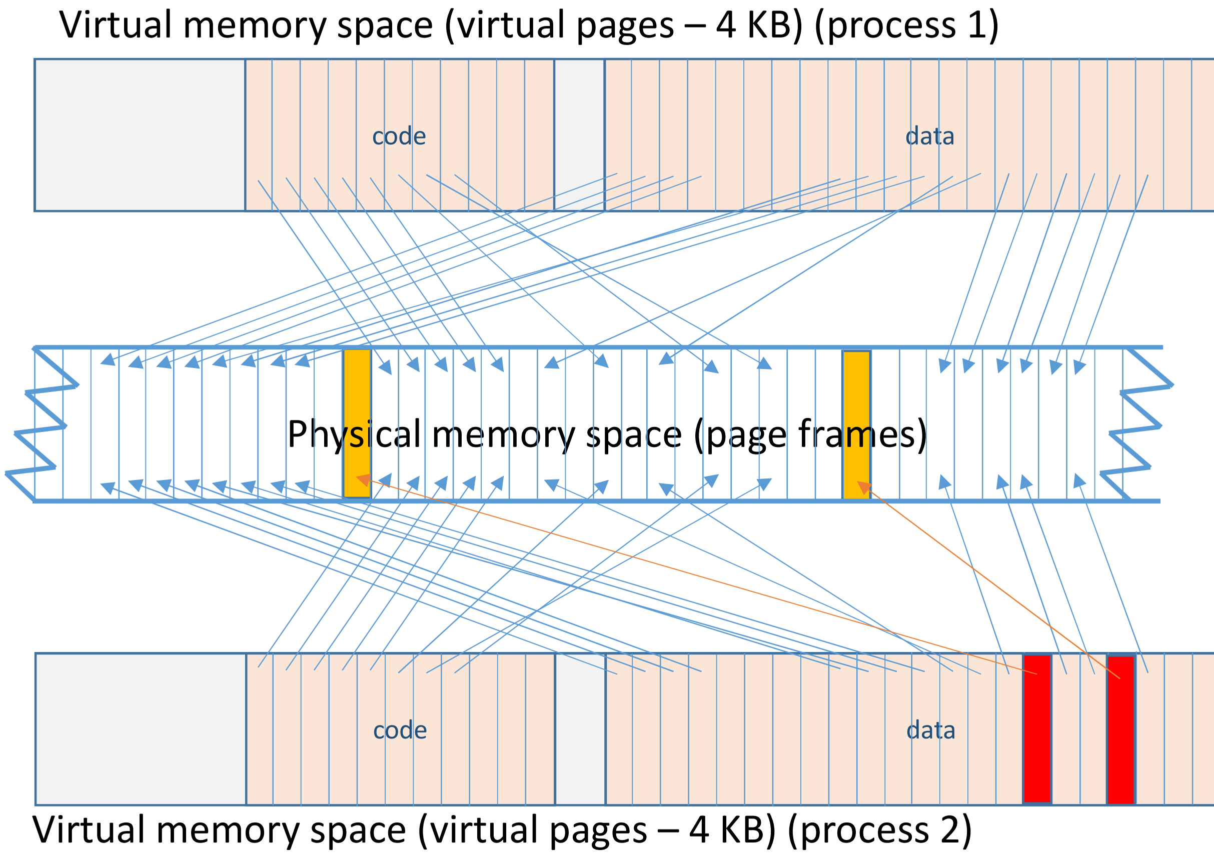
Memory Management on fork

- On fork(), copy-on-write
  - Page changes are protected on both processes
  - While data is only read (code is only fetched)
    - No page duplication
  - As soon a store operation is done, that page is duplicated



