Introducción al kernel Linux

Matías Zabaljáuregui
matiasz@info.unlp.edu.ar

- Repaso del modelo kernel / proceso
- Contextos de ejecución
- Kernels Unix y Linux
- Reentrancia y Sincronización
- Diferencias con el user space

Repaso:
¿Que es el Kernel?

- El kernel como máquina ampliada o maquina virtual: proveé un entorno de ejecución a las aplicaciones.
- El kernel como manejador de recursos: interactúa con la plataforma de hardware.

El Modelo Proceso / Kernel

- Los procesos son entidades dinámicas, programas en ejecución que consumen recursos.
- El kernel es un administrador de procesos que crea el entorno necesario para que los procesos puedan ejecutarse.
- El modelo asume que los procesos que requieren un servicio del kernel utilizan construcciones específicas llamadas **system calls**.
- Se necesita soporte del hardware para implementar el modelo con dos estados de la CPU: uno privilegiado y uno no privilegiado.

User Mode vs Kernel Mode

- Una CPU tiene distintos modos de ejecución, con distintos privilegios. Por ejemplo x86 tiene 4 niveles de privilegios.
- El modelo que estudiamos utiliza dos modos de ejecución.
- Cuando un programa se ejecuta en User Mode tiene ciertas reestricciones de acceso a memoria o hardware.
- Cuando una rutina se ejecuta en Kernel Mode accede a todos los recursos sin restricciones.
- Cada tipo de CPU proveé instrucciones especiales para pasar de User Mode a Kernel Mode y viceversa.

Kernel Space vs User Space

- El kernel vive en un estado elevado del sistema, comparado con las aplicaciones del usuario. Este estado, que incluye un espacio de memoria protegido, acceso pleno al hardware y el procesador ejecutando en Kernel Mode, es conocido como **Kernel Space**.
- Por otro lado las aplicaciones de usuario se ejecutan en el **User Space**. Sólo ven un subconjunto de los recursos de la máquina y no pueden realizar ciertas funciones directamente. En este estado el procesador se ejecuta en User Mode.
- Un programa normalmente se ejecuta en User Space y cambia al Kernel Space sólo cuando requiere un servicio provisto por el kernel. Cuando el kernel satisface el pedido, vuelve a setear el procesador en User Mode y la aplicación vuelve a ejecutarse en User Space.

