DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

System Programming - Kernel 64-Bit

Organización del Computador II

Integrante	LU	Correo electrónico
Facundo Linari	591/16	facundo.linari@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

El objetivo del presente trabajo es implementar un sistema mínimo que permita correr un máximo de 32 tareas concurrentemente a nivel de usuario, usando los lenguajes C/ASM para la arquitectura x86-64 de Intel. Además, el sistema será capaz de captar cualquier excepción que alguna de estas tareas genere y quitarla.

En las próximas secciones se detallarán los módulos implementados para levar a cabo dicho sistema. El resultado final será un kernel multitarea perfectamente funcional y capaz de correr tareas que están en memoria de manera estática. La implementación actual solo posee un máximo de 4 tareas diferentes pero fácilmente extendible.

Para este trabajo se utilizará como entorno de pruebas el programa Bochs. El mismo permite simular una computadora IBM-PC compatible desde el inicio, y realizar tareas de debugging. Todo el código provisto para la realización del presente trabajo está ideado para correr en Bochs de forma sencilla.

1.1. Instrucciones de uso

Para utilizar el programa se necesita tener instalado Bochs-2.6.9[1] con x86-64 habilitado.
Para ejecutar el kernel utilizar el *Makefile* cumpliendo los requerimientos[2].

El juego para probar el sistema implementado consiste en crear bacterias que se mueven libremente y colocar comida.

C: Crear bacteria.(Máximo 32)

F: Crear comida en posición del cursor.(Máximo 32)

WASD: Mover cursor.

Flechas: Modificar vida inicial de las bacterias.

2. Desarrollo

Los archivos que conforman este trabajo práctico se encuentran en la carpeta *src* y son los siguientes:

- *Makefile* - encargado de compilar, generar y correr el *floppy disk*.
- *bochsrc* y *bochsdbg* - configuración para inicializar Bochs.
- *boot.asm* - bootloader.
- *kernel.asm* - código de inicialización del kernel, estructuras y tarea inicial.
- *defines.h* - constantes y definiciones.
- *syscalls.h* - interfaz a utilizar en C para los llamados al sistema.
- *i386.h* - funciones auxiliares para utilizar assembly desde C.
- *idt.h* y *idt.c* - función para completar entradas en la IDT.
- *game.h* y *game.c* - implementación de los llamados al sistema.
- *tss.h* y *tss.c* - función para inicializar TSS.
- *isr.h*, *isr.mac* y *isr.asm* - definiciones de las rutinas para atender interrupciones.
- *sched.h* y *sched.c* - rutinas asociadas al scheduler.

[1] Se puede obtener de `https://sourceforge.net/projects/bochs/files/bochs/2.6.9/`
[2] La regla *run* asume que Bochs está agregado al *PATH* del usuario

- *mmu.h* y *mmu.c* - rutinas asociadas a la administración de memoria.
- *screen.h* y *screen.c* - rutinas para pintar la pantalla.
- *a20.asm* - rutinas para habilitar y deshabilitar A20.
- *pic.c* y *pic.h* - funciones habilitar pic, deshabilitar pic, fin intr pic1 y resetear pic.
- *master.c*, *manager.c*, *player.c* y *enemy.c* - código de las tareas.

A continuación se detallará cómo fueron implementadas las partes más importantes.

2.1. Bootloader

El bootloader simplemente copia[3] el código del kernel a partir de la dirección 0x1000 y las tareas a partir de 0x8000 y luego salta a la dirección 0x1000 para comenzar a correr el kernel. Esto se puede ver en la figura 1.

