Jaurías: Programación de Sistemas Operativos

M. I. Gamarra, D. Santos y P. N. Alcain

12 de diciembre de 2015

1. Introducción

El trabajo práctico consiste en la implementación de un pequeño Sistema Operativo que dé soporte para que corra una version del juego **Jaurías**. El desarrollo se basó en una serie de ejercicios que sirvieron como pautas de ordenamiento para la implementación del sistema.

Los ejercicios estaban enfocados en la implementación del SO a través de ciertas etapas específicas para el funcionamiento del sistema.

Las etapas desarrolladas son:

- Tabla de Descriptores Globales (`GDT`): Armamos la tabla `GDT`para direccionar los primeros `500 MB` que contengan los niveles *usuario* y *kernel* y tanto datos como códigos. La segmentación es del tipo *flat* (es decir, todos los segmentos comparten el espacio total de los `500 MB`) ya que los permisos luego se conforman con la paginación.

- Tabla de Descriptores de Interrupción (`IDT`) de Procesador: Armamos la tabla correspondiente a `IDT`para las interrupciones del procesador.

- Inicialización de Directorio de Páginas (`PD`): Creamos e inicializamos el directorio páginas y las tablas de pagina para el *kernel* con *identity mapping* para el mapeo de las páginas. Una vez que está mapeada la primera entrada, activamos paginación.

- Unidad de Manejo de Memoria (`MMU`): Escribimos una unidad de memoria para manejar la paginación dinámica, que mapea y desmapea páginas a medida que las tareas lo requieren.

- Tabla de Descriptores de Interrupción (`IDT`) de Usuario: Completamos la `IDT`con las rutinas de atención de interrupciones del usuario (teclado, reloj y servicio `0x46`). Además se cambió el handler de las otras interrupciones para que, ante cualquier excepción generada por una tarea asociada a un jugador, ésta sea eliminada. La interrupción `0x46` es una `syscall` que programamos de acuerdo a las acciones de las tareas.

- Tabla de Tareas (`TSS`): Armamos la tabla de `TSS`con 18 entradas en total (la tarea inicial, la tarea `Idle` y las tareas de los 8 perros para cada jugador). Luego completamos en la `GDT`los selectores de todas las táreas. Completamos cada entrada de la `TSS`con el código de ejecución común a los perros, `0x401000`. Como `cr3` cambia según el tipo de perro, `0x401000` está mapeado a distintas posiciones de memoria y, en consecuencia, ejecuta código distinto. Aquí estamos usando claramente la ventaja de la paginación.

- Scheduler: Programamos un scheduler de tipo Round-Robin que intercambia las tareas, de modo tal de poder simular el procesamiento "simultáneo" de tareas.

- Modo Debug: La idea era implementar esta característica del juego, que pone el sistema en pausa y muestra el estado de los registros del procesador cuando una tarea genera una excepción.

2. El kernel

La tarea clave del kernel es pasar de modo real a modo protegido. Una vez en modo protegido, ya tenemos un kernel de 32 bits funcionando. Para pasar a modo protegido necesitamos definir la `GDT`.

2.1. Ejercicio 1: GDT

En este ejercicio armamos la GDT. Es simplemente un arreglo de entradas con la siguiente estructura:

```
struct str_gdt_entry {
  unsigned short  limit_0_15;
  unsigned short  base_0_15;
  unsigned char   base_23_16;
  unsigned char   type:4;
  unsigned char   s:1;
  unsigned char   dpl:2;
  unsigned char   p:1;
  unsigned char   limit_16_19:4;
  unsigned char   avl:1;
  unsigned char   l:1;
  unsigned char   db:1;
  unsigned char   g:1;
  unsigned char   base_31_24;
} __attribute__((__packed__, aligned (8)));

/* sizeof(struct str_gdt_entry) = 8 bytes */
```

Hay un detalle específico a este TP en particular, y es que además del `NULL_DESCRIPTOR` en la posicion 0, recién llenamos la `GDT`a partir del índice 8. La segmentación, como ya dijimos, es de tipo *flat*, así que todas las entradas de la `GDT`tienen que direccionar los primeros `500 MB`. Para poder direccionar tanta memoria, necesitamos que la granularidad sea 1 así direcciona de a páginas de `4 KB`. La primera entrada de la GDT es una entrada nula y, a partir de la octava entrada, cargamos los descriptores correspondientes a cada nivel (0 y 3) para código y datos respectivamente. La única diferencia entre cada par de segmentos en el mismo nivel es únicamente el campo tipo. El límite es `500 MB/4kb - 1= 0x1F400 - 1 = 0x1F3FF`, la última página direccionada. Así, por ejemplo, un descriptor de segmento de código de nivel 0 es de la forma:

- **límite** = `0x001F3FF` (`limit(0:15) = 0xF3FF` y `limit(16:19) = 0x1`).
- **base** = `0x0` (`base(0:15) = 0x0` y `base(16:19) = 0x0`).
- **tipo** = `0xA` (de acuerdo a la tabla 3-1 de Intel. También se puede pensar en setear cada bit por separado) que indicar que es de código y permitir que se pueda leer y ejecutar. (o sea, el tipo de memoria y el privilegio)
- En el bit de sistema le pusimos 1 para indicar que no es de sistema.
- Le asignamos privilegio de kernel (DPL = 0).
- Lo marcamos como presente.
- En el campo AVL pusimos 0 al igual que en el campo *l* pues trabajamos con la arquitectura de 32 bits y como trabajamos con segmentos de 32 bits marcamos el bit DB como uno.

El resto de las definiciones es análoga. Una vez hecho esto, preparamos el sistema para pasar a modo protegido. Para pasar a modo protegido, lo que realizamos fue:

- Deshabilitar las interrupciones, usando **cli**.
- Habilitamos A20 (llamando a la rutina `habilitar_A20`).
- Cargamos la GDT utilizando la instrucción `lgdt[GDT_ DESC]`, donde `GDT_DESC` es la base en la que definimos la la GDT.
- Seteamos el bit PE del registro CR0 realizando un OR entre lo que contiene este registro y **0x1**
- Finalmente realizamos el salto a `(0x8*8):modo_rotegido`, pues en la posición **0x8** de la GDT se encuentra el selector de código de nivel 0 y `modo_protegido` es la etiqueta donde empezamos a definir el código correspondiente al modo protegido.

Estando en modo protegido, seteamos los correspondientes registros de segmento de datos y el segmento extra para video (en `fs`). Finalmente, seteamos la pila en la dirección `0x27000`, una vez en modo protegido, cargamos los registros `ebp` y `esp` con el valor `0x27000`.

Específicos de este ejercicio

Finalmente, para terminar el ejercicio, inicializamos la pantalla llamando a la función proporcionada por la cátedra `screen_inicializar()` la cual pinta el área de la pantalla con los colores que se muestran en el enunciado.

2.2. Ejercicio 2: Interrupciones del procesador

Completamos las entradas de la IDT para las entradas 0-19 (20 a 31 estan reservadas). Todas estas entradas fueron inicializadas con los siguientes valores:

- En el campo offset(15:0) se colocan los últimos 2 bytes de la dirección donde se encuentra la rutina de atención de la interrupción en cuestión.
- En el selector de segmento ponemos `0x0008`, que corresponde a un segmento de código de nivel 0.
- En los atributos de la entrada `IDT` usuario colocamos `0xEE00` (macro `ID_ENTRY`) que indica que estamos trabajando con una interrupt gate, de tamaño 32 bits, `DPL` 3 y presente.
- En los atributos de la entrada `IDT` supervisor colocamos `0x8E00` (macro `IDT_ENTRY`) que indica que estamos trabajando con una interrupt gate, de tamaño 32 bits, dpl 0 y presente.
- En el campo `offset(32:16)` colocamos los primero 2 bytes de la dirección donde se encuentra la rutina de atención de la interrupción en cuestión.

```
struct str_idt_entry {
  unsigned short offset_0_15;
  unsigned short segsel;
  unsigned char zero;
  unsigned char type:4;
  unsigned char s:1;
  unsigned char dpl:2;
  unsigned char p:1;
  unsigned short offset_16_31;
} __attribute__((__packed__, aligned (8)));

/* sizeof(struct str_idt_entry) = 8 bytes */
```

La macro sencillamente pone en el handler el código de ejecución definido en `isr.asm`. Éste sólo imprime en pantalla el tipo de interrupción.

Específicos de este ejercicio

Para probar el uso de estas rutinas por parte del procesador, realizamos una división por cero, y efectivamente, se nos imprime en pantalla el mensaje de `Divide Error`.

2.2.1. Ejercicio 3: Iniciar el directorio de página

La inicialización del directorio y la tabla de páginas la completamos sin hacer uso de la función inicializar_dir_kernel sino que lo hamos directamente en kernel.asm de la siguiente forma:

- Inicializamos todas las posiciones del Page Directory (PD) en 0x00000002.
- Apuntamos la primer página de PD a la page table y seteamos los bits de presente y lectura/escritura
- Realizamos el identity mapping para las `0x1000` cada páginas de la `PAGE TABLE` y activamos la lectura/escritura y el bit de presencia de cada página. (atributos)

Para relacionar una entrada en la tabla de páginas con una posicíon de memoria física, necesitamos dividir la posición por `0x1000` (`4 KB`), que es el tamaño de las páginas. Esto es lo que nos permite direcccionar hasta `4 GB` de memoria con la estructura de directorio+tabla de paginación. esto lo cumplimos al utilizar como posición inicial del Page Directory a la `0x27000` y a la del Page Table `0x28000`.

```
struct str_pd_entry {
  unsigned char present:1;
  unsigned char rw:1;
  unsigned char user_supervisor:1;
  unsigned char page_level_write_through:1;
  unsigned char page_level_cache_disable:1;
  unsigned char accessed:1;
  unsigned char ignored:1;
  unsigned char page_size:1;
  unsigned char global:1;
  unsigned char disp:3;
  unsigned int  base_dir:20;
} __attribute__((__packed__));
/* sizeof(struct str_pd_entry) = 4 bytes */

struct str_pt_entry {
  unsigned char present:1;
  unsigned char rw:1;
```

```
    unsigned char user_supervisor:1;
    unsigned char page_level_write_through:1;
    unsigned char page_level_cache_disable:1;
    unsigned char accessed:1;
    unsigned char dirty_bit:1;
    unsigned char page_table_attr_index:1;
    unsigned char global:1;
    unsigned char disp:3;
    unsigned int  base_dir:20;
  } __attribute__((__packed__));
  /* sizeof(struct str_pt_entry) = 4 bytes */
```

Para activar paginación cargamos en el campo base del `cr3` la dirección del `PD`y seteamos el bit de paginación con `or cr3, 0x80000000`.

Específicos de este ejercicio

Testeamos el ejercicio desmapeando la última página `0x3FF000`.

2.2.2. Ejercicio 4: Unidad de Manejo de Memoria

Escribimos una rutina (`inicializar_mmu`) que inicializa las estructuras globales necesarias para administrar la memoria en el area libre (un contador de paginas libres):

- `PRIMER_PAG_LIBR: 0x101000`
- `ULTIMA_PAG_LIBR: 0x3FE000`

Además implementamos algunas funciones para el mapeo de las páginas:

- `int mmu_proxima_pagina_fisica_libre()`
- `void mmu_mapear_pagina (uint virtual, uint cr3, uint fisica, uint attrs)`

Esto fue necesario para escribir la rutina `mmu_inicializar_memoria_perro` encargada de inicializar un directorio de páginas y una tabla de páginas para cada una de las tareas.

Específicos de este ejercicio

Con todo implementado inicializamos la tarea de un perro y la intercambiamos con el kernel, luego cambiamos de el color del fondo del primer caracter. Para luego volver a la normalidad, con esto nos aseguramos el correcto funcionamiento de lo recién implmentado.

2.3. Ejercicio 5: Interrupciones del Usuario

Agregamos 3 entradas a la IDT:

- `32` para reloj
- `33` para teclado
- `0x46` para syscall

Las primeras 2 entradas las completamos de la misma forma que hicimos con las interrupciones del procesador en el punto 3. Es decir, para completar la entrada de `IDT`pusimos como offset el campo `offset(15:0)` la direccióon donde se encuentra la rutina de atención de la interrupción en cuestión, en el selector de segmento ponemos 0x08 (que corresponde a un segmento de código de nivel 0) y en los atributos colocamos `0x8E00` (que indica que estamos trabajando con una interrupt gate, de tamaño 32 bits, dpl 0 y presente). La entrada de IDT correspondiente a la interrupcion `0x46`, la completamos exactamente de la misma forma, pero poniendo el dpl en 3 (o sea attrs en 0xEE00), pues es una interrupcion de tareas.

Luego hay que escribir los handlers de las interrupciones:

2.3.1. Reloj

Esta rutina, por cada tick, llama a la función `proximo_reloj`, que realiza la animación de un cursor rotando. La función `proximo_reloj` está definida en `isr.asm`.

