DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Lucas Iván Kruger	799/19	Lucaskruger10@gmail.com
Franco Colombini	341/18	francocolombini2013@gmail.com
Gonzalo Monasterio	153/18	gonzamonas@gmail.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

En el presente trabajo vamos a diseñar e implementar un sistema operativo simple, y un juego (Tutunito) que correrá en dicho sistema, utilizando sus funcionalidades. Vamos explicar y detallar su funcionamiento y nuestras experiencias al desarrollarlo. Para poder correr el sistema se uso el programa Bochs, el cual nos permite emular una computadora IBM con arquitectura Intel-x86.

Figura 1: Imagen tomada del sistema corriendo el juego de Tutunito en Bochs.

2. Caracteristicas del sistema

En esta seccion se mencionaran, de manera general, los componentes necesarios para que el sistema pueda funcionar correctamente. En las siguientes secciones se profundizara respecto a la implementacion de los mismos. Para que el sistema a implementar pueda funcionar, es necesario contar con:

- **La GDT** (global descriptor table) que referencia a distintos segmentos, tanto de codigo como de datos y de TSS (task state segment, los cuales permiten la ejecucion de tareas.
- **Las tareas** que correran como parte del juego, tendran sus correspondientes entradas en la GDT y cada un de ellas referenciara a la TSS de la misma.
- **La IDT** (Interrupt descriptor table) que le permitira al sistema atender las excepciones que ocurran. Esta tabla tendra distintas entradas, cada una asociada a las excepciones que genera el procesador, de manera que cuando una de estas ocurra, el procesador disponga del codigo necesario para atender la excepcion.
- **El scheduler** que permita decidir el orden de ejecucion de las tareas.
- **El codigo del kernel** se encargar a de realizar el pasaje a modo protegido y activar el mecanismo de paginacion, entre otras cosas, para poder comenzar con la ejecucion del sistema.

3. Caracteristicas del juego de Tutunito

En esta seccion vamos a mencionar algunos puntos importantes para poder entender el juego de Tutunito, para mas detalle se puede revisar la documnentacion en el enunciado[].

- **Dinamica general de juego:** En el juego hay dos equipos: Amalin y Betarote. Ambos son pueblos de una raza ficticia llamada ”Lemming”. Cada equipo aparece en esquinas contrarias de la pantalla. La idea es que cada equipo creara Lemmings cada cierto tiempo, que iran avanzando con el fin de llegar al otro lado de la pantalla y poder ganar el juego. El sistema se encarga de jugar, es decir, los lemmings toman decisiones solos.
- **Aparicion y desaparicion de personajes:** Cada equipo tendra su zona de ”spawn”donde se creara un lemming cada cierta cantidad de clocks. Si ese lugar esta ocupado no se crearan. Cada equipo puede crear hasta 5 lemmings, si cada cierto tiempo mas largo hat 5 lemmings en algun equipo, el mas antiguo de ellos sera desalojado. Los desalojos pueden darse por lo visto arriba o por alguna accion, ya sea que el lemming exploto, puso un puente u otro lemming lo mato.
- **Acciones:** Las tareas de los lemmings tienen acceso a distintas syscalls para realizar acciones:
  - **Move:** Recibe una direccion y si el lemming no tiene nada delante se movera. En caso que no pueda moverse, devolvera un numero que da aviso de la razon.
  - **Explode:** el lemming explota, destruyendose a el y a todo a su alrededor (excepto el agua).
  - **Bridge:** Rebice una direccion y, si el lemming tiene agua en frente, construye un puente sobre el agua. En caso de que pueda o que no, el lemming que realizo la accion morira.
- **Puntos:**Los puntos para cada equipo se suman cada vez que un Lemming nace para ese quipo.
- **Relojes:** Los relojes en pantalla buscan mostrar que Lemmings estan vivos actualmente.
- **Modo Debug:** Este modo detiene el juego y nos muestra una pantalla donde podemos algunos registros importantes, asi como datos de la pila y demas.
- **Fin del juego:** Cuando un Lemming de algun equipo toca el lado contrario del mapa, entonces todo su equipo gana y el juego se termina.

4. Implementacion:

En esta seccion se describira, paso por paso, lo realizado en cada uno de los archivos que fueron modificados, a fin de otorgar una idea lo suficientemente clara de las decisiones de implementacion que se optaron. Especıficamente, se espera que sirva como guıa de referencia para poder entender el comportamiento de las funciones implementadas.

4.1. defines.h

Define los indices en la GDT para cada uno de los descriptores de segmentos (ademas de otras direcciones importantes).

4.2. game.c

No definimos funciones en esta seccion.

4.3. gdt.c

Define los descriptores de segmento globales. El primero debe ser el descriptor nulo. Después se definen son dos descriptores de nivel 0 y 3 para código y datos y descriptores de TSS para 5 tareas de Amalin y 5 tareas de Betarote. Los dpl de las tareas Amalin y Betarote son de nivel 3. El lımite de los segmentos se calcula como la cantidad de bloques o bytes menos uno (esto segun est e seteado el flag de granularidad o no) que entran en la cantidad de memoria que se quiere direccionar.

La estructura para una entrada en la GDT esta declarada en el archivo gdt.h y se compone de los siguientes campos:

```
typedef struct str_gdt_entry {
uint16_t limit_15_0;
uint16_t base_15_0;
uint8_t base_23_16;
uint8_t type : 4;
uint8_t s : 1;
uint8_t dpl : 2;
uint8_t p : 1;
uint8_t limit_19_16 : 4;
uint8_t avl : 1;
uint8_t l : 1;
uint8_t db : 1;
uint8_t g : 1;
uint8_t base_31_24;
} __attribute__((__packed__, aligned(8))) gdt_entry_t;
```

4.4. idt.c

Define las entradas para la Interrupt descriptor table. Se definen 20 entradas para las excepciones del procesador, del 0 al 19, las entradas para las interrupciones de reloj y teclado, 32 y 33 y las entradas correspondientes a las syscalls move, bridge y explode, con los numeros 88, 98 y 108 respectivamente. Las syscalls tendran un dpl=3, para que puedan ser llamadas por las tareas, que corren en nivel 3, mientras que las restantes entradas de la tabla tendran un dpl=0. El selector de segmento de todas las entradas es de nivel 0.

4.5. isr.asm

Contiene las rutinas de atencion de interrupciones y las syscalls especificadas en el enunciado. Las variables currentSelector y currentOffset contienen el selector de segmento de la siguiente tarea a ejecutar. En la seccion correspondiente al scheduler se especificara con mas detalle el funcionamiento del mismo. Estas variables se modifican en la rutina de atencion del reloj. A continuación se hace un repaso de las diferentes rutinas y syscalls implementadas y una descripcion de las mismas:

- **Rutina de atención del reloj:** Llama a la función schednextTask (definida en el scheduler) que devuelve el selector de segmento de la siguiente tarea a ejecutar. Para preservar la alternancia en la ejecución, se posee un variable, ultimoPueblo, que indica a que pueblo le corresponde el turno (seteada en el scheduler).Además, posee una variable que acumula la cantidad de clocks para que al llamar a schednextTask se cheque si la variable es divisible por 401 en tal caso se chequea si alguno de los dos pueblos tiene menos de 5 lemmings en juego y si es así se crea uno nuevo.Luego, se chequea si la variable es divisible por 2001 y en tal caso si alguno de los dos pueblos tiene 5 Lemming en juego se desaloja 1.
- **Rutina de atención del teclado:** Imprime el scancode .Tambien esponderá a la tecla Y, activando y desactivando el modo debug, utilizando para esto la variable debug. Además, cuando se desactiva el modo debug, realiza un llamado a la función printMap para restablecer la pantalla.
- **Syscall Move:** Toma como parámetro un numero del 0 al 3 el cual determina para que lado va a moverse y luego llama a ala función moveLemming la cual intenta mover al Lemming un casillero del mapa y retorna por eax un numero que representa si se pudo mover o si tuvo algún problema.Luego, salta a la tarea Idle para completar el quantum que este corriendo.
- **Syscall Explode:** No toma parámetros ni retorna. Llama a a la función explodeLemming la cual convierte en pasto todo lo que se encuentra a un manhatan de distancia (excepto que sea agua) y

a si mismo.Desaloja la tarea y cualquier tarea que haya muerto por la explosión.Finalmente, salta a la tarea Idle para completar el quantum que este corriendo.

- **Syscall Bridge:** Toma como parámetro un numero del 0 al 3 el cual determina para que lado va a crear un puente si es que hay agua en ese casillero. Desaloja la tarea. Finalmente, salta a la tarea Idle para completar el quantum que este corriendo.

4.6. kernel.asm

Contiene todo el codigo del kernel necesario para poder correr el sistema. Realiza el salto a modo protegido, establece los selectores de segmento y la pila, inicializa la pantalla, el manejador de memoria, el directorio de paginas, habilita paginacion, inicializa las estructuras de las tareas, el scheduler, carga la IDT, configura el controlador de interrupciones, carga la tarea inicial, inicializa el juego, habilita las interrupciones y salta a la tarea Idle.

