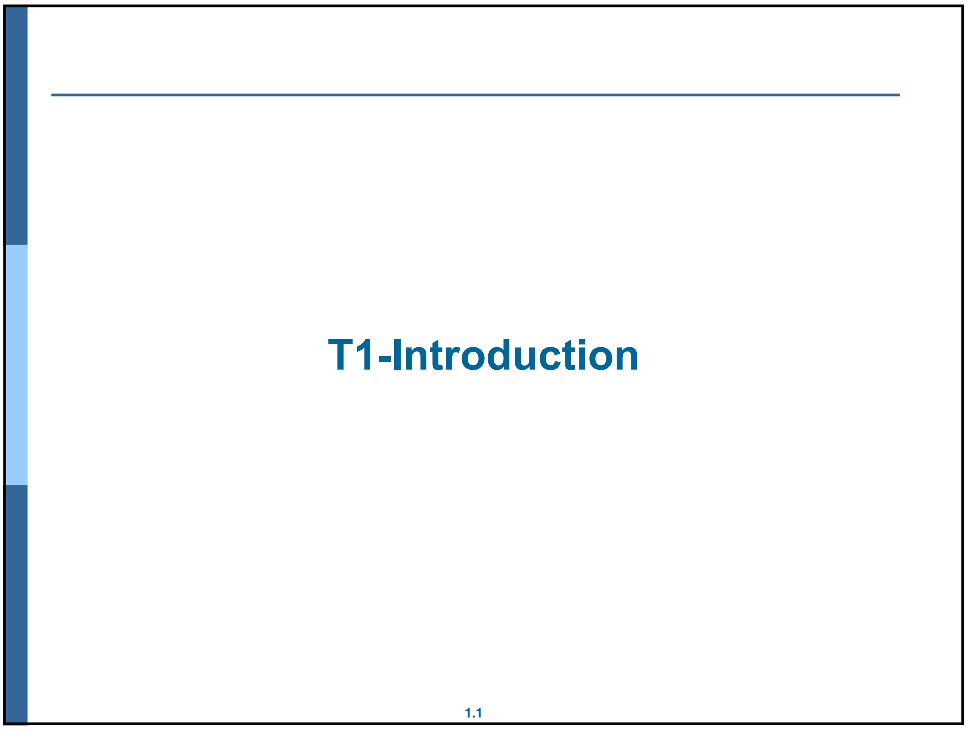
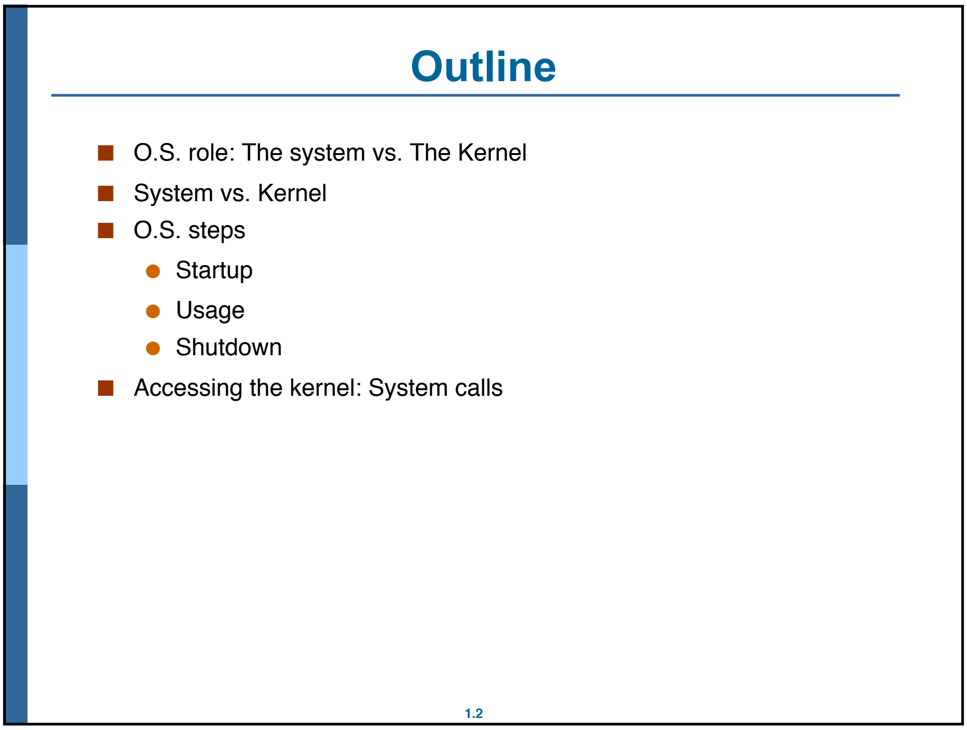


