DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

subtitulo del trabajo

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Nombre	XXX/XX	`mail`
Nombre	XXX/XX	`mail`

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 el presente trabajo se describe la problemática de ...

Índice

1. Objetivos generales **3**

2. Desarrollo **3**
2.1. Segmentación y manejo de excepciones 3
2.2. Paginación 3
2.2.1. Directorio de kernel 4
2.2.2. Directorios de tareas 4
2.3. Interrupciones 5
2.3.1. Interrupción de reloj 5
2.3.2. Interrupciones de teclado 5
2.3.3. Servicios de sistema 5
2.4. Gestión de tareas 5
2.5. Scheduler de tareas 6
2.5.1. Desalojo de tareas 7

3. Conclusiones y trabajo futuro **7**

1. Objetivos generales

El objetivo de este Trabajo Práctico es ...

2. Desarrollo

A lo largo de este trabajo haremos referencia a identificadores y símbolos utilizados en el código. Para ello usaremos este formato: `variable_o_funcion`.

2.1. Segmentación y manejo de excepciones

De acuerdo a lo indicado en el enunciado, definimos segmentos de código y datos en la GDT a partir del indice 4, un par con privilegios de kernel y otro con privilegios de usuario. Direccionamos los primeros 878MB de memoria con estos segmentos. Para ello, establecimos la base de cada segmento en la dirección 0x0 y, para poder representar el número en 20 bits, calculamos el limite de cada uno en bloques de 4KB. Adicionalmente, definimos un segmento de datos con privilegios de kernel para la memoria de video, basado en la dirección 0xB8000. Dado que las dimensiones de la pantalla son 80x50 caracteres y que para representar cada uno se necesitan dos bytes, el tamaño de este segmento es de 8000 Bytes. En función de esto definimos el limite del segmento en 7999, su ultimo byte direccionable.

Inicialmente, resolvimos el manejo de excepciones definiendo los primeros veinte indices (de 0 a 19) en la tabla de descriptores de interrupción. Para ello utilizamos un macro en el cual referimos cada entrada N al segmento de código de kernel, con el offset correspondiente su rutina de atención _isr$N$. En cuanto los atributos de cada descriptor, dejamos en 1 el bit de presencia, asignamos nivel de privilegios de kernel y usamos el tipo *interrupt*.

Para atender excepciones escribimos rutinas que muestran sus mensajes de error por pantalla. Definimos cada _isr$N$ con un macro en el cual obtenemos y mostramos el N-ésimo mensaje de un arreglo donde los guardamos previamente.

Este comportamiento luego fue revisado para la implementación del modo debug y el desalojo de tareas (sección 2.5.1)

2.2. Paginación

Implementamos paginación de dos niveles, donde construimos las estructuras necesarias en el area libre de memoria, que comienza en la dirección *0x100000*.
Para el manejo de páginas libres utilizamos el puntero `proxima_pagina_libre` que denota la dirección de la próxima página libre, al que inicializamos apuntando al inicio del área libre. Administramos este puntero con la función `mmu_proxima_pagina_fisica_libre`, que devuelve un puntero a la proxima página e incrementa el valor de `proxima_pagina_libre` en 4KB.

A través de estos mecanismos, una vez que ubicamos un directorio de páginas en una página libre, hacemos uso de las funciones `mmu_mapear_pagina` y `mmu_unmapear_pagina` para agregarle y sacarle mapeos de páginas.

En la función `mmu_mapear_pagina` tomamos los siguientes atributos:
METER ESTO EN UNA TABLA:
virtual : dirección virtual a mapear
cr3 : la dirección física del page directory al que queremos agregar una página
fisica : una dirección física
attr : los atributos a definir en las entradas de la tabla y directorio.

A partir de estos parámetros, al mapear una página tenemos en cuenta primero si es necesario crear la entrada en el page directory, revisando el bit **P** de la entrada correspondiente a la dirección virtual. De ser así obtenemos una nueva página libre para la tabla de páginas, que inicializamos en 0 para todas sus entradas, y agregamos la entrada en el directorio apuntando a ella usando los atributos de `attr`. Luego de obtener la page table, sea recién creada o previamente existente, asignamos la dirección física alineada a 4KB con los atributos pedidos a la entrada indicada por `virtual`. Hecho esto quedan mapeadas las direcciones `virtual` a `fisica`, y llamamos a la función `tblflush` para limpiar la TLB.

En el contexto de este trabajo solo usamos los bits **P**, **R/W** y **U/S** de los atributos, que coinciden en ambas estructuras. Como en el esquema que utilizamos no necesitamos entradas en tablas con atributos distintos a los definidos en la entrada de directorio correspondiente, usamos los atributos definidos en `attr` para todas las estructuras requeridas en cada mapeo. Asimismo, como veremos en las secciones subsiguientes, el area libre esta mapeada por identity mapping para el kernel y todas las tareas. Debido a esto la dirección física del directorio de páginas se corresponde con su dirección virtual en todo contexto.