Esta rutina entonces es la entrada `_isr32` en `isr.asm`. Hace los siguientes pasos:

1. desactivamos interrupciones `cli`
2. preservamos registros `pushfd`
3. preservamos EFLAGS `pushad`
4. comunicamos al PIC que se atendió la interrupción `fin_intr_pic1`

5. llamamos a la rutina `proximo_reloj`

6. restauramos EFLAGS `popfd`

7. restuaramos registros `popad`.

8. activamos interrupciones `sti`

9. retornamos `iret`

En el caso de la rutina de teclado, hacemos esencialmente lo mismo, pero movemos al registro `eax` la tecla apretada mediante `in al, 0x60`. ponemos `eax` en la pila y llamamos a la funcion `game_atender_teclado`. Luego restauramos `eax` y su ruta.

La función `game_atender_teclado` es la que se encarga de realizar alguna acción segun la tecla que fue presionada.

Para la rutina de la `_isr46` por ahoa sólo modifica el vaor de `eax`, pero luego va a modificarse.

2.3.2. Ejercicio 6: Tabla de Tareas

```
struct str_tss {
  unsigned short   ptl;
  unsigned short   unused0;
  unsigned int     esp0;
  unsigned short   ss0;
  unsigned short   unused1;
  unsigned int     esp1;
  unsigned short   ss1;
  unsigned short   unused2;
  unsigned int     esp2;
  unsigned short   ss2;
  unsigned short   unused3;
  unsigned int     cr3;
  unsigned int     eip;
  unsigned int     eflags;
  unsigned int     eax;
  unsigned int     ecx;
  unsigned int     edx;
  unsigned int     ebx;
  unsigned int     esp;
  unsigned int     ebp;
  unsigned int     esi;
  unsigned int     edi;
  unsigned short   es;
  unsigned short   unused4;
  unsigned short   cs;
  unsigned short   unused5;
  unsigned short   ss;
  unsigned short   unused6;
  unsigned short   ds;
  unsigned short   unused7;
  unsigned short   fs;
  unsigned short   unused8;
  unsigned short   gs;
  unsigned short   unused9;
  unsigned short   ldt;
  unsigned short   unused10;
  unsigned short   dtrap;
  unsigned short   iomap;
} __attribute__((__packed__, aligned (8)));

/* sizeof(struct str_tss) = 104 */
```

Como primer paso, definimos las entradas para la tarea inicial e idle en la `GDT`dejando algunos valores para completar desde `tss.c`:

- `base_0_15`
- `base_23_16`
- `base_31_24`
- `limit_0_15`

Estos valores se van a definir una vez inicializada la tabla con `tss_inicializar`.

Para inicializar la `TSS`de la tarea `Idle`, realizamos una función (`tss_initializar_tarea_idle`) que mapea a dirección virtual del código de la tarea con la dirección física del código de idle y completa la entrada `TSS`con los siguientes valores:

- `cr3` es el del del kernel (no cambiamos el `PD`);
- `eip` es la posición del código de la tarea en el `PD`del kernel (porque es el que seteamos en `cr3`).
- `EFLAGS` en `0x202`, es decir con interrupciones en 1 y presente;
- Como la tarea IDLE corre en nivel 0, se mantienen los registros de segmentación del kernel;

- Se apuntan el esp0, el esp y el ebp a donde se aloja la pila de la tarea IDLE

La función `tss_inicializar_tarea` tiene como fin completar la `TSS`de una tarea en particular, la que se pasa por parámetro y lo hace de la siguiente manera:

- `cr3` el directorio de páginas de la tarea (a priori no tiene por qué ser el del kernel);
- `eip` a donde se encuentra la tarea en el `PD`;
- `EFLAGS` nuevamente en `0x202`;
- Como las tareas corren en nivel 3 y para cumplir con lo pedido, se almacenan los siguientes valores para los registros de segmentación:
  - `es = ss = ds = fs = gs = 0x33`, que es un segmento de datos de nivel 3;
  - `cs = 0x2B`, que es un segmento de código de nivel 3;
- Tanto `esp` como `ebp` se ubican en la dirección de la pila de la tarea que es `TASK_STACK`;
- El `esp0` en la dirección física de la pila de nivel 0 de la tarea.
- El `ss` de nivel 0 en el segmento de datos de nivel 0 ubicado en la posicipón `0x60`;

En `tss_initializar_tarea_inicial` no hace falta setear valores para la `TSS`puesto que nunca se va a saltar a esa tarea, sólo nos interesa setear los valores de la GDT que no pudimos setear en la inicializaciónn.