4.7. mmu.c

Contiene todas las funciones necesarias para utilizar el manejador de memoria.

- **void mmu_init(void):** Setea el comienzo de las paginas libres del kernel.
- **paddr_t mmu_next_free_kernel_page(void):** devuelve la direccion de la siguiente pagina libre.
- **paddr_t mmu_init_kernel_dir(void):** Inicializa el directorio del kernel. Completa las entradas del directorio y las entradas de la tabla asociada a la primera entrada del directorio.
- **void mmu_map_page(uint32_t cr3, vaddr_t virt, paddr_t phy, uint32_t attrs):** Mapea una pagina con el cr3, la direccion virtual, fısica y el nivel con el que se desea, indicado por el parametro attrs. Para esto descompone la direccion virtual y setea los campos correspondientes del directorio y la tabla que sean necesarios. Para setear la tabla pide una nueva pagina en memoria utilizando mmu next free kernel page.
- **paddr_t mmu_unmap_page(uint32_t cr3, vaddr_t virt):** Desmapea una pagina con el cr3 pasado por parametro a partir de la direccion indicada.
- **paddr_t mmu_init_task_dir(uint32_t task):** Inicializa un directorio para una tarea Lemming, solicitando una nueva pagina fısica para esto. Mapea el kernel (4MB) con nivel 0, mapea 4 paginas con nivel user segun indica el enunciado y copia el codigo desde el kernel a las posiciones indicadas.

4.8. sched.c

Define el comportamiento del scheduler. A continuacion se enumeran las variables y estructuras utilizadas y sus significados (algunas estan declaradas en otros archivos pero son inicializadas por sched_init):

- **void sched_init(void)**: Inicializa las variables y estructuras para que el scheduler funcione correctamente. Estas serian:
- Dos listas (puebloA y puebloB) de 5 tareas las cuales tienen los selectores de las tareas y si estas están activadas.
- Dos variables que representan cual es la ultima tarea que se corrió para cada pueblo.
- La variable ultimoPueblo que registra cual fue el ultimo pueblo que corrió.
- Las variables LemmingsA y LemmingsB las cuales registran la cantidad de Lemmings para cada pueblo.

- Las Listas OrdenDesalojoA y OrdenDesalojoB las cuales determinan en que orden se tienen que desalojar los Lemmings.
- **uint16_t sched_next_task(void)**: Se encarga de decidir el orden de ejecución de las tareas. Primero chequea si el juego termino o si esta en modo debug, devolviendo en esos casos el selector de segmento de la tarea idle. Si ninguna de estas condiciones se cumple, según el turno, recorre los arreglos puebloA y puebloB , hasta que encuentra la próxima tarea viva y devuelve el selector correspondiente, habiendo antes actualizado la variable ultimoPueblo según corresponda.Además dependiendo de la cantidad de clocks que van Crea un Lemming o lo desaloja.
- **void CrearLemming(void)**: Si la posición adonde se va a crear el Lemming no esta ocupada se busca la primer tarea del pueblo correspondiente que no este activada y se la activa, se actualiza la cantidad de Lemmings , se modifica el puntaje, se actualiza la posicion del Lemming y se muestra el Lemming en el mapa, ademas se lo agrega en la primer posicion no ocupada del orden de desalojo.
- **void DesalojarLemming(void)**: Se desaloja al priemr lemming que esta en la lista de orden de Desalojo, la misma se actualiza para ya no tener en cuenta a este Lemming. Se desactiva la tarea, se actualiza el contador de Lemmings y se lo elimina del mapa.
- **void matarLemmingSiEsta(uint16t x, uint16t y)**: Se llama a esta funcion luego de que un Lemming explota.Se busca en la lista de posiciones de cada pueblo para ver si algun Lemming esta en la posición pasado por parametro y se lo desaloja.
- **void posLemmingActual(uint16t *x, uint16t *y)**:Modifica las variables pasadas por parametro para que retornen la posicion del Lemming actual.
- **char getMapPosition(uint16t x, uint16t y)**:Retorna el carácter que se encuentra en las posiciones pasadas por parámetro del mapa.
- **void modifyMapa(uint16t x, uint16t y, char letra, uint16t colour)**:Modifica el mapa en la posicion pasada por parametro y pone el caracter pasado por parametro.
- **void printMapa()**: Printea cada posicion del mapa y le asigna color dependiendo del carácter.
- **int chequearMapeoDireccion(int32_t dir)**:Chequea si la direccion pasada por parametro esat mapeada y en ese caso retorna la direccion Fisica, caso contrario retorna -1.
- **void pageFault()**:Chequea si a la tarea que se le activo el page fault se le pueden mapear mas tareas y de ser asi lo hace. Caso contrario la desaloja y la elimina del mapa.
- **void printearReloj(uint16_t x,uint16_t y)**:Imrpime la animacion del reloj.
- **void printearVivos()**: Llama a la funcion printeaReloj para cada Lemming que esta vivo.
- **void printearRelojQuieto(uint16_t x,uint16_t y):** printea el reloj sin animacion en x,y.
- **void debug_mode_switch():** sirve como interruptor para el modo debug.

4.9. screen.c

- **void printlibretas()**:Printea las Libretas en la primera linea.
- **void printScanCode(uint8_t scancode)**: Printea el scancode de la letra del teclado presionada en la posición de la pantalla arriba a la derecha.
- **void uint8_t print(const char* text, uint32_t x, uint32_t y, uint16_t attr)**: Imprime el texto pasado por parametro en las posiciones x e y de la pantalla. Esta funcion vino de la catedra, una observacion es que las posiciones x e y estan intercambiadas, primero se debe pasar la posicion y y luego la x, asi (const char* text, uint32_t y, uint32_t x, uint16_t attr) siendo x las filas, y las columnas.

- **void print_dec(uint32_t numero, uint32_t size, uint32_t x, uint32_t y, uint16_t attr)**: Imprime un numero decimal por pantalla en las posicines x e y.
- **void print_hex(uint32_t numero, uint32_t size, uint32_t x, uint32_t y, uint16_t attr)**: Imprime un numero hexadecimal por pantalla en las posiciones x e y.
- **void print_char(const char* text, uint32_t x, uint32_t y, uint16_t attr)**: Imprime un caracter por pantalla en las posiciones x e y.
- **void escanearRegistros(int exception, int32_t* esp3, int16_t ss, int16_t gs, int16_t fs, int16_t es, int16_t ds, int32_t edi, int32_t esi, int32_t ebp, int32_t esp, int32_t ebx, int32_t edx, int32_t ecx, int32_t eax, int32_t err, int32_t eip, int16_t cs, int32_t eflags):** recibe por parametros todo lo que estaba en la pila de nivel 3, en particular recibe todo lo necesario para llenar los campos del debugger. De esa manera logramos guardar todos los registros y estado de la pila.
- **uint8_t encontramosErrorCode(uint32_t intr):** Se fija si la interupcion que le llego por parametro tiene error code o no en la pila.
- **void printRegs()**: printea los registros y demas valores necesarios para el modo debug, tambien si ocurrio o no una excepcion y lo que sea que haya en el stack de nivel 3.

4.10. tss.c

Contiene la declaración de las tss de las tareas (Idle, inicial, 5 Lemmings del pueblo A y 5 Lemmings del pueblo B) y las funciones encargadas de iniciar las mismas.

- **tss initial, tssIdle** Contienen las TSS de las tareas a las que hacen mención.
- **tssLemmingsA y tssLemmingsB:** arreglos de 10 posiciones, donde cada una contiene una TSS para el Lemming correspondiente (5 Amalin, 5 Betarote).

En cuanto a las funciones se tiene:

- **void tss_init(void):**Inicializa las tss's.
- **void tss_init_task(uint32_t player):** Inicializa una tss de LemmingA o LemmingB, segun corresponda. Algunos consideraciones importantes: el campo eip tiene la direccion desde donde comienza a ejecutar la tarea, segun especifica el enunciado. Para el campo cr3 se utiliza la funcion mmu init-task dir que devuelve la direccion de un cr3 ya preparado. Para el stack de nivel 0 se pide una nueva pagina fisica y se lo situa al final de la misma. Ademas modifica el arreglo tssLemmingA (o tssLemmingB) con la tss creada.
- **void tss_init_task(uint32_t player):** Inicializa una tss de LemmingA o LemmingB, segun corresponda. Algunos consideraciones importantes: el campo eip tiene la direccion desde donde comienza a ejecutar la tarea, segun especifica el enunciado. Para el campo cr3 se utiliza la funci on mmu init - task dir que devuelve la direccion de un cr3 ya preparado. Para el stack de nivel 0 se pide una nueva pagina fisica y se lo situa al final de la misma. Ademas modifica el arreglo tssLemmingA (o tssLemmingB) con la tss creada.
- **void tss_initTaskIdle(void):** Inicializa la task idle con los datos correspondientes de acuerdo al enunciado, con el eip en 0x0001C000 e interrupciones activadas.