Programación en modo Kernel y Desarrollo de drivers

Carlos Manuel Duclos Vergara
carlos@embedded.cl
Embedded CL

Advertencias

- APAGAR LOS CELULARES!

Contenidos

- Estructura de un sistema Unix / Linux
  - Principios rectores
  - Componentes
  - Funcionamiento
- Programación en modo kernel
  - Diferencias con la programación de aplicaciones
  - Ventajas y desventajas
  - Cuando programar en modo kernel y cuando no

Contenidos (2)

- Funcionamiento de un sistema Linux
  - Booteo
  - Inicializacion del kernel
  - Inicializacion del sistema de usuario
  - Drivers
- Desarrollo de un driver de ejemplo
  - Estructura básica de un módulo
  - Nuevo sistema de compilación
  - Funcionalidades básicas que se deben implementar
  - Interactuando con interrupciones
  - Condiciones de carrera

Acerca de mi

- Ingeniero Informático de la UTFSM
- Fundador de Embedded CL
  http://www.embedded.cl
- Colaborador del proyecto µCLinux y también del proyecto ReactOS
- Actualmente trabajo portando Linux a diferentes arquitecturas de hardware desarrolladas por mi compañía

Preliminares

- Este taller es más una transferencia de experiencias más que de conocimientos. Internet está llena de información sobre la programación para el kernel de Linux
- Este taller no pretende crear Kernel Hackers, pero si pretende dotar a los participantes con las herramientas necesarias para llegar a serlo

Preliminares (2)

- Preguntas previas al público
  - Hexadecimal 0x5AC7
  - Static y Volatile
  - Proceso de compilación
  - Experiencia previa con el Kernel
    - Compilación del Kernel

Estructura de un sistema Unix / Linux

05 diapositivas

Estructura de un sistema Unix / Linux

Principios rectores:

- Sistema multiusuario
- Simple
- Fácilmente expandible

Recordar que Unix surgió en 1969, y que Linux es un “clon” de Unix

Estructura de un sistema Unix / Linux (2)

Componentes:

- Kernel
  - Administración de los recursos
  - Interacción con el hardware
  - Procesos y comunicación entre procesos

Estructura de un sistema Unix / Linux (3)

- Librería C
  - Encargada de ofrecer (“exportar”) las funcionalidades ofrecidas por el kernel hacia las aplicaciones de usuario
  - Es la interfaz principal a través de la cual los procesos pueden comunicarse con el kernel
  - Implementa gran parte de funcionalidades extra para hacer más fácil la programación de aplicaciones

Estructura de un sistema Unix / Linux (4)

- Init
  - Es el primer proceso creado por el Kernel, sin Init no hay sistema Unix.
  - Es un proceso encargado de crear los “niveles” de ejecución y encargado de ofrecer los servicios asociados a cada uno de ellos
  - Es un proceso especial, que no responde a las señales tradicionales y que recibe un tratamiento especial por parte del kernel

Estructura de un sistema Unix / Linux (5)

- Shell
  - Es la interfaz entre el usuario y el sistema
  - Actualmente ha perdido importancia, pero sigue siendo una herramienta poderosa y muy utilizada
  - Su principal virtud es su simplicidad, lo que permite una fácil integración con otras aplicaciones de usuario

Programación en modo kernel

17 Diapositivas

Programación en modo Kernel

Existen 2 tipos de situaciones:

- Programar aplicaciones de usuario de forma tal de que el usuario pueda sacar provecho de su sistema computacional
- Programar funcionalidades en el kernel de forma tal de ayudar a los desarrolladores de aplicaciones a entregar mejores herramientas a los usuarios

LAS DOS SON EQUIVALENTES EN COMPLEJIDAD

Programación en modo Kernel (2)

Tipos de funcionalidades entregadas por el Kernel:

- Acceso a hardware
- Acceso a sistemas de archivos
- Control y intercomunicación de procesos
- Administración de los diferentes recursos disponibles en el sistema computacional
- Soporte para protocolos de comunicación

Programación en modo Kernel (3)

La programación en modo Kernel debiera utilizarse en situaciones donde el rendimiento es crítico y en aquellas situaciones donde se requiere acceso a hardware que no es accesible a las aplicaciones de usuario.

La programación en modo Kernel no sirve para todas las situaciones

Programación en modo Kernel (4)

En general la programación en modo Kernel debe restringirse al mínimo posible.

Hay funcionalidad que puede ser provista de mejor manera por aplicaciones de usuario, no hay para que involucrar al Kernel.

Muchas veces la mejor solución involucra un poco de programación en modo Kernel y una aplicación de usuario que se encarga del resto

Diferencias con la programación de aplicaciones de usuario

Principales diferencias:

- Alcance
  - Una aplicación sólo afecta a quien la utiliza y sólo de manera temporal
  - El kernel afecta a todos los usuarios y aplicaciones

Diferencias con la programación de aplicaciones de usuario

- Consumo de memoria
  - El kernel reside en memoria física, por consiguiente cualquier byte usado en modo kernel es un byte menos para el resto del sistema
  - Las aplicaciones residen en memoria virtual, por consiguiente pueden ser intercambiadas (swap) o bien eliminadas de la memoria si el kernel así lo dispone

Diferencias con la programación de aplicaciones de usuario

- Aislamiento
  - Un error en una aplicación de usuario por lo general queda aislado a sólo esa aplicación
  - Un error en el kernel afecta a todo el sistema y por lo general hace el sistema irrecuperable

Comparación

Ventajas de la programación en modo Kernel:

- Puede otorgar mejor rendimiento
  - Notar que si no se programa correctamente el desempeño puede empeorar de manera trágica
- Acceso a recursos no disponibles para aplicaciones de usuario
  - Llamadas no disponibles a través de la biblioteca C
  - Recursos de hardware

Comparación (2)

Desventajas:

- Los recursos disponibles en modo Kernel son por lo general muy limitados
  - Ausencia de printf (Cuidado con el uso de printk)
  - No hay aritmética de punto flotante
- El flujo del tiempo no es *lineal*
  - Una aplicación comienza y termina de manera lineal
  - El uso de locks y llamadas al scheduler hacen que la predicción del flujo del tiempo sea muy difícil

Comparación (3)

- Consumo de memoria
  - Como se dijo anteriormente el kernel utiliza memoria física no intercambiable, por consiguiente hay que ser muy cuidadoso con el uso de memoria
- No existe retroalimentación ni interacción posible por parte del usuario
  - Lo máximo que se puede lograr es la modificación de algunos parámetros generales de funcionamiento

Cuando programar en modo kernel

- Cuando se requiera acceso directo al hardware
  - Es decir cuando se requiera un acceso más complejo que lo que ofrecen iopl(2) e ioperm(2) ofrecen
  - Cuando se requiere trabajar directamente sobre las interrupciones
  - Cuando se requiere un manejo muy fino de la memoria o del scheduler

Cuando programar en modo Kernel

- Para implementar protocolos de comunicación
  - Es prudente implementar sólo los protocolos de comunicación que son independientes de un hardware específico, tomar el caso del protocolo PPP
  - En algunos casos es necesario dividir la implementación en dos partes, de forma tal de permitir que el protocolo en si sea tratado por el kernel y la interacción con el hardware y el usuario se realice a través de una aplicación de usuario

Cuando programar en modo kernel

- Cuando se requiera implementar políticas especiales
  - Por ejemplo cuando se requiera cambiar el scheduler por otro (RTAI Linux)
  - Cuando se requieren políticas especiales de manejo de memoria
  - Para programar políticas de control de recursos! (SELinux)

Cuando programar en modo kernel

- Cuando las circunstancias lo ameriten
  - Es decir, hay casos en los que no es fácil clasificar el por qué de la necesidad de programar en modo kernel
  - La experiencia indica que se presentan situaciones donde es preferible programar en modo kernel, tal como dice el dicho: “A ojo de buen cubero”

Cuando NO programar en modo kernel

- Cuando lo que se quiere implementar sea más simple de realizar a través de una aplicación de usuario
  - No es necesario reinventar la rueda, muchas veces es posible solucionar los problemas a través de una aplicación de usuario
  - El agregar funcionalidad sin un fundamento de peso por lo general trae más problemas que soluciones (X + Manejadores de escritorio versus todo integrado como Windows)

Cuando NO programar en modo kernel (2)

- Cuando se requiera interacción por parte del usuario
  - El kernel por definición no incorpora mecanismos para interactuar con el usuario
  - Aún cuando es posible implementar algunos mecanismos de interacción, el rendimiento decae en forma casi exponencial
  - En estos casos es mejor separar el problema en dos partes e implementar la parte interactiva en una aplicación de usuario

Cuando NO programar en modo kernel (3)

- Cuando no haya un beneficio claro para todo el sistema
  - Nótese que el beneficio debe ser claro para el sistema en cuestión, no para todos los sistemas existentes ni todos los casos posibles
  - Hay que tener cuidado de no implementar funcionalidad ya existente que podría ser mejor aprovechada a través de una biblioteca de sistema

Funcionamiento de un sistema Linux

10 Diapositivas

Booteo

Booteo:

- El booteo es el proceso mediante el cual el computador es llevado desde un estado no operacional a un estado operacional
- Nótese que no involucra sistemas operativos ni bootloaders

Booteo (2)

Podemos clasificar el booteo en tres etapas:

- POST
  - Power On Self Test
- Bootloader
  - Pequeño(s) programa(s) encargado de tomar el control del dispositivo una vez finalizado el POST
  - Su principal función es la de preparar el ambiente para la ejecución de un sistema operativo

Booteo (3)

- Sistema operativo
  - Set de rutinas y bibliotecas diseñadas para utilizar los recursos computacionales existentes
  - La parte central de un sistema operativo es llamada Kernel (Núcleo)
  - El Kernel por sí sólo no conforma un sistema operativo, hacen falta bibliotecas de sistema y aplicaciones que permitan la utilización de los recursos

Booteo (4)

Booteo en un sistema Linux:

- Comúnmente vía bootloader
- El booteo de Linux se compone de tres partes:
  - Inicialización de hardware
  - Inicialización del Kernel
  - Inicialización de ambiente de usuario

Inicialización de hardware

- Archivos head.S, crt0_ram.S o crt0_rom.S
- Consiste en “ubicar” el Kernel en el hardware correspondiente
- Termina con la llamada *start_kernel()*

Inicialización del Kernel

- Consiste en la ejecución de 2 rutinas principales:
  - *start_kernel()*
  - *rest_init()*
- La rutina *start_kernel()* es la encargada de inicializar cada uno de los subsistemas del kernel:
  - Memoria
  - I/O
  - Scheduler

Inicialización del Kernel (2)

- La rutina *rest_init()* es la encargada de poner en marcha el proceso **init** y de esta manera liberar el scheduler

Inicialización del ambiente de usuario

- Una vez creado el proceso **init** (PID 1), este se encarga de utilizar la llamada *fork()* para ir creando los procesos necesarios
- Dependiendo del sistema la inicialización del ambiente de usuario puede abarcar distintas áreas y distintos procesos

Drivers

¿Qué es un driver?

Un driver es un software diseñado para cumplir una función específica

Nótese que un driver puede no tener nada que ver con el Kernel

Clasificación de drivers

- Por función
  - Acceso a hardware
  - Implementación de protocolos
- Por nivel de ejecución
  - Kernel
  - Usuario

Desarrollo de un Driver para Linux

17 Diapositivas

Tipos de drivers en Linux

- Forma de linkeado:
  - Modulares
  - Integrados
- Tipos de acceso
  - char
  - block
  - network
  - hotplug y derivados
  - misceláneos

Drivers tipo char

- Se llaman drivers tipo char ya que el acceso a ellos es byte a byte
- Son el tipo más simple de drivers y los más utilizados
- Están pensados para interactúar con aplicaciones más que con el resto del Kernel

Drivers modulares

- Pueden ser integrados a medida que el sistema los requiere
- Facilitan enormemente el desarrollo ya que normalmente pueden ser cargados/descargados según necesidad
- Mientras no estén cargados no ocupan memoria

Elementos básicos de un driver modular

Un módulo básico está compuesto de dos funciones:

- **inicialización**
  - No tiene nombre predeterminado, se asigna mediante el uso de la función auxiliar *module_init()*
- **salida**
  - Tampoco tiene nombre predeterminado, se asigna mediante el uso de la función auxiliar *module_exit()*

Elementos básicos de un driver modular (2)

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int inicializacion_serial(void)
{
   printk(KERN_ALERT "Hola Encuentro Linux 2004!\n");
   return 0;
}

static void fin_serial(void)
{
   printk(KERN_ALERT “Adios Encuentro Linux 2004!\n”);
   return 0;
}
module_init(inicialización_serial);
module_exit(fin_serial);
```

Elementos básicos de un driver modular (3)

Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m       := serial.o
    serial-objs := serial_interface.o
else
    KDIR        := /lib/modules/$(shell uname -r)/build
    PWD         := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif

clean:
    rm -f .*.cmd *.o *.ko *.mod.c
```

Métodos de acceso al hardware

Existen 2 modelos de acceso al hardware:

- Memory Mapped (1 sólo espacio de direcciones)
  - Usado en muchas arquitecturas ya que permite la escritura de un código mucho más simple
- IO Memory (2 o más espacios de direcciones)
  - Usado en la arquitectura x86, también en la Sparc y UltraSparc ya que permite encapsulación y separación de código

Utilización del hardware

El hardware es controlado mediante la escritura y lectura de registros de control y status.

