Guía de creación de Kernel - TP ARQUI - ITBA

Lpinilla

30 Octubre 2018

Abstract

Este documento es una guía para ayudar en el desarrollo del kernel en el Trabajo Prctico de la materia Arquitectura de las computadoras.

Disclaimer: Este documento no fue revisado por ningún miembro de la cátedra. Puede contener errores. Queda en su propia responsabilidad utilizar el contenido de este documento.

Consideraciones previas

w to enable Las versiones de Ubuntu mayores a 16.04 presentan problemas al querer manipular la IDT, se recomienda utilizar Docker **desde cero.**

Debuggeando QEMU con GDB

Es posible conectar el debugger GDB con QEMU para que poder examinar nuestro SO. Para usarlo al máximo se necesita un manejo avanzado de GDB. Sin embargo, se puede sacar información valiosa sin tener conocimientos avanzados (como ver si se están pasando bien los argumentos entre llamadas a funciones).

Para conectar los dos programas tenemos que primero agregar dos parmetros al comando de ejecución de QEMU: tenemos que agregar los flags *"-s -S"* antes del "*-hda*, esto es para que el QEMU se quede esperando en localhost:1234.

Ahora le tenemos que decir a GDB que se conecte a esa dirección, ejecutamos el programa y simplemente corremos *target remote localhost:1234* y listo, ya están los dos programas conectados.

PIC (Programmable Interface Controller)

Lo primero que hay que entender es que las computadoras tienen 2 PIC. El "MasterPIC" y el "SlavePIC". el slave esta "colgado" del master (cascading), por lo tanto se puede deshabilitar o ignorar lo que venga de l desde el master (IRQ02).
Cada PIC tiene 2 entradas, una para leer los datos y otra para configurarlos, el master esta en 0x21 y 0x20. El slave en 0xA1h y 0xA0 respectivamente.

IRQ

Cada PIC tiene 8 entradas, 1 bit por cada entrada $\longrightarrow$ 2bytes para configurarlos.

Las entradas del IRQ son:

IRQ	Descripcin
MasterPIC	
0	Timer-Tick
1	Keyboard
2	Cascade (used internally by the two PICs. never raised)
3	COM2 (if enabled)
4	COM1 (if enabled)
5	LPT2 (if enabled)
6	Floppy Disk
7	LPT1 / Unreliable "spurious" interrupt (usually)
SlavePIC	
8	RTC
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2-Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

Masking del PIC

Para cambiar la configuración del PIC tenemos que hacer un masking de bits. Para deshabilitar algo, tenemos que ponerlo en 1, en caso contrario esta habilitado.

Supongamos que queremos solamente habilitar el teclado, para eso habra que solamente dejar en 0 el bit 1 del MasterPic. Para eso debemos encontrar el valor en hexa que deja al primer byte del pic en 1101.

Recordando que cada PIC tiene 2 bytes para configurarlo, tenemos que hallar el valor.

En este caso, queremos que este todo en 1 menos el anteúltimo bit: 1111-1101. La primer parte corresponde al valor F mientras que la segunda al D. Por lo tanto, el valor a mandarle al MasterPIC es FD.

```
picMasterMask:
push rbp
mov rbp, rsp
mov ax, di
out      21h, al
pop rbp
retn
```

Análogamente configuramos el slavePIC con el valor FF para no habilitar ninguna otra interrupcion.

Interrupciones de Hardware

Recordando que una interrupción es un evento externo que ocurre, veamos como podemos realizar driver de un periférico específico (el teclado).

Cadena de ejecución de interrupciones

Veamos como resuelve la pc cuando se interactúa con el teclado.

1. Se aprieta una tecla en el teclado.
2. El teclado envía una interrupción al PIC.
3. El PIC recibe la interrupción desde el IRQ01 y se fija si la deja pasar o no.
4. Si la deja pasar, le indica al procesador que tiene una interrupción.
5. El procesador le indica si esta listo o no para recibir interrupciones.
6. Si esta listo, le informa al PCI. Este le envía cual de sus interrupciones se activó.
7. El microprocesador con esa información va a buscar a la IDT el registro correspondiente a la interrupción IRQ01.
8. El microprocesador le indica al PIC que la interrupción terminó.

En la IDT, el PIC está mapeado directamente, osea que IRQ0 arranca en X0h, IRQ1 en X1h, ... donde X en principio es 0 Pero acá hay un problema ya que las primeras 32 entradas de la IDT son excepciones por lo tanto se pisarían las entradas, por eso, se "mueve" el inicio de las IRQ. En este caso sería simplemente que X valga 2.

Por lo tanto, la tabla de los IRQ arranca en 20 (32 en hexa).

Creación de Interrupciones

Para crear un driver tenemos que manejar las interrupciones del periférico (ej Teclado). Para eso, tenemos que hacer un par de cosas:

1. Crear la entrada en la IDT.
2. En la entrada de la IDT, asignar un puntero a función que va a ser la rutina de ejecución de la interrupción.

3. De ser necesario, llamar desde la rutina a una función en C.

Por ejemplo: Se crea la interrupción en la entrada 21h. $\longrightarrow$ la interrupción llama a la rutina de asm $\longrightarrow$ la rutina llama a una función de C que se encarga de interactuar con la lectura del teclado.

Hay que entender que la rutina apuntada por la IDT no es lo que finalmente será lo que lee del teclado, esta rutina llama a una función de C que se va a encargar de eso y de proveer más funcionalidades.

Excepciones

Las excepciones son interrupciones que lanza el mismo procesador, por eso, se crean de la misma manera que las interrupciones.

Esta es una tabla con las excepciones más comunes.

Código	Decripción
0h	Division By zero
1h	Single-step interrupt (see trap flag)
2h	NMI
3h	Breakpoint (callable by the special 1-byte instruction 0xCC, used by debuggers)
4h	Overflow
5h	Bounds
6h	Invalid Opcode
7h	Coprocessor not available
8h	Double Fault
9h	Coprocessor Segment Overrun (386 or earlier only)
Ah	Invalid Task State Segment
Bh	Segment not present
Ch	Stack Fault
Dh	General Protection Fault
Eh	Page Fault
Fh	reserved
10h	Math Fault
11h	Alignment Check
12h	Machine Check
13h	SIMD Floating-Point Exception
14h	Virtualization Exception
15h	Control Protection Exception

Teclado

Una vez que se pueda leer del teclado y mostrar lo que se aprieta, podemos ver que las teclas no corresponden con lo que se escribe, esto es debido a que falta hacer un mapeo 1:1 con el teclado y sus respectivos valores.

Para resolver esto simplemente tenemos que tener el mapa de teclado y mapear las teclas antes de devolverlas.

Entradas comunes

Entradas más utilizadas cuando se lee en asm.

Entradas	Dispositivo
20-21h	MasterPIC
A0-A1h	SlavePIC
60-64h	Keyboard
70-71h	RTC
0xB6h	PCSpeaker

Pasaje de Parámetros en 64 bits

Este es el orden en el que se pasan

1. rdi.
2. rsi.
3. rdx.
4. rcx.
5. r8.
6. r9.

Modo de video

El modo de video es una forma distinta de leer la pantalla por parte del driver del monitor.

El estándar VGA-VESA nos permite varios modos de video para elegir, los distintos estándares suelen estar compuesto por lo siguiente:

1. Aspect Ratio.
2. Display Size.
3. Display Resolution.
4. Color depth.
5. Refresh Rate

Cada estándar distinto provee distintas opciones para estos items.

Para el TP, como tenemos que hacer un juego, voy a elegir el "Modo 13h" que fue bastante utilizado en videojuegos en su época por tener un fácil acceso a memoria ("Chunky Graphics").

Siguiendo una documentación en internet, para cambiar el modo el modo 13, tengo que poner AH en 00h y en AL el modo que quiero, en este caso AX quedaría: 0x0013h

Para entrar en modo video tengo que cambiar el bit "cfg_vesa" a 1 en Pure64/src/sysvar.asm

En el mismo archivo podemos encontrar toda la información de VESA bajo el comentario del mismo nombre.

Para cambiar el modo, hay que hard-codear la instrucción en asm dentro de Pure64/src/Pure64.asm simplemente basta con poner:

```
;cambiar el modo 13h
mov ax, 0x13
int 0x10
```

La resolución de este modo es 320x200px. Es importante recordar el número 320 (o 3200) ya que lo podemos llegar a usar más adelante.

Una idea para testear: El cambio de modo lo hacemos con la interrupción 10h del BIOS que después nuestra IDT pisa.. si se puede cambiar de lugar el puntero de la rutina de ejecución de la entrada 10 a otra zona de memoria, creando "mi propia" excepción, podría cambiar de modo texto a modo video cuando quiera.