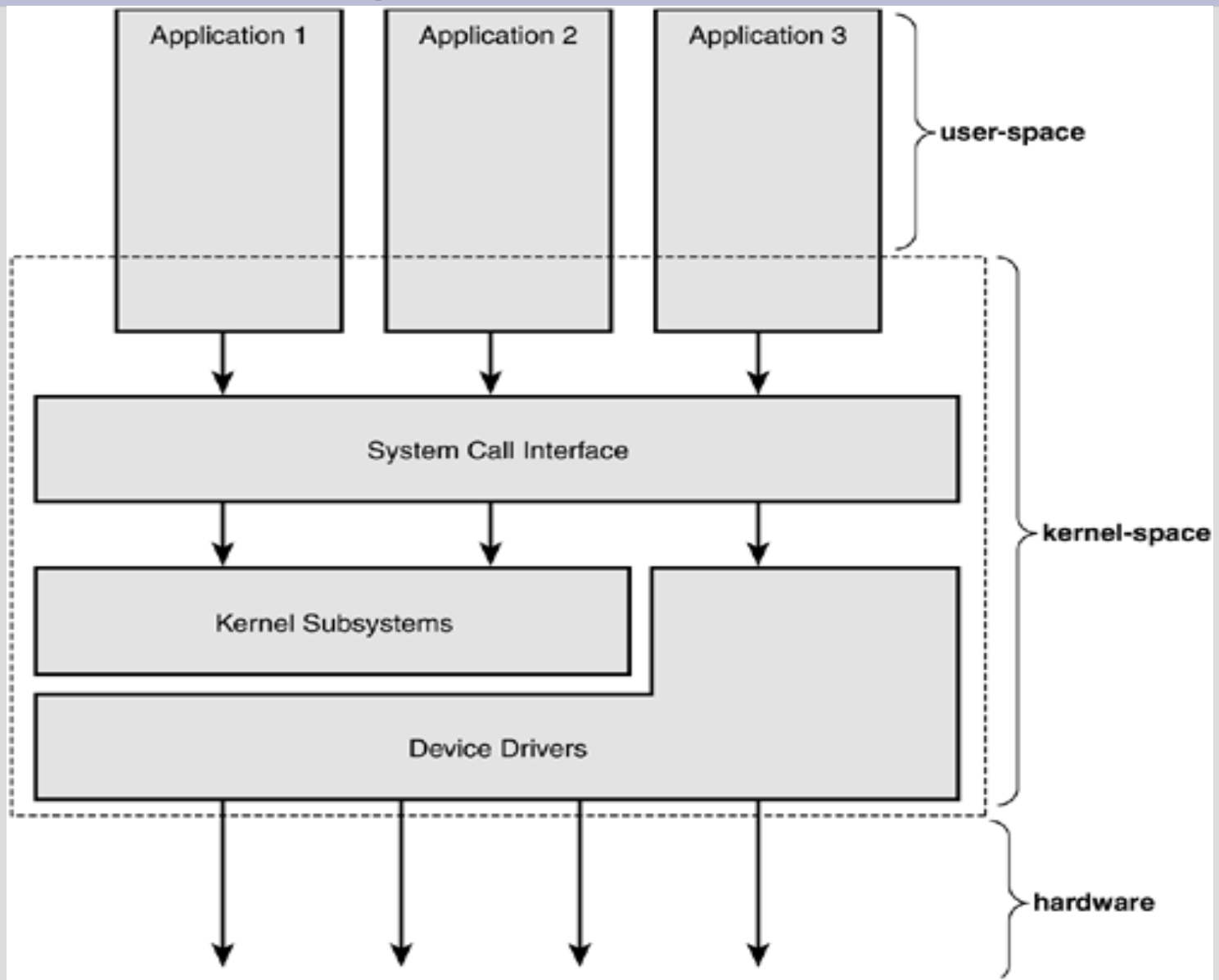
System Calls

- Las aplicaciones invocan funciones del kernel a través de las System Calls. Típicamente se llaman a funciones de librerías, las cuales invocan systems calls para realizar su trabajo.

- Algunas funciones de librerías invocan a más de una system call o llaman a una system call sólo como un paso en su procesamiento. Ejemplo: printf().
- Algunas funciones tienen una relación de uno a uno con system calls. Ejemplo: open().
- Otras funciones, no deberían hacer uso de system calls. Por ejemplo: strcpy ().

- En una system call, se dice que el kernel se está ejecutando en nombre de la aplicación. Es más, se dice que la aplicación está ejecutando una función en kernel space y el kernel está corriendo en el contexto del proceso de usuario (process context).

Relación entre procesos, kernel y hardware



Interrupciones y Excepciones

- Una interrupción suele ser definida como un evento que altera la secuencia de instrucciones ejecutadas por un procesador. Esos eventos corresponden a señales eléctricas generadas por hardware desde dentro y fuera de la CPU.

- Suelen clasificarse en:
  - **Excepciones:** producidas por la unidad de control de la CPU.
  - **Interrupciones:** generadas por otros dispositivos de hardware.

- Cuando una señal de interrupción llega a la CPU, el kernel debe realizar ciertas tareas que generalmente son implementadas desde un módulo llamado **manejador de interrupción**.

Interrupt Context

- Los manejadores de interrupciones no se ejecutan en el contexto de un proceso.
- En cambio, se ejecutan en un contexto especial de interrupción, pero los ciclos de CPU consumidos se computan al proceso interrumpido.
- Este contexto existe únicamente para permitir que un manejador responda rápidamente a una interrupción y luego termina.

Contextos de Ejecución del Kernel (vistos hasta ahora)

- Estos contextos representan **momentos de actividad del kernel**.

- De hecho, en Linux, se puede generalizar que cada procesador de una máquina está haciendo alguna de estas tres cosas en un momento dado:

- En **kernel space**, en "**process context**", ejecutando en nombre de algún proceso.
- En **kernel space**, en "**interrupt context**", manejando una interrupción.
- En **user space**, ejecutando código de un proceso de usuario.

Kernel Threads

Además de procesos de usuario, Linux incluye algunos "procesos" privilegiados, con las siguientes características:

- Normalmente son creados durante el arranque del sistema y permanecen vivos hasta que se apaga la máquina.
- Corren en kernel space, no necesitan tablas de página para mapear memoria de usuario
- No interactúan con usuarios, por lo que no requieren dispositivos de E/S (terminales)
- Contexto más liviano, por lo tanto el switch es más rápido que el de un proceso de usuario

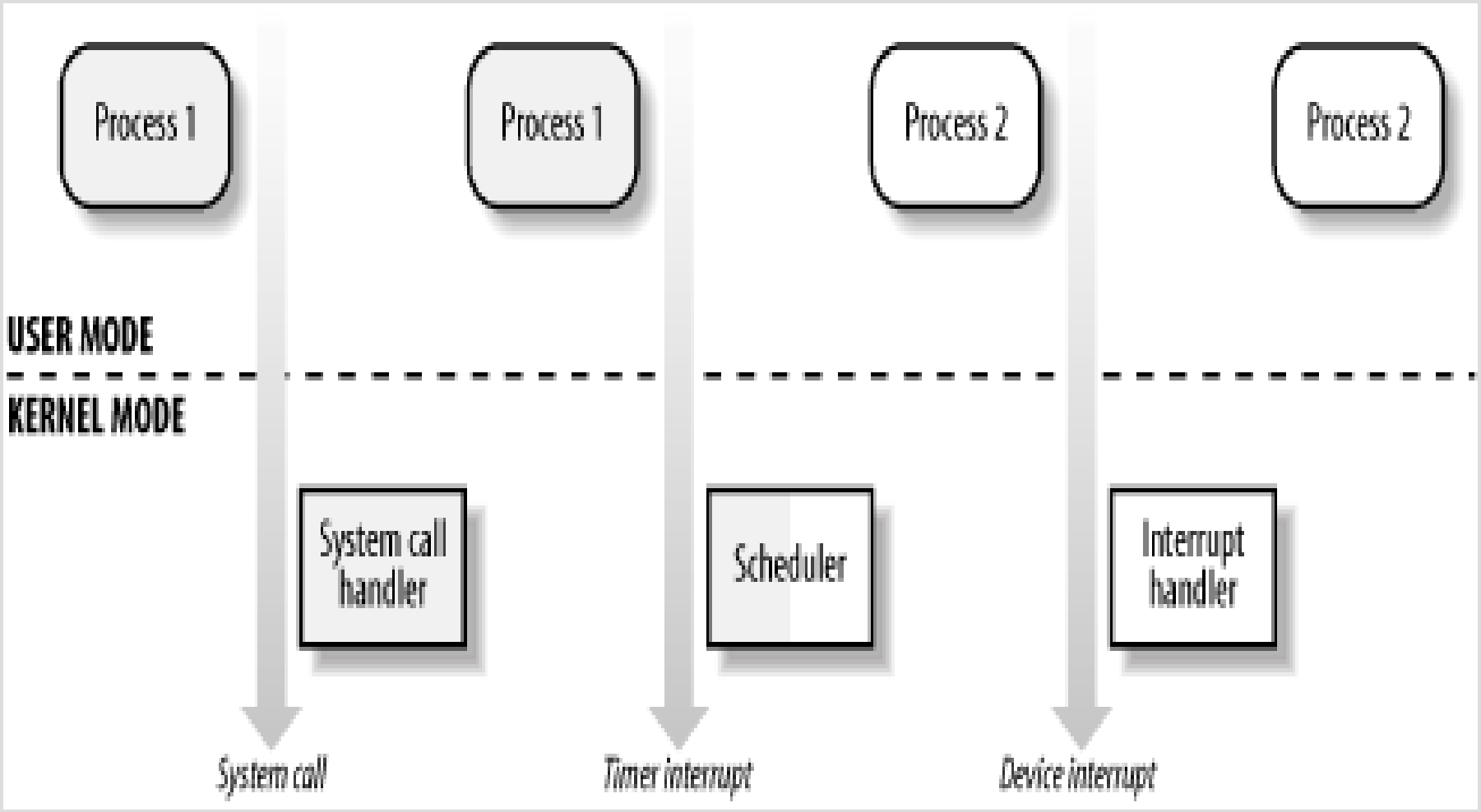
Funciones Diferibles

- El manejador de interrupciones no es un contexto en el que puedan realizarse cualquier tipo de acción. Por ejemplo, **no debe bloquearse nunca.**
- Las operaciones largas y no críticas deberían ser diferidas ya que mientras el manejador se está ejecutando, las señales en la línea IRQ correspondiente son temporalmente ignoradas.
- Linux implementa esto usando dos tipos de funciones diferibles (“bottom halves”) del kernel: “softirqs” y “tasklets”
- Este tipo de funciones suele ser planificada por el programador desde un manejador de interrupciones (o de otro contexto) para que se ejecute asincrónicamente en algún momento “seguro” (elegido por el kernel) en el futuro cercano.

Contextos de Ejecución del Kernel (Completo)

- Un proceso invoca una **system call**.
- Una CPU ejecutando un proceso eleva una **excepción**. El kernel maneja la excepción “en nombre” del proceso que la causó.