T1-Introduction

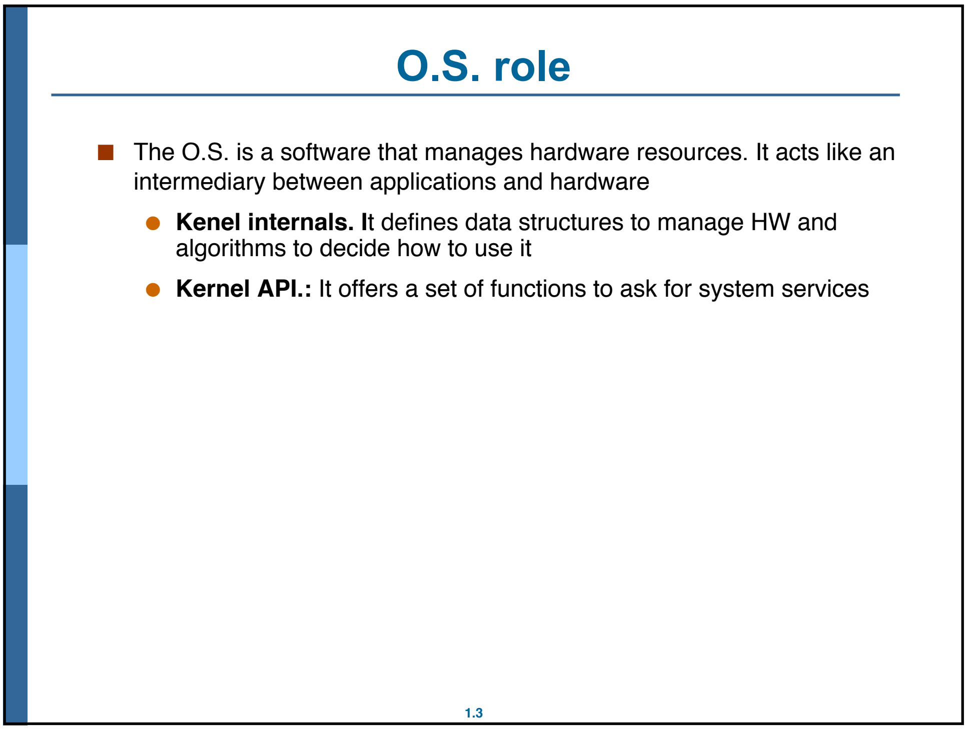
1.1



Outline

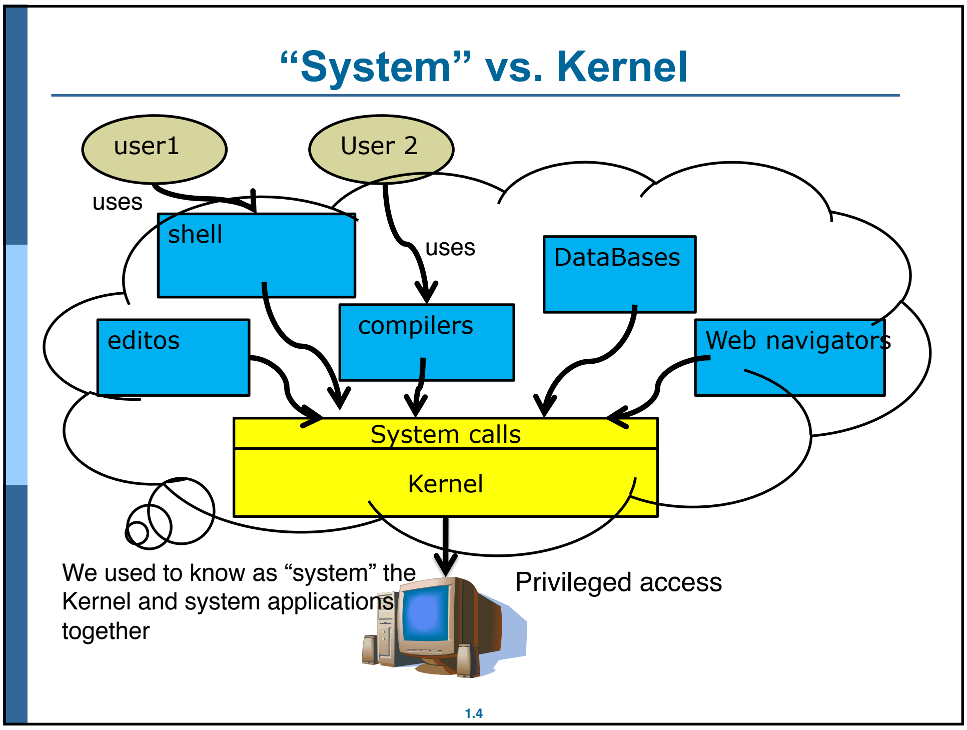
- O.S. role: The system vs. The Kernel
- System vs. Kernel
- O.S. steps
  - Startup
  - Usage
  - Shutdown
- Accessing the kernel: System calls

1.2



O.S. role

- The O.S. is a software that manages hardware resources. It acts like an intermediary between applications and hardware
  - **Kenel internals. I**t defines data structures to manage HW and algorithms to decide how to use it
  - **Kernel API.:** It offers a set of functions to ask for system services



