nombre todavía no pensado

Un kernel emulador

Aldana Ramirez
Martín Villagra

29 de septiembre de 2014

Resumen

El objetivo del trabajo es construir un emulador de la videoconsola Nintendo Entertaiment System(también conocida como Family) que se ejecute sin ningún sistema operativo de por medio.

Índice general

Capítulo 1

Introducción

Brainstorm!
algo historia de la consola, que fue la más exitosa de su época, como trascendio, que era barata,etc
porque la elegimos: es una de las consolas mas investigadas y toqueteadas que alguna vez existio.
porque queremos independizarnos del sistema operativo: para poder correr nuestro emulador en máquinas mas restringidas, para que en un futuro sea posible correrlo en procesadores de bajo poder, se podría hacer un clon de la consola a muy bajo precio, un arduino cuesta 10 20 dolares. en marcado libre las consolas originales están como 1500 pesos :o
*modo de trabajo: se busca no reinventar la rueda, el projecto se construyo tomando como guia otros kernels y emuladores ya existentes**explicar como se encontró y utilizo la información necesaria para hacer el proyecto*
la mayoría se va a hacer en c, solo se usa assembler si es necesario *se asume conocimientos en c por parte del lector y un manejo de inglés*
comentar que se van a usar referencias a los códigos del emulador

1.1. Estructuración

En un principio dividimos en dos partes principales el projecto.
Una es el **emulador** propiamente dicho, es decir la que se encarga de hacer todo lo que hacía internamente la consola.
La otra parte es el **kernel** encargada de inicializar todo lo necesario para que el emulador funcione, así como proveerle funciones de bajo nivel tales como escribir en pantalla o reservar memoria. Al no tener un sistema operativo detrás, funciones como malloc y free que cualquier programador de C supone siempre presentes deben ser implementadas por el kernel.

Capítulo 2

El Kernel

Un **kernel** (de la raíz germánica Kern, núcleo, hueso) se define como la parte que se ejecuta en modo privilegiado (conocido también como modo núcleo), es decir con acceso irrestricto a todo el hardware del sistema. Es el principal responsable de facilitar a los distintos programas acceso seguro al hardware de la computadora y de gestionar recursos.
En su más simple escencia es simplemente un código ejecutable, el cual es generado con un compilador como cualquier otro. La diferencia es que no tiene dependencias con librerías del sistema, tales como stdio.h o stdlib.h.

2.1. Carácteristicas

Empecemos por decir que lo que aquí se presenta está muy lejos de un sistema operativo completo. Se priorizó la simplicidad recortando todo lo que estaba de más, reduciendo al mínimo las capacidades del sistema. Por ejemplo la mayoría de los sistemas operativos pueden leer un programa y ejecutarlo. Nuestro querido kernel carece de esa posibilidad. A continuación se enumeran las carácteristicas principales del sistema.

- Dentro de la familia de arquitecturas x86[1] se eligió i686[2]. Se prefirió esta por sobre x86_64[3] ya que tiene esta última no es soportada nativamente por GRUB(ver sección 2.3). Respecto al formato de los ejecutables se eligió el que implementan los sistemas Unix[4](como

[1] `http://en.wikipedia.org/wiki/X86`
[2] `http://en.wikipedia.org/wiki/P6_(microarchitecture)`
[3] `http://en.wikipedia.org/wiki/X86-64`
[4] `http://en.wikipedia.org/wiki/Unix`

Linux[5]), el ELF [6].

- Es un kernel monolítico, esto significa que todos los drivers y los servicios necesarios para el funcionamiento completo del sistema están incluídos en dentro del mismo kernel. Sistemas operativos como Linux no siguen esta filosofía, ya que implicaría cargar todos los drivers al incio del sistema, el cual tiene un costo de memoria muy alto. Esto no representa un problema para nosotros puesto que la cantidad de drivers y servicios ofrecidos por el kernel es muy reducida.

- Monotarea: Solo puede realizar una tarea a la vez, en nuestro caso el emulador. Para simplificar aún más incluso, el emulador es verdaderamente la *única* tarea que el kernel puede ejecutar. Podría verse al emulador como un solo programa standalone, que no necesita ningún sistema operativo para ejecutarse.