- Un dispositivo eleva una señal de **interrupción** a la CPU para indicar que se completó una operación de E/S.
- Un **kernel thread** es elegido por el scheduler.
- Una **softirq** es ejecutada.

Ejemplo de ejecución



Kernels Unix y Linux

- Debido a sus antepasados comunes, los kernels Unix modernos, además de sus APIs comparten varias cuestiones de diseño.
- Con algunas excepciones, un kernel Unix es típicamente un binario monolítico y estático. Es decir, existe como una imagen ejecutable única y de gran tamaño que corre en un único espacio de direccionamiento
- Los sistemas Unix normalmente requieren hardware de administración de memoria (MMU) el cual proveé protección de memoria y un espacio virtual único a cada proceso. Linux tiene sus variantes para hardware embebido. Ejemplo: uClinux.

Diseño Monolítico versus Microkernels

Existen dos grandes clasificaciones de kernels con respecto a su diseño:

- kernels monolíticos
- microkernels

Otras categorías, nanokernels, exokernels, etc, se mencionan principalmente en entornos de investigación.

Kernels Monolíticos

- Representan el diseño más simple y fue la única opción hasta los 80s.
- Se implementan en forma de un gran “programa” corriendo en un único espacio de direccionamiento.
- La comunicación dentro del kernel es trivial porque todo corre en kernel space en el mismo espacio de direccionamiento: se invocan funciones.
- Los defensores de este modelo citan la simplicidad y performance como principales ventajas.
- La mayoria de los kernels Unix son monolíticos.

Microkernels

- La funcionalidad del kernel está distribuida entre procesos separados, usualmente llamados servers.
- En teoría, sólo los procesos que realmente lo requieren son los que deberían ejecutarse en kernel space, y el resto en user space.
- Se incluye algún mecanismo de comunicación entre procesos, que ofrece comunicación entre servers a través de mensajes.
- Diseño modular. Permite fácil reemplazo de subsistemas.