"System" vs. Kernel

user1

User 2

uses

shell

uses

DataBases

editos

compilers

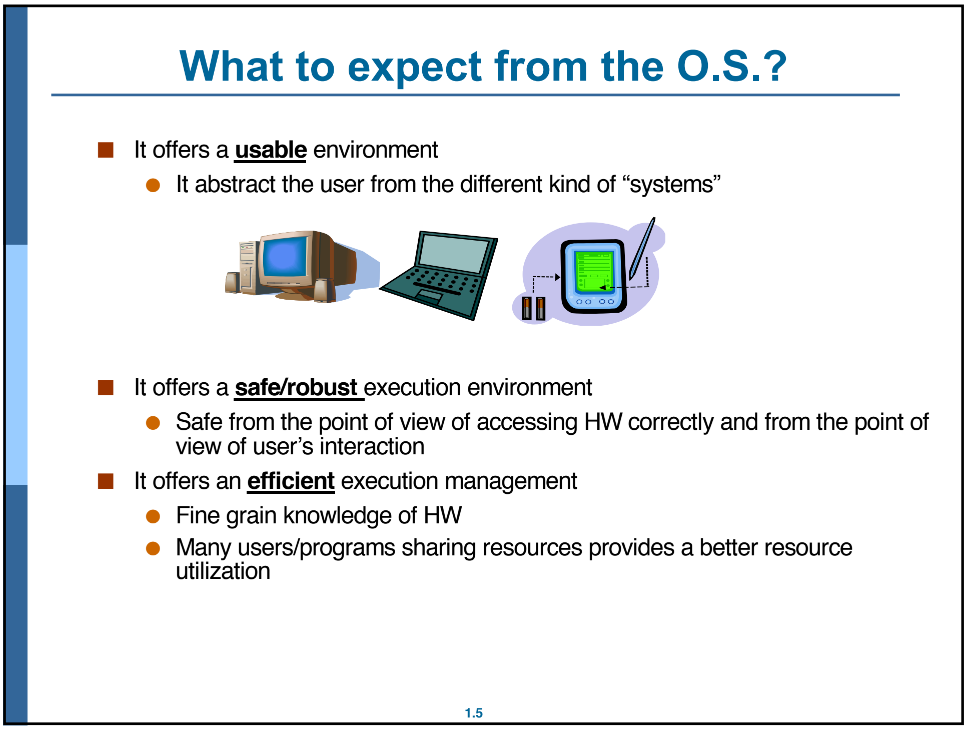
Web navigators

System calls

Kernel

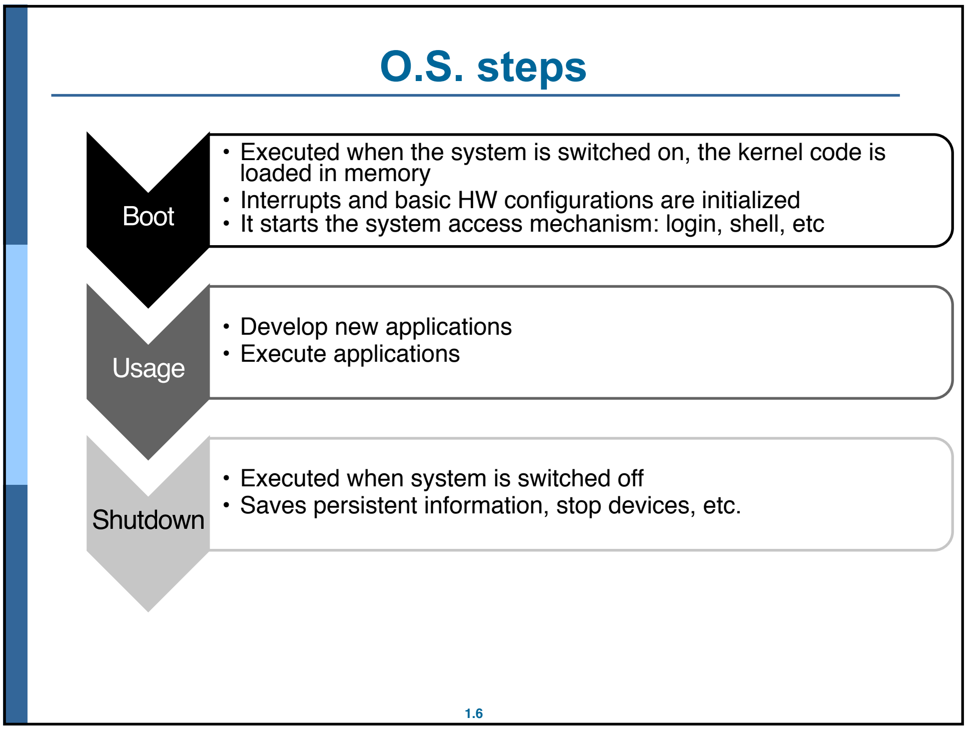
We used to know as "system" the Kernel and system applications together

Privileged access



What to expect from the O.S.?

- It offers a **usable** environment
  - It abstract the user from the different kind of “systems”
- It offers a **safe/robust** execution environment
  - Safe from the point of view of accessing HW correctly and from the point of view of user’s interaction
- It offers an **efficient** execution management
  - Fine grain knowledge of HW
  - Many users/programs sharing resources provides a better resource utilization



O.S. steps

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: login, shell, etc

Usage
- Develop new applications
- Execute applications

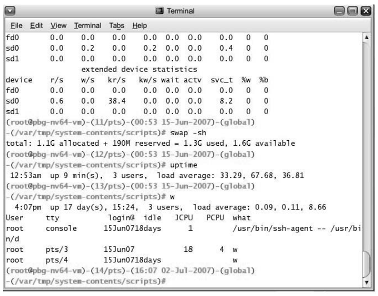
Shutdown
- Executed when system is switched off
- Saves persistent information, stop devices, etc.

Boot

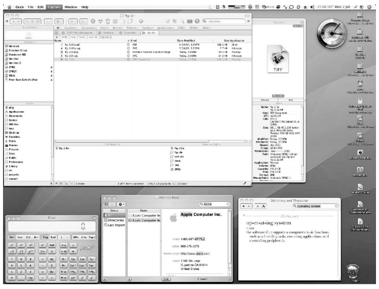
- A first piece of software is automatically loaded in memory by the hardware. This minimum boot code is in charge of startup the rest of the kernel
    - Loaded from the disc or other device
    - Loaded from network
    - Showing a list of available options to boot (boot loader) (for example GRUB) [1]
- Once the system to boot is selected
    - It is fully copied in memory
    - All the HW structures required are initialized
    - The kernel gets the full control of HW access
        - All the HW mechanisms to automatically execute code are captured by the kernel
            - Interrupts/exceptions/system calls

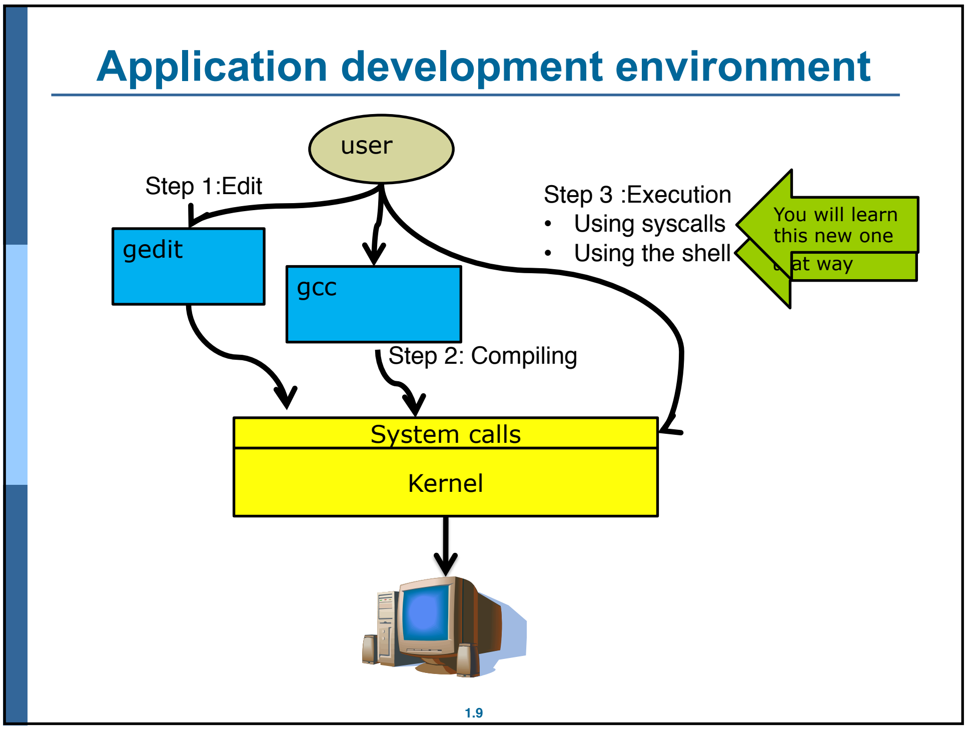
Using the system…

- When using the system, there are two main ways:
    - Using some interactive tool such as shells or graphical environments
        - This is an indirect access to the kernel, since "services" are requested to these tools (execute program X), and tools translate user request to kernel system calls
    - Using directly system calls



```
fd0       0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd0       0.0    0.2    0.0    0.2  0.0  0.0    0.4   0   0
sd1       0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
                 extended device statistics
device    r/s    w/s   kr/s   kw/s wait actv  svc_t  %w  %b
fd0       0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd0       0.6    0.0   38.4    0.0  0.0  0.0    8.2   0   0
sd1       0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# uptime
 12:53am  up 9 min(s),  3 users,  load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# w
  4:07pm  up 17 day(s), 15:24,  3 users,  load average: 0.09, 0.11, 8.66
User     tty           login@  idle   JCPU   PCPU  what
root     console     15Jun0718days      1          /usr/bin/ssh-agent -- /usr/bi
n/d
root     pts/3       15Jun07             18      4  w
root     pts/4       15Jun0718days                  w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-(/var/tmp/system-contents/scripts)#
```





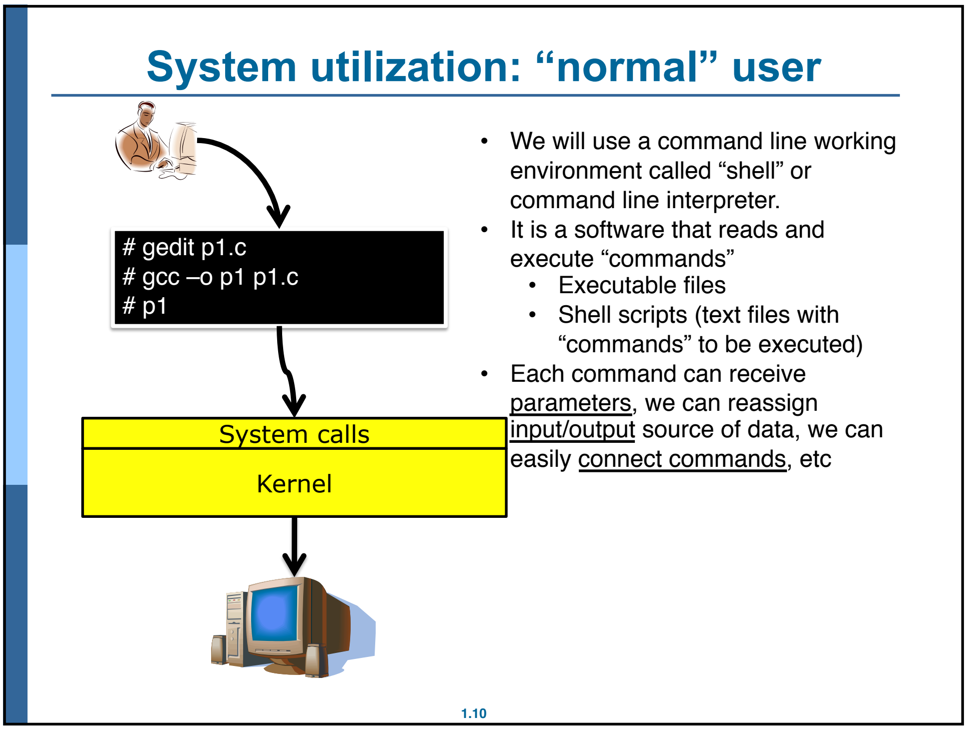
Application development environment

Step 3 :Execution

- Using syscalls
- Using the shell

You will learn this new one that way

1.9



System utilization: “normal” user

```
# gedit p1.c
# gcc –o p1 p1.c
# p1
```

- We will use a command line working environment called “shell” or command line interpreter.
- It is a software that reads and execute “commands”
  - Executable files
  - Shell scripts (text files with “commands” to be executed)
- Each command can receive parameters, we can reassign input/output source of data, we can easily connect commands, etc

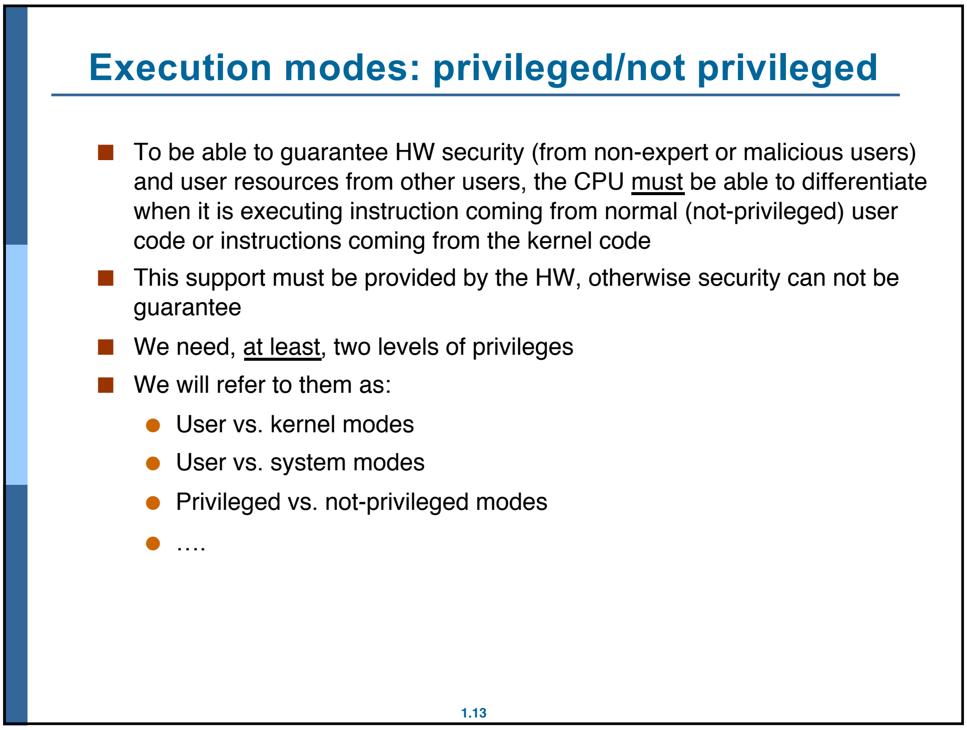
1.10

System utilization: “normal” user

```
Ej1 # command1
Ej2 # command2 par1 par2
Ej3 # command2 par1 par2 < input_data
Ej4 # command2 par1 par2 < input_data > output_data
Ej5 # command2 par1 par2 l command3 par1 par2 par3
```

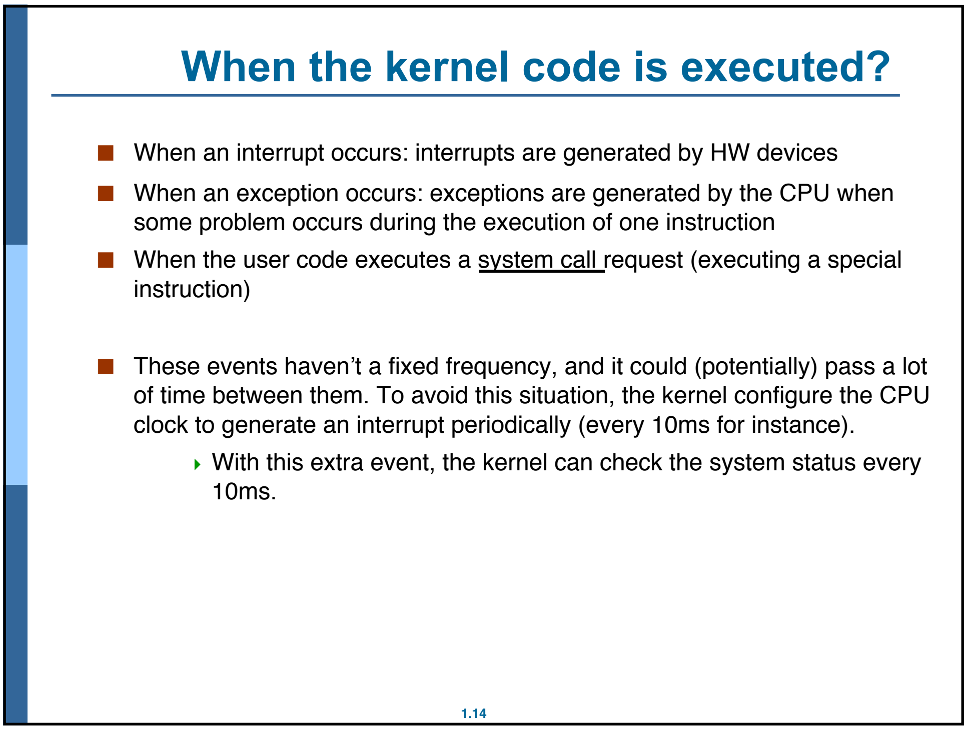
- Ej1: Command1 is an executable file without parameters
- Ej2: Command2 is an executable 2 with 2 parameters
  - Input/output data are read/written from/to the console
- Ej3: We can reassign standard input by using special character <
  - Input data will be gathered from file input_data file rater than from console
- Ej4: We can reassign standard output by using special character >
  - Data generated will be written in output_data file rather than to the console
- Ej5: We can connect two commands by using special character l
  - Output data of command2 will be the input data of command3

ACCESSING THE KERNEL CODE



Execution modes: privileged/not privileged

- To be able to guarantee HW security (from non-expert or malicious users) and user resources from other users, the CPU must be able to differentiate when it is executing instruction coming from normal (not-privileged) user code or instructions coming from the kernel code
- This support must be provided by the HW, otherwise security can not be guarantee
- We need, at least, two levels of privileges
- We will refer to them as:
  - User vs. kernel modes
  - User vs. system modes
  - Privileged vs. not-privileged modes
  - ….



When the kernel code is executed?

- When an interrupt occurs: interrupts are generated by HW devices
- When an exception occurs: exceptions are generated by the CPU when some problem occurs during the execution of one instruction
- When the user code executes a system call request (executing a special instruction)
- These events haven’t a fixed frequency, and it could (potentially) pass a lot of time between them. To avoid this situation, the kernel configure the CPU clock to generate an interrupt periodically (every 10ms for instance).
  - With this extra event, the kernel can check the system status every 10ms.

Access through system calls

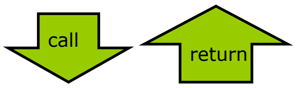
- System calls are explicit requests for some kernel service [2]
  - For instance: Linux version 2.4.17 has 1100 system calls
- From the point of view of programmers they have the same look and feel that a library function call

```
####> man printf 3
printf(3)              linux programmer's manual              printf(3)
name
     printf,  fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
     vsnprintf - formatted output conversion

synopsis
     #include <stdio.h>

     int printf(const char *format, ...);
......
```

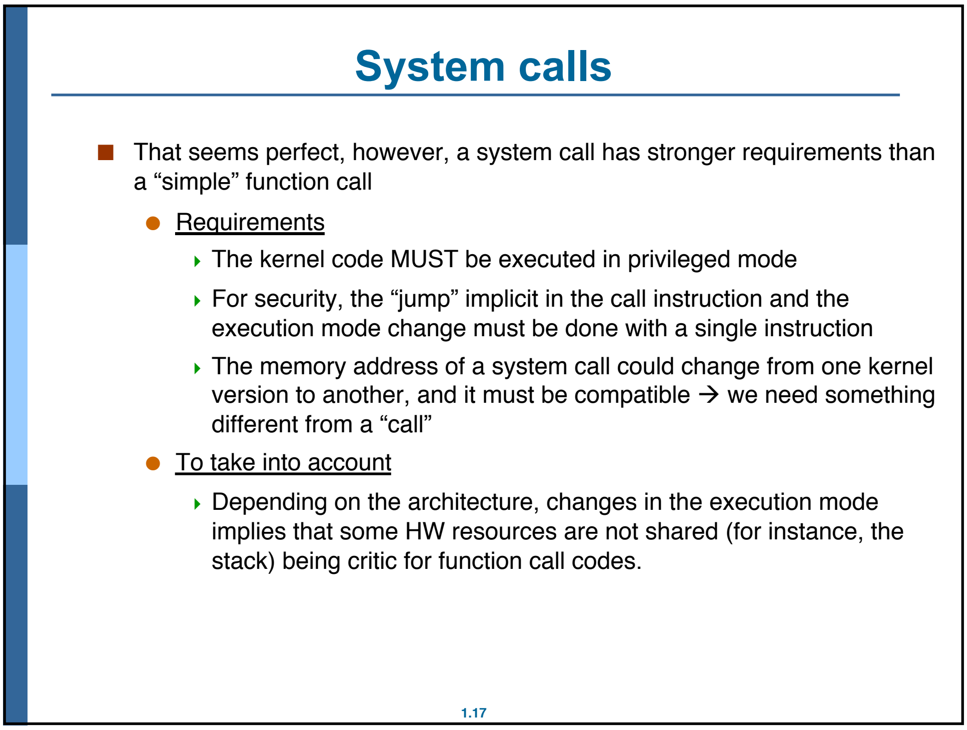
```
####> man write 2
WRITE(2)               Linux Programmer's Manual              WRITE(2)
NAME
     write - write to a file descriptor
SYNOPSIS
     #include <unistd.h>
     ssize_t write(int fd, const void *buf, size_t count);
......
```

System calls

- Programmers insert a function call in their codes, and the compiler generates the low level code automatically
  - Pushing arguments in the stack
  - Calling the function address
  - Getting return value (for instance from register eax)

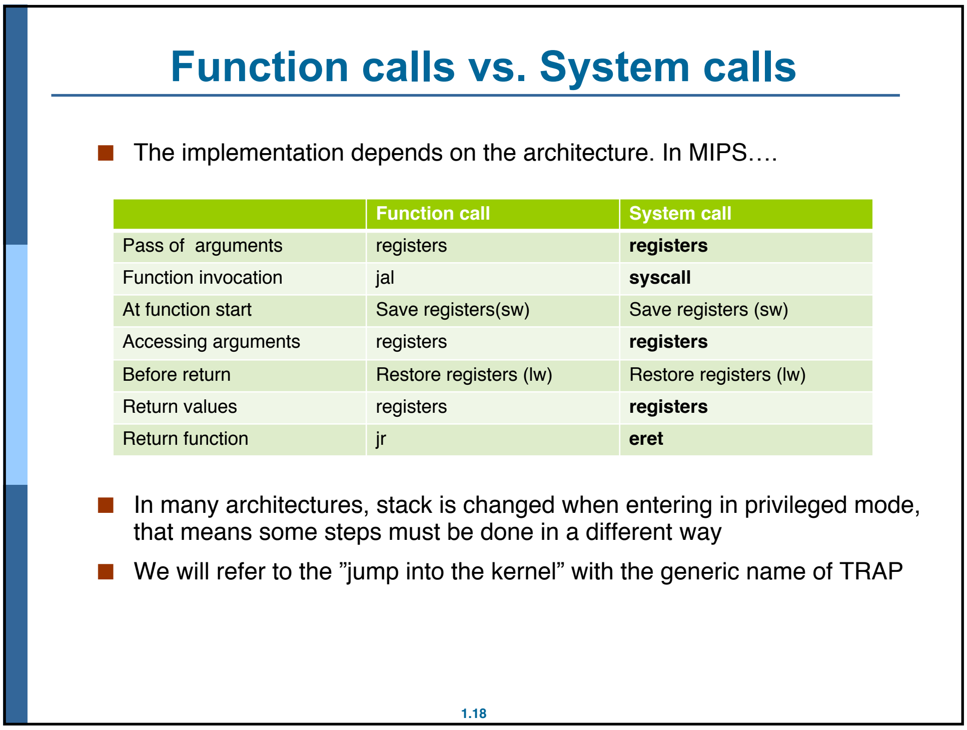


- The system call code is in the kernel code, and the compiler also generates the code for it:
  - Reserving space for local variables in the stack
  - Accessing arguments from the stack
  - Returning values (for instance using register eax)



System calls

- That seems perfect, however, 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 something different from a “call”
  - To take into account
    - Depending on the architecture, changes in the execution mode implies that some HW resources are not shared (for instance, the stack) being critic for function call codes.

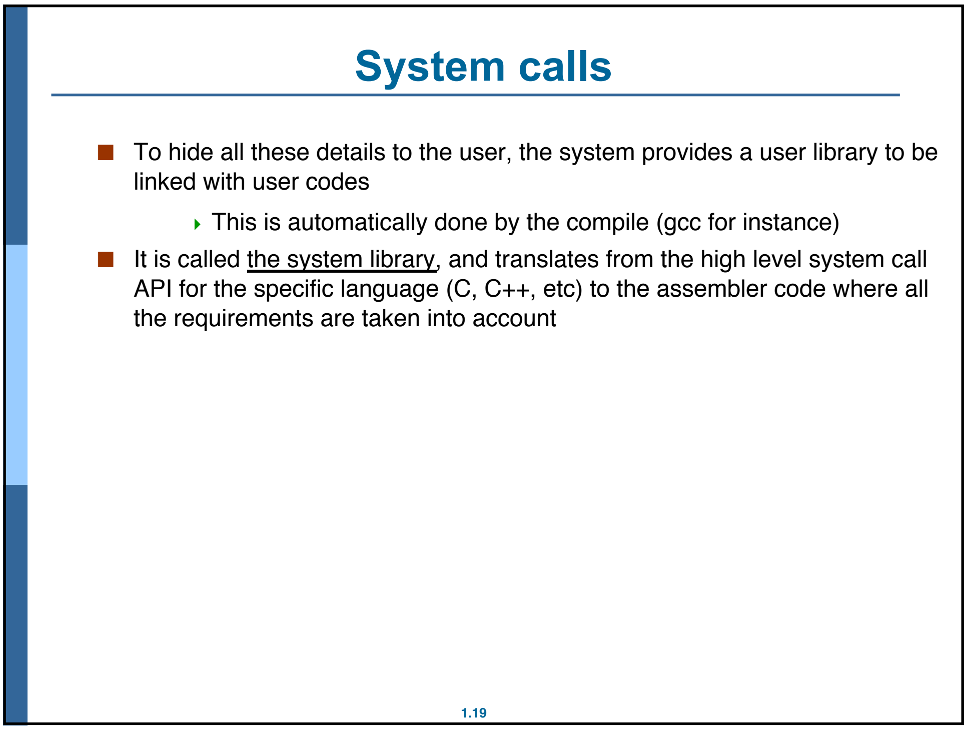


Function calls vs. System calls

- The implementation depends on the architecture. In MIPS….