Memory Management on fork

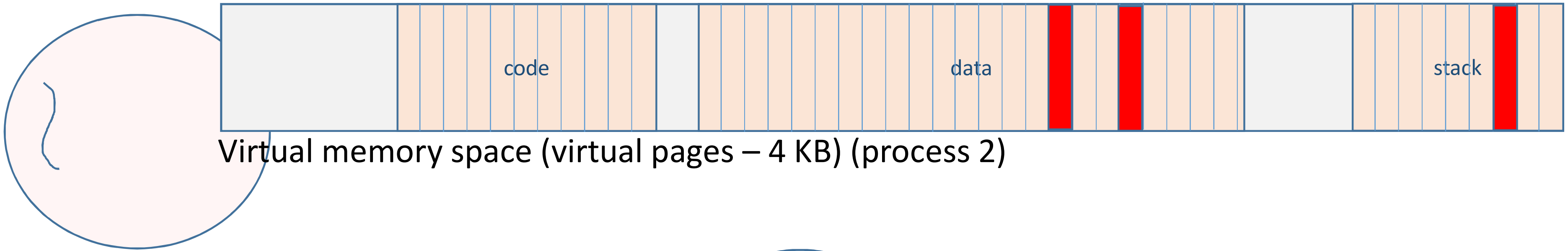
- Copy-on-write
  - Process 2 tries to modify a data page
    - Read-only!
  - Protection exception is raised
- The OS
  - Gets a free page frame
  - Copies the page to the new frame
  - Redirects process 2 to the new frame
- This process is repeated for each modification access



Memory Management on exec

- Actions on exec()

child



- Address space is completely renewed
  - New page tables, TLB flushed

child

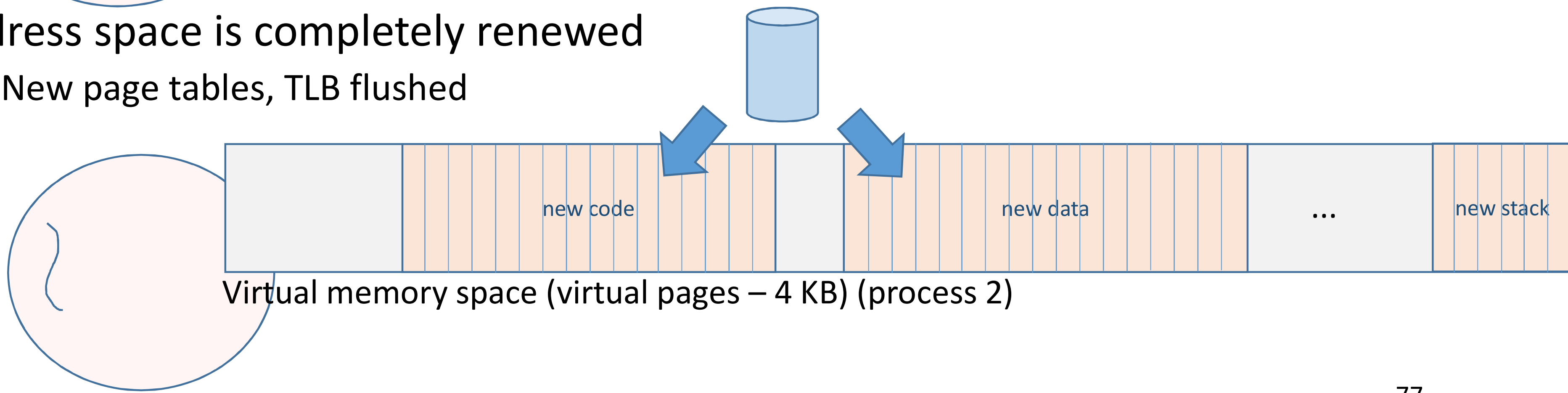


Table of Contents

- Operating System
- Basic Concepts
- Access to Kernel functions
- Process management
- Memory subsystem
- Other important OS concepts and tasks

- NOTE: I/O subsystem and filesystems will be studied in future lessons

Other important OS concepts and tasks

- The OS is in charge of many other management tasks
  - I/Osubsystem
  - Filesystem
  - Inter-process communication
- Some of them will be addressed in the two final lessons of this course

- New technologies are arising to provide new functionalities to the OS
  - Distributed systems
  - Virtualization technologies used for different management purposes
  - Accelerators (GPUs, FPGAs…) management

Bibliography

- Computer Organization and Design (5th Edition)
  - D. Patterson and J. Hennessy
  - http://cataleg.upc.edu/record=b1431482~S1*cat
    - Introduces hardware support for OS

- Operating System Concepts (John Wiley & Sons, INC. 2014)
  - Silberschatz, A; Galvin, P. B; Gagne, G.
  - http://cataleg.upc.edu/record=b1431631~S1*cat
    - Introduces the presented concepts about OS

Next steps

- Support to the programming environment
  - Execution environment
  - Parallelism
  - Performance analysis