Kernel Linux 2 - System calls y Módulos

Explicación de práctica 2

Sistemas Operativos

Facultad de Informática
Universidad Nacional de La Plata

2024

Agenda

Contenido:

- Kernel
- System calls
- Módulos

Agenda

Contenido:

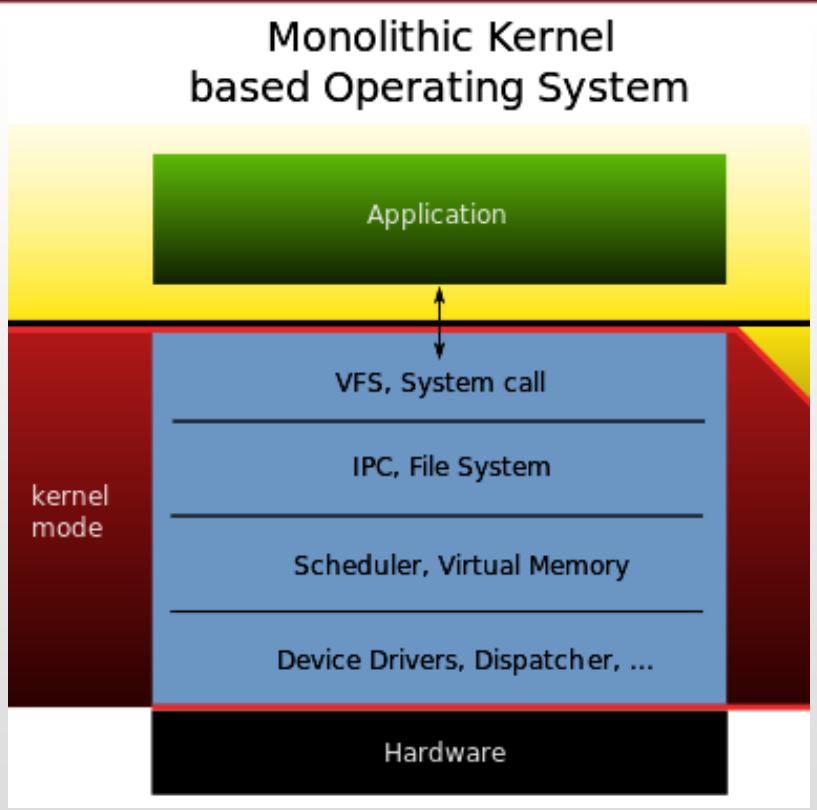
- Kernel
- System calls
- Módulos

Kernel Linux

- 28 millones de líneas de código descontando comentarios y líneas vacías
- El 68.8 % del código son drivers
- Difícil de comparar con Windows por falta de datos oficiales y porque en Linux la mayoría de los drivers son parte del kernel
- Tasa de errores en drivers con respecto al Kernel: 7 veces más
  - Fuente:http://pdos.csail.mit.edu/6.097/readings/osbugs.pdf (estudio en versiones entre 1.1.3 y 2.4.0)

Kernel Monolítico - Memoria compartida

- Componentes linkeados en un único binario en memoria.
- Memoria compartida (es importante la sincronización)
- Scheduler, Drivers, Memory Manager, etc en un mismo espacio de memoria.