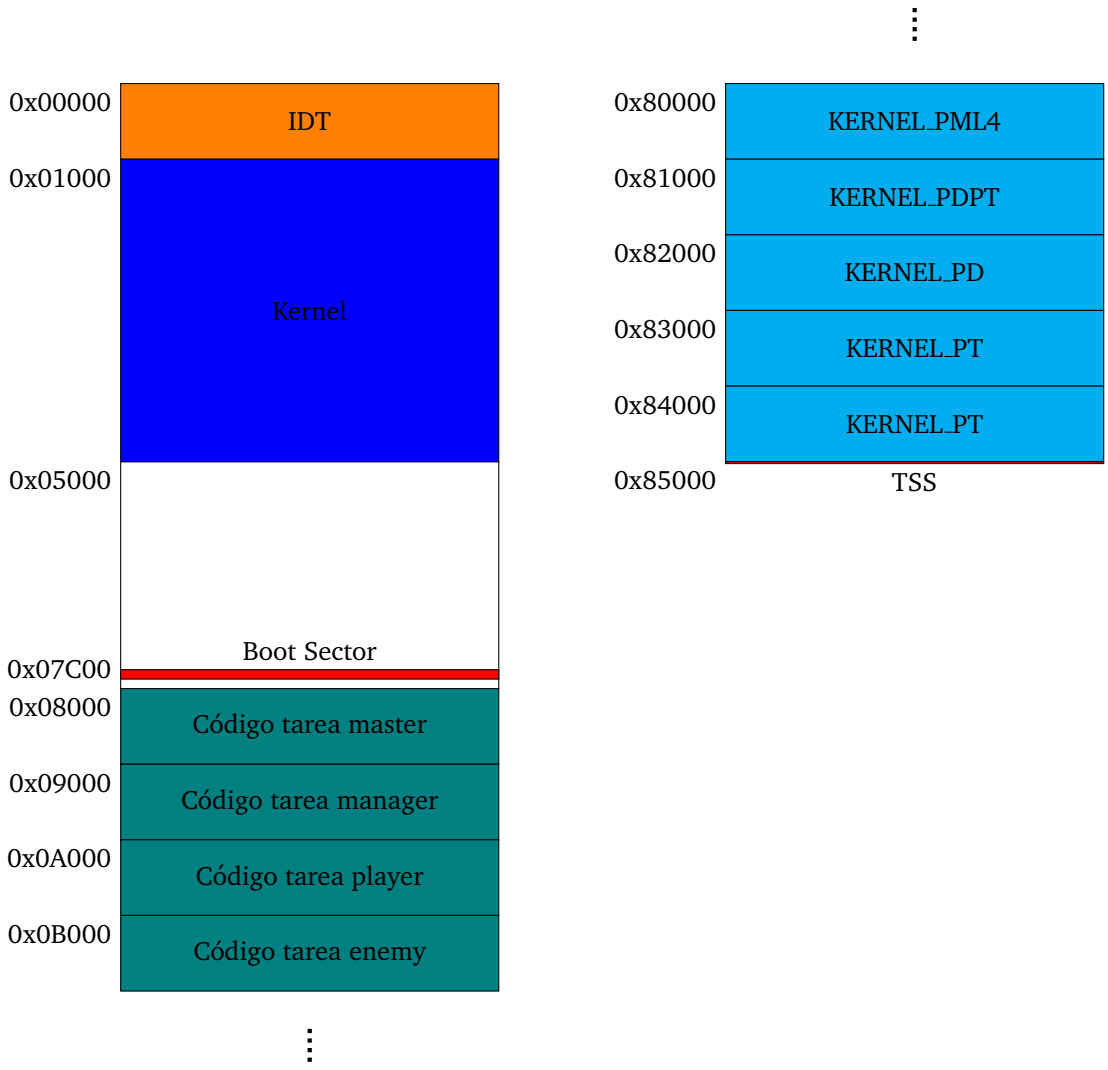

Figura 1: Mapa de la organización de la memoria física del kernel.

Se decidió separar el código del kernel y de las tareas para evitar pisar al **Boot Sector** que es donde se llama a la rutina de acceso al disco.

[3]Utilizando la BIOS con la interrupción 0x13

2.2. GDT

Como solo se van a utilizar los anillos 0 y 3 de privilegio, alcanza con 4 descriptores de segmento (2 de código y 2 de datos). También se tiene un descriptor de código de 32 bits para el pasaje a modo protegido y luego llegar a modo largo. Finalmente se tiene un único descriptor de TSS.

Índice	Límite	Base	Tipo	S	DPL	P	AVL	L	DB	G
0	0	0	0	0	0	0	0	0	0	0
1	0xFFFFF	0x00000000	0xA	1	0	1	0	0	1	1
2	0x00000	0x00000000	0xA	1	0	1	0	1	0	0
3	0x00000	0x00000000	0xA	1	3	1	0	1	0	0
4	0xFFFFF	0x00000000	0x2	1	0	1	0	0	1	1
5	0x00000	0x00000000	0x2	1	3	1	0	0	0	0
6	0x00067	0x00085000	0x9	0	0	1	0	0	0	0

Cuadro 1: Descriptores en la GDT.

Se puede encontrar definida a la GDT en *kernel.asm* en la ”Sección de datos”.

2.3. IDT

La función **inicializar_idt** se encarga de inicializar toda la tabla de descriptores de interrupción a partir de la dirección 0x00000, eliminando la IVT colocada por la BIOS.

Las primeras 21(0-20) entradas de la IDT se llenan con puertas de interrupción para poder captar todas las excepciones del procesador con DPL 0 porque no es deseable que una tarea nivel usuario pueda llamarlas. En la figura **2** se puede observar un ejemplo de los descriptores puestos en la IDT.

Reservado

Offset 32-63
(isrn_physical_address >> 32) and 0xFFFF

Offset 16-31	P	DPL		TYPE				IST
(isrn_physical_address >> 16) and 0xFFFF	1	0 0	0	1 1 1 0	0 0 0	0	0	0 0 0

Selector de segmento	Offset 0-15
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0	(isrn_physical_address) and 0xFFFF

Figura 2: Puerta de interrupción para excepción **n**.

Además de las excepciones, se tienen las siguientes puertas de interrupción de 64 bit con la información de la tabla **2**:

Índice	Offset	Selector de segmento	DPL	P	IST	Atiende
32	& isr32	0x10	0	1	0	Reloj
33	& isr33	0x10	0	1	0	Teclado
255	& isr255	0x10	3	1	0	Syscall

Cuadro 2: Descriptores de reloj, teclado y syscall en la IDT.

Notar que no se utiliza el mecanismo IST[4], ya que es más sencillo tener solo una pila nivel 0 por tarea.

[4]Interrupt Stack Table

2.4. ISR

2.4.1. Reloj

Guarda el contexto de la tarea actual en su PCB correspondiente, llama a **next** para pasar a la siguiente tarea y carga el PCB de dicha tarea. También carga los selectores de segmento que corresponda.

2.4.2. Teclado

Para el manejo del teclado se cuenta con un arreglo de 256 bytes para guardar los scan codes de las teclas oprimidas. Luego, esta interrupción llama a **add_key** para que luego una tarea llame a **get_key** y obtener el scan code agregado. Dicho arreglo se maneja como una cola, con un índice de inicio y otro de fin.

2.4.3. Syscall

Llama a **manage_syscall** pasando los parámetros correspondientes (ver sección 2.7) y retorna el resultado por rax, el cual es el único registro que altera.

2.4.4. Excepción

Las rutinas de atención de excepción desalojan (ver sección 2.6.2) a la tarea que generó la excepción y esperan a la interrupción de reloj para cambiar de tarea.

2.5. MMU

Todas las tareas poseen el mismo tamaño y es fijo. Este es 8 marcos de pagina del area del kernel(ver figura 4) y 2 del area de tareas(ver figura 5).

Por lo tanto, el área total de las tareas es un mega del área del kernel y 256 KB del área de tareas.

Al tener fija la memoria ocupada por cada tarea con un simple bitmap se puede administrar la memoria.

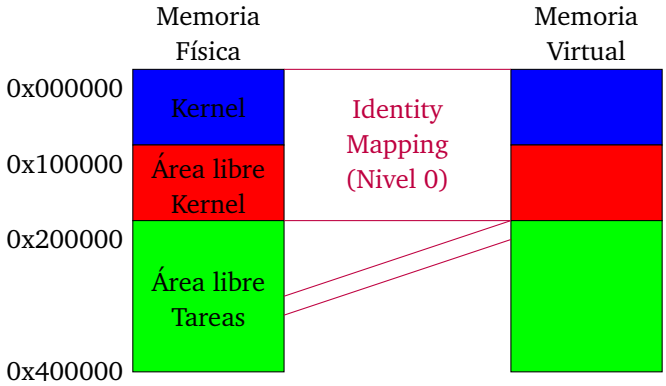

Figura 3: Mapa de memoria de la tarea.

Para mappear/desmappear se requiere la dirección de la pml4 y se agregan las estructuras necesarias secuencialmente de la siguiente forma:

```
PML4 - PDPT0 - PD0 - PT0 - PT1 - PT2 - ... - PT511 - PD1 - PT512 - PT513 - ...
```

Cabe destacar que hay que usar hasta PT2 para no pisar memoria destinada a otro fin.

2.5.1. Haciendo zoom en el área de una tarea específica:

En el área libre del kernel están las estructuras de paginación (5 marcos de página), el PCB y la pila nivel 0.

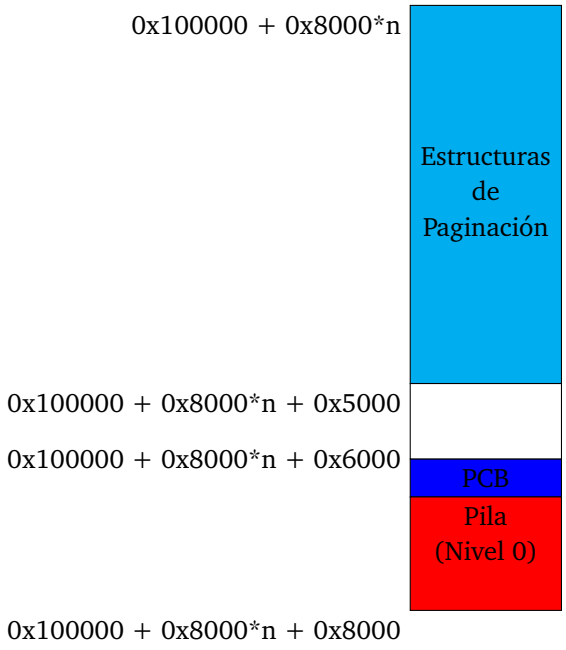

Figura 4: Memoria de la tarea **n** en área del kernel.

El área de tareas consiste en 2 marcos de página: uno para su código y el otro para datos y pila.

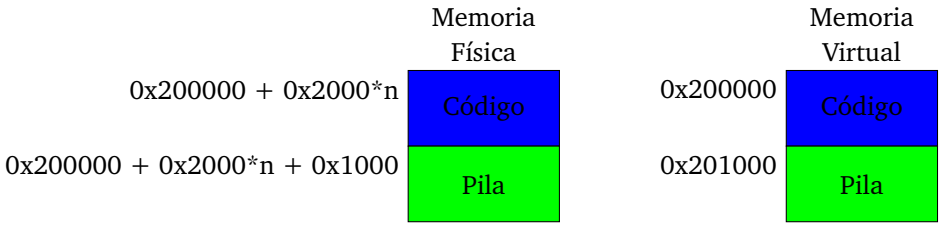

Figura 5: Memoria de la tarea **n** en área de tareas.

2.6. Scheduler

El scheduler está compuesto por un arreglo de 32 *pi*[5] que itera secuencialmente con una política *Round-robin*. Un *pi* tiene toda la información ligada a una tarea.
Esto es:

- El estado actual: -1: Free, 0: Ready, 1: Running, 2: New y 3: Blocked
- El tipo de tarea: 0: Master, 1: Manager, 2: Player, 3: Enemy
- Cantidad de turnos que corrió (para debugging)
- Permisos: ver sección 2.8
- Buzón: para comunicación entre procesos

El ID de un proceso está dado por la posición en dicho arreglo y del ID se puede obtener su **cr3** y **PCB**.

[5] process info

2.6.1. PCB

Para el cambio de tarea es necesario guardar el contexto. Para esto, se utiliza el process control block (PCB) cuyo contenido se puede ver en la figura 6. No hay necesidad de guardar los selectores de segmento ya que se pueden inferir por el nivel de privilegio en el cual corre la tarea.

rip	136
rflags	128
r15	120
r14	112
r13	104
r12	96
r11	88
r10	80
r9	72
r8	64
rsp	56
rbp	48
rdi	40
rsi	32
rdx	24
rcx	16
rbx	8
rax	0

Figura 6: Process Control Block.

2.6.2. Inicialización y eliminación de tarea

La función **start_task** crea las estructuras de paginación en su área del kernel, copia el código de la tarea e inicializa un proceso. Para facilitar los mapeos se utiliza el **cr3** del kernel que tiene identity mapping de los primeros 4 MB. Esto permite ahorrar desmapeos al momento de copiar el código de la tarea.

Para eliminar una tarea se coloca el estado Free en el *pi* correspondiente.

2.7. Syscalls

Las tareas pueden utilizar la interrupción 255 para hacer llamados a sistema (ver sección 2.3). En la tabla 3 se detallan todos los servicio del sistema disponibles. El pasaje de parámetros a los servicios del sistema sigue la convención C. Se pasa rdi para el número de servicio y rsi, rdx y rcx para los parámetros 1, 2 y 3 respectivamente. El resultado retorna en rax.

Número	Tipo	Parámetro 1	Parámetro 2	Parámetro 3	Descripción
1	*exit*	-	-	-	Finalizar proceso (genera una excepción para que sea desalojado)
2	*create task*	tipo	permisos	mensaje inicial	Crear tarea del tipo, permisos y mensaje inicial deseado
3	*read*	-	-	-	Leer mensaje y eliminarlo
4	*write*	tarea	mensaje	-	Mandar mensaje a la tarea deseada en caso de haber espacio disponible
5	*unblock*	tarea	-	-	Desbloquear tarea deseada
6	*block*	tarea	-	-	Bloquear tarea deseada
7	*print*	modo	mensaje	posición y atributos	Imprimir en pantalla
8	*get id*	-	-	-	Obtener ID del proceso
9	*get key*	-	-	-	Obtener una tecla presionada

Cuadro 3: Syscalls.

2.7.1. Print

Mediante la BIOS se configuró el video a modo texto 80x50 con color.
El llamado al sistema print varía el formato de su segundo parámetro dependiendo del modo:

1. *Texto:* puntero a arreglo de *char* con el mensaje que se quiere imprimir.
2. *Número decimal:* número que se imprimirá en decimal.
3. *Número hexadecimal:* número que se imprimirá en hexadecimal.
4. *Bloque:* tamaño de bloque y caracter a repetir en forma de bloque. (Ver figura 7)

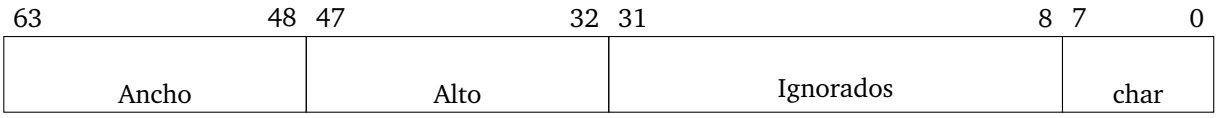

Figura 7: Segundo parámetro de syscall print con modo bloque.

En cambio, el tercer parámetro sigue siempre el mismo formato que se muestra en la figura 8.

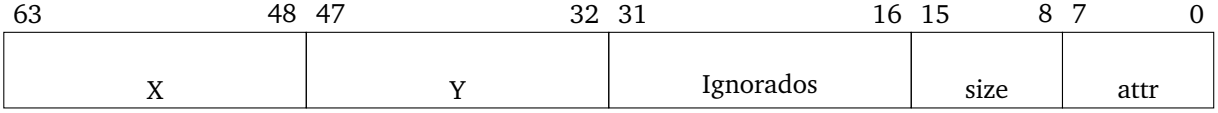

Figura 8: Tercer parámetro de syscall print.

Notar que solo se utiliza **size** al imprimir un número.

2.8. Permisos

En el *pi* del proceso se almacenan los permisos. En la figura 9 se puede ver cómo son almacenados y qué hace cada uno.

15	14	4	3	2	1	0
S	Ignorados	B	W	T	P	K

Figura 9: Almacenamiento de permisos de un proceso.

- S: sistema, corre en nivel 0 en caso de estar activado
- B: block, puede bloquear/desbloquear tareas
- W: write, puede enviar mensajes a tareas
- T: task, puede crear tareas
- P: print, puede imprimir en pantalla
- K: keyboard, tiene acceso al buffer de teclas oprimidas

2.9. Tareas de ejemplo

Las únicas que se utilizan son *master.c* y *player.c*.

La tarea *master.c* es la encargada de recibir las teclas oprimidas, crear las bacterias (instancias de *player.c*), mantener sus contadores de vida, manejar la comida y comunicarse con las bacterias para ver hacia qué dirección se van a mover.

La tarea *player.c* simplemente envía a la tarea padre (*master.c*) la dirección a la cual se quiere mover y termina su ejecución en caso de recibir un mensaje pidiendo eso.

3. Bonus

En la carpeta *Dungeon OS* se encuentran otros códigos de tareas que implementan otro juego que está incompleto por falta de memoria, ya que la tarea *manager.c* requiere más que 4 KB. Este otro juego sirve de ejemplo de comunicación entre procesos sin necesidad de usar el mecanismo de bloqueo. Esta alternativa no es recomendada ya que genera mucho *busy waiting*, pero en este caso se hizo debido a que es un juego por turnos y que no corre varias tareas a la vez.

4. Conclusión

Saltar de 32 a 64 bit no es un gran desafío. Hay que estar atento en el cambio de contexto pues hay que hacerlo por software.

4 KB de memoria para código solo permite generar tareas sencillas.

5. Bibliografía

- Osdev:
  - `https://wiki.osdev.org/Creating_a_64-bit_kernel`
  - `https://wiki.osdev.org/Memory_Map_(x86)`
  - `https://wiki.osdev.org/Text_UI`
  - `https://wiki.osdev.org/ATA_in_x86_RealMode_(BIOS)`

    - https://wiki.osdev.org/InlineAssembly

- http://www.brokenthorn.com/Resources/OSDev5.html
- https://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html
- http://www.ousob.com/ng/asm/index.php
- Manual de Intel