Ni intento ENTERTAINMENT KERNEL

Martín Villagra

29 de noviembre 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

La consola NES[1]fue una de las consola más vendidas de todos los tiempos. En 2009, la misma fue nombrada la mejor consola de videojuegos de la historia por IGN[2]. Gracias a su popularidad y simplicidad muchas personas se interesaron en el funcionamiento de la consola, dando lugar a una abundante documentación al respecto.
Fue por estas dos razones que se dicidió emular está consola en particular. Por otro lado, se eligió realizar un kernel para no depender de ningún sistema operativo. Esto introduce la posibilidad de ejecutar el emulador en máquinas más restringidas, que obviamente resultarían de menor costo que una computadora o la consola original.
Debido a la gran extensión del trabajo, se decidió seguir la filosofía de 'no reinventar la rueda'. El proyecto se construyó tomando como base otros kernels y emuladoras ya existentes, mencionados en la bibliografía (Capítulo 4).

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.

[1] `http://en.wikipedia.org/wiki/Nintendo_Entertainment_System`
[2] `http://uk.ign.com/top-25-consoles/1.html`

Capítulo 2

El Kernel

SACAR:En su más simple escencia es simplemente todos los kernels 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 o stdlib.
El diseño del kernel está muy lejos de lo que hoy en día se conoce como un sistema operativo completo. Se priorizó la simplicidad recortando todo lo que estaba de más, reduciendo al mínimo las capacidades del sistema.

2.1. Carácteristicas

- Dentro de la familia de arquitecturas x86 se eligió i686. Se prefirió esta por sobre x86_64 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(como Linux), el ELF [1].

- 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

[1] `http://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

verse al emulador como un solo programa standalone, que no necesita ningún sistema operativo para ejecutarse.

- Sin memoria virtual: 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.*

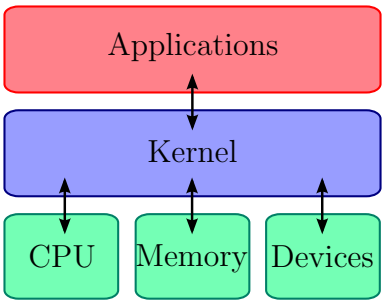


Figura 2.1: Layout estándar de un kernel.

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 los pixeles de la pantalla. Ver sección 2.4.
- Detectar pulsaciones del teclado. Ver sección 2.5.
- Poder reservar memoria dinámica. Ver sección 2.6.

- Cargar de alguna forma los juegos. Ver sección 2.7.

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

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. Existe un standard llamado Multiboot[2] 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 y ejecutar el kernel. Ya iniciado por GRUB, está va a ser la secuencia de arranque:

2.3.1. boot.s[3]

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 (ya en C) que continua con la inicialización del sistema.

2.3.2. kernel_entry[4]

En está función se inicializan en el orden dado las siguientes funciones:

- Pantalla y consola: Se hace primero para poder ver cualquier error en los siguientes pasos.

[2] `http://en.wikipedia.org/wiki/Multiboot_Specification`
[3] Ubicado en kernel/arch/x86/generic/init/boot.s
[4] Ubicado en kernel/arch/x86/generic/init/kernel_entry.c

- Tablas globales de descripción (GDT): Indican al procesador como reaccionar antes las distintas interrupciones (a que dirección saltar).
- Programmable Interrupt Counter (PIT): Configura el procesador para que llame a determinada función cada cierta frecuencia fija.
- Page Memory Management (PMM): Ver sección 2.6.
- Teclado: Ver sección 2.5.

2.3.3. kmain[5]

En esta función inicia el sistema virtual de archivos(VFS) y una pequeña librería de timers que no fue usada en el proyecto.

