Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Organizacion del Computador II 1C 2021

Trabajo Práctico 3: Los Lemmings

Grupo 5

Integrante	LU	Correo electrónico
Cantini Budden, Sebastian	576/19	sebascantini@gmail.com
Seisdedos, Jose Luis	797/19	jlseisdedos11@gmail.com
Secin, Lucia Maria	748/19	luciamsecin@gmail.com

I. Introducción:

Este informe hablará de principios básicos de $SystemProgramming$. En particular de diseñar uno con el fin de poder jugar un juego llamado $Lemmings$, donde podemos ver una de sus tradiciones más preciadas, Tutunito. Tutunito es un juego que ambos pueblos lemmings comparten donde deben atravesar un mapa lleno de obstaculos y con terrenos tenuosos, con el fin de llegar al pueblo del contrincante. El primero en llegar ganará.

En este trabajo práctico veremos cosas como kernel, paginación, utilización de interrupciones y tareas para darle vida a este mundo de Lemmings Y permitirles completar su juego favorito.

II. Juego

El juego es bastante simple. Brevemente hablamos de cómo funciona en la introducción. Hay dos pueblos en esquinas opuestas del mapa y los lemmings competirán para ver que pueblo logra antes que uno de sus lemmings llegue al otro primero.

En esta carrera los lemmings se cruzarán con dos obstáculos principales, las paredes y agua, ambas obstruyendo el camino. Los lemmings son ingeniosos y tienen métodos para lidiar con cada uno de éstos.

Estos alegres lemmings tienen tres acciones, moverse en una de las cuatro (arriba, abajo, izquiera y derecha) direcciones, dar su vida para transformarse en puentes sobre el agua o sacrificarse en una explosión con el fin de destruir paredes y otros lemmings que se interpongan en su camino (no destruye puentes).

Sin embargo, en este juego hay reglas, cada pueblo sólo puede tener cinco lemmings corriendo la carrera a la vez. Esto implica que una vez que libera a cinco lemmings, no podrá liberar más hasta que uno de los liberados muera. Los lemmings mueren cuando explotan, se transforman en puentes o eventualmente la vejez puede finalizar la vida de estos pueblerinos.

Las reglas también especifican que debe haber un sistema de turnos, esto implica que un mismo equipo no puede hacer dos acciones consecutivas (a menos que el otro equipo no tenga ningún lemming despachado) y que dentro de un equipo, un mismo lemming no puede hacer su próxima acción a menos que todos los otros lemmings de ese equipo hayan tenido su turno (a menos que sea el único lemming despachado de ese pueblo).

III. Segmentacion

Nuestro trabajo práctico empezó con la Tabla de Descriptores Globales (GDT). Esta se encarga de contener una descripción que definirá los segmentos que utilizaremos en nuestro sistema. Estos segmentos se definen de la siguiente manera:

```
[GDT_IDX_NULL_DESC] = {
        .base_15_0 = 0x0000,
        .base_23_16 = 0x00,
        .base_31_24 = 0x00,
        .limit_15_0 = 0x0000,
        .limit_19_16 = 0x0,
        .type = 0x0,
        .s = 0x00,
        .dpl = 0x00,
        .p = 0x00,
        .db = 0x0,
        .g = 0x00,
        .l = 0x0,
        .avl = 0x0},
```

Fig. 1. Ejemplo de entrada de GDT.

El segmento anterior es el segmento nulo (notar que tiene cero en todas las variables). Como se puede ver, se utilizan muchas variables para definir un segmento. Veamos una breve descripción de cada una.