Para la función `mmu_unmapear_pagina` requerimos una dirección virtual y el directorio en el cual queremos deshacer el mapeo. Con estos datos buscamos la entrada en el directorio, la cual de no estar presente no hacemos nada, y la entrada correspondiente en la tabla apuntada por ella. Anulamos ésta última, dejando todos sus bits en cero. Luego verificamos si hay al menos una entrada presente en la tabla y de no ser así, anulamos también la entrada del page directory. Finalmente, limpaiamos la TLB con la función `tblflush`.

A continuación detallamos los procedimientos a través de los cuales construimos los directorios que usamos en este trabajo tanto para el kernel, como para las tareas del juego.

2.2.1. Directorio de kernel

Siguiendo las indicaciones del enunciado, para el directorio del kernel mapeamos las direcciones *0x000000* a *0x3FFFFF* usando *identity mapping*. Para lograrlo en primer lugar inicializamos los primeros 4KB de memoria a partir de la dirección *0x27000* con ceros, donde luego definimos el directorio de páginas del kernel. Ubicamos la primer página del directorio en la dirección *0x28000*, con atributos de lectura y escritura, nivel de privilegios cero y el bit presente activo. Luego, para mapear el sección de memoria pedida, llenamos la tabla con las direcciones de los primeros 1024 bloques de 4KB de memoria física. De esta manera definimos la última entrada de la tabla con la dirección base 0x3FF000, permitiendo direccionar las siguientes 4096-1 direcciones. Al igual que la definición de la tabla de páginas en el directorio, cada página fue definida con atributos de lectura y escritura, privilegios de kernel y bit presente activo.

Teniendo armado un directorio de páginas con una tabla de páginas, habilitamos paginación moviendo la dirección del directorio a CR3 y levantando el bit correspondiente en CR0.

2.2.2. Directorios de tareas

Controlamos la inicialización de directorios para tareas con la función `mmu_inicializar_dir_tarea`. Para ella requerimos los parámetros `tipo`, que indica el tipo de tarea se esta mapeando, y `fisica`, que define la dirección de memoria física dentro del mapa que debemos mapear. Denotamos el tipo de cada tarea con el enum `task_type`, cuyos valores para tareas sanas, de virus A y B son: `H_type(0)`, `A_type(1)` y `B_type(2)`.

En esta función debemos copiar el código adecuado y construir el directorio y páginas necesarias para que una tarea encuentre sus instrucciones a partir de la dirección virtual *0x8000000* y pueda mapear otra página en la posición en el mapa a la dirección virtual *0x8001000*. Para ello usamos el directorio apuntado por CR3 al momento en que se llamó la función para copiar el código, mapeando la dirección física con identity mapping temporalmente usando atributos de kernel y escribiendo el código de la tarea. Obtenemos éste último calculando el desplazamiento a partir de la dirección *0x10000*, que resulta del producto entre el tamaño de página y el orden en memoria correspondiente al tipo de tarea. Una vez

escrito el código, desmapeamos la página usada del directorio actual y construimos el de la tarea. Con este objetivo obtenemos una página para el directorio, que inicializamos con ceros, y seguimos el mismo procedimiento descrito para el directorio de kernel. Una vez mapeada el area de kernel en el directorio de la tarea, vinculamos la posición física indicada a la dirección virtual 0x8000000. Consideramos que en un principio la tarea no esta atacando (mapeando la dirección) a otra tarea, por lo que también vinculamos la posición virtual 0x8001000 a la misma dirección física.

2.3. Interrupciones

2.3.1. Interrupción de reloj

2.3.2. Interrupciones de teclado

2.3.3. Servicios de sistema

2.4. Gestión de tareas

REVISAR/REESCRIBIR DESDE ACA

Para tareas usamos una inicial, con tss vacia, a donde metemos el contexto al momento de realizar el primer salto. Este salto lo hacemos justo despues de habilitar interrupciones y es la ultima linea de código de kernel que se ejecuta.

Todos los descriptores de TSS que cargamos estan en el anillo de privilegio 0. La tss de la tarea inicial la cargamos vacía y ubicamos su descriptor en el indice 9 de la GDT. Para la tarea idle usamos un descriptor en el indice 10 de la gdt, apuntando a una tss donde definimos un contexto de kernel (esp a la base del stack de kernel, directorio de paginacion, segmentos de datos y codigo de kernel).
REVISAR/REESCRIBIR HASTA ACA

Para gestionar el estado de las tareas del juego utilizamos una matriz, `tareasInfo`. Esta matriz tiene dimensiones 3x15, para acomodar todas las tareas en juego. La primer coordenada se corresponde con el enum `task_type`, mientras que la segunda indica el indice o id de tarea. Siempre que se accede guardamos el recaudo de mantenernos solo en los indices válidos para cada tipo de acuerdo a la función `task_type_max`, que devuelve 14 para tareas sanas, y 4 para los virus. El tipo de esta matriz es una estructura que llamamos `task_info`. En ésta guardamos los siguientes campos:

METER ESTO EN UNA TABLA:
alive : flag para denotar si una tarea esta en juego.
owner : dueño de la tarea. Utiliza el tipo enumerado `task_type`
x,y : coordenadas en el mapa donde esta ubicada la tarea
mapped_x y mapped_y : coordenadas en el mapa de la pagina mapeada por la tarea
gdtIndex : indice de la gdt correspondiente a esta tarea

Para manejar los TSSs de las tareas del juego optamos por una matriz, llamada `tss_directory`, de la estructura `tss`. Esta matriz tiene las mismas dimensiones que `tareasInfo`, de manera que los indices de cada tss se corresponden con los de la tarea cuyo contexto representa.