2.4. Pantalla

Se aprovecho la información brindada por GRUB. La misma nos provee un puntero a un buffer lineal de la pantalla, así como sus dimensiones y profundidad (bits por pixel). En la inicialización se calcula cuantos carácteres entran por línea y se configura la manera adecuada para escribir un pixel (dependiendo de la profundidad), entre otras cosas. Teniendo esto se implementaron diferentes funciones para modificar la pantalla (escribir una imagen, un carácter, hacer un clear, *etc*)[6].
El emulador obtiene el puntero al buffer lineal y va modificandolo con ayuda de la función setpixel, configurada previamente.
Respecto a imprimir texto, las funciones involucradas en el proceso se pueden ver en los siguientes diagramas:

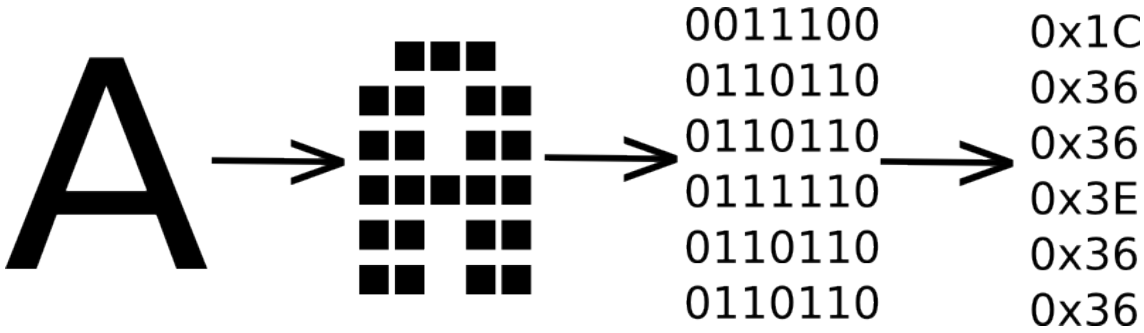
printf parsea la entrada, y por cada caracter que tiene que escribir llama a putchar. putchar calcula en que posición del buffer tiene que ir el carácter y llama a texmode_write, que guarda finalmente el carácter con la información del color en el buffer.

[5]Ubicado en kernel/kmain.c
[6]kernel/graphics/video.c

Este carácter no se muestra hasta que se llama a video_flush_console, el cual vuelca el buffer de la consola a pantalla, carácter por carácter.

2.4.1. Fuente

Para escribir un carácter en pantalla, se guardó para cada uno de los posibles carácteres la información para dibujarlo.



Este fue el procedimiento para generar la fuente de 16x16 de la consola, el cual nos evita tener que realizar a mano cada carácter.

1. Se descargó de internet la fuente deseada, inspirada en la que usaban los juegos de la consola (formato ttf)
2. Se creó una imagen conteniendo todos los carácteres en una cuadricula (formato xfc)
3. Se exportó la imagen a formado RAW (extensión .data)
4. Se hizo makearray.cpp[7], que carga la imagen y para cada fila calcula una máscara de bits de sus pixeles y las muestra como un arreglo de C.
5. Se agregó el mismo al codigo fuente del kernel[8].

[7] en utils/fuentes
[8] graphics/rsrc/nesfont.h, usando en graphics/video.c

2.5. Teclado

Para detectar cuando se presiona una tecla, se configuró la interrupción IRQ1 para que llame a la función kbd_handler[9]. En la misma se lee del puerto 0x60 para obtener el scancode que contiene información sobre si se soltó o presionó una tecla y cual fue. La función getchar que proporciona el kernel se implementó usando una arreglo circular como buffer. Sin embargo esta función no es utilizada por el emulador ya que sólo necesita saber el estado de los controles en un momento dado.