	Function call	System call
Pass of arguments	registers	**registers**
Function invocation	jal	**syscall**
At function start	Save registers(sw)	Save registers (sw)
Accessing arguments	registers	**registers**
Before return	Restore registers (lw)	Restore registers (lw)
Return values	registers	**registers**
Return function	jr	**eret**

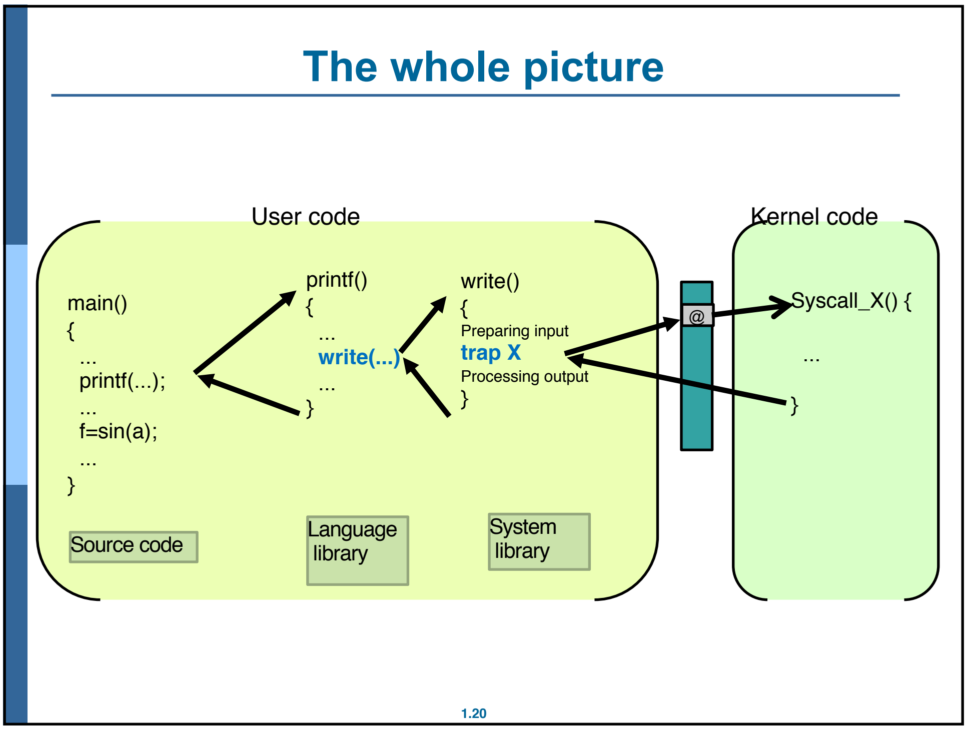
- In many architectures, stack is changed when entering in privileged mode, that means some steps must be done in a different way
- We will refer to the ”jump into the kernel” with the generic name of TRAP



System calls

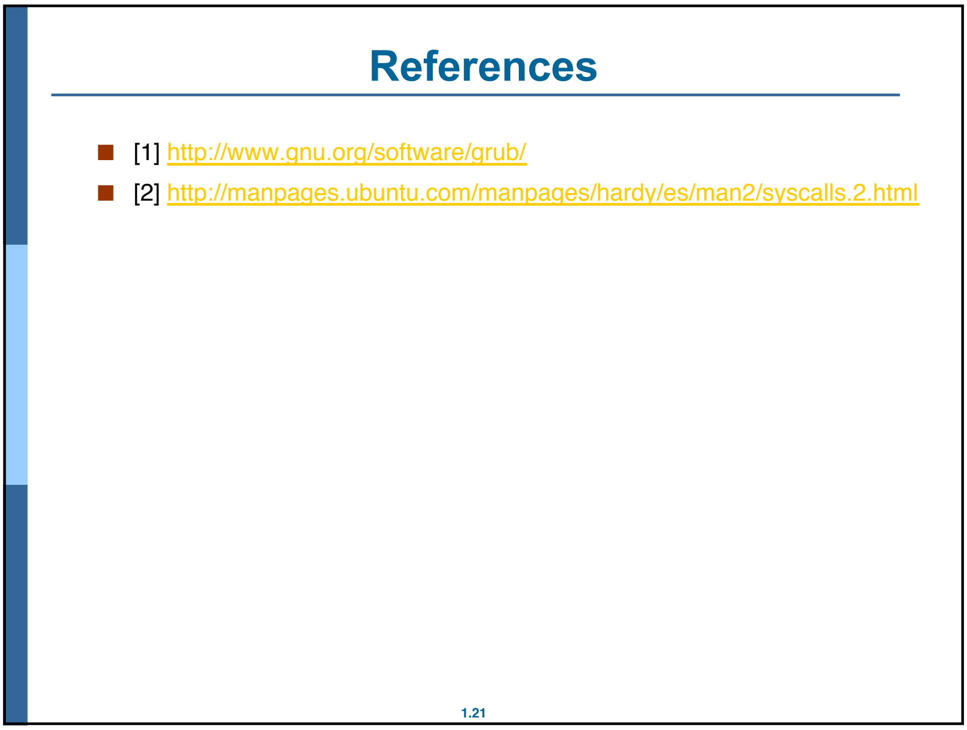
- To hide all these details to the user, the system provides a user library to be linked with user codes
  - This is automatically done by the compile (gcc for instance)
- It is called the system library, and translates from the high level system call API for the specific language (C, C++, etc) to the assembler code where all the requirements are taken into account

1.19



The whole picture

1.20



References

- [1] http://www.gnu.org/software/grub/
- [2] http://manpages.ubuntu.com/manpages/hardy/es/man2/syscalls.2.html