Utilizamos también estos indices para definir en que posición de la GDT ubicamos el descriptor de TSS, correspondiendo cada posición válida en la matriz con una ubicación en la GDT. De esta manera, partiendo del indice 11 de la tabla de descriptores globales, los primeros 15 consecutivos corresponden a las tareas sanas, los siguientes 5 a tareas del virus A y los últimos 5 a tareas del virus B. En función de esto, para cada tarea de `tareasInfo` calculamos su posición en la GDT con la siguiente formula: $11 + offset(tipo) + indice$

Basándonos en estas variables y estructuras, al lanzar una tarea creamos su contexto a partir del tipo de tarea, su indice en `tareasInfo`, su indice en la GDT, y la dirección de memoria física correspondiente a su posición en el mapa. Para ello generamos un directorio de paginas siguiendo el procedimiento descrito en la sección 2.2.2 y obtenemos una página de memoria libre para la pila de nivel 0 de la tarea. Luego cargamos su descriptor en la entrada de GDT indicada y llenamos la tss ubicada en `tss_directory[tipo][indice]` con los siguientes datos:

METER ESTO EN UNA TABLA:
esp0 : pagina de memoria libre
ss0 : seg datos kernel
cr3 : dir paginas creado para la tarea
eip : 0x1000
esp y ebp : final de la pagina de codigo de la tarea (0x8001000)
regs de prop gral : 0
flags : 0x202
cs : seg codigo user
ds y demas seg datos : seg datos user

Al inicializar las estructuras del juego y scheduler, dejamos en 0 todos los campos de cada entrada en `tareasInfo`. Hecho esto se lanzan las 15 tareas sanas en posiciones al azar dentro del mapa usando el procedimiento que describimos en esta sección. A continuación se detalla cómo implementamos los saltos entre tareas y utilizamos las estructuras que introducimos aquí.

2.5. Scheduler de tareas

Una vez terminada la configuración del procesador y construidas las estructuras necesarias, saltamos a la tarea idle. A partir de ese momento controlamos los saltos entre tareas a través de un scheduler. Como se mencionó en la sección **??**, en cada interrupción de reloj el código del scheduler decide si debe saltar, y de ser asi a que tarea, hasta el próximo tick de reloj.

Manejamos esto con la función `sched_proximo_indice`, que retorna el indice en la gdt de la tarea a la que se debe saltar, o cero si no es necesario cambiar de tarea.

Ésta se apoya en las siguientes variables:

`en_idle`, un flag que denota si el último salto fue a la tarea idle
`tareasIndices`, un arreglo de tres posiciones indexado por el enum `task_type`, que guarda cual fué la última tarea que se ejecutó para cada tipo. Tambien nos referiremos a este como el indice actual para un tipo
`tareasInfo`, la estructura descrita en la sección anterior. `currentType`, de tipo `task_type`, que guarda el tipo de la tarea corriendo actualmente.
`currentIndex`, un entero que indica el indice de la tarea correindo actualmente.

Estas variables se inicializan a la vez que `tareasInfo`, con todos los valores en 0. En primer lugar controlamos si estamos en un estado de interrupción por debug, en cuyo caso no debe cambiarse de tarea. Más detalles sobre este comportamiento se describen en la sección **??**.

Para buscar el indice en la GDT de la próxima tarea iteramos circularmente tres veces, guardando en la variable `nextType` el tipo que estamos considerando en cada iteración. Ésta variable toma inicialmente el tipo siguiente al actual, siguiendo el orden de tareas sanas luego virus A y por último virus B. Al ciclar tres veces, `nextType` toma el valor del tipo actual en la última vuelta, para manejar el caso en que todas las tareas corriendo son del mismo tipo.

A partir de esto, obtenemos el indice de la última tarea ejecutada para este tipo y partiendo de el, recorremos circularmente todos los indices válidos para el tipo en orden de menor a mayor, sin pasar por el último utilizado.

Si la tarea correspondiente al par tipo-indice con los cuales estamos iterando tiene su flag `alive` alto en `tareasInfo`, bajamos el flag `en_idle`, actualizamos las variables `currentType` y `currentIndex` al ti-

po e indice que encontramos, así como la entrada correspondiente en `tareasIndices`, y devolvemos el indice en la GDT que almacenamos en `tareasInfo`.

Si al ciclar sobre los indices del tipo `nextType` no encontramos ninguna tarea con indice distinto al actual y la entrada en `tareasInfo[nextType][indiceActual]` tