Kernel Linux 2 - System calls y Módulos

Explicación de práctica 2

Sistemas Operativos

Facultad de Informática
Universidad Nacional de La Plata

2025

Agenda

Contenido:

- Kernel
- System calls

Agenda

Contenido:

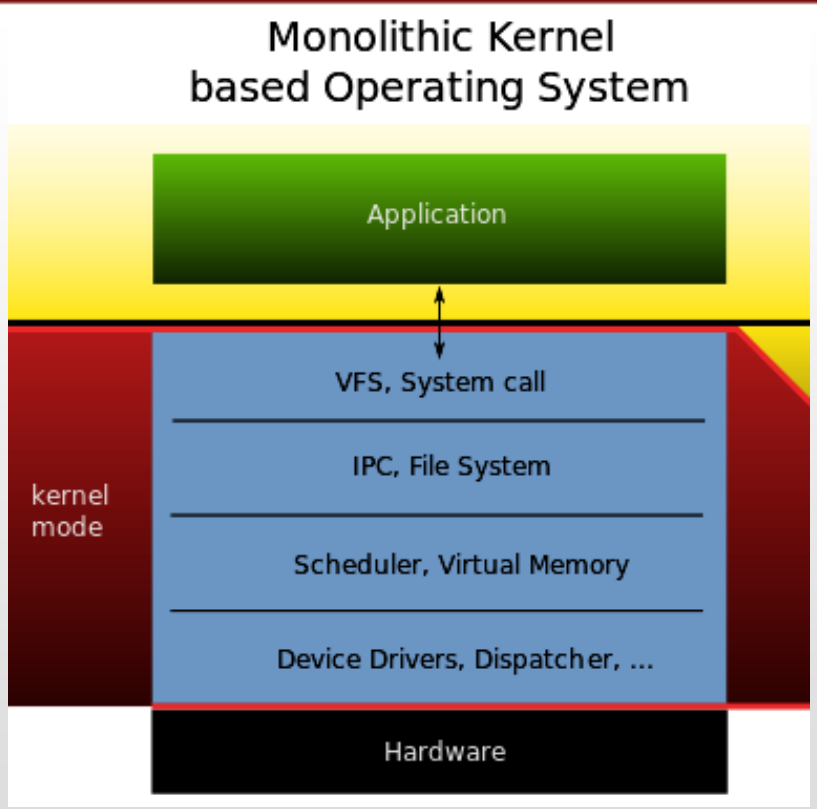
- Kernel
- System calls
- Módulos

Kernel Linux

- https://docs.kernel.org/index.html
- ~ 40 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.

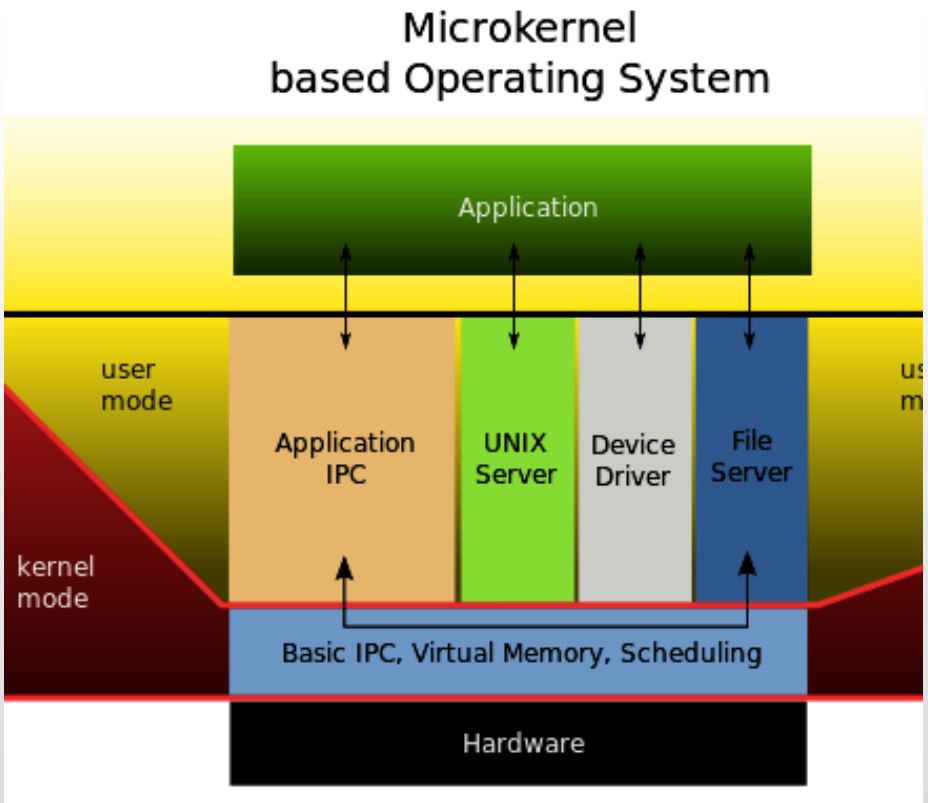


Kernel Monolítico - Operating System Crash

- ¿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

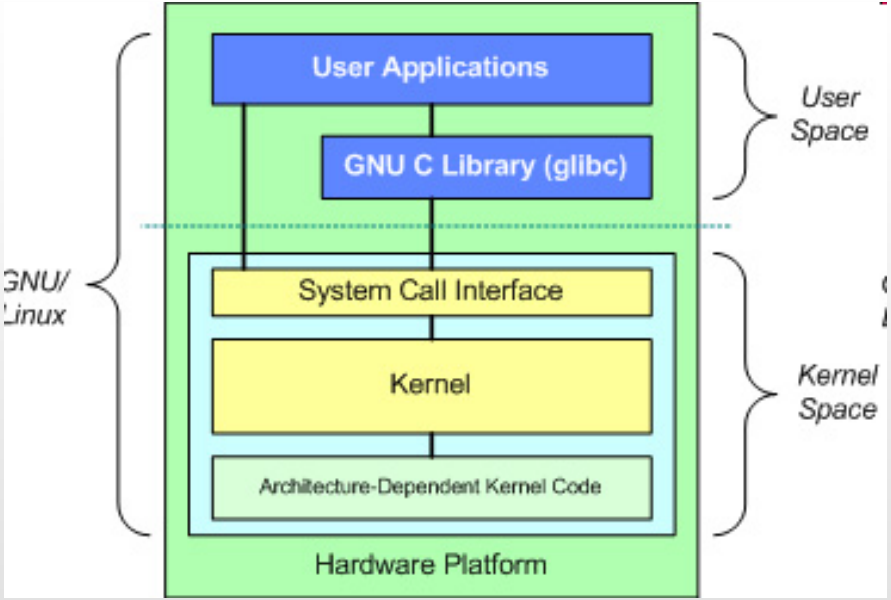
Agenda

Contenido:

- Kernel
- System calls
- Módulos

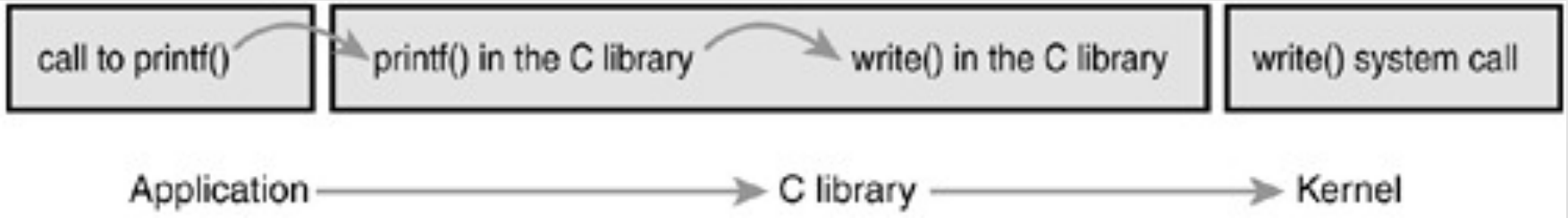
API del Sistema Operativo

- 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 de glibc
  - int rc = chmod(“/etc/passwd” , 0444);
- Invocación explícita utilizando la System Call syscall provista por glibc
  - Declarada en el archivo de headers de C: unistd.h
    - long int syscall (long int sysno, ...)
- Ejemplo utilizando syscall:
  - rc = syscall(SYS_chmod, “/etc/passwd”, 0444);

Ejemplo

```
#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);
}
```

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:

```
#include <stdio.h>
#include <sys/types.h>
#include <syscall.h> // SYS_getpid = 39
#include <unistd.h>
int main(void) {
        long id = syscall(SYS_getpid);
        printf ("El pid del proceso es:\n", id);

}
```

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

```
#include <syscalls.h>
#include <linux/unistd.h>
#include <stdio.h>
#define sys_newcall 351

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
...
```