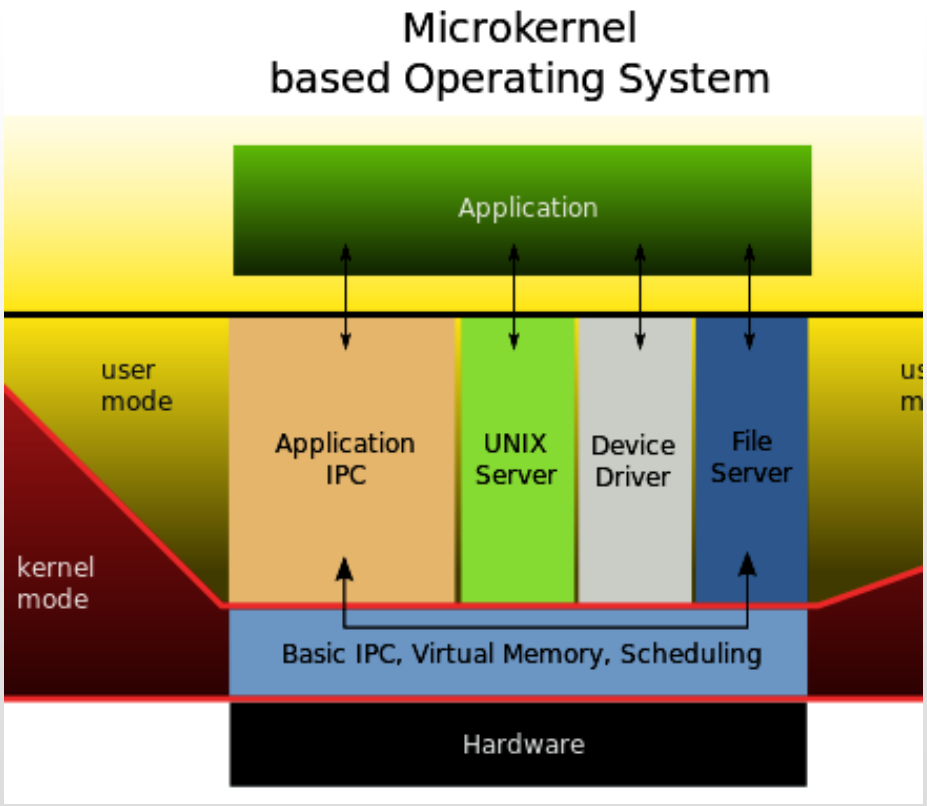


Kernel Monolítico - Operating System Crash

- ¿Qué 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 decisiones vinculadas a la performance tomadas por limitaciones de hardware tomadas hace mucho tiempo.
- ¿Hoy en día la decisión sería 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)



Microkernel

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

Torvalds – Tanembaum debate

- Post en comp.os.minix (29/01/1992): "LINUX is obsolete"
- El kernel Linux está muy acoplado a los procesadores x86 (poca portabilidad)
- Escribir un kernel monolítico en 1991 es un “retroceso gigante a los años 70”

- MINIX tiene defectos de diseño inherentes (como la falta de multithreading)
- El diseño de microkernel es superior desde un punto de vista “teórico y estético”
- Diseñó Linux específicamente para el Intel 80386 en parte de forma intencional como un ejercicio para él mismo por eso su API es más simple y eso lo hace más portable que MINIX

https://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate

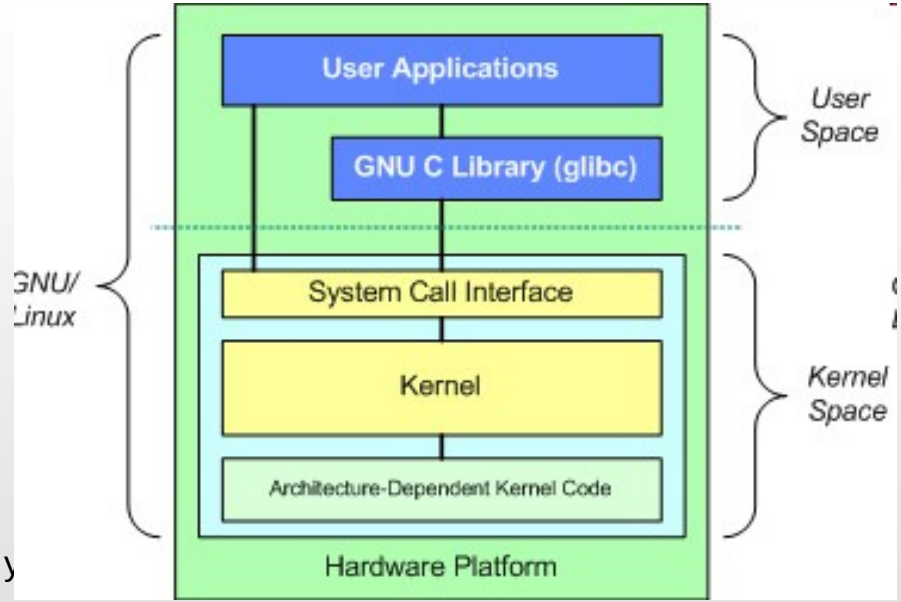
Agenda

Contenido:

- Kernel
- System calls
- Módulos

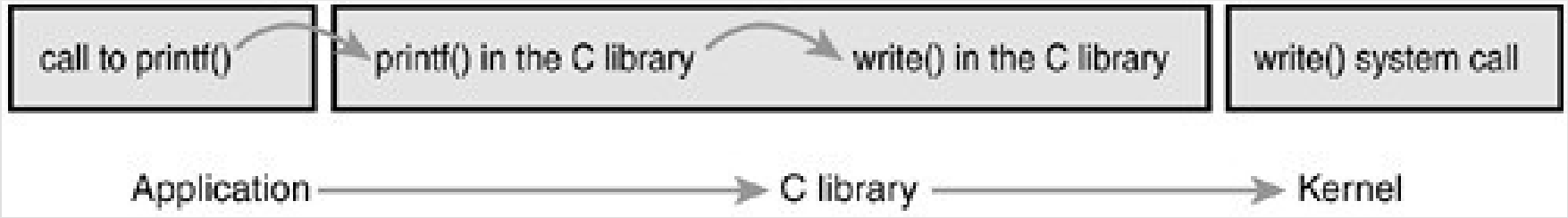
API del Sistema Operativo

- 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 estándar de C
  - Es una interfaz entre aplicaciones de usuario y las System Calls(System Call Wrappers).



POSIX APIs

- Gran parte de la funcionalidad de la libc y de las system calls está definida por el estándar POSIX
  - Su propósito 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



System Calls - Repaso

- Son llamados al kernel para ejecutar una función específica que controla un dispositivo o ejecuta una instrucción privilegiada
- Su propósito 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?

Invocando System Calls

- Utilizando los **wrappers** específicos de glibc
  - int rc = chmod(“/etc/passwd” , 0444);
- Invocación explícita utilizando la **el wrapper** syscall provisto por glibc
  - Declarada en el archivo de headers de C: unistd.h
    - long int syscall (long int sysno, ...)
- Ejemplo utilizando **el wrapper** syscall:
  - rc = syscall(SYS_chmod, “/etc/passwd”, 0444);

Ejemplo

```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 wrapper general */
        syscall(SYS_gettimeofday, &tv, NULL);
}
```

Interrupciones y System Calls

- Invocación dependiente de la arquitectura de la CPU:
  - Intel x86 usa el mecanismo de interrupciones con la instrucción int 0x80.
  - AMD64/x86_64 define la instrucción syscall.
  - En ARM se usa la instrucción svc (Supervisor Call).
- Una librería de espacio de usuario (libc) nos provee una abstracción:
  - Sarga el índice de la system call y sus argumentos en los registros correspondientes
  - Invoca la instrucción necesaria para invocar la syscall.
  - Recupera el valor retornado por la syscall.
- A través de la estructura syscall table y el índice se determina que handler function invocar.

Consideremos el siguiente caso:

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

int main(void) {
      int p_id= syscall(SYS_getpid);
      printf("El pid es %d\n",p_id);
      return 0;
}
```

System Calls e Interrupciones (cont.)

El compilador (en AMD64) generará algo parecido a:

```
...
mov     $39, %rdi
mov     $0, %rax
call    syscall@PLT
mov     %rax,-0x8(%rbp)
...
```

System Calls e Interrupciones (cont.)

El código de la función syscall en la libc es:

```
...
<syscall>:
endbr64
mov    %rdi,%rax
mov    %rsi,%rdi
mov    %rdx,%rsi
mov    %rcx,%rdx
mov    %r8,%r10
mov    %r9,%r8
mov    0x8(%rsp),%r9
syscall
...
```

Desarrollando una System Calls en GNU Linux

- 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_ y __x64_sys_).

/usr/src/linux-<X>/arch/x86/entry/syscalls/syscall 64.tbl

```
number  abi      name     entry point
...
351     common   newcall  __x64_sys_newcall
```

Desarrollando una System Calls en GNU Linux

- Desarrollando una System Calls en GNU Linux
- 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 cómo pasar los parámetros:
    - en x86 (32 bits) por stack
    - en otras arquitecturas pueden ir por registros

/usr/src/linux-<x>/include/linux/syscalls.h

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

Desarrollando una System Calls en GNU Linux

- Debemos definir nuestra syscall en algún punto del árbol de archivos del kernel.
- Podemos utilizar algún archivo existente.
- Podemos incluir un nuevo archivo y su correspondiente Makefile.

Manualmente:

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

De la forma recomendada:

```
SYSCALL_DEFINE1(newcall, int, a)
{
        printk("calling newcall... ");
        return a+1;
}
```

- ¿printk? ¿Por qué no printf?

Desarrollando una System Calls en GNU Linux

- ¡Recompilar el Kernel!
  - Idem práctica 1

Invocando explícitamente nuestra System Call

```c
#include <syscalls.h>
#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 invocadas por un proceso**
- man strace

strace a.out > /dev/null

```
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!”, 11) = 11
...
```

- "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

- `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.

- 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");
}
```

- 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

- 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");
```

- 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)
```

- 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

- 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);
```

¿Preguntas?