DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming - Divıdete y come

Organización del Computador II

Segundo Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Chiara Tarzia	368/17	dulcechiara@gmail.com
Maximiliano Martino	123/17	maxii.martino@gmail.com
Guido Lipina	18/17	gmlipina@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

Resumen

En este trabajo práctico implementamos, usando los conceptos de *System Programming*, un sistema mínimo que simula un juego entre dos jugadores sobre un tablero. Además, éste captura los problemas que pueden generar las tareas, y los resuelve de una manera adecuada.

Índice

1. Introducción

En este trabajo práctico tendremos una primera aproximación al *System Programming*, a través de la implementación de un juego. En específico, programaremos todas las partes básicas de un sistema operativo, incluyendo el kernel, GDT, IDT, ISR, MMU, PIC, TSS y scheduler.

Además, el sistema que implementaremos simulará un juego entre dos jugadores, en el que cada uno comenzará con una tarea en el tablero de peso 64, y en cada turno, estas pueden dividirse y/o moverse con el objetivo de comer fruta. Las frutas representan una cantidad fija de puntos que el jugador captura al comerlas, y al final del juego, gana el que tenga mayor puntaje.

Fruta	Puntos
Uva	16
Banana	32
Frutilla	64

La distancia máxima que puede moverse y mirar cada tarea depende de su peso, y a su vez este depende de la cantidad de veces que la tarea se dividió anteriormente de la siguiente manera:

Peso	Distancia máxima a mover	Distancia máxima a mirar
64	1	64
32	2	32
16	4	16
8	8	8
4	16	4
2	32	2
1	64	1

Por último, si dos tareas de distintos jugadores se mueven a una misma casilla se suman los pesos de todas las tareas por jugador y sobreviven las de mayor peso total. El peso total de las correspondientes al otro se agrega al puntaje del que las comió, al igual que el puntaje de la fruta de esa posición (si había). Si tenían el mismo peso, desaparecen todas.

2. Desarrollo

En esta sección describiremos y explicaremos en orden nuestro código de sistema para que corra el juego.

2.1. Modo Real y segmentación

El primer paso para inicializar un sistema es ingresar a modo protegido. Para esto, luego de saltar a la sección de `start`, deshabilitamos las interrupciones con `cli`, cambiamos el modo de vídeo a 80x50 y habilitamos A20. Con la función `lgdt` cargamos la GDT (que explicaremos en la sección GDT), y por último, antes de hacer el `jmp far`, seteamos el bit `PE` de `CR0`.

2.2. GDT

Para la GDT tenemos dos estructuras: la de descriptor, que es la que cargamos con `lgdt` y sus parámetros son su largo y su dirección, y la de entry. En esta guardamos todos los atributos que debe tener una GDTEntry: dirección de `base` y `límite`, `tipo, s, p, avl, l, db` y `g`.

Los entradas que le cargamos a nuestra GDT son:

- Un descriptor nulo con todos sus parámetros en 0.
- Dos segmentos para código nivel 0 y 3.
- Dos segmentos para datos nivel 0 y 3.

- Un segmento adicional que describa el área de la pantalla en memoria que pueda ser utilizado sólo por el kernel.
- La TSS de la tarea *inicial*.
- La TSS de la tarea *idle*.
- Relleno de las posiciones restantes para que la GDT tenga 50 posiciones.

2.3. Modo Protegido

Una vez que ingresamos a Modo Protegido, nuevamente deshabilitamos las interrupciones con `cli` y establecemos los selectores de segmentos. Para esto, cargamos el selector de segmento para datos de nivel 0 a `ds, es, gs` y `ss`, y el selector para vídeo en `fs`. También establecemos la base de la pila de kernel como $0x27000$ cargándole ese valor a `ebp` y `esp`.

Luego, pintamos la pantalla de negro y gris de la manera que queremos que sea nuestro tablero, e inicializamos el manejador de memoria (*MMU*) y el directorio de páginas. Cargamos este último a `CR3` y habilitamos la paginación modificando `CR0`.