- Base: Esta variable está dividida en tres. Ocupando un total de treinta y dos bits, esta nos marca el principio del segmento que estamos describiendo.
- Limit: El limite o limit, está también particionado en dos, ocupando un total de veinte bits. Similar a la base, esta variable define una posición del segmento, en vez de marcar donde comienza éste, define donde finaliza.
- Type: La tercer variable define cómo se va a utilizar el segmento. Entre las opciones podemos ver el tipo dos ($0x2$), que nos dice que el segmento es de datos y se utilizará para escritura y lectura, y el tipo nueve ($0xA$), que nos indica que el segmento es de código y este segmento será utilizado exclusivamente para ejecutar código. Estos dos tipos son los que utilizaremos principalmente para nuestros segmentos.
- S: Este bit es el bit de sistema que indica si el segmento pertenece a uno de los segmentos que utilizará el sistema.
- DPL: el descriptor de nivel de privilegio nos indica, claramente, el nivel del privilegio del segmento. Al utilizar dos bits, este puede tomar cualquier valor entero entre el cero y el tres. Estos definen una jerarquía de privilegios con el nivel cero siendo el más privilegiado. Definiremos nuestros segmentos

con el privilegio cero.
- P: el bit de presente indica si este segmento está presente en memoria. Como no tenemos muchos segmentos y adicionalmente este trabajo práctico se trata de implemtentar un sistema básico, este siempre estará en uno.
- D/B: Este bit nos indica si el segmento utiliza código o data de 32-bit o de 16-bit. Siempre pondremos este bit en cero.
- G: la granularidad nos indica la escala de la variable de límite descripta anteriormente. Cuando este bit está en uno, nos indica que el límite debe ser interpretado como unidades de 4-KBytes mientras que si está en cero se interpreta en Bytes, nosotros no necesitamos tanto tamaño para los segmentos entonces este siempre estará en cero.
- L: este flag nos indica si el segmento contiene código de modo 64-bit nativo. Nuevamente este bit permenecerá en cero ya que no nos aplica.
- AVL: los bits disponibles y reservados, son bits disponibles para uso del software del sistema.

En definitiva segmentos son es una porción de memoria continua que empiezan en la $base$ y terminan en $base + limite$. En el segmento se encuentra datos o código (dependiendo el tipo) y también tiene un propósito definido por el tipo del segmento (de cero a siete es datos mientras que ocho en adelante es código). Como dijimos antes, mayormente utilizaremos dos de los dieciseis modos de los cuales ocho son de datos y los otros ocho son para código.

Nuestra GDT contiene un total de dieciocho entradas. Las cuales son las siguientes:

- Descriptor Nulo: Siempre presente.
- CS0: Segmento de código nivel cero.
- CD0: Segmento de datos nivel cero.
- CS3: Segmento de código nivel tres.
- CD3: Segmento de datos nivel tres.
- VS0: Segmento de video, es el que utilizaremos para imprimir en pantalla para poder ver como nuestros paqueños pueblerinos recorren el mapa.
- TSS Idle: El Segmento de nuestra mas importante. Esta es una tarea nula, lo que significa que los lemmings no hacen nada mas que descansar. Explicaremos mas adelante el uso de esta tarea en la sección Tareas.
- TSS Init: El segmento de nuestra tarea inicial que utilizaremos más adelante en la sección Scheduler.
- Lemmin 0-9: Estos diez segmentos son nuestros lemmings.

Muchos de estos segmentos los explicaremos en mayor detalle en la secciones siguientes.

IV. Interrupciones

Las interrupciones son similares a las funciones, y estas tienen el poder de interrumpir al procesador con el fin de evitar fallas, generar advertiencias, debuggear y lo que más impacta a nuestro trabajo práctico, poder cambiar entre las acciones que hacen nuestros pequeño lemmings.

Las interrupciones, similar a los segmentos también tienen una tabla como la GDT, esta es la IDT o table de descripción de interrupciones definida en $idt.c$. Estas descripciones son mas simples que las descripciones de segmento pero cumplen un propósito similar. Nuestra IDT tiene un total de veinticinco interrupciones definidas en $isr.asm$

- Primeras: Las primeras veinte interrupciones son del sistema, estas incluyen interrupciones cuando se intenta de hacer una división por cero o cuando hay un problema en la paginación o tareas (mas información en Paginación y Tareas respectivamente).
- isr32: Esta es la interrupción mas importante y la que mas interactúa con nuestro sistema. Esta es la interrupción del reloj. Isr32 se encarga de cambiar la tarea activa con el fin de darle un turno a cada lemming. Hablaremos más de tareas y de orden de tales en Tareas y Scheduler respectivamente.
- isr33: Interrupción del teclado. Esta interrupción se genera cada vez que un usuario presiona una tecla del teclado. En nuestro trabajo práctico no cumple ningún rol importante mas allá de activar y desactivar el modo debugger cuando se presiona la tecla ”Y”.
- isr88: Esta es nuestra primera interrupción dedicada a los lemmings. Esta interrupción será llamada exclusivamente cuando un lemming quiera moverse tal como indica la sección Juego.
- isr98: La segunda interrupción de los lemmings. Esta interrupción será llamada cuando un lemming decida sacrificarse explotando.
- isr108: Esta es la última interrupción que nuevamente pertenece a los lemmings. Esta indicará que los lemmings se transformaron en puentes.

V. Paginación

La paginación, como la segmentación, es un método de mapeo a memoria donde se divide la memoria en bloques de 4KB llamadas páginas. Este sistema mapea una dirección física usa tres capas para su mapeo.

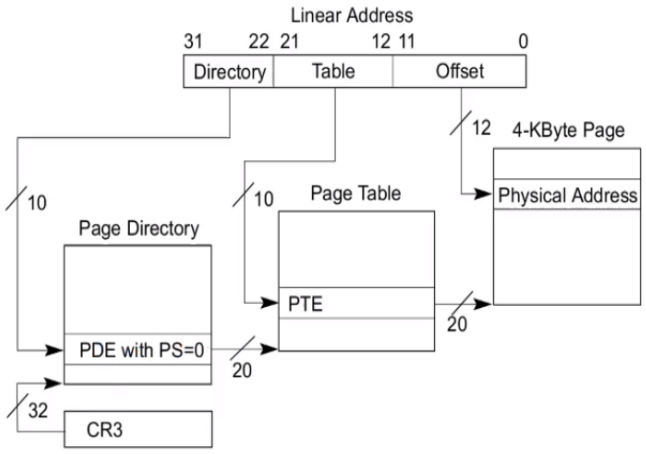

Fig. 2. Inicialización de los lemmings.

Cuando la paginación está activa, el direccionamiento se resuelve tomando el registro CR3 del contexto actual (que indica la base del directorio de páginas) y luego se le suman los diez bits más significativos de la dirección lineal para obtener la base de la tabla de páginas a la cual, a su vez, se le suman los próximos diez bits de la dirección lineal como offset con el fin de obtener la dirección física de la base de la página mapeada. A esto, finalmente, se le suman los últimos tres nibbles de la dirección lineal como offset para poder conseguir la dirección física. En contexto de nuestro trabajo práctico, utilizamos este método para resolver las direcciones de las tareas llevadas a cabo por nuestro sistema. Normalmente, en un sistema operativo, la segmentación es poco usada, por ende, esta implementación es muy importante para nosotros ya que es lo que más se asimila a un sistema operativo real.

VI. Tareas

Las tareas son los procesos principales de nuestro juegos. Como vimos en la sección Segmentación, tenemos definidos doce segmentos de tareas, TSS Idle, TSS Init y nuestros diez lemmings. Las tareas son conocidas como TSS o Task-State Segment, en otras palabras, es un segmento que contiene el entorno de la tarea. Esto significa que se guardan todos los registros y toda la información que utiliza el procesador para completar una tarea particular. Así, podremos cambiar de tarea, y cuando finalmente tengamos que continuar una tarea anterior, cargamos el entorno de la tarea para que continúe como si jamás hubiese cambiado.

Llamamos $lemming$ al conjunto de instrucciones en los archivos $taskLemmingA$ y $taskLemmingB$. Cada TSS comenzará al principio del código y mantendrá cada entorno de ejecución logrando asi que cada Lemming pueda ejecutar todos sus deberes de principio a fin, avanzando al principio de su turno. Cada vez que se haga un cambio de tarea, el contexto del procesador se guardará en este segmento y después se cargará cuando vuelva a ser su turno. Este proceso lo veremos más en detalle en Scheduler.

Un TSS se define de la siguiente forma:

31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

Fig. 3. Estructura de TSS.

Como se puede ver en la imagen, una tss ocupa cientocuatro bytes de memoria (por ende el límite en la GDT será $0x68$). Y se ve gráficamente como se guardan todos los registros de propósito general y los registros de segmento, todo lo que afecta el entorno de ejecución. Como dijimos antes, tenemos doce segmentos:

- TSS Init: Este es un TSS nulo
- TSS Idle: Este TSS apunta a la posición de memoria donde está el código de $idle.asm$ un ciclo infinito que imprime un reloj en la esquina inferior derecha.
- Lemming x: Nuestros diez lemmings. Estos están enumerados desde el cero hasta el nueve. Los Lemmings pares pertenecen al pueblo $Amalin$, mientras que los impares al pueblo $Betarote$.

Como la mayoría de nuestros TSS son lemmings, a continuación veremos como fueron inicializados.

Como podemos ver, esta función que inicializa lemmings recibe un número entero. Este número deberá ser un número decimal de un dígito ya que será utilizado para identifica que lemming será el que se inicializará. Como los pares pertenecen a un equipo y los impares

```
void task_init(int8_t num){

  uint32_t cr3 = mmu_init_task_dir(((num % 2) == 0) ? TASK_LEMMING_A_CODE_START : TASK_LEMMING_B_CODE_START);

  tss_array[num].ss0 = GDT_OFF_DS_0;
  tss_array[num].esp0 = mmu_next_free_kernel_page() + PAGE_SIZE - 1;

  tss_array[num].eip = TASK_CODE_VIRTUAL;
  tss_array[num].esp = TASK_STACK_BASE;
  tss_array[num].cr3 = cr3;

  tss_array[num].cs = GDT_OFF_CS_3 | 3;
  tss_array[num].ds = GDT_OFF_DS_3 | 3;
  tss_array[num].es = GDT_OFF_DS_3 | 3;
  tss_array[num].fs = GDT_OFF_DS_3 | 3;
  tss_array[num].gs = GDT_OFF_DS_3 | 3;
  tss_array[num].ss = GDT_OFF_DS_3 | 3;

  tss_array[num].ebp = TASK_STACK_BASE;
  tss_array[num].eflags = 0x202;
  tss_array[num].iomap = 0x68;

  gdt[gdt_array[num]].base_15_0 = (uint32_t)&tss_array[num] | 0;
  gdt[gdt_array[num]].base_23_16 = ((uint32_t)(&tss_array[num]) >> 16) | 0;
  gdt[gdt_array[num]].base_31_24 = ((uint32_t)(&tss_array[num]) >> 24) | 0;
}
```

Fig. 4. Inicializacion de los lemmings.

al otro, nos tenemos que asegurar de pasar la dirección fisica correcta ya que ambos equipos tienen su código en una dirección distinta.

VII. Scheduler

El scheduler es el código que se encarga de decidir que TSS va a correr y cuando. Como dijimos en la sección Interrupcion, cada vez que se recibe una interrupción de reloj, ésta le pregunta al scheduler como continuar, esto se hace con un Switch. Un switch es el proceso en el cual el entorno del procesador se guarda dentro de una TSS y se carga otro, logrando un cambio de tareas exitoso. Para poder lograr el switch vamos a necesitar una nueva TSS y una anterior para guardar el entorno actual. Sin embargo, la primera vez que se hace un switch, no existe una tarea original para poder guardarlo, por ende tenemos TSS Init con el propósito de ser el segmento de guardado del primer switch.

El tiempo entre una interrupción de clock y lo siguiente es lo que vamos a considerar como *quantum*. El quantum es el tiempo que tendrá un lemming para hacer su acción ya que esta interrupción le pregunta al scheduler cual es la próxima tarea a ejecutarse y generar el switch. Un quantum es un tiempo fijo, por lo que no es algo que necesariamente nuestros lemmings vayan a ocupar completamente luego de hacer su acción. Luego de que los lemmings se muevan, exploten o se transformen en puentes, puede existir una parte del quantum que aún no ocurrió. Esto se resuelve llenando al resto del quantum con la tarea idle. Cada vez que se genere una interrupción de acción de lemming, antes de finalizar la interrupción, se hace un switch con TSS Idle y se mantiene allí hasta que vuelva a ser el turno del lemming correspondiente.

Como dijimos antes, el scheduler se encarga de decidir el orden de las tareas, o en nuestro caso, lemmings. Por las reglas del juego sabemos que cada pueblo puede tener cinco lemmings despachados en un mismo momento, por ende optamos por dos arrays, uno para cada pueblo, que tengan los lemmings de ese pueblo. El scheduler tendrá que ir saltando de array en array avanzando el índice de cada uno con el fin de que los equipos jueguen de modo intercalado y además dentro de un equipo hacer que un lemming no pueda ir dos veces sin que el resto del equipo haya tenido un turno, tal como dicen las reglas en la sección Juego

VIII. Pantalla

En esta seccion discutiremos como manejamos el uso de la pantalla. La pantalla con la que trabajamos es 80x50 la cual dividimos en dos, una que es 80x40 donde esta el mapa y una zona inferior 80x10 donde ponemos informacion como cuantos lemmings uso cada pueblo y un reloj que nos dice a que lemming le pertenece el turno.

El debugger se genera reescribiendo en la parte superior un rectángulo negro de 60x40. Este imprime que hubo en los registros durante la última interrupción. Además pausa el juego y se puede acceder apretando la tecla ”Y”. En caso de no haya habido nunguna excepción, se puede acceder a este modo igual pero los registros no mostrarán nada de valor.

Como al salir del modo debugger tenemos que poder ver el mapa, volvemos a imprimir este en su totalidad. Para el manejo del mapa, tenemos dos mapas separados, uno que contiene el mapa sin lemmings, con todos los puentes y cambios que se hagan en el ambiente. El segundo mapa es identico al primero pero contiene a los lemmings, este es el mapa que el usuario verá en pantalla. Cuando salimos de modo debugger, imprimimos el segundo mapa para visualizar el juego tal como lo veíamos antes. La funcionalidad del primer mapa es saber que oculta cada lemming, ya que con el segundo mapa no sabemos si un lemming está parado sobre pasto o un puente, eso solo lo sabremos mirando el segundo mapa.

IX. Kernel

El kernel es el primer paso de nuestro sistema. Este se encarga de habilitar interrupciones por primera vez, inicializar los segmentos principales, imprimir el mapa en pantalla e inicializar la GDT e IDT. Mientras que lo que mayormente hace es inicializar, es el paso mas crucial y frágil ya que un error puede arrastrase hasta el

punto donde el sistema devuelva un montón de errores en forma de interrupciones o directamente no funcione en absoluto.

X. Páginas compartidas y mapeo a demanda

Para la excepción 14 (Page fault), se pedía que asignáramos páginas a demanda a los lemmings, estas páginas estando en las direcciones de memoria que comparten todos los lemmings del mismo equipo. Para cada equipo creamos dos arrays de 4096 entradas cada uno, en estos arrays se guardan las direcciones virtuales ya mapeadas para cada equipo y sus respectivas direcciones físicas a las que fueron mapeadas. Cuando ocurre un page fault, si el lemming está intentando entrar a una posición de memoria prohibida, es decir fuera del área compartida, es desalojado. En cambio, si la dirección es válida, primero se fija si ya fue mapeada en el array, si es así, lo mapeamos a la misma dirección física que ya figura, en caso contrario usamos una nueva pagina libre de las disponibles al usuario. Para estos dos últimos casos no se desaloja al lemming del sistema, y se continúa saltando a la idle cuando finaliza la ISR 14.