2.6. Administración de Memoria

En primer lugar, se dividió la memoria disponible en páginas de 4KB cada una. Se decidió utilizar la librería liballoc, cuyo funcionamiento se explica más adelante. La librería requiere que implementemos dos funciones: liballoc_alloc y liballoc_free[10].
liballoc_alloc recibe la cantidad de páginas contiguas que se necesitan reservar y debe devolver un puntero a la primer página.
liballoc_free es para liberar las páginas que reservamos anteriormente, recibe el puntero que habíamos devuelto y de alguna forma deberíamos marcar esas páginas como libres para ser reutilizadas.
Por cada página se guarda un bit, para saber si esa página está reservada (1) o libre (0). Como en cada entero podemos guardar 32 bits, podemos empaquetar todos los bits en $\lceil \frac{x}{32} \rceil = \lfloor \frac{x+31}{32} \rfloor$ enteros, donde x es la cantidad de páginas. Utilizando máscaras de bits podemos realizar las operaciones necesarias más rápidamente. En particular se necesitaron tres funciones auxiliares[11].

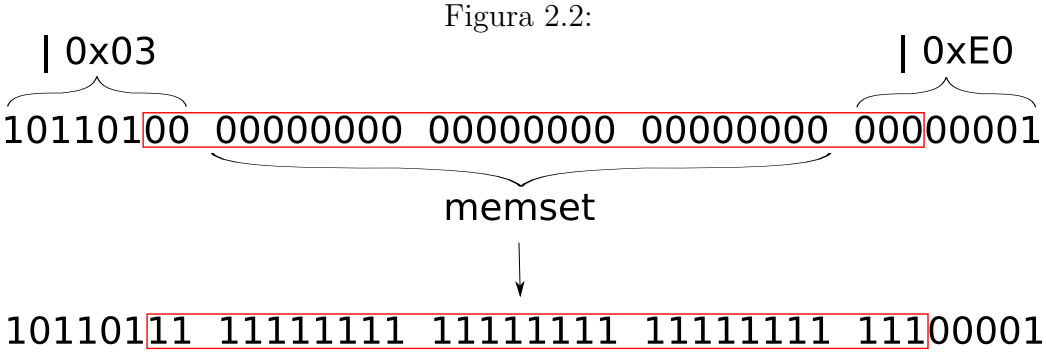
- bitmap_set(l, r): Dado un intervalo en el bitset, setear todos los bits que estan dentro de él. Es usada para marcar las páginas como ocupadas.
- bitmap_clear(l, r): Análogo a la anterior, pero para limpiar los bits.
- fits_inside(x, v): Busca dentro de un entero v (32 bits) si hay x bits libres contiguos.

Por ejemplo, para implementar bitmap_set, primero se activaron los bits de los extremos usando máscaras de bits y luego los restantes mediante una

[9] kernel/arch/x86/generic/drivers/keyboard.c
[10] implementadas en kernel/arch/x86/pmm.c
[11] implementadas en kernel/arch/x86/pmm.c

lamada a memset. Se muestra un ejemplo en la figura 2.2 (usando 8 bits en lugar de 32).

