kernel II - System Calls y Módulos

Explicación de práctica 3

Sistemas Operativos

Facultad de Informática
Universidad Nacional de La Plata

2019

Agenda

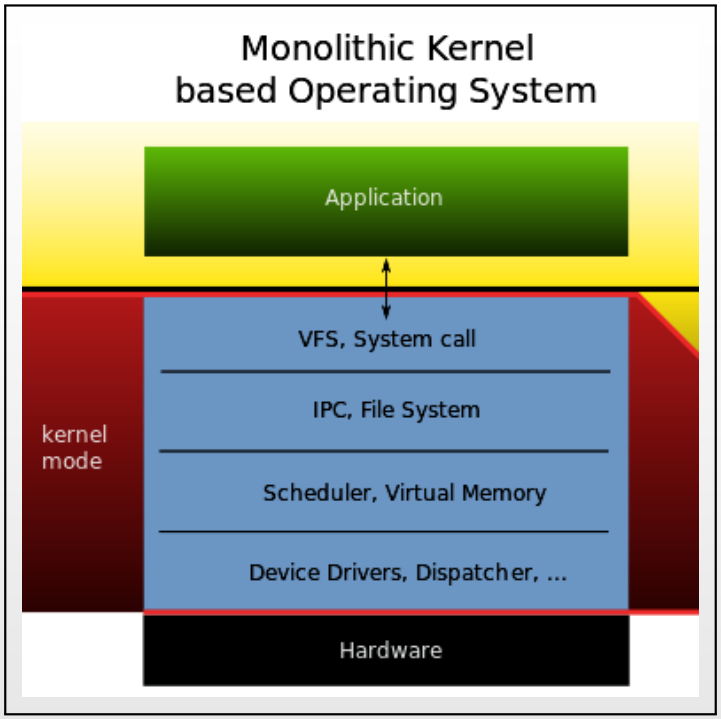
1. Kernel
2. System Calls
3. Módulos

Agenda

1. Kernel
2. System Calls
3. Módulos

- 2,4 millones de lineas de código(y contando...)
- El 70 % del código son drivers
- Windows: mas del doble de lineas!
- Tasa de errores en drivers con respecto al Kernel: 7 veces mas
  - Fuente:http://pdos.csail.mit.edu/6.097/readings/osbugs.pdf
- Comparación con la aeronaútica:
  - Aislamiento de fallas
  - Un problema en el toilet no afecta al sistema de navegación!

Kernel Monolítico - Memoria compartida

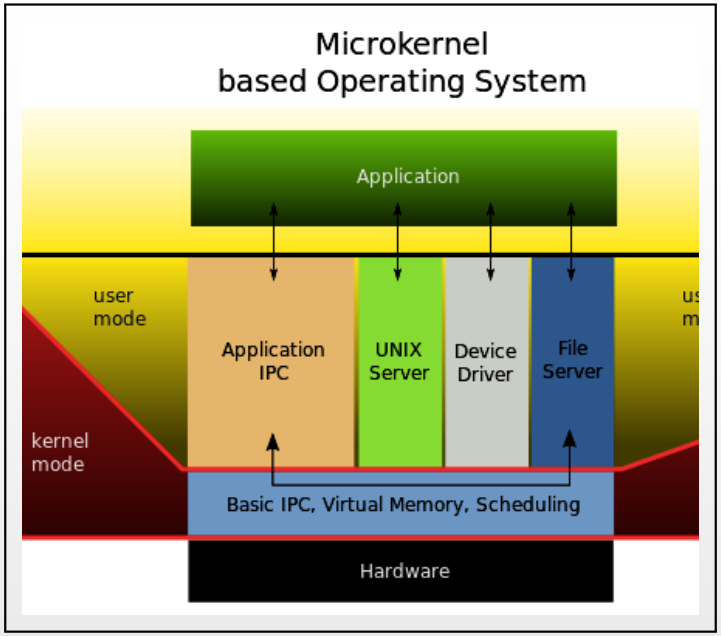


- Componentes linkeados en un mismo binario en memoria.
- Memoria Compartida(¡sincronización!)
- Scheduler, Drivers, Memory Manager, etc. mismo espacio en memoria