2.4. MMU

Para administrar la memoria libre mantendremos dos variables: `unsigned int nextFreeKernelPage` y `unsigned int nextFreeTaskPage`. Estas guardan la próxima página de memoria libre de kernel y de tareas respectivamente.

Para inicializar la MMU utilizamos la función `void mmu_init()` que inicializa las dos variables antes descriptas, apuntando a la primer direción de memoria libre.

Además, implementamos `uint32_t mmu_nextFreeTaskPage()` y `uint32_t mmu_nextFreeKernelPage()`, las cuales devuelven la próxima página libre y actualizan el valor de las variables anteriores, respectivamente.

Aquí también nos encargamos de mapear y desmapear direcciones. Para mapear direcciones implementamos `void mmu_mapPage(uint32_t virtual, uint32_t cr3, uint32_t phy, uint32_t attrs)` que dado un CR3, una dirección virtual, una dirección física y los atributos, mapea la dirección virtual a la dirección física en el contexto del `CR3`, con los atributos indicados. Para lograr esto, de la dirección virtual obtenemos el índice del Page Directory (`virtual>>22`) y el índice del Page Table (`((virtual<<10)>>22)`, luego seteamos la página correspondiente con los atributos indicados, y hacemos que apunte a la dirección física indicada (seteando `pt[indicePT].base = phy >>12`).

En cambio, para desmapear una página implementamos `void mmu_unmapPage(uint32_t virtual, uint32_t cr3)` que desmapea la dirección virtual pasada por parámetro en el contexto del `CR3` indicado. Para lograr esto, de la dirección virtual obtenemos el índice del Page Directory (`virtual>>22`) y el índice del Page Table (`((virtual<<10)>>22)` y seteamos el atributo `present` de esa tabla en 0.

Para inicializar un directorio de páginas y tablas de páginas para una tarea utilizamos la función `pd_entry* mmu_initTaskDir(uint8_t is_copy, uint32_t team)` que pide dos páginas del área libre de tareas (utilizando `mmu_nextFreeTaskPage()`) que utilizamos para almacenar el directorio de paginas y el directorio de tablas. Adicionalmente, pedimos dos paginas libres más, donde copiamos el código de la tarea y mapeamos dicha página a partir de la dirección virtual $0x08000000$(128MB), con los atributos correspondientes (`p=1, us=1, rw=1`) utilizando `mmu_mapTaskPages((uint32_t)pd, PLT1, PLT2, attrs, is_copy, team)`.

La función `mmu_mapTaskPages((uint32_t)pd, PLT1, PLT2, attrs, is_copy, team)` se encarga de mapearle el código y datos a la tarea, preguntando si la misma es una copia (*Divide*), o es la original (*isCopy*: este solo se usa para las 2 primeras tareas, después todas las tareas nuevas que se creen serán mediante divide y se copiara el código y datos desde la dirección $0x08000000$ del `cr3` de la tarea actual). Para ello utilizamos `mmu_mapPage`, anteriormente descrita, y la función `void mmu_copy(uint32_t dir_inicio, uint32_t dir_final, uint32_t dst)`, que se encarga de copiar una determinada dirección de memoria.

Por ultimo, la función `void mmu_initKernelDir()` se encarga de inicializar el directorio del Kernel.

2.5. TSS

Para inicializar la *TSS*, desde el kernel llamamos a la funcion `tss_init`,la cual inicializa los descriptores de la *TSS* que se ubican con la *GDT* de la tarea *idle*, la tarea *inicial* y las dos primeras tareas del juego (una para cada equipo).

2.6. IDT

El siguiente paso que nos toca realizar es inicializar la *IDT* llamando a `idt_init` desde el kernel. Esta función aprovecha la macro provista por la cátedra e inicializa en la *IDT* las entradas para las 20 excepciones requeridas por el sistema. Además inicializa una interrupción para el teclado y una para el reloj. Por último, inicializa en las posiciones $0x47$, $0x48$ y $0x49$ los syscalls read, move y divide respectivamente.

Luego, nuevamente en el kernel, con la función `lidt` cargamos la dirección del descriptor de la *IDT*.

2.7. Controlador de interrupciones

Para configurar las interrupciones, desde el kernel llamamos a las funciones del *PIC* provistas por la cátedra `pic_disable`, `pic_reset` y `pic_enable`.

2.8. Juego

2.8.1. Inicialización y variables globales

Utilizamos las siguientes variables globales a lo largo de la ejecución del juego:

- `CURRENT_TEAM`: Si es 0 el equipo actual es A y si es 1 es B
- `CURRENT_TASK`: Entero del 0 al 9 que indica que tarea se esta ejecutando. En conjunto con `CURRENT_TEAM` permite saber cual de las dos tareas se estan ejecutando
- `FRUIT_COUNT`: Indica la cantidad de frutas actualmente en el juego. En el momento en que no hay mas frutas el juego termina
- `POINTS_A/B`: Dos variables que indican la cantidad de puntos de cada equipo

Luego de inicializar las tareas *inicial*, *idle* y la primera de cada equipo inicializamos la pantalla del juego.

2.8.2. Funcionamiento de la pantalla

Para el funcionamiento de la pantalla utilizamos una matriz de 50x50. Cada posición de la matriz indica de acuerdo a su valor lo que hay en esa celda en ese momento:

Valor	Significado
0	No hay nada
1	Tarea de equipo A
2	Tarea de equipo B
16	uva
32	banana
64	frutilla

Para inicializar la pantalla seteamos la posición inicial de las frutas, las 2 tareas iniciales y todos los detalles con información extra a la izquierda del tablero, es decir, los puntajes de los respectivos equipos e información de las tareas.

2.8.3. Interrupción del reloj y scheduler

Nuestra implementación de la interrupción del reloj incluye el código de interrupción genérico con el agregado de que preguntamos si estamos en Modo Debug o no, junto con el procedimiento a realizar en caso de que salte una excepción si este está activo. Se hablará más de esto último en la sección *Modo Debug*.

La función `sched_next_task` devuelve el descriptor de segmento de la proxima tarea a ejecutarse. Para esto utilizamos la variable global `CURRENT_TASK`, dos vectores de `info_task` para recorrer las tareas y la variable global `CURREN_TEAM`. También chequeamos que solo se ejecuten tareas ”vivas”. En caso de que haya terminado el turno, es decir, se ejecutaron una vez todas las tareas vivas de A y B, se ejecutan las funciones `resolver_conflictos()` y `actualizar_pantalla()` y se setea como tarea actual la primer tarea viva del equipo A.

2.8.4. Información de las tareas

Usamos una estructura llamada `info_task` que contiene toda la información de una tarea dada:

Nombre	Información de tarea
Weight	Peso actual de la tarea
Max_move	Maxima cantidad de celdas que la tarea dada puede moverse
Max_look	Maxima cantidad de celdas que la tarea dada puede ver
X	Posicion en el eje x de la tarea
Y	Posicion en el eje y de la tarea
Is_alive	Indica si la tarea esta viva y puede ejecutar codigo o no
Revive	Indica si al final del turno hay que revivir”la tarea

2.8.5. Resolver conflictos

La función `resolver_conflictos()` se ejecuta al final de cada turno y chequea en el siguiente orden:

- Si hay varias tareas de distintos equipos que estén vivas (`is_alive =1`) y en una misma posición del tablero. En cuyo caso ”mata”las tareas del equipo cuyo peso total sea menor que las del otro equipo y las suma al puntaje del equipo vencedor.
- Si hay tareas y frutas en una misma posición se eliminan las frutas y se contabilizan los puntos al equipo correspondiente.
- Se reviven las tareas que tengan el flag `revivir` en 1. Estos puntos se ejecutan en este orden para evitar conflictos como por ejemplo que 2 equipos se contabilicen puntos de una misma fruta.

2.8.6. Actualizar Pantalla

Luego de ejecutarse `resolver_conflictos()` se ejecuta `actualizarPantalla()`. Esta recorre la posición de todas las tareas vivas, actualiza la matriz del tablero con la información correspondiente y pinta el nuevo estado del juego en la pantalla.

2.8.7. Syscalls

Para implentar los `syscalls` utilizamos en lo posible funciones de C que son llamadas desde `isr.asm`.