- Sin memoria virtual: En un sistema operativo clásico cada proceso tiene sus propio mapa de direcciones virtuales. A medida que el proceso va solicitando más memoria se busca espacio en la memoria física y luego se le asigna la encontrada a una dirección virtual, la cual es usable por el proceso. Este sistema tiene ventajas tales como la posibilidad de restringir la memoria accesible por un proceso. Es destacable que la memoria física no tiene porque restringirse solo a la RAM[7], puede apuntar incluso a una posición en el disco duro. Optamos por no incluir esta opción, para simplificar el sistema. Esto significa que los punteros contienen la dirección de memoria física y que es posible acceder y leer la memoria usada por otros procesos libremente.

- Sin userspace: Normalemente un sistema operativo puede dividirse en tres capas de abstracción diferentes, el hardware, el kernel y finalmente el userspace, que es donde se ejecutan las aplicaciones como Chromium, Notepad, *etc.*

[5] `http://en.wikipedia.org/wiki/Linux`
[6] `http://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
[7] Random Access Memory

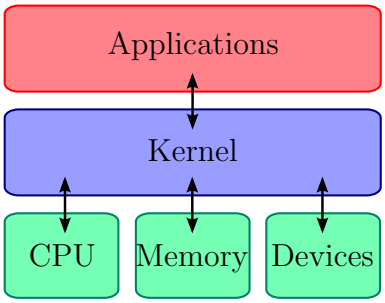

Figura 2.1: Layout estándar de un kernel.

En nuestro caso no tenemos userspace: el emulador, nuestro único programa a ejecutar, está embebido en el mismo kernel. Esto simplifica enormemente el diseño del kernel.

2.2. Funciones

Antes de hacer el kernel, hay que tener muy en claro que es exactamente lo que tiene que hacer. Surge la pregunta entonces ¿Qué necesita como mínimo nuestro emulador? Se muestra, en orden de importancia:

- Iniciar el sistema y ejecutar el emulador. Ver sección 2.3.
- Modificar libremente pixeles de la pantalla
- Detectar pulsaciones del teclado
- Poder reservar memoria dinámica *explicar que es*
- Cargar de alguna forma los juegos
- Ejecutar sonido
- Escribir en algún medio persistente el estado actual del juego(Guardar la partida)

Pues bien, el kernel tiene que ser capaz de proveer funciones que faciliten cada una de estas tareas. En las siguientes secciones se detallaran cada una de ellas.

2.3. Booteo

explicar que es un bootloader y la bios muy brevemente Para evitar tener que lidiar con la BIOS y otras interfaces de bajo nivel, se eligió utilizar un bootloader ya existente y ampliamente usado: GRUB. *poner que carajos es GRUB*. En particular existe un standard llamado Multiboot[8] que especifica como estructurar un kernel para que el mismo pueda ser cargado por GRUB(o por cualquier otro bootloader que implemente Multiboot). En particular se debe elaborar un header al inicio del kernel. En esta cabecera se determina por ejemplo que modo se prefiere(texto o consola) y que función va a ser la primera en ser llamada. A su vez GRUB se comunica con la BIOS entre otras cosas y recolecta información de la máquina que luego es recibida convenientemente por nuestro kernel. De esta forma al encender la máquina se iniciará GRUB, el mismo va a poder detectar nuestro kernel y lo va a ejecutar.

2.3.1. boot.s[9]

El problema es que al momento en que GRUB ejecuta el kernel el stack pointer no está inicializado por lo que no es posible que la función inicial sea en C. Debemos comenzar en assembler, inicializar el stack pointer y ahí si pasar a C. Esta es la razón de la existencia del archivo boot.s, que contiene el comienzo de nuestro kernel. Allí se incluye el header de Multiboot, el espacio para el stack y una función, la primera en ser llamada al iniciar el sistema, que inicializa el stack pointer y llama a nuestra función kernel_entry[10](ya en C) que continua con la inicialización del sistema.

[8] `http://en.wikipedia.org/wiki/Multiboot_Specification`
[9] Ubicado en kernel/arch/x86/generic/init/boot.s
[10] Ubicado en kernel/arch/x86/generic/init/kernel_entry.c