Microkernels

Existen costos asociados al modelo:

- Pasaje de mensajes tiene mayor overhead que invocacion directa a funcion.
- Se producen context switches de kernel-space a user-space y viceversa

En consecuencia, todos los sistemas basados en microkernels ahora ejecutan sus servers en kernel space. El kernel de Windows NT y Mach son ejemplos de microkernels y ninguno corre los servers en user space en sus últimas versiones.

Linux: monolítico moderno

Linux es un kernel monolítico, sin embargo, toma muchas de las buenas ideas de los microkernels:

- Diseño modular
- Scheduleable / preemptible (KCPs)
- Soporte para kernel threads
- Modulos Dinámicos en demanda

Por otro lado, no sufre de la pérdida de performance asociada al modelo de microkernel:

- Todo corre en kernel mode
- La comunicación es a través de invocaciones funciones

Ejemplos

Kernels Monolíticos:

- Linux
- Kernels Unix tradicionales, como los BSDs
- Solaris

Microkernels famosos:

- Aix (IBM eServers pSeries, utilizando procesadores de la familia IBM POWER de 32 y 64bits)
- Minix
- Mach, usado en GNU Hurd y Mac OS X entre otros
- QNX (real time)

Reentrancia

- Todos los kernels Unix son reentrantes, es decir, más de un proceso puede estar en kernel mode a la vez
- Si ocurre una interrupción de hardware, un kernel reentrante es capaz de suspender el proceso que se está ejecutando en ese momento, **aunque esté en kernel mode**. Esta capacidad es importante para lograr eficiencia.
- Una forma de lograr reentrancia es usar sólo funciones reentrantes (como lo hacen algunos kernels de tiempo real), aunque esto no es siempre posible.
- Otra forma es permitir funciones no reentrantes y utilizar **mecanismos de locking**.

Reentrancia y su impacto en la organización del kernel

- Un camino de control del kernel (KCP) denota una secuencia de instrucciones ejecutadas por el kernel. En el caso más simple, el kernel ejecuta un KCP completo, sin ser interrumpido. Pero hay otros casos.
- system call que se bloquea --> scheduler invoca a otro proceso --> el nuevo proceso invoca una system call (dos KCP relacionados con dos procesos distintos).
  - mientras un proceso está en kernel mode ocurre un page fault --> se ejecuta el handler (dos KCP que se ejecutan en nombre del mismo proceso)
  - mientras un proceso está en kernel mode ocurre una interrupción --> se ejecuta el handler (dos KCP que se ejecutan en el “contexto del mismo proceso”, pero no necesariamente en nombre del mismo)
  - ocurre una interrupción y un proceso de mayor prioridad puede ser ejecutado --> por ser preemptive, el primer KCP se suspende y la CPU ejecuta otro KCP en nombre del proceso con mayor prioridad.

Concurrencia y Sincronización en el kernel Linux

- Es un sistema operativo reentrante, multi usuario y multi tarea.
- Soporta multiprocesamiento, por lo que cierto código del kernel ejecutándose en procesadores distintos podría acceder al mismo recurso al mismo tiempo.
- Las interrupciones ocurren asincrónicamente con respecto al código ejecutado en un momento dado.
- Es preemptive (desde el 2.6). Por lo tanto, cierto código del kernel puede ser interrumpido en favor otro código que acceda al mismo recurso.
- El kernel debe proveer mecanismos de sincronización (http://linux.linti.unlp.edu.ar/kernel/wiki/images/0/07/Concurrencia-sincronizacion-linux.pdf)

Algunas formas de sincronizar los KCPs

- Deshabilitación de la capacidad de preemption
- Deshabilitación de las interrupciones
- Semáforos
- Spinlocks

Deshabilitación de la capacidad de preemption

- solución drásticamente simple de algunos kernels Unix tradicionales
- cuando un proceso se ejecuta en modo Kernel, no puede ser suspendido y sustituido por otro.
- Entonces, en un sistema UP, todas las estructuras de datos no modificadas por interrupciones o excepciones estarán seguras.
- Los procesos en modo Kernel pueden ceder voluntariamente la CPU
- Este mecanismo no sirve en sistemas SMP, ya que dos KCPs en distintas CPUs pueden acceder concurrentemente al mismo dato.
- El kernel Linux permite desactivar la capacidad de preemption antes de entrar a una sección crítica, como medida de sincronización.

Deshabilitación de las interrupciones

- Se desactivan todas las interrupciones del hardware al entrar a la sección crítica y se habilitan al salir
- Problemas de performance
- En sistemas SMP, desabilitar las interrupciones en la CPU local no es suficiente y se debe usar otro mecanismo

Semáforos

- Una variable entera asociada a la estructura de datos que queremos protejer.
- Una lista de procesos y dos operaciones atómicas: down() y up().
- Los **procesos se bloquean** y se agregan a la lista cuando la estructura está siendo usada.
- Efectivo tanto en UP como en SMP.

Spinlocks

- Si la sección crítica es pequeña (se requiere poco tiempo para acceder/modificar el dato) un semáforo puede ser ineficiente.
- Cuando un KCP encuentra el lock tomado, ejecuta un loop cerrado (busy waiting).
- Sólo en SMP

Algunos cambios con respecto al user space

El kernel tiene algunas diferencias comparadas con las aplicaciones de usuario que, aunque no necesariamente lo hagan más difícil de programar que el user space, ciertamente presentan un desafío con algunas reglas nuevas.

Las diferencias más importantes son:

- No poseé protección de memoria como el user space.
- No puede realizar operaciones de punto flotante fácilmente.
- El kernel tiene una pila pequeña y de tamaño fijo.
- El kernel no tiene acceso a la librería de C.
- Está codificado con GNU C.

No hay Protección de Memoria

- Cuando una aplicación de usuario intenta un acceso a memoria ilegal, el kernel maneja la excepción producida, envía SIGSEGV y mata al proceso.
- Las violaciones de memoria del kernel resultan en “oops”. Un “oops” es un error principal en el kernel, que produce un mensaje de error en la consola, hace un dump de los registros y proveé un back trace.
- Si el oops ocurre en contexto de interrupción, en la tarea idle (pid 0) o la tarea init (pid 1) se produce un “panic” (caida instantánea del sistema).
- En otros casos, el kernel mata el proceso y trata de continuar ejecutándose, aunque podría quedar en un estado inconsistente.

Memoria no paginable

- La memoria del kernel no es paginable.
- Esto es una decision de diseño.
- Por lo tanto, cada byte *consumido* por el kernel es un byte menos de memoria física disponible

Uso complicado de punto flotante

- Cuando un proceso en user space usa instrucciones de punto flotante, el kernel normalmente maneja la situación (trap).
- Pero usar operaciones de punto flotante en el kernel requiere escribir y leer manualmente los registros del FPU (o similar). Por eso la opción utilizada es la más sencilla: NO SE USA PUNTO FLOTANTE EN EL KERNEL

Pila pequeña y de tamaño fijo

- En el user space, los procesos pueden alocar muchas variables en la pila, incluyendo estructuras grandes y arreglos de muchos elementos.
- La pila del kernel es pequeña y de tamaño fijo. El tamaño exacto varía con la arquitectura. En x86, el tamaño es configurable en momento de compilación y puede ser de una o dos páginas.
- Históricamente, la pila del kernel es de dos páginas, lo que generalmente implica que es de 8KB en arquitecturas de 32 bits y de 16KB en arquitecturas de 64 bits.
- Cada proceso tiene hay una pila distinta en kernel mode.

No libc

- A diferencia de las aplicaciones de usuario, el kernel no se linkea con la librería de C estandar (en realidad, con ninguna librería).
- La principal razón (además del problema tipo “huevo-gallina”) es la velocidad y el tamaño.
- Muchas de las funciones usuales de la libc han sido implementadas dentro del kernel. Por ejemplo, las funciones de strings están en lib/string.c. Es necesario incluir <linux/string.h>
- Por lo tanto, el kernel no tiene acceso a printf(), pero puede usar printk(), la cual copia el string formateado en el buffer de log del kernel, el cual será leido normalmente por syslog.