Para completar los descriptores anteriormente nombrados, utilizamos una función que se llama `tss_inicializar`, la cual se encarga de llamar a :

- `tss_initializar_tarea_inicial`
- `tss_initializar_tarea_idle`

Una vez hecho todo eso, para ejecutar la tarea `Idle`, primero llamamos a `inicializar_scheduler`, que inicializa todas las estructuras necesarias para arrancar el scheduler. Posteriormente cargamos el selector de la tarea inicial en un registro, es decir, el `0x40` y lo cargamos en el task register. Luego de cargar el task register, hacemos un cambio de tareas pasando a la tarea idle mediante un `0x48:0`. La CPU correrá la tarea idle y cuando se produzca la próxima interrupción de reloj saltará a la primera tarea.

2.3.3. Ejercicio 7: Scheduler

Implementamos la función `sched_inicializar` seteando:

Primero para la `Idle`:

- `scheduler.tasks[MAX_CANT_TAREAS_VIVAS].gdt_index = COD_TAREA_IDLE_DIR`
- `scheduler.tasks[MAX_CANT_TAREAS_VIVAS].perro = NULL`

Luego para cada perro en el arreglo de tareas:

- `scheduler.tasks[i].gdt_index = (GDT_IDX_TSS_BASE_PERROS_A + i)`
- `scheduler.tasks[i].perro = NULL`

y

- `scheduler.tasks[MAX_CANT_PERROS_VIVOS + i].gdt_index = (GDT_IDX_TSS_BASE_PERROS_B + i)`
- `scheduler.tasks[MAX_CANT_PERROS_VIVOS + i].perro = NULL`

aquí el índice `i` corre hasta 8, seteando los 8 perros de cada jugador.

Y por último, el estado inicial del `scheduler`, con los jugadores:

- `scheduler.current = 0`
- `scheduler.indice_ultimo_jugador_A = 1`
- `scheduler.indice_ultimo_jugador_B = 9`

Implementamos la función `sched_proxima_a_ejecutar`, cuya funcionalidad es hacer el switcheo round-robin de tareas entre los perros de cada jugador. Intercala las ejecuciones de los perros de cada jugador a la vez sin repetir el perro en caso de que tengan vivo mas de uno.

Específico al problema es cómo se guarda la lista de tareas del scheduler. Lo más importante es tener las rutinas que la manejen adecuadamente. Básicamente, tenemos que poder:

- Agregar tarea
- Remover tarea
- Siguiente tarea

Es natural suponer que la estructura de datos óptima de la lista de tareas del scheduler va a depender mucho de la naturaleza del SO. En nuestro caso, la lista de tareas es un array, así que quitar una tarea es disminuir la cantidad de perros del jugador en cuestión y liberar la última posición. Agregar una tarea es aumentar la cantidad de perros del jugador y setear la entrada en la `TSS`de acuerdo al tipo de perro y al nombre del jugador.

Así, `sched_proxima_a_ejecutar`, va cambiando entre los jugadores (en caso de que los dos tengan tareas activas) y dentro de cada jugador toma la acción del siguiente perro. Cada interrupción de reloj llama a `sched_proxima_a_ejecutar` en caso de que haya un perro vivo y en caso de que no haya tareas en ninguno de los jugadores, ejecuta la tarea `Idle`.

Finalmente, los movimientos de los perros requieren privilegios de kernel para, por ejemplo, pintar el mapa. Como esto no pueden hacerlo las tareas directamente (no tienen los permisos), hay que hacerlo a través de una interrupción. En este caso, modificamos la interrupción de `syscall 0x46` para que comunique al kernel la acción del perro. Como esta interrupción la tiene que pedir el usuario, la entrada de `IDT`de `0x46` tiene que tener `DPL = 3`. Una vez que se lanza la interrupción, el kernel la maneja y decide qué hace. Es decir, la interrupción sólo es un *pedido* al kernel. Éste después decide si le da curso y cómo, pudiendo incluso desalojar otra tarea.

2.3.4. Modo Debug

Tal como se especificó en el enunciado, las excepciones del procesador activan el modo debug en el juego, en el cual se setea la variable booleana `debug`. Así, cuando se produzca la próxima excepción causada por una tarea se imprime el cartel con el tipo de excepción que produjo y el valor de cada uno de los registros de la `TSS`de esa tarea y el estado de la pila. Por último, al volver a presionar una tecla se desactiva el modo Debug y se reanuda el juego desde el estado anterior.