Específicamente los puertos seriales compatibles con el 8250 tienen los siguientes registros:

- Lectura
  - +0 Receiver buffer
  - +1 Interrupt Enable Register
  - +2 Interrupt Identification Register

Utilización hardware (2)

  - +3 Line Control Register
  - +4 Modem Control Register
  - +5 Line Status Register
  - +6 Modem Status Register
- Escritura
  - +0 Transmit buffer
  - +2 FIFO Control Register

Utilización hardware (3)

- Extendidos
  - +0 Divisor Latch Low Byte
  - +1 Divisor Latch High Byte

Todos estos registros están definidos en *serial_definitions.h*

Métodos específicos para un driver tipo char

Los métodos clásicos que un driver tipo char debe implementar son:

- *open*
- *read*
- *write*
- *release* (NO close!)

Métodos específicos para un driver tipo char (2)

- **Open**
  - Es el método de inicializar el dispositivo si no está inicializado
  - Debe ubicar el hardware y configurarlo
- **Release**
  - Es el método que se invoca cuando no hay más referencias a un módulo
  - Su misión es devolver y deshacer todo lo hecho por *open()*

Métodos específicos para un driver tipo char (3)

- **Read**
  - Es el encargado de buscar información desde el dispositivo
  - Usualmente bloquea a los procesos que lo llaman
- **Write**
  - Es el encargado de enviar información al dispositivo
  - Usualmente bloquea a los procesos que lo llaman

Coordinación de eventos

Cuando se interactúa con hardware generalmente se requiere responder a eventos:

- Fin de dispositivo
- Lectura no válida
- Buffer listo para iniciar transmisión

Es por esto que se hace necesario contar con primitivas de sincronización

Coordinación eventos (2)

Técnicas de sincronización:

- Semáforos
  - Útiles para evitar que dos procesos se estorben al acceder a un dispositivo
- Listas de espera
  - Útiles cuando se desea esperar una respuesta de hardware

Interrupciones

Las interrupciones son eventos externos al sistema computacional que provocan un cambio en el flujo normal de las instrucciones

Facilitan la tarea de coordinar eventos entre diversos elementos de hardware y software

Interrupciones (2)

En Linux las interrupciones deben ser solicitadas al Kernel y luego devueltas:

- **request_irq**
  - Solicita una interrupción e instala un handler para su administración
- **free_irq**
  - Devuelve una interrupción solicitada por *request_irq()*

Interrupciones (3)

- **request_irq**(irq, handler, flags, "name", data)
  - irq: número de la interrupción solicitada
  - handler: manejador de la interrupción
  - flags: tipo del manejador y la interrupción
  - data: información privada que se le entrega al manejador
- **free_irq**(irq, data)
  - irq: número de la interrupción a devolver
  - data: información privada que se le entregó al handler

Interrupciones (4)

- **request_irq**
  - request_irq(4, interrupcion_serial, SA_INTERRUPT, "serial-el", NULL);
- **free_irq**
  - free_irq(4, NULL);

Condiciones de carrera

Una condicion de carrera se define como una secuencia de operaciones que entregan diferentes resultados dependiendo del orden en el que son ejecutados

Para evitar las condiciones de carrera hay que usar primitivas de sincronización y bloqueo

Condiciones de carrera (2)

- **Corrupción de datos**
  - Se produce cuando dos threads interactúan con el dispositivo a la vez
- **Falsos positivos y Verdaderos negativos**
  - Se producen cuando se gatilla un evento de manera artificial o cuando se pierde un evento debido a que no se atendió en el tiempo correspondiente
- **Manejadores de interrupciones**
  - No pueden bloquearse, por lo general introducen condiciones de carrera

Condiciones de carrera (3)

- **Handler interrupción**
  - p->estado |= DATOS_DISPONIBLES;
- **Write**

```
lock( p );
if( !(p->estado & DATOS_DISPONIBLES ))
.....
unlock( p );
```

Referencias

- http://lwn.net
- http://www.xml.com/ldd
- Listas de correo