DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming - Saltadores y Cazadores

Organización del Computador II

Primer Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Bogetti Gianfranco	693/15	gianbogetti7@hotmail.com
Feliu Santiago	644/15	santiagofeliu96@gmail.com
Ingaramo Pablo	544/15	pablo2martin@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. Desarrollo

1.1. Segmentación

La segmentación solicatada para este trabajo práctico sera una segmentación flat con base en $0x0$ y límite de 314 MB de memoria. Por lo pedido tendrémos cuatro selectores de segmentos, dos para código nivel 0 y nivel 3, y otros dos para datos nivel 0 y nivel 3 tambien.

```
PARTES EN COMÚN DE SELECTORES DE SEGMENTO:
        (uint16_t)    0x39FF,         /* limit[0:15]  */
        (uint16_t)    0x0000,         /* base[0:15]   */
        (uint8_t)     0x00,           /* base[23:16]  */
        (uint8_t)     0x01,           /* s            */
        (uint8_t)     0x01,           /* p            */
        (uint8_t)     0x01,           /* limit[16:19] */
        (uint8_t)     0x00,           /* avl          */
        (uint8_t)     0x00,           /* l            */
        (uint8_t)     0x01,           /* db           */
        (uint8_t)     0x01,           /* g            */
        (uint8_t)     0x00,           /* base[31:24]  */
```

Los selectores se diferenciarán solo por el tipo y dpl, de la siguiente manera:

- **Código nivel 0**: tipo: $0xA$ dpl: $0x0$
- **Código nivel 3**: tipo: $0xA$ dpl: $0x3$
- **Datos nivel 0**: tipo: $0x2$ dpl: $0x0$
- **Datos nivel 3**: tipo: $0x2$ dpl: $0x3$

Estos 4 selectores estarán posicionados respectivamente en las posiciones 20, 21, 22 y 23 de la GDT.

1.2. Interrupciones

Tendremos una **IDT** con las primeras 32 entradas utilizadas para identificar las excepciones de intel, luego tendremos en las entradas 32 y 33 interrupciones procedentes del pic, clock y teclado respectivamente, por otro lado utilizaremos la entrada 102 ($0x66$) para detectar interrupciones de software.

Todas estas tendran sus respectivos descriptores de segmento de puerta de interrupción alojadas en la **IDT**, utilizando el segmento en la posición 20 de la **GDT**, con dpl $0x0$, exceptuando la interrupción $0x66$ que tendra dpl $0x3$ ya que querremos que sea llamada por el usuario.

1.3. Paginación

En este TP utilizamos identity mapping para el sector del kernel, contenido entre las direcciones $0x00000000$ a $0x003FFFFF$, en cambio las tareas estaran mapeadas en la dirección virtual $0x8000000$ mientras que la memoria física corresponderá al sector de Área libre tareas ($0x400000$ a $0x7FFFFF$)

El directorio de páginas de kernel estará en la dirección $0x27000$ y la primer tabla de páginas estará en la dirección $0x28000$ pedido por el enunciado.

El primer descriptor de tabla de páginas en el directorio de páginas tendra los siguientes atributos:

```
        .p = 1,
        .rw = 1,
        .us = 0,
        .pwt = 0,
        .pcd = 0,
        .a = 0,
        .ignored1 = 0,
        .ps = 0,
        .ignored4 = 0,
        .dir_table = 0x28
```

Mientras que los atributos de los descriptores de páginas en la tabla de páginas estaran descriptos de la siguiente manera:

```
        .p = 1,
        .rw = 1,
        .us = 0,
        .pwt = 0,
        .psv = 0,
        .a = 0,
        .d = 0,
        .pat = 0,
        .g = 0,
        .ignored = 0,
        .dir_memory = i + 1024*j
```

Con $i \in [0..,1023]$

1.4. Tareas

En total se tendrán 12 tareas, una tarea inicial, una llamada idle y otras 5 para cada jugador. Los descriptores de TSS de estas tareas se encontraran en la **GDT** y ocuparan las posiciones 1 hasta la 12.

La tarea inicial será utilizada solo para poder lograr el primer salto de tarea, por lo tanto la tss estará en blanco, ya que nunca se utilizará.

La tss de la tarea idle sera de la siguiente manera:

```
        .ss0 = 0xB0,
        .eflags = 0x202,
        .eip = 0x14000,               /* Pocición de código idle */
        .ss = 0xB0,
        .es = 0xB0,
        .cs = 0xA0,
        .ds = 0xB0,
        .fs = 0xB0,
        .gs = 0xB0,
        .iomap = 0xFFFF
```

Donde $0xA0$ es el segmento de código de nivel 0 y $0xB0$ es el segmento de datos de nivel 0. En cuanto al resto:

- **CR3**: Dirección del directorio de páginas. Coincide con el que utiliza el kernel
- **ESP0**: Página física libre kernel
- **ESP** y **EBP**: Utilizan el final de una página fisica libre del kernel

Inicialmente las tss de las demas tareas tendrán los siguientes campos idénticos:

```
        .ss0 = 0xB0,
        .eflags = 0x202,
        .esp = 0x8000000 + 4096,
        .ebp = 0x8000000 + 4096,
        .eip = 0x8000000,
        .ss = 0xBB,
        .es = 0xBB,
        .cs = 0xAB,
        .ds = 0xBB,
        .fs = 0xBB,
        .gs = 0xBB,
        .iomap = 0xFFFF
```

Donde $0xAB$ es el segmento de código de nivel 3, $0xBB$ es el segmento de datos de nivel 3 y $0xB0$ es el segmento de datos de nivel 0. Mientras que cada tarea tendrá su propio directorio de páginas en donde tendran mapeados en identity mapping el sector del kernel y el código de la tarea en la dirección $0x800000$. En cuanto el **ESP0** apuntara al final de una página física libre del kernel.

1.5. Scheduler

El scheduler tendrá los siguientes elementos:

- Dos *jugador_t*, jugador A y jugador B.
  - Un *uint8_t* indice_tarea, índice que tendra la tarea que se este ejecutando actualmente.
  - Un *uint8_t* jugador_actual, indica jugador del turno actual (0=A—1=B).
  - Un *uint8_t* caracteres_reloj[], arreglo con los 4 posibles caracteres del reloj.
  - Un *uint16_t** ultima_pantalla, puntero a la copia de la pantalla antes de pausar el juego.
  - Un *uint32_t* debugging, indica si se esta en modo debugging.
  - Un *uint32_t* paused, indica si el juego esta pausado.
  - Un *uint32_t* reestablecer_pausa, indica que el juego esta volviendo de la pausa.
- Los jugadores estarán constituidos de la siguiente manera:
  - *tarea_t** cazadores, puntero al primer cazador en un total de 4.
  - *tarea_t** saltadora, puntero a la tarea saltadora.
  - *tarea_t** ultimo_cazador, puntero al último cazador utilizado por el jugador.
  - *uint8_t* cant_vidas, cantidad de vidas restantes del jugador.
- La estructura tarea_t estará constituida de la siguiente manera:
  - *uint16_t* indice_tss, índice de la tss en la **GDT**
  - *uint32_t* base_codigo, dirección física de la base del código de la tarea.
  - *uint8_t* indice, indicador del número de tarea en el context del jugador.
  - *uint8_t* indice_reloj, indice del ultimo caracter de reloj escrito en pantalla, contexto de tarea.
  - *tarea_t** siguiente, puntero a siguiente tarea.
  - *tarea_t** anterior, puntero a tarea anterior.

