Clase 3

Materia	ISO Teoría
Estado	Done
Fecha	@August 30, 2022

System Call

Las llamadas al sistema es la forma en la que, los *programas de usuario* pueden acceder a los servicios del sistema operativo. Estas **funciones llaman al kernel y este les da servicio-**

Como dije antes: son funciones. Los parámetros asociados a las llamadas pueden pasarse de varias maneras: por registros, bloques o tablas en memoria ó la pila.

```
count = read (file , buffer, nbytes);
```

> **Se implementan desde modo usuario pero se ejecutan en modo Kernel.**

Para activar una SysCall se indica el número de syscall que se quiere ejecutar y los parámetros de esta. Luego se emite una interrupción para pasar a modo **kernel** y gestionar la systemcall. El manejador de interrupciones (*syscall handler*) evalúa la SysCall y la ejecuta**. Un poco mejor y más claro es así:**

Como detalle, es importante saber en qué **arquitectura se está trabajando**, ya que de esto depende la tabla. Con la arquitectura cambia los números de instrucciones (y una instrucción en los procesadores de 64 que hace todo un poco más eficiente), pero la libreria esta amortigua los cambios para que

Pasos más o menos detallados:

- El modo de implementación es mediante una **librería.**
- Se apila el parámetro y se hacen sucesivos pushs.
- **Ahí la librería pone el código a la llamada en un registro**. Aquel que programa tiene la responsabilidad de saber el número de registro pues eso depende del kernel, ya que **el registro corresponde a la instrucción.**
- Hay una excepción, por lo tanto comienza la rutina de interrupción y se pasa a modo Kernel. (*Recordar que las*

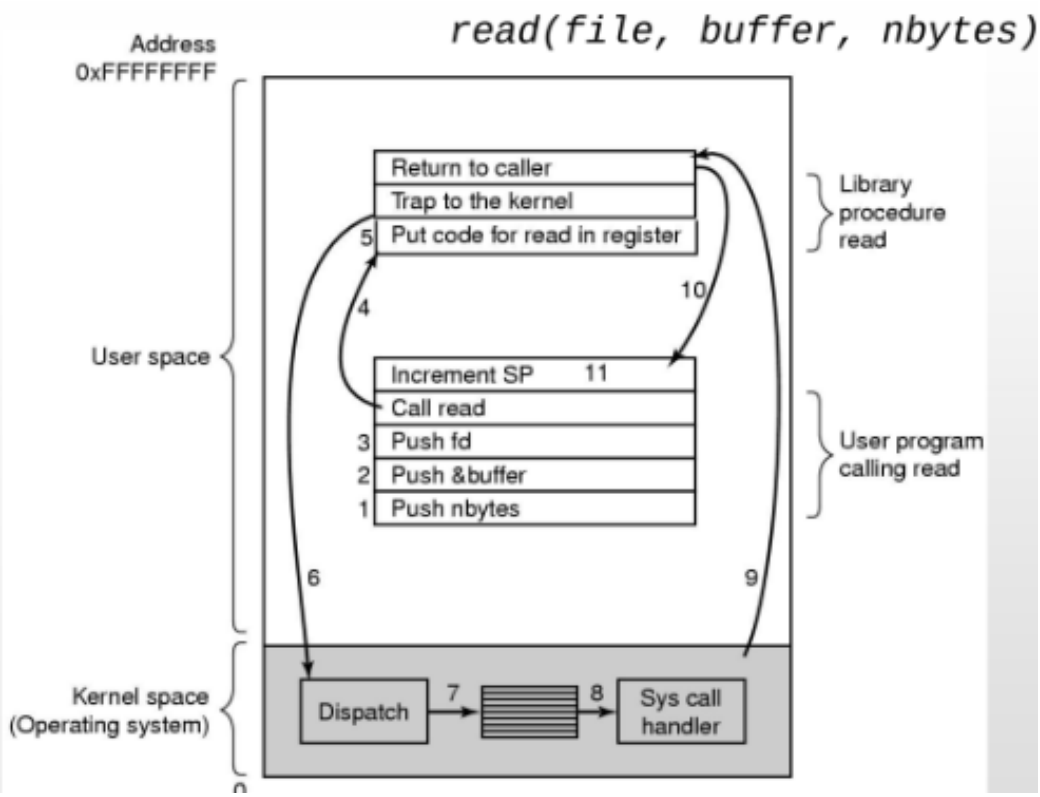
los programas puedan programarse independientemente de la arquitectura.

    Como detalle del detalle(?) las cosas no funcionan igual en *linux y en windows* porque las llamadas a sistema **se llaman diferentes** y algunas ni siquiera tienen equivalencias.

*excepciones son la única manera de **llamar** al modo Kernel*)

- El que ejecuta el `in/out` es el **Kernel**, ya que son instrucciones privilegiadas.

Hay **dos pilas**, una para el modo usuario y otra para el modo Kernel, que está protegida.



Todo en linux es un archivo, por lo tanto **todo programa en linux tiene 3 archivos que le corresponde:**

- Un archivo para la entrada standard,
- Un archivo para la salida standar,
- Y un archivo para los errores standar.

Categorías

- **Control de Procesos**

```
pid = fork() #create a child process identical to the parent
pid = waitpid (pid , &statloc, options) # wait for a child to terminate
s = execve (name ,argv , enviromp) # replace a process' core image
exit (status)  # terminate a process exceution and return status
```

- **Manejo de archivos**

```
fd = open (file , how , ...)  #open a file for reading, writing or both
s = close (fd) # close an open file
n = read (fd , buffer , nbytes) #  read data from a file into bufffer
n = write (fd , buffer , nbytes) # write data from a buffer into a file
position = lseek(fd , offset , whence)# move the file pointer
s = stat (name , &buf)#get a file's status information
```

- Manejo de dispositivos
- Mantenimiento de información del sistema
- Comunicaciones

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Ejemplos de System Call

Recordar: Las cosas no funcionan igual en *linux y en windows* porque las llamadas a sistema **se llaman diferentes** y algunas ni siquiera tienen equivalencias.

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Programar una SysCall

Para programar el clásico “Hello World” se necesitan mínimo hacer 2 llamadas al sistema:

1. Escribir en pantalla un mensaje de `syscall write`
2. Terminar la ejecución de un proceso `syscall exit`

Para obtener información sobre estas llamadas se puede usar los manuales del sistema. El comando man permite acceder a distintos tipos de documentación, en particular a información referida a systemcalls. Estos manuales permiten saber cuáles son los parámetros.

```
write (man 2 write)
exit (man exit)
```

Para indicarle al sistema operativo lo que queremos hacer (write o exit) es necesario saber cuál es el **número asociado que tiene cada una de las Syscall** (que depende de la arquitectura)

En arquitectura de 32 bit

`write` → syscall número 4

`exit` → syscall número 1

`EAX` → lleva el número de syscall que se desea ejecutar

En arquitectura de 64 bit

`write` → syscall número 1

`exit` → syscall número 60

`EAX` → lleva el número de syscall que se desea ejecutar

`EBX` → lleva el primer parámetro

`ECX` → lleva el segundo parámetro

`int 80h` → Inicia la systemcall.

```
start:
;sys_write (stadout , message, lengh)
mov eax , 4 ; sys_write syscall
mov ebx , 1
mov ecx , message ; message address
mov edx , 14 ; el largo del string
int 80h
;sys_exit (return_code)
mov eax , 1 ;sys_exit syscall
mov ebx, 0
int 80h

section.data
  message: db "Hello, world!",0x0A
;message and newline
```

`RDI` → lleva el primer parámetro

`RSI` → lleva el segundo parámetro

`syscall` → Inicia la systemcall.

```
;sys_write (stadout , message, lengh)
mov rax , 1 ; sys_write syscall
mov rdi , 1
mov rsi, message ; message address
mov rdx , length ;el largo del string
syscall
;sys_exit (return_code)
mov rax, 60 ;sys_exit syscall
mov rdi , 0
syscall


section.data
  message: db "Hello, world!",0x0A
;message and newline
  length: equ 14;
```

En resumen:

Los manuales del sistema indican los parámetros necesarios para activar una **system call**. Dependiendo de la arquitectura va a cambiar el número de la syscall utilizado para una función determinado y la manera de pasar los parámetros al kernel.

- Los procesadores 32 y 64bit usan un esquema de registros diferentes y usan instrucciones diferentes, inclusive para activar las systemcalls.
  - `int 80h` para los de 32 bits.
  - `syscall` para los de 64bits.

Tipos de Kernel

Como sabemos, el kernel es **la primer capa entre el hardware y el resto.**

Kernel monolítico

El kernel monolítico es una visión **más clásica**, toda la funcionalidad del sistema operativo está en el núcleo. Cuándo se invoca al kernel, se llama al mismo, se resuelve la situación por la cuál se llamó y se vuelve al modo usuario.

Microkernel

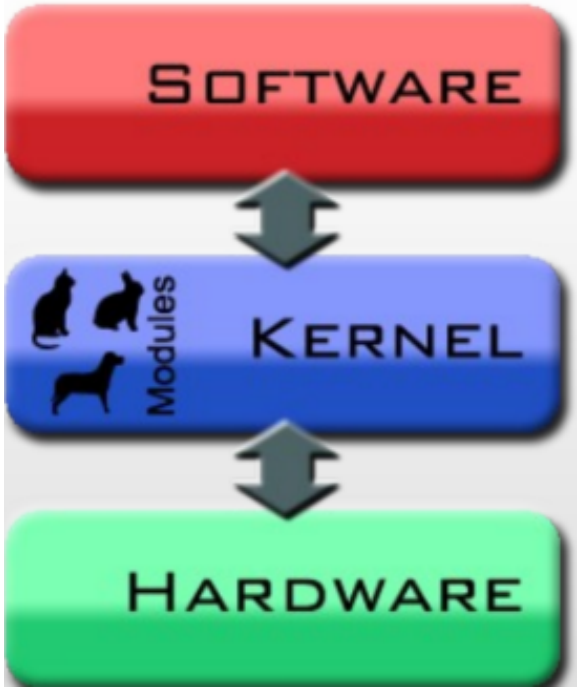
Se trata de hacer que el **kernel sea lo más chico posible,** dejando en modo usuario servicios *que se ejecutan como apoyo al kernel.*

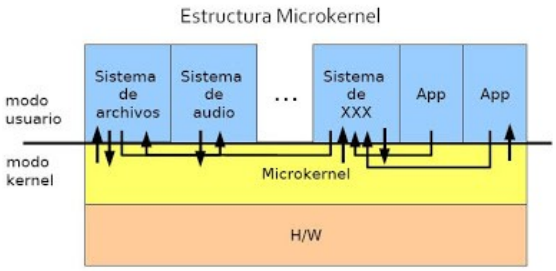
- Es seguro y la computadora trata de estar lo menos posible en modo kernel.

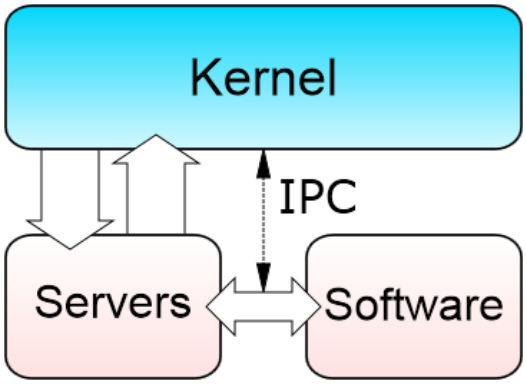
Es más rápido y eficiente, ya que reduce los cambios de modo.

En resumen: introduce más errores pero es más eficiente.

- Hay menos librerias corriendo en modo kernel.
- Menos chances de bugs en modo kernel.







Monolítico vs Microkernel