La pantalla generalmente puede ser descripta por los siguientes valores:

Width	Cuantos pixels hay en una linea horizontal
Height	Cuantas lineas horizontales hay
Pitch	Cuantos bytes en VRAM hay que saltear para ir un pixel abajo
Depth	Cuantos pixeles de colores se tiene
PixelWidth	Cuantos bytes en VRAM hay que saltear para ir un pixel a la derecha

Conocimiento General

Escribir al monitor es símplemente escribir de una forma específica a una dirección de memoria. En este caso la dirección de memoria es 0xB8000 (sacada del manual de Pure64).

La forma en la que tenemos que escribir es B,R,G con las letras que corresponden a Blue, Red y Green. Si te preguntas por qué ese formato y no el

más conocido (rgb) es por la forma de "decodificar" el color como un número. El color originalmente es un número que tiene todo junto sus valores de rgb, por ejemplo el número puede que tenga esta forma 0x255255255. En este caso estarámos metiendo el color (255,255,255) que representa al blanco.

La forma en la que compacta el número es por medio de shifteos. Hace "color & 255" para obtener el azul, "(color << 8) & 255" para obtener el verde y "(color << 16) & 255" para obtener el rojo.

Imprimir mis propios caracteres

Para imprimir mis propios caracteres tengo que tener el dibujo de cada caracter, ese dibujo se suele hacer en formato de bitmap. Esto funciona de la siguiente manera:

Teniendo una matriz de datos:

1111
1000
1111

se puede representar la forma del caracter "C". Entonces lo que tenemos que hacer es tener para cada caracter su mapa (bitmap) y hacer algo simple como:

```
for(int i = 0; i < altura_char; i++){
        for(int j = 0; j < ancho_char; j++){
                if( hay un 1){
                        pintar_con_un_color(x,y,color);
                }else{
                        pintar_con_otro_color(x,y,color2);
                }
        }
}
```

Obviamente podemos ver que esto no es muy eficiente ya que maneja pixel a pixel.

Bitmap Encoding

Como se pudieron dar cuenta, lo anterior si bien funciona, puede ocupar mucho espacio en memoria. Entonces lo que podemos hacer es armar un tipo de codificación sencillo para codificar las filas del bitmap (este es uno de varios métodos para codificar).

Sabiendo que en realidad un char (estándar) ocupa 8 bits de largo y 16 bits de algo, eso significa que tenemos 16 filas de 8 bits (o 1 byte). Entonces si podemos codificar cada fila ya sería mejor.

Siguiendo con el ejemplo anterior de la "C" (para simplificar el ejemplo, nuestro char sería de 4x3). Podríamos codificar cada fila por el valor numérico que representan esos bits, en este caso sería:
0xF
0x8
0xF
Entonces de esta manera tenemos codificado el bitmap.

Optimización

Verificar de a 1 bit a la vez e imprimir 1 pixel es algo sumamente ineficiente, más sabiendo que cada char ocupa 1 byte de largo, entonces nos podemos preguntar: Por que no ir copiando directamente de a bytes?

Esto se puede hacer pero es mucho más difícil ya que se tendría que "empaquetar" (shifteando bits) el color deseado para que se imprima en todos los bits correspondientes en los lugares que correspondan.

Programación de videojuegos en bajo nivel

En nuestra consigna de TP nos tocó realizar el juego PONG pero las técnicas que vamos a explicar sirven otros tipos de juegos también.

Antes (70's - 80's) los juegos se hacían en Assembler porque las computadoras (y después las consolas) no podían procesar algo de alto nivel como C o no era conveniente (solo había 40kb de RAM) , a medida que mejoraron las tecnologías, les permitieron a los desarrolladores poder programar en C pero cuando debían realizar alguna configuracíon como por ejemplo cambiar el color de texto, tenían que cambiar parámetros mediante rutinas en Assembler.

Con el TP tenemos un caso parecido, podemos programar toda la lógica en C pero para poder graficar tenemos que hacerlo mediante syscalls (o creando una librería que se maneja con las syscalls) para no tocar la memoria directamente.