2. Resolución de problemas

1. *a*) Este punto pedia completar la **GDT** con 4 descriptores de segmentos, los cuales fueron explicados en la sección de segmentación.

   b) Una vez creados los descriptores de segmentos, en particular el descriptor que es de tipo código y con dpl 0 se podrá pasar a modo protegido, para ello se habilita *A20*, se carga la **GDT** y se setea el bit 0 del registro *CR0* para poder activar segmentación. Luego se ejecuta la instrucción *jmp* para saltar al modo protegido utilizando el segmento descripto anteriormente. Luego se setean los demas registros de segmentos con el descriptor de datos con dpl 0 y se coloca el valor *0x27000* en los registros *ESP* y *EBP* para establecer la pila en esa posición.

   c) Se declara un nuevo descriptor de segmento, en este caso para definir el espacio que ocupa la pantalla. para ello se situa la base del segmento en la dirección *0xB8000* y el límite sera de *0x1F40*, mientras que los demas atributos seran iguales a los del descriptor de segmento de datos de nivel 0, diferenciandose solo en que el bit **G** estará apagado.

d) Lo que creemos que se intenta lograr con este ejercicio, es que observemos la diferencia entre dirección lógica y dirección lineal (en este caso igual a la física porque no esta activada la paginación), ya que la unidad de segmentación es la encargada de transformar la dirección lógica a dirección lineal. Por lo tanto como el nuevo segmento que se creo en el punto anterior tiene base en el principio de la pantalla, se tendra que trabajar con las direcciones lógicas comenzando desde *0x0* hasta *Fin de pantalla AQUI*.

2. *a*) Este punto fue descrito anteriormente en la sección de interrupciones.
   b) Para probarlo se hizo una división por 0 que esta comentada en el kernel. Como también esta comentado la impresión que se hacia en la interrupción.

3. *a*) *, b), c) y d)* La mayoría fue descrita en la sección de interrupción, cabe remarcar que se utiliza la función **fin_intr_pic** para poder avisarle al pic que se atendió la interrupción en las interrupciones de reloj y de teclado. También se utilizan en todas las interrupciones las funciones **pushad** y **popad** para mantener los estados de los registros antes de llamarse a la interrupción.

4. *a*) En la función **mmu_inicializar_dir_kernel** se crean las estructuras que representan al directorio de páginas de kernel y a la primer tabla de páginas del kernel, en la posición pedida y con los atributos ya mencionados en la sección de paginación.
   b) Para activar paginación primero se coloca en el **CR3** la dirección de memoria en donde está alojado el directorio de páginas creado anteriormente, y luego se setea el bit mas significativo del registro **CR0**.

5. *a*) **mmu_inicializar** se encargará de inicializar dos variables a las que llamaremos **prox_pag_libre_kernel** y **prox_pag_libre_tarea** las cuales serán de tipo *uint32_t* e indicarán donde comienza la próxima página libre del kernel y de tarea, respectivamente. Ademas existirán **mmu_prox_pag_fisica_libre_kernel** y **mmu_prox_pag_fisica_libre_tarea** que devolveran el índice de los contadores y lo harán avanzar.
   b) En **mmu_inicializar_dir_tarea** debemos incicializar un directorio para la tarea, para ello se tomarán los parámetros *uint8_t** codigo que será un puntero al código a copiar de la tarea y *uint32_t** dir_fisica_codigo que se utilizará para devolver la dirección física en donde estará el código de la tarea, su utilidad se verá más tarde con la explicación del scheduler.
   Esta función creará un nuevo directorio de páginas, en la cual el primer descriptor de tabla de páginas será idéntico al del kernel. Luego se pide con la función **mmu_prox_pag_fisica_libre_tarea** una página para poder copiar el código de la tarea, pero antes hay que mapearla en alguna dirección lineal libre, para ello utilizamos la dirección $0x400000$, esta página se mapea utilizando la función **mmu_mapearPagina** explicado en el próximo punto, que mappeara la dirección física en la dirección lineal dada, con los atributos que recibirá, los cuales serán iguales al kernel con excepción de que será accesible para usuario.
   Una vez mapeada la nueva página se le copia el código, se la unmmapea y finalmente se le mappea en el nuevo directorio de páginas.
   c)
   - **mmu_mapearPagina** recibirá como parámetros la dirección virtual, el directorio de páginas, la dirección física y los atributos que se querran colocar. Utilizaremos la dirección virtual para poder acceder a los distintos descriptores e ir colocando los atributos pedidos, en el caso de encontrar que algún descriptor no se encuentra presente se le asignará uno nuevo en la dirección que nos otorgue la función **mmu_prox_pag_fisica_libre_kernel**. Una vez llegado al descriptor de páginas, se le copiaran los 20 bits más significativos en el atributo base de página.
   - **mmu_unmapearPagina** recibirá la dirección virtual y el directorio de páginas. Luego se buscará el descriptor de tabla de páginas y el director de páginas pertenecientes a la dirección lineal para apagar el bit de presente de ambos.

6. *a*) Las entradas en la gdt para las tss de cada tarea son las descriptas anteriormente en la sección de tareas. La tarea inicial estará en el índice 1 de la **GDT**, mientras que la tarea Idle se encontrará en el 2. En cuanto a las tareas de los jugadores se les irá asignando a partir de la creación de cada tarea.

b) Los atributos de la tarea Idle también fue descrito en la sección Tareas.

c) La función encargada de inicializar una tarea será llamada **tss_nueva_tarea** que recibirá como parámetros *uint32_t* tipo y *uint32_t** dir_fisica_codigo, 'tipo' será la variable que decidirá cual es el código de la tarea, 0 = saltadora A; 1 = cazadora A; 2 = saltadora B; 3 = cazadora B. Luego hace todo lo que ya se enunció en la sección Tareas.

Finalmente se utiliza una función llamada **prox_entrada_libre_gdt** que devuelve la próxima entrada libre de la **GDT** para poder colocar allí el descriptor TSS que poseerá los siguientes atributos:

```
.base_0_15 = tss_tarea;
.base_23_16 = tss_tarea >> 16;
.base_31_24 = tss_tarea >> 24;
.p = 1;
.limit_0_15 = 0x67;
.type = 0x9;
.dpl = 0x3;
.db = 0x1;
```

Donde tss_tarea es la dirección a la tss y devuelve el índice de la entrada en la **GDT**.

d) La entrada de la tarea inicial en la **GDT** se encontrará definida de la siguiente manera:

```
(uint16_t)    0x0067,             /* limit[0:15]  */
(uint16_t)    tss_inicial,        /* base[0:15]   */
(uint8_t)     tss_inicial >> 16,  /* base[23:16]  */
(uint8_t)     0x09,               /* type         */
(uint8_t)     0x00,               /* s            */
(uint8_t)     0x00,               /* dpl          */
(uint8_t)     0x01,               /* p            */
(uint8_t)     0x00,               /* limit[16:19] */
(uint8_t)     0x00,               /* avl          */
(uint8_t)     0x00,               /* l            */
(uint8_t)     0x01,               /* db           */
(uint8_t)     0x00,               /* g            */
(uint8_t)     tss_inicial >> 24,  /* base[31:24]  */
```

Donde tss_inicial estará definido como el parámetro que tome la función **tss_inicializar** y el límite es $0x67$ porque es el tamaño que ocupa la tss.

e) Mientras que la entrada de la tarea Idle en la **GDT** se encontrará definida de la misma forma, con la diferencia que la base será distinta.

f) La tarea inicial se cargará utilizando la instrucción **ltr** con el registro ax, en el registro debe estar el selector de descriptor correspondiente para la tarea. Y luego se utiliza **jmp** con el selector del descriptor de la tarea Idle.

7. *a*) Para inicializar el scheduler se utiliza la función **inicializar_sched** que inicializará los valores de las estructuras del scheduler, entre ellas:

```
indice_tarea = 4;
jugador_actual = 1;
debugging = 0;
paused = 0;
reestablecer_pausa = 0;
ultima_pantalla = (uint16_t*) mmu_prox_pag_fisica_libre_kernel();
mmu_prox_pag_fisica_libre_kernel();

jugador_A = (jugador_t*) mmu_prox_pag_fisica_libre_kernel();
jugador_B = (jugador_t*) mmu_prox_pag_fisica_libre_kernel();
```

Para remarcar:

- ultima_pantalla toma dos páginas libres porque no entra en una sola página.
- el juego empieza con jugador actual en 1 e índice tarea 4, para que **sched_proximoIndice** devuelva la tarea saltadora del Jugador A.

b) **sched_proximoIndice** será una función que devuelva el próximo índice dependiendo del estado de indice_tarea y jugador_actual como se observa en la Figura 1.

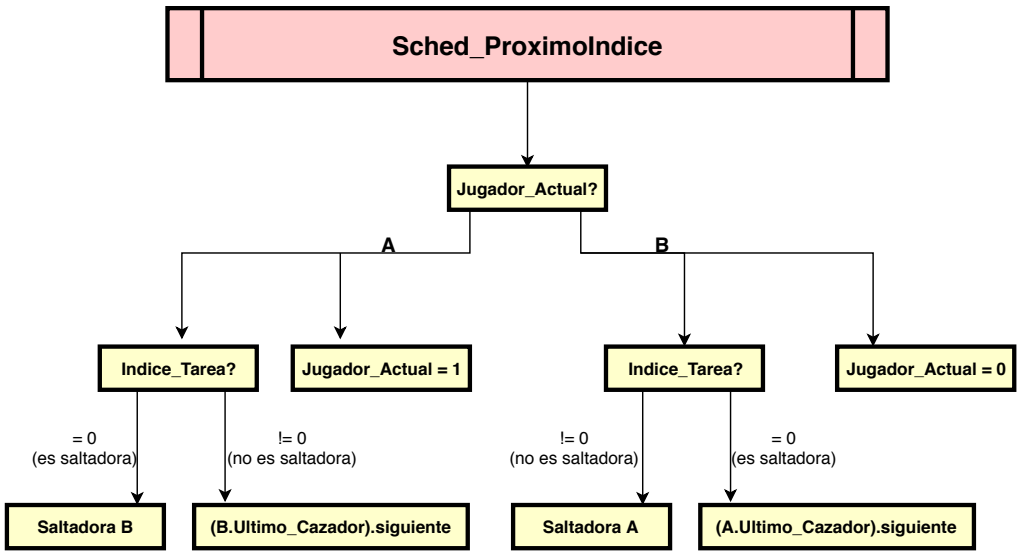

Figura 1

c) En la rutina de atención de la interrupción 0x66 primero se chequea cual de las tres syscalls hay que llamar y luego se pasan los parámetros necesarios por la pila antes de hacer el llamado a game_escribir, game_leer o game_numero, siempre respetando el comportamiento requerido por la cátedra para cada syscall.

En el caso de game_numero, retornamos el valor actual de la variable indice_tarea del scheduler. Mientras que en game_leer y game_escribir se utiliza la misma idea que en la función **mmu_inicializar_dir_tarea** en donde se mapea la página física a una dirección lógica libre y la dirección física se toma del atributo $base\_codigo$ de la tarea saltadora del jugador contrincante. Luego se escribe o lee según la función que fue llamada y se unmapea la página.

d) Para saltar de una tarea a otra en el código de atención de la interrupción de reloj llamamos a la función sched_proximoIndice que hace lo que se detalla en la Figura 1 y retorna el índice en la GDT de la próxima tarea a ejecutar.