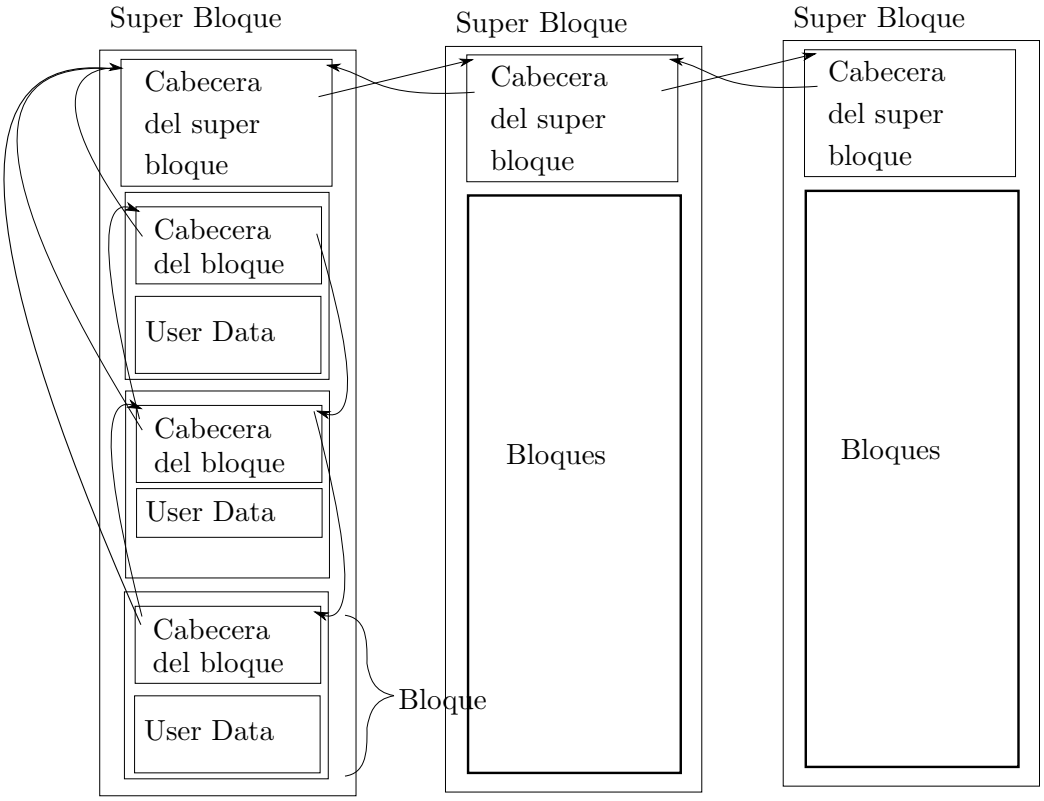
Figura 2.2:



2.6.1. liballoc

Internamente para cada conjunto de páginas reservadas se crea un 'super bloque'. Cuando se llama a malloc se busca un super bloque con la suficiente memoria libre y se crea un bloque dentro de él, que va a contener la memoria que se va a otorgar al llamante.
Un dato interesante es que como estamos implementando malloc no podríamos pedir más memoria dinámica para guardar información sobre los bloques. La solución a esto fue guardar esta información directamente al comienzo de cada bloque. Para recorrer estas estructuras se usan listas enlazadas, los punteros al siguiente/anterior tambień se encuentran en las cabeceras de cada bloque, como se puede apreciar en la figura 2.3.

Figura 2.3:



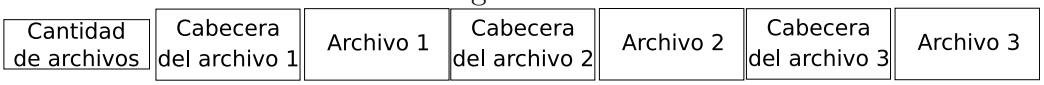
Otra ventaja de este sistema es que uando se recibe un puntero a la User Data, se puede acceder a su cabecera y obtener la información del bloque al que pertenece.

2.7. Sistema virtual de archivos

Necesitamos poder cargar los juegos desde algún lugar para poder emularlos. Como leer de discos puede ser relativamente complejo, se opto por un initrd[12]. Esto es básicamente un archivo que se carga directamente en la RAM al inicio del sistema. Afortunadamente la carga de este archivo es realizada enteramente por GRUB antes de bootear el kernel. GRUB proporciona un puntero al inicio del archivo en RAM y el kernel se encarga de procesarlo.

[12] `http://en.wikipedia.org/wiki/Initrd`

Figura 2.4:



El formato del initrd que se utilizó se ejemplifica en la figura 2.4. El mismo consta simplemente de los archivos colocados uno tras otro,junto con una cabecera que contiene el tamaño, nombre y un valor para verificar integridad. Para crear este initrd se programó initrd_gen[13], el cual recibe los archivos y crea el la imagen lista para ser cargada por GRUB. En el archivo initrd.c[14] se implementó la inicialización del initrd, junto con sus procedimientos para leer y listar los archivos.

[13] utils/initrd_gen.c
[14] kernel/fs/initrd.c

Capítulo 3

Emulador

Capítulo 4

Bibliografía