Las distintas "partes" de un juego de este estilo son

1. Ciclos del juego.
2. Lógica del juego.
3. Parte gráfica del juego.

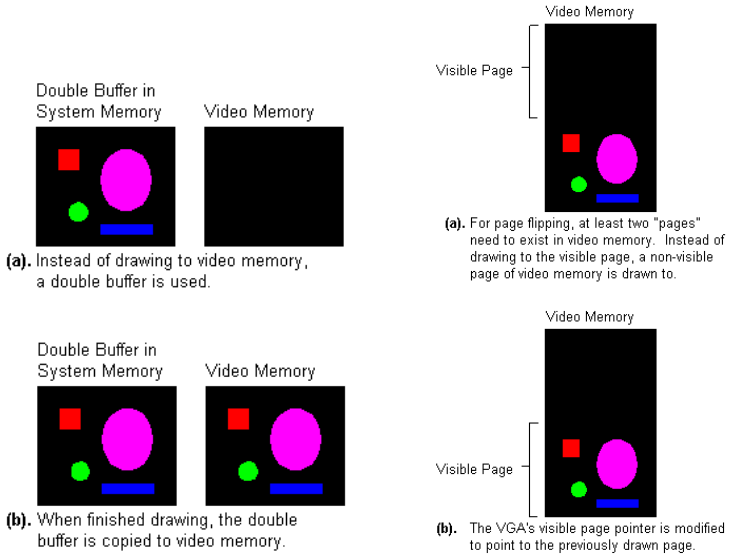
El o los ciclos del juego es el ciclo principal que va a estar corriendo mientras se esté ejecutando el programa. Adentro de este ciclo puede existir una máquina

de estados para guiar al jugador mientras se ejecute ("menú", "en juego", "fin del juego").

La lógica del juego es donde va a estar toda la programación de la jugabilidad, en nuestro caso, el puntaje, el control de las teclas para jugar y la "física" de la pelotita que rebota.

La parte gráfica es la que se encarga de dibujar todo. En bajo nivel, dependiendo el juego, esto podría a llegar a ser más complicado que la lógica del juego ya que tenemos que recordar que no tenemos forma de "pintar" cosas más que llamando a las syscalls y recordando que si cambiamos lo que el usuario esta viendo mientras lo esta viendo, puede generar errores conocidos como "tearing" o "flickering".

Double Buffering o Page Flipping



(a) Double Buffering (b) Page Flipping

Figure 1: Overall caption

El Double Buffering y el Page Flipping son dos técnicas conocidas que se encargan de hacer lo mismo de manera distinta. La idea es que en vez de modificar la pantalla mientras se esta mostrando, se utilice una zona de memoria auxiliar para que sería nuestra "segunda" pantalla.

En Page Flipping, si la memoria es lo suficientemente grande, podríamos llegar a tener parte de la memoria que el monitor no esta "viendo", entonces cuando este se llena o queremos cambiarlo, podemos decirle que "mire" a la otra parte que cambiamos.

Nosotros vamos a implementar Double Buffering porque VGA Mode 13h no tiene espacio suficiente asignado para "más de una pantalla", así que lo primero que hacemos es declarar un array de char[] del tamao de la resolución (en nuestro caso, 320 * 200) que va a actuar como nuestro segundo buffer o "shadow_buffer".

Dependiendo de las especificaciones, podemos hacer las cosas más o menos eficiente. Mode 13h no nos deja simplemente cambiar el puntero a su dirección de memoria así que tenemos uqe copiar todo el buffer cuando queremos "cambiarlos". Con Page Flipping habría que ver si el modo nos deja cambiar el puntero o hay que también copiar de una zona de memoria a otra.

Colisiones en videojuegos 2D

Las colisiones en 2D suelen ser bastante simples, depende de los objetos a colisionar.

Si el juego utiliza matrices de posiciones

Si utilizan una matriz de posiciones internamente en el backend, podría a llegar a ser tan sencillo como fijarse si la posición i-j que corresponde con la pelota, está sobre la posición i-j que corresponde con la barra.

Si el juego no utiliza matrices de posiciones

La idea principal es entender que un objeto (en el caso del pong, la pelota y las barras del jugador) tienen un punto que es a partir de donde las dibujamos. Por ejemplo, podemos tener que la pelota tenga una posición x = .. e y = .. ; a partir de esto podemos hacer algo como

```
draw_filled_circle(int x, int y, int size){
        ...
}
```

Suponiendo que la función "*draw_filled_circle*" dibuja un círculo a partir de los puntos x e y.

Con esto en mente, para ver si dos pelotas chocan, tenemos simplemente que verificar si la distancia que hay entre las pelotas es menor al radio de estas. (*Nota:* Esta comparación se tiene que hacer todo el tiempo si el juego es "a tiempo real").

Volviendo al caso del PONG, tenemos que verificar si la distancia entre la pelota esta en el "área" de la barra, esto quiere decir si su posición x

esta entre barra.x y barra.x+ancho de barra y si ball.y esta entre barra.y y barra.y+$altura\_de\_la\_barra$.

Estas cuentas pueden variar, depende mucho de donde esta ubicado del punto a partir de donde se dibuja el objeto, si está en el centro del objeto, si está en la esquina superior izq, derecha, etc.