e) La rutina de excepción la modificamos de manera que cuando una tarea causa una excepción se llama a la función $mantenimiento\_scheduler$. Esta función se encarga de desalojar la tarea actual y luego carga la próxima tarea o, en el caso que el jugador actual se haya quedado sin vidas o sin tareas cazadoras, se termina el juego. Si la tarea que causa la excepción es una tarea cazadora, esa tarea se borra de la lista $cazadores$ del jugador actual y el $ultimo\_cazador$ del jugador actual pasa a ser el cazador anterior al $ultimo\_cazador$ en la lista de cazadores. En caso de que la tarea que causa la excepción es una saltadora el jugador actual pierde una vida lo cual se ve reflejado en la pantalla en el sector ”vidas”. Ademas se resetea la tss de la saltadora con los valores iniciales y una pila nueva, y también se copia la página donde se encuentra el código con el código original. Al retornar la función $mantenimiento\_scheduler$ se salta a la tarea Idle por el resto del quantum.

f) El comportamiento del modo debugg se puede ver en el diagrama de la Figura 2.

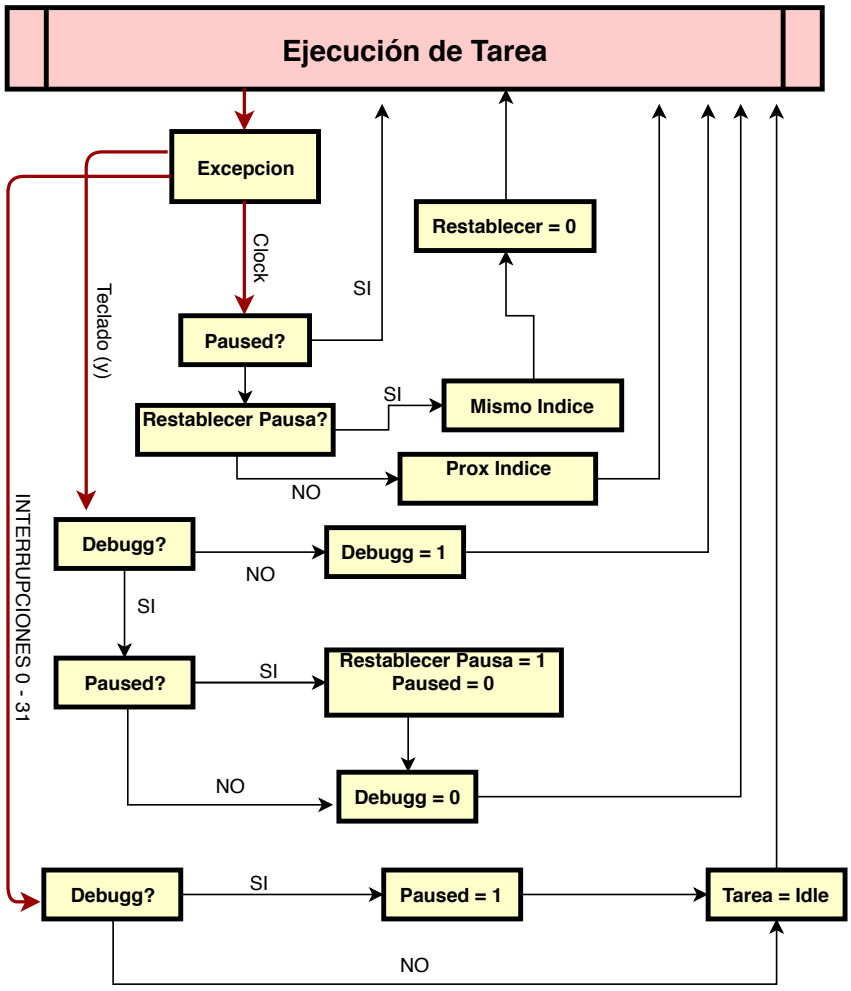
Ejecución de Tarea
Excepcion
Clock
Teclado (y)
INTERRUPCIONES 0 - 31
Paused?
SI
Restablecer = 0
Restablecer Pausa?
SI
Mismo Indice
NO
Prox Indice
Debugg?
NO
Debugg = 1
SI
Paused?
SI
Restablecer Pausa = 1
Paused = 0
NO
Debugg = 0
Debugg?
SI
Paused = 1
Tarea = Idle
NO

Figura 2