Kernel Monolítico - Operating System Crash

- ¿Que sucede si hay un error en un driver?
  - Windows: BSD(blue screen of death).
  - Unix: Kernel Panic.
- Un único gran componente linkeado en un mismo espacio de direcciones implica un módulo muy grande y complejo.
- La razón de tener un único gran componente linkeado en un mismo espacio de direcciones se debe a cuestiones de performance por limitaciones de hardware tomadas hace mucho tiempo.
- ¿Hoy en dia la decisión seria la misma?

Microkernel - Procesos de usuario



- Componentes del kernel en distintos procesos de USUARIO
- Kernel minimalista(comunicación con el hard e IPC)
- IPC (¡Computación distribuida!)
  - Scheduler, Drivers, Memory Manager en distintos procesos de Usuario
  - IPC es parte del Kernel(muchos cambios de modo)

- Pros
  - Facilidad para desarrollar servicios del SO.
  - Los bugs existen y existirán siempre, entonces deben ser aislados.
  - Kernel muy pequeño, entonces mas fácil de entender, actualizar y optimizar.
- Contras
  - Baja performance
  - La computación distribuida es inheremente mas compleja que la computación por memoria compartida
  - No fue adoptado masivamente por la industria(ej. Minix)

Torvalds – Tanembaum debate

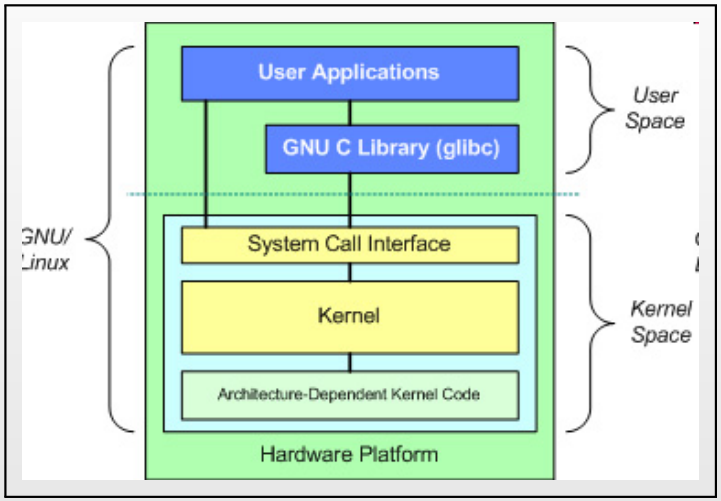
vs

http://en.wikipedia.org/wiki/Tanenbaum

Agenda

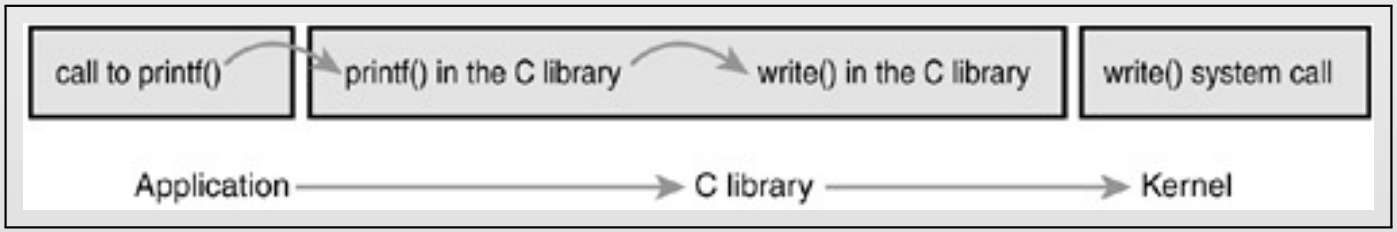
1. Kernel
2. System Calls
3. Módulos

- Los SOs proveen un conjunto de interfaces mediante las cuales un proceso que corre en espacio de usuario accede a un conjunto de funciones comunes



- En UNIX la API principal que provee estos servicios es libc:
  - Es la API principal del SO
  - Provee las librerías estandar de C
  - *Es una Interface entre aplicaciones de usuario y las System Calls(System Call Wrappers).*

- La funcionalidad anterior está definida por el estandar POSIX
- Su próposito es proveer una interfaz común para lograr portabilidad
- En el caso de las System Calls, el desarrollador generalmente interactúa con la *API* y NO directamente con el Kernel
- En UNIX por lo general cada función de la API se corresponde con una System Call



- Son llamados al kernel para ejecutar una función específica que controla un dispositivo o ejecuta una instrucción privilegiada
- Su próposito es proveer una interfaz común para lograr portabilidad
- Su funcionalidad se ejecuta en modo Kernel pero en contexto del proceso
- Recordar
  - Cambio de Modo
  - ¿Como se pasa de modo usuario a modo Kernel?

- Utilizando los wrappers de glibc
  - `int rc = chmod(``/etc/passwd'' , 0444);`
- Invocación explícita utilizando la System Call syscall provista por glibc
  - Definida en la libreria unistd.h
    - `long int syscall (long int sysno, ...)`
  - Ejemplo utilizando syscall:
    - `rc = syscall(SYS_chmod, ``/etc/passwd'', 0444);`

```c
#include <stdlib.h>
#include <sys/syscall.h>
#include <sys/time.h>
#include <unistd.h>
#define SYS_gettimeofday 78

void main(void){
    struct timeval tv;
    /* usando el wrapper de glibc */
    gettimeofday(&tv, NULL);
    /* Invocación explícita del  system call */
    syscall(SYS_gettimeofday, &tv, NULL);
}
```

- La manera en que una system call es llevada a cabo dependerá del procesador.
  - `Los procesadores x86 se basan en el mecanismo de interrupciones.`
- Interrupción enmascarable int 0x80 en Linux.
  - `Se usa el vector 0x80 para transferir el control al kernel. Este vector de interrupción esta inicializado durante el startup del sistema [ref].`
- Una librería de espacio de usuario(libc) carga el índice de la system call y sus argumentos, la interrupción enmascarable por software 0x80 es invocada, la cual resulta en el cambio de modo.
- A través de la estructura sys_call_table y el registro eax como índice se determina que *handler function* invocar.

Consideremos el siguiente caso:

```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#define sys_getpid 20
int main(void) {
  long ID = syscall(SYS_getpid);
  printf ("El pId del proceso es:\n", ID);
}
```

El compilador generará algo parecido a:

```
...
movzwl 12(%esp),%eax
movl  %eax,4(%esp)
movl $20,%eax
movl 4(%esp),%ebx
int $0x80
movl  %eax,%edx
testl  %edx,%edx
jge L2
negl  %edx
movl  %edx,_errno
...
```

- Debemos identificar nuestra syscall por un número único(syscall number).
- Agregamos una entrada a la syscall table.
- Debemos considerar el sys call number.
- Ver que el código fuente organizado por arquitectura.
- Respetar las convenciones del Kernel(ej. prefijo sys_).

/usr/src/linux-⟨X⟩/arch/x86/entry/syscalls/syscall_32.tbl

⟨number⟩ ⟨abi⟩ ⟨name⟩ ⟨entry point⟩

...

351 i386 newcall sys_newcall

- Debemos declarar nuestra system call (prototipo de de la syscall)
- Los parámetros a system calls deben ser realizados por medio del stack
- Informamos de esto al compilador mediante la macro asmlinkage
  - asmlinkage instruye al compilador a pasar parámetros por stack y no por ejemplo en registros

/usr/src/linux-4.15/include/linux/syscalls.h

```
asmlinkage long sys_newcall(int i);
```

- Debemos definir nuestra syscall en algún punto del árbol de fuentes.
- Podemos utilizar algún archivo existente.
- Podemos incluir un nuevo archivo y su correspondiente Makefile.
  - Ver apuntes adjuntos

En algún archivo ya incluido en los fuentes del Kernel...

```
asmlinkage int sys_newcall(int a) {
printk("calling newcall... ");
return a+1;
}
```

- ¿printk?, ¿porque no printf?

- **Recompilar el Kernel!**
  - Idem Práctica 2

Invocando explícitamente nuestra System Call

```c
#include <linux/unistd.h>
#include <stdio.h>
#define sys_newcall 351
int main(void) {
  int i = syscall(sys_newcall,1);
  printf ("El resultado es: %d\n", i);
}
```

- **Reporta las system calls que realiza cualquier programa**
- man strace
- Opción útil -f (tiene en cuenta procesos hijos)

strace a.out (hola mundo)

```
execve("./syscall.o", ["./syscall.o"], [/* 19 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ—PROT_WRITE,
MAP_PRIVATE—MAP_ANONYMOUS, -1, 0) = 0x7f12ea552000
...
write(1, "hola mundo!", 11hola mundo!) = 11
...
```

- Realizar un parche que contenga los cambios asociados a la siguiente *system call*:
  - Simplemente debe imprimir el texto "Hello world" a través de la función *printk*
  - Se debe hacer sobre la versión utilizada en la práctica anterior
- Tips:
  - Utilizar explicación y práctica para guiarse
  - `http://lxr.free-electrons.com/`

DESAFIO - Implementación

- ¿Dónde incluir la implementación?:
  - Dos opciones:
    - Incluir el código en un fichero existente
    - Agregar un nuevo fichero ⟩ modificar *Makefile* existente
- Crearemos un nuevo fichero *.c* bajo el directorio ⟨*source*⟩*/kernel*

```
#include <linux/syscalls.h> /* For SYSCALL_DEFINEi() */
#include <linux/kernel.h>

SYSCALL_DEFINE0(mysyscall)
{
  printk(KERN_DEBUG "Hello world\n");
  return 0;
}
```

DESAFIO - Modificar Makefile

```
#
# Makefile for the linux kernel.
#

obj-y = ...
        async.o range.o groups.o smpboot.o mysyscall.o
...
```

```c
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>
#include <stdio.h>

#define __NR_MYSYSCALL -num-syscall-

int main() {
  printf("Invocando system call...\n");
  return syscall(__NR_MYSYSCALL);
}
```

```
$ gcc mysyscall.c -o mysyscall
$ ./mysyscall
```

```
## Crear parche
$ diff -urpN linux-4.4.6 linux-4.4.6-modificado > patch
    -4.4.6-so2016
```

¡No olvidar ejecutar!:

```
$ make mrproper
```

Agenda

¿Que son los Módulos del Kernel?

- "Pedazos de código" que pueden ser cargados y descargados bajo demanda
- Extienden la funcionalidad del kernel
- Sin ellos el kernel sería 100 % monolítico
  - Monolítico "hibrido"
- No recompilar ni rebootear el kernel

Comandos relacionados

- `lsmod`
  - Lista los módulos cargados (es equivalente a cat /proc/modules)
- `rmmod`
  - Descarga uno o más módulos
- `modinfo`
  - Muestra información sobre el módulo
- `insmod`
  - Trata de cargar el módulo especificado
- `depmod`
  - Permite calcular las dependencias de un módulo
  - depmod -a escribe las dependencias en el archivo /lib/modules/version/modules.emp
- `modprobe`
  - Emplea la información generada por depmod e información de /etc/modules.conf para cargar el módulo especificado.

¿Como creamos un módulo?

- Debemos proveer dos funciones:
  - Inicialización: Ejecutada cuando ejecutamos insmod.
  - Descarga: Ejecutada cuando ejecutamos rmmod.

```
#include <linux/module.h>
#include <linux/kernel.h>
int init_module(void){
        printk(KERN_INFO "Hello world 1.\n");
return 0;
}

void cleanup_module(void){
        printk(KERN_INFO "Goodbye world 1.\n")
            ;
}
```

¿Como creamos un módulo?(cont.)

- También podemos indicarle otras funciones.
  - `module_init()`
  - `module_exit()`

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
static int hello_init(void){
        printk(KERN_INFO "Hello! \n");
return 0;}

static void hello_exit(void){
printk(KERN_INFO "Goodbye! \n");}

module_init(hello_init);
module_exit(hello_exit);
```

- Se definen con la macro `module_param`
  - `name`: Es el nombre del parámetro expuesto al usuario y de la variable que contiene el parámetro en nuestro módulo
  - `type`: byte, short, ushort, int, uint, long, ulong ,charp ,bool, invbool
  - `perm`: Especifica los permisos al archivo correspondiente al módulo el sysfs

```
static char *user_name = "";
module_param(user_name, charp, 0);
MODULE_PARM_DESC(user_name,"user name");
```

Al cargar el módulo indicamos el valor del parámetro:

```
$ sudo insmod hello.ko user_name=colo
```

- Construimos el Makefile

```
obj-m += hello.o
all:
make -C /lib/modules/$(shell uname -r)/build M=$(pwd)
    modules
clean:
make -C /lib/modules/$(shell uname -r)/build M=$(pwd)
    clean
```

- Compilamos

```
    $ make
```

- Entendemos por dispositivo a cualquier dispositivo de hard: discos, memoria, mouse, etc
- Cada operación sobre un dispositivo es llevada por código específico para el dispositivo
- Este código se denomina “driver” y se implementa como un módulo
- Cada dispositivo de hardware es un archivo (abstracción)
- Ejemplo: /dev/hda
  - En realidad no es un archivo.
  - Si leemos/escribimos desde él lo hacemos sobre datos “crudos” del disco (bulk data).
- Accedemos a estos archivos mediante operaciones básicas (espacio del kernel).
  - read, write: escribir y recuperar bulk data
  - ioctl: configurar el dispositivo

¿Que son los Dispositivos? (Cont.)

- Podemos clasificar el hard en varios tipos.
  - Dispositivos de acceso aleatorio(ej. discos).
  - Dispositivos seriales(ej. Mouse, sonido,etc).
- Acorde a esto los drivers se clasifican en:
  - Drivers de bloques: son un grupo de bloques de datos persistentes. Leemos y escribimos de a bloques, generalmente de 1024 bytes.
  - Drivers de carácter: Se accede de a 1 byte a la vez y 1 byte sólo puede ser leído por única vez.
  - Drivers de red: tarjetas ethernet, WIFI, etc.

- Major y Minor device number.
  - Los dispositivos se dividen en números llamados major device number. Ej: los discos SCSI tienen el major number 8.
  - Cada dispositivo tiene su minor device number. Ejemplo: /dev/sda major number 8 y minor number 0
- Con el major y el minor number el kernel identifica un dispositivo.
- kernel_code/linux/Documentation/devices.txt

```
# ls -l /dev/hda[1-3]
brw-rw---- 1 root disk 3, 1 Abr 9 15:24 /dev/hda1
brw-rw---- 1 root disk 3, 2 Abr 9 15:24 /dev/hda2
brw-rw---- 1 root disk 3, 3 Abr 9 15:24 /dev/hda3
```

- Representación de los dispositivos(device files)
- Por convención están en el /dev
- Se crean mediante el comando mknod.

```
mknod[- m<mode >] file[b|c ]major  minor
```

- b o c: según se trate de dispositivos de caracter o de bloque.
- El minor y el major number lo obtenemos de kernel_code/linux/Documentation/devices.txt

- Necesitamos decirle al kernel:
  - Que hacer cuando se escribe al device file.
  - Que hacer cuando se lee desde el device file..
- Todo esto lo hacemos en un módulo.
- La struct file_operations:
  - Sirve para decirle al kernel como leer y/o escribir al dispositivo.
  - Cada variable posee un puntero a las funciones que implementan las operaciones sobre el dispositivo.

- Mediante la struct file_operations especifico que funciones leen/escriben al dispositivo.

```
struct file_operations my_driver_fops = {
read: myDriver_read,
write: myDriver_write,
open: myDriver_open,
release: mydriver_release};
```

- En la función module_init registro mi driver.

```
register_chrdev(major_number, "myDriver", &
   my_driver_fops);
```

- En la función module_exit desregistro mi driver.

```
unregister_chrdev(major_number, "myDriver");
```

¿Como creamos un driver?(Cont..)

- Operaciones sobre el dispositivo
  - Escritura del archivo de dispositivo

```
echo "hi" > /dev/myDeviceFile
```

```
ssize_t myDriver_write(struct file *filp, char
    *buf,size_t count, loff_t *f_pos);
```

- Lectura del archivo de dispositivo

```
cat /dev/myDeviceFile
```

```
ssize_t myDriver_read(struct file *filp, char
   *buf,size_t count, loff_t *f_pos)
```

¿Como creamos un driver? (cont.)

- Parámetros de las funciones funciones:
  - `struct file`: Estructura del kernel que representa un archivo abierto.
  - `char *buf`: El dato a leer o a escribir desde/hacia el dispositivo(espacio de usuario)
  - `size_t count`: La longitud de a leer de `buf`.
  - `loff_t *f_pos`: La posición actual en el archivo

Ejemplo de Módulos - Implementación

Sección "Desarrollando un Driver" de la Práctica 3(Ejercicio guiado)

Mecanismo alternativo: /proc

- El /proc es un sistema de ficheros virtual
  - No ocupa espacio en disco
- Al leer o escribir en un archivo de este sistema del /proc se ejecuta una función del kernel que devuelve o recibe los datos
  - Lectura: read callback
  - Escritura: write callback
- En Linux, /proc muestra información de los procesos, uso de memoria, módulos, hardware, ...

Mecanismo de interacción entre el usuario y el kernel

Los módulos pueden crear entradas /proc para interactuar con el usuario

- Crear un módulo del kernel con funciones `init_module()` y `cleanup_module()`
- Definir variable global de tipo struct `file_operations`
  - Especifica qué operaciones en el /proc se implementan y su asociación con las funciones del módulo

```
struct file_operations fops = {
.read = myproc_read,
.write = myproc_write,
};
```

- En la función de inicialización, crear la entrada del /proc con la función `proc_create()`:

```
struct proc_dir_entry *proc_create(const
    char *name,umode_t mode,struct
    proc_dir_entry *parent,const struct
    file_operations *ops);
```

- Parámetros:
  - `name`: Nombre de la entrada
  - `mode`: Máscara octal de permisos (p.ej., 0666)
  - `parent`: Puntero al directorio padre (NULL → directorio raíz)
  - `ops`: Puntero a la estructura que define las operaciones
- En la función cleanup del módulo, eliminar la entrada /proc creada

```
void remove_proc_entry(const char *name,
    struct proc_dir_entry *parent);
```

/proc

Módulo que se comunica con el espacio de usuario a través de una entrada del /proc(analizar el código)

Utilizar:

1. Archivo .c
2. Makefile

Lo compilamos:

```
$ sudo make
$ insmod modulo_so.ko
```

Escribimos y leemos al /proc/module_so y analizamos el log

```
$ cat miArchivo > /proc/module_so
$ dmesg
$ cat /proc/module_so
$ dmesg
```

EXPERIMENTO - Explicación

- Implementar un módulo que encienda los leds del teclado en base a un número escrito en una entrada del */proc*.
- Cadena puede incluir los caracteres '1','2' y '3':
  - Si aparece el '1' → encender Num Lock (`bit 1` encendido)
  - Si aparece el '2' → encender Caps Lock (`bit 2` encendido)
  - Si aparece el '3' → encender Scroll Lock (`bit 0` encendido)
- Tips:
  - KBD Driver
  - Privilegios de *root*
  - En la `VM` hay que asociar el dispositivo `USB`

- Utilizar:
  - Fichero .c
  - Makefile

```
#Compilamos el módulo
$ make
```

```
#Insertar el módulo
$ sudo insmod procleds.ko

#Encender leds
$ sudo echo 1 > /proc/leds
$ sudo echo 123 > /proc/leds

#Eliminar módulo
$ sudo rmmod procleds
```

¿Preguntas?