

# Organización del Computador II

Departamento de Computación  
Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

7 de diciembre de 2017

## Trabajo Práctico Número 3

**Jan Michael Vincent**

Integrante	LU	Correo electrónico
Zar Abad, Ciro Román	129/15	ciromanzar@gmail.com
Vercinsky, Iván	141/15	ivan9074@gmail.com
Puterman, Lucas	830/13	lucasputerman@gmail.com

## Índice

<b>1. Ejercicio 1</b>	<b>4</b>
1.1. Completar la GDT . . . . .	4
1.2. Pasar a modo protegido . . . . .	4
<b>2. Ejercicio 2</b>	<b>5</b>
2.1. Interrupt Descriptor Table (IDT) . . . . .	5
<b>3. Ejercicio 3</b>	<b>5</b>
3.1. Inicializar directorio del kernel . . . . .	5
3.2. Activar paginación . . . . .	5
<b>4. Ejercicio 4</b>	<b>6</b>
4.1. Memory Management Unit (MMU) . . . . .	6
4.2. Inicializar directorio de las tareas . . . . .	6
4.3. Mapear páginas . . . . .	6
<b>5. Ejercicio 5</b>	<b>7</b>
5.1. Rutina de atención de reloj . . . . .	7
5.2. Rutina de atención de teclado . . . . .	7
5.3. Rutina de atención de syscall . . . . .	7
<b>6. Ejercicio 6</b>	<b>7</b>
6.1. Task State Segment (TSS) . . . . .	7
6.1.1. Idle . . . . .	7
6.1.2. Tareas . . . . .	8
6.1.3. Salto a tareas manual . . . . .	9
<b>7. Ejercicio 7</b>	<b>9</b>
7.1. Scheduler . . . . .	9
7.1.1. variables . . . . .	9

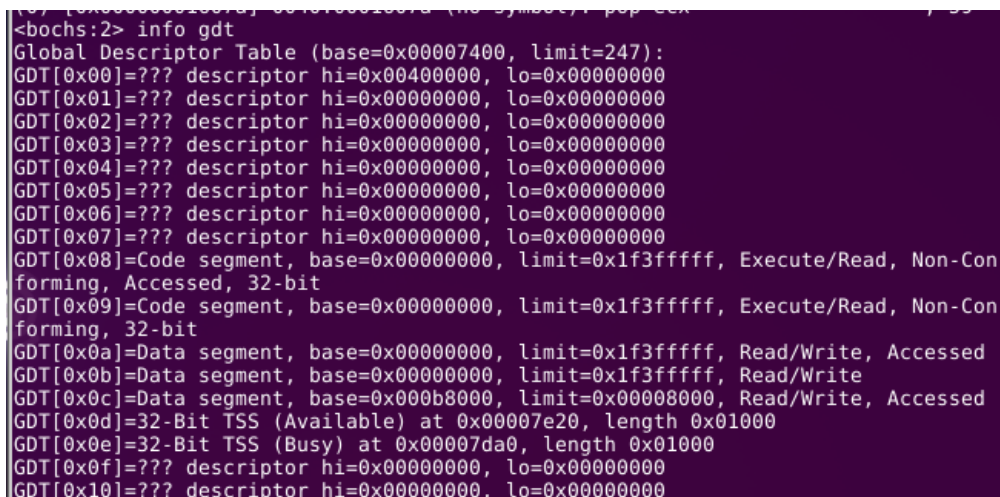
7.1.2. Conmutación . . . . .	10
7.2. Syscalls . . . . .	10
7.3. Juego . . . . .	10
7.4. Final del juego . . . . .	11
7.5. Desalojo de tareas . . . . .	12
7.6. Modo debug . . . . .	12

## 1. Ejercicio 1

### 1.1. Completar la GDT

Como se puede observar en la captura 1, dejamos la primer posición de la Tabla de Descriptores Globales en 0 como lo especifica INTEL, y las siguientes 7 posiciones sin usar. Luego, las posiciones 8, 9, 10 y 11 fueron completadas para representar los segmentos de código de nivel 0 y 3, y los segmentos de datos de nivel 0 y 3.

Para poder direccionar 500MB de memoria, fue necesario setear los bits de granularidad de estos 4 segmentos en 1, y el límite en 500MB - 1, es decir 0x1F3FFFFF.



```

<bochs:2> info gdt
Global Descriptor Table (base=0x00007400, limit=247):
GDT[0x00]=??? descriptor hi=0x00400000, lo=0x00000000
GDT[0x01]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x02]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x03]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x04]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x05]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x06]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x07]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x08]=Code segment, base=0x00000000, limit=0x1f3fffff, Execute/Read, Non-Con
forming, Accessed, 32-bit
GDT[0x09]=Code segment, base=0x00000000, limit=0x1f3fffff, Execute/Read, Non-Con
forming, 32-bit
GDT[0x0a]=Data segment, base=0x00000000, limit=0x1f3fffff, Read/Write, Accessed
GDT[0x0b]=Data segment, base=0x00000000, limit=0x1f3fffff, Read/Write
GDT[0x0c]=Data segment, base=0x000b8000, limit=0x00008000, Read/Write, Accessed
GDT[0x0d]=32-Bit TSS (Available) at 0x00007e20, length 0x01000
GDT[0x0e]=32-Bit TSS (Busy) at 0x00007da0, length 0x01000
GDT[0x0f]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x10]=??? descriptor hi=0x00000000, lo=0x00000000

```

Figura 1: Estado de la GDT luego de iniciar el kernel

### 1.2. Pasar a modo protegido

Para pasar a modo protegido es necesario tener activada la segmentación. Para eso utilizamos la tabla del ejercicio 1.1 y la cargamos en el registro GDTR. Luego prendimos el primer bit del cr0 e hicimos el far jump al segmento de código definido en la GDT.

En *kernel.asm* tuvimos que indicar antes de la etiqueta que usamos para saltar al modo protegido, que las instrucciones iban a estar alineadas a 32 bits, usando la instrucción BITS 32. Esto es así porque le indicamos a la entrada de la GDT que los segmentos serán de 32 bits.

## 2. Ejercicio 2

### 2.1. Interrupt Descriptor Table (IDT)

La función `idt_inicializar()` completa la tabla de interrupciones, utilizando la macro provista por la cátedra. Se completaron las primeras 20 entradas correspondientes a las excepciones del procesador, las interrupciones externas que se mapean a través del PIC: 32 (clock), 33 (teclado) y por último la interrupción de software (syscall) en la entrada 70.

Las rutinas para atender las excepciones son dos, divididas entre las excepciones que generan error code y las que no. En un principio estas rutinas imprimían en pantalla el número de interrupción y el código de error si correspondiese. Luego fueron reemplazadas por tareas más específicas que serán detalladas más adelante.

## 3. Ejercicio 3

### 3.1. Inicializar directorio del kernel

Para inicializar el kernel, debíamos mapear con *identity mapping* los primeros 4MB de memoria.

Para ello hicimos una función que se encarga de mapear una dirección virtual a una física en un `cr3` pasado por parametro.

Esta función se encarga de inicializar las entradas en el directorio de páginas en caso de que no estuviera presente. Para ello debíamos indicar si las páginas iban a ser de usuario/supervisor o escritura/lectura.

### 3.2. Activar paginación

Para activar paginación debíamos tener el esquema precargado y luego debíamos activar paginación levantando en el `cr3` el bit de paginación

## 4. Ejercicio 4

### 4.1. Memory Management Unit (MMU)

Es la unidad que se encarga de la paginación. En el tp la usamos para crear las estructuras de la paginación de las tareas. Mover las tareas por el mapa, copiar el código de las tareas y copiar valores a la pila de cada tarea.

### 4.2. Inicializar directorio de las tareas

El directorio de tareas debía tener mapeado con identity mapping el área del kernel. Para ello utilizamos la función `mmu_mapear_pagina` con los mismos índices de virtual y física con permisos de supervisor y solo lectura.

Luego cada tarea debía mapear la dirección virtual `0x400000` a una posición física en el mapa que dependía de la posición de la tarea. Para ello usamos la posición `x` e `y` para calcular un offset lineal al cual le sumábamos la base física del mapa para obtener la dirección física a la cual debíamos mapear la `0x400000`. En este caso era con permisos de usuario y escritura (Ya que la misma página se iba a usar para la pila de la tarea)

Luego debíamos mapear las posiciones adyacentes a la tarea. Para ellos aplicamos el mismo esquema, en base a la posición `x,y` de la tarea calculábamos la posición física. Pero además calculábamos las posiciones virtuales. Luego mapeamos cada virtual con su correspondiente física. Y, en este caso era con permisos de usuario y solo lectura.

### 4.3. Mapear páginas

Para mapear páginas hubo que descomponer la dirección virtual para acceder a las entradas del directorio y tabla de páginas. Con los primeros 10 bits accedemos a la entrada del directorio, luego con los siguientes 10 a la entrada de la tabla de páginas. Y luego con los últimos 12 a la posición de memoria dentro de la página. En esa posición escribíamos en el campo `address` la dirección física a la cual queríamos mapear, si era de lectura o escritura, si era de supervisor o usuario, etc.

En caso de que alguna entrada de las estructuras no estuviera presente había que inicializarla indicando el tamaño de cada página, los permisos, etc.

## 5. Ejercicio 5

Como mencionamos antes, las interrupciones externas de clock, teclado y software se mapearon en la IDT en las posiciones 32, 33 y 70 respectivamente.

### 5.1. Rutina de atención de reloj

Se escribió la rutina de atención de la interrupción 32 en assembly, para que haga un call a la función `game_tick()`.

Más adelante se fue ampliando para que la interrupción se ignore si el juego esta pausado o terminado, y se reemplazó la llamada a `game_tick()` por `sched_tick()`.

### 5.2. Rutina de atención de teclado

Se escribió la rutina de atención de la interrupción 33 en assembly, de forma que haga un call a la función `screen_pintar_tecla(uint tecla)` que imprime el código ASCII de la tecla en la esquina superior derecha de la pantalla.

Más adelante se fue ampliando de a poco para agregar funcionalidad a las teclas `shift` derecha e izquierda, y a la tecla `y`.

### 5.3. Rutina de atención de syscall

Se escribió la rutina de atención de la interrupción `0x46`, para que reemplace el valor del registro `eax` por `0x42`. Más adelante se modificará para atender el valor de `eax` por `0x42`. Posteriormente este comportamiento va a ser modificado para atender el servicio del sistema.

## 6. Ejercicio 6

### 6.1. Task State Segment (TSS)

#### 6.1.1. Idle

En el archivo `tss.h` se inicializó una instancia de la estructura `tss` llamada `tss_idle`, que se completa llamando a la función `tss_inicializar_idle()`.

Dado que el segmento de esta tarea debe mapear una página de 4KB de memoria, a partir de la posición física 0x16000 hasta 0x16FFF, se seteó al límite en la entrada de la GDT en 0x0FFF con la granularidad en 0 (ya que así el límite indica la última porción de memoria de 1 byte accesible por la tarea) y en la base la ubicación de la tss.

La TSS de la tarea idle se completó en `eip` con la posición virtual en donde comenzará a ejecutarse el código correspondiente, que por estar mapeada con *identity mapping* es igual a la física.

Al estar la pila en la misma página que el resto de los datos, se inicializa `esp` y `ebp` al final de la misma, en 0x1000.

Los `eflags` se inicializan en 0x202, ya que tienen que tener el bit 2 prendido siempre, y el 9 que indica las interrupciones encendidas.

El `cr3` es compartido con el kernel por lo que también es 0x27000.

El `dpl` se setea en 0, ya que la tarea debe correr con el mismo nivel de privilegio que el kernel. Y por supuesto, no olvidamos el bit de presente prendido.

### 6.1.2. Tareas

Las tareas fueron lanzadas atendiendo interrupciones del teclado o a través de la lógica de la syscall `mover`. Para ello debimos calcular la posición física de donde se estaba lanzando la tarea para armar la estructura de paginación.

Una decisión que tomamos fue que todas las tareas del mismo jugador compartieran las mismas páginas físicas de las tablas de páginas. De esta manera cada vez que un explorador mapeaba nuevas posiciones, de manera automática, eran mapeadas por todas las demás tareas del mismo jugador. Porque todas accedían al mismo array de páginas.

O sea, cada tarea tenía su propio directorio de tablas de página en el cual se registraban las entradas para mapear el mapa, las cuales apuntaban a las mismas direcciones físicas donde ya estaba definidas la tabla de páginas.

Luego el `eip` dependía del tipo de pirata que el jugador estaba lanzando. Y por enunciado inicializamos el `esp` y `ebp` en el final de la página del código.

Un detalle que tuvimos que tener en cuenta, fue que las tareas reciben 2 parámetros, para ellos desplazamos el `esp` dos posiciones para arriba, dejando espacio para los parámetros.

Además las tareas tenían una pila de nivel cero que por enunciado debía



estar en una pagina libre dentro de la zona libre del kernel. Para ello, al momento de construir la TSS pediamos una nueva pagina y le sumabamos el offset para que quede al final de la pagina.

### 6.1.3. Salto a tareas manual

Para saltar a una tarea, primero debimos cargarla el descriptor de la TSS en la GDT.

Para construir el descriptor de la TSS, debimos armar un esquema de paginación. Para ello simulamos una posición, un pirata y un jugador. Luego construimos el esquema en base a las definiciones del trabajo práctico, y le asignamos al cr3 la dirección del directorio de páginas.

Luego le indicamos el `eip`, `esp` y `ebp` que correspondia.

Para cargarlo en la GDT, forzamos un indice, que sabiamos estaba libre, y guardamos en esa posicion la información de la TSS.

Luego en el `kernel.asm` hicimos el `jmp` a ese indice de la GDT, indicando el `ti` y el `rpl` ambos en 0.

## 7. Ejercicio 7

### 7.1. Scheduler

La estructura del scheduler consiste en una serie de variables que guardan datos del estado del juego, además, se encarga de agregar y borrar piratas de la conmutación de tareas y de conmutar las tareas mediante la función `sched_proxima_a_ejecutar`

#### 7.1.1. variables

Las variables de `sched` son las siguientes:

- *ultimo\_jugadorA* y *ultimo\_jugadorB*: guarda el índice del último pirata que se ejecutó para cada jugador 0, se incializan out of bounds con `MAX_CANT_PIRATAS_VIVOS`
- *jugador\_actual* guarda el indice del jugador que tiene turno actualmente

- *modo\_debugg*, *juego\_pausado*, *juego\_terminado*: trackean con 0 o 1 si el juego se encuentra en alguno de esos estados

### 7.1.2. Conmutación

Las tareas se conmutan utilizando la función `sched_tick` que es llamada en cada ciclo del clock. Esta llama a `sched_proxima_a_ejecutar` que trabaja de acuerdo a lo pedido por el enunciado, es decir, cambia entre jugadores por cada ciclo de clock (si ambos jugadores tienen piratas) y dentro de cada jugador itera las tareas linealmente. La función `sched_proxima_a_ejecutar` devuelve el índice de la GDT de la tarea. Luego se llama a `game_tick` que actualiza los relojes en la pantalla.

## 7.2. Syscalls

Las syscalls entran por la interrupción 70 que compara el registro `eax` para saber de que syscall se trata. Luego, llama a las funciones de `game.c` que se encargan de ejecutar el código correspondiente a cada una de las syscalls.

## 7.3. Juego

Todo lo relacionado al juego en si es manejado desde `game.c`. El kernel llama a la función `game_inicializar` que inicializa las variables de la clase y los dos jugadores que jugarán al juego.

*game\_inicializar* pide páginas de memoria para cada jugador que serán compartidas por los piratas de cada uno, compartiendo así las direcciones físicas todos los piratas de un mismo equipo. Además setea las direcciones de los puertos de ambos jugadores y su índice.

*game\_jugador\_lanzar\_pirata*: Es la función que se encarga de lanzar nuevos piratas al juego. El pirata solo será lanzado si el jugador tiene espacio disponible para el mismo. Esta función inicializa la estructura del pirata y llama a *tss\_inicializar\_pirata* que se encarga de inicializar la TSS del pirata. Luego, se pinta el pirata en la pantalla y se llama a *sched\_inicializar\_jugador* que agrega el pirata al scheduler.

*game\_matar\_pirata*: Se encarga de eliminar piratas que deben morir en el juego. Esta función se encarga de marcar al pirata como muerto dentro del

arreglo de piratas del jugador, se podría pensar como hacer algo equivalente a poner el bit de presente en 0 en una estructura, ya que solo se va a conmutar a la tarea si el pirata tiene *esta\_vivo* en 1. Por último, la función chequea si queda algún pirata vivo de ese pirata y limpia el reloj en la pantalla.

*game\_syscall\_pirata\_posicion*: Esta función responde a *syscal\_posicin* y se encarga de devolver la posición en el mapa del pirata pasado por parametro codificada como pide el enunciado.

*game\_syscall\_pirata\_cavar*: Esta función responde a la *syscal\_cavar* y es llamada por mineros. La función chequea que en la posición desde donde fue llamada por la tarea haya un tesoro, y que este aún tenga monedas. Si esto es así, extrae una moneda del tesoro y llama a *game\_jugador\_anotar\_punto*. Si en la posición no había tesoro, o había un tesoro sin monedas, se mata a la tarea que ejecutó la syscall.

*game\_syscall\_pirata\_mover*: Esta función responde a la *syscal\_mover* y se encarga de mover a los piratas alrededor del mapa. La función chequea tanto como que el pirata se esté moviendo a un lugar valido del mapa, como a un lugar explorado y de no ser así levanta una excepción y mata la tarea. Una vez realizados estos chequeos, si el pirata que llamo a la función es un explorador, se llama a *game\_chequear\_botin* función que marca como exploradas las posiciones aledañas y lanza un explorador si había un tesoro en alguna de ellas además de pintar la pantalla en estas posiciones. Luego, se llama a *screen\_pirata\_movimiento* que pinta la posición nueva y la anterior del pirata en la pantalla como muestran las figuras del enunciado. Por último, *mmu\_mover\_pirata* se encarga de mapear y mover el código de la tarea a su nueva posición.

Además, como decisión de diseño, utilizamos una matriz auxiliar para cada jugador llamadas **posiciones\_exploradas** en las que guardamos que posiciones de la matriz fueron exploradas (valor 1), y en cuales hay botines descubiertos y ya vacios (valores 2 y 3 respectivamente), para simplificar el algoritmo de dibujar los caracteres en el mapa según su estado.

## 7.4. Final del juego

El juego termina o bien cuando no quedan tesoros en el mapa, o bien cuando la variable de *game.c tiempo\_restante* llega a 0. Esta baja uno por cada tarea que se corre y se restaura a su valor original cuando un jugador anota un punto. Al terminar el juego, se muestra por pantalla que jugador resultó ganador. Por decisión de implementación del juego, en caso de empate,

se declara como ganador al equipo rojo.

## 7.5. Desalojo de tareas

El desalojo de tareas consiste en 2 pasos. Primero, marcar el bit de presente en la GDT en 0. Y, segundo, indicarle al struct del jugador que corresponda, que tiene un slot libre para lanzar otra tarea.

## 7.6. Modo debug

Dado que el modo debug debía activarse a través del teclado, y pausar el juego al generarse una excepción en una tarea, el primer paso fue agregarle funcionalidad a la interrupción de teclado para que al apretar la tecla `y`, el modo se active. Luego que las rutinas de las interrupción de reloj, teclado, software y excepciones del procesador sean ignoradas al estar pausado, excepto la interrupción del teclado que debe reanudar el juego.

Por último, se reemplazaron las rutinas de atención de las excepciones (interrupciones 0 a 19), para que llamen a una función en `c`, que imprime por pantalla los datos de la TSS, los cuales se consiguen leyendo el TR actual. Además, los últimos 5 valores pusheados en la pila se pasan como parametro a la función.

Al salir de la pausa, se desaloja la tarea de la forma que está explicada anteriormente, y se salta a la tarea idle para que en el siguiente ciclo de clock el scheduler salte a otra tarea si correspondiera.