Extensiones al lenguaje C

- Linux no se programa en un ANSI C estricto. Los desarrolladores usan varias extensiones del lenguaje C especificadas por ISO C99 y GNU C.

- Esto hace que Linux sea inseparable de gcc (el GNU Compiler Collection, que contiene el compilador usado para compilar el kernel y casi cualquier cosa escrita en C en un sistema Linux).

- Recientemente otros compiladores, como el compilador C de Intel han llegado a soportar suficientes características como para poder compilar el kernel Linux.

ISO C99

- ISO C99 es la última revisión importante del estandar ISO C. Agrega algunas mejoras a la revisión anterior, ISO C90, incluyendo inicializadores nombrados para estructuras, un tipo *complex* (este no puede ser usado de manera segura en el kernel) y un tipo *boolean* explícito.
- Introducción: *Open source development using C99* (http://www.ibm.com/developerworks/library/l-c99.html)
- Ejemplo de inicializadores:

```
struct {float x, y, z;} s = { .y = 1.0, .x = 3.5, .z = 12.8};
```

GNU C

Las desviaciones más interesantes y, tal vez menos conocidas, del estandar ANSI C son aquellas provistas por GNU C.

A continuación se mencionan aquellas que suelen aparecer en el código del kernel:

- Funciones Inline (también en C99)
- Assembly Inline
- Optimización de Saltos

Funciones Inline

- Con una función inline, el código de la función es insertado en el lugar de cada invocación. Esto elimina el overhead del salto y el regreso (register saving and restore) y permite una mejor optimización del código (porque el compilador puede optimizar juntos al código invocante y el código invocado).
- La desventaja es el incremento en el tamaño del código, el cual consume más memoria y espacio en cache de instrucciones (footprint).
- Los desarrolladores usan funciones inline para funciones pequeñas y de tiempo de respuesta rápido. Se prefieren éstas antes que macros complicadas por razones de seguridad de tipos.
- Una declaración ejemplo:

  static inline void convertir (unsigned long valor) {...}
- Se suelen definir en los headers y como son estáticas, no se exportan a otros archivos.

Assembly Inline

- El compilador gcc permite embeber instrucciones assembly en funciones C utilizando la directiva al compilador *asm().*

- El kernel Linux está programado en C en su mayoría y se usa assembly en las partes de código de bajo nivel dependientes de la arquitectura y aquellas funcionalidades cuya eficiencia es crítica.

Optimización de Saltos

El compilador gcc tiene una directiva que permite la optimización de saltos condicionales, dependiendo de la probabilidad de que el salto se produzca o no.

En el kernel se usan las macros likely() y unlikely() para indicar esa probabilidad.

Por ejemplo, en una sentencia if como la siguiente:

```
if (seCumpleAlgo)
{
        /* ... */
}
```

Optimización de Saltos

- Indicamos que es muy probable que esa condición se cumpla en ese momento de la ejecución.

```
if (likely(seCumpleAlgo))
{
        /* ... */
}
```

Estas directivas resultan en una mejora de performance cuando la predicción resulta correcta, pero significan un costo si la predicción es incorrecta.

- Un uso muy común para likely() y unlikely() son la verificación de errores. En el kernel, unlikely() es bastante más usado que likey() ya que las sentencias if tienden a indicar algún caso especial.

Referencias

- Understanding the Linux kernel. 3era edición.
- Linux Kernel Development. 2da edición.
- Understanding Linux Network Internals.
- The Linux TCP/IP Stack: Networking for Embedded Systems.
- Linux Device Drivers. 3era edición.
- Linux Kernel in a Nutshell.
- Internet

“Just a Program”

“The kernel is indeed a unique and inimitable beast: No memory protection, no tried-and-true libc, a small stack, a huge source tree. Despite this, however, **the kernel is just a program**.”

PREGUNTAS ?