2.8.8. Read

Read recibe dos enteros que indican la posición en la que una tarea esta leyendo, busca en la matriz de tablero que es lo que hay en esa posición y devuelve acorde a lo especificado en el enunciado:

Valor de retorno	Elemento en el tablero
0	Si la tarea quiere leer mas lejos de lo que puede
10	Si no hay nada en esa posición del tablero
20	Si hay una tarea del mismo equipo que al de la tarea leyendo
30	Si hay una tarea del equipo contrario que al de la tarea leyendo
40	Si hay una fruta en esa posición del tablero

2.8.9. Move

Move recibe un numero sin signo que indica la cantidad de celdas que se mueve la tarea y una dirección. Como el tablero es circular es importante calcular exactamente a donde se mueve la tarea. Para esto calculamos el valor absoluto de lo que se mueve la tarea módulo el tamaño del tablero y esto se lo sumamos al eje correspondiente de la tarea. En el caso de que se quiera mover hacia arriba o hacia la izquierda le restamos este valor al tamaño del tablero para garantizar que se mueva en la dirección correcta y volvemos a calcular el módulo con el tamaño del tablero.

2.8.10. Divide

Divide es la syscall más complicada ya que es importante que se copie correctamente toda la información de la tarea.

Lo primero que hacemos es validar si la tarea se puede dividir o no. Puede haber 2 razones por los cuales una tarea no pueda dividirse: La primera es si la tarea que quiere dividirse pesa 1 y la segunda es si no quedan tareas disponibles a las cuales dividirse. En este último caso, el peso de la tarea se divide de todas formas pero no se crean tareas nuevas.

Luego buscamos una tarea disponible para dividirse recorriendo nuestro vector de `info_task` del equipo actual y le copiamos toda la información de la tarea (nuevo peso, posición, etc.).

Por último utilizamos *tss_init_task*, para inicializar la nueva tarea correctamente. Esta función recibe una tarea, un equipo, un flag que indica si la tarea es una copia de otra o si es nueva, los registros `ebp`, `eip` y una dirección de retorno. A partir de esto, hace todo el mapeo y copiado de datos y código de la tarea y setea todos los datos en la *TSS*.

La dirección de retorno es la ultima línea de *divide* que chequea si tiene que devolver 0 o 1 dependiendo de la tarea que esta corriendo actualmente (La tarea copiada va a empezar a correr en esta linea en el siguiente turno).

2.8.11. Modo Debug

Para implementar el *Modo Debug*, utilizamos dos variables. La primera, `modoDebug`, que se activa la primera vez que se presiona la tecla *Y*, y se desactiva cuando esta se presiona nuevamente si todavía no ha ocurrido una excepción.

Si ocurriese una excepción mientras el *Modo Debug* está activo, en la rutina de atención de excepciones movemos el valor de todos los registros a una estructura auxiliar que definimos (`infoActual`) y luego llamamos a `imprimirPantallaDebug` que se encarga de imprimir estos valores de forma ordenada en pantalla. Cuando ocurre esto, utilizamos otra variable auxiliar, `modoDebugPantalla`, que nos indica si ocurrió una excepción y estamos mostrando los valores en pantalla. Si presionamos la tecla *Y* en este estado, se desaloja la tarea actual mediante la función `matarTareaActual`, que simplemente setea a la tarea como muerta (`is_alive` = 0), y continua la ejecución del juego normalmente.

2.9. Conclusión

A lo largo de todo el trabajo se presentó la enorme cantidad de inconvenientes que conlleva trabajar sobre la nada. Para realizar el trabajo fue imprescindible entender lo que estaba sucediendo en el kernel a muy bajo nivel, pues constantemente surgen inconvenientes cuya explicación no suele ser evidente.

También algo que notamos es que la dificultad de la implementación asciende al infinito si uno lo permite. En todo momento se presentan encrucijadas sobre cómo realizar algo específico teniendo una libertad casi absoluta al tener permisos de kernel. En general en esos momentos intentamos optar por la

implementación mas clara con una lógica mas solida perdiendo en algunos casos algo de performance. Sin embargo lo que se gano con eso fue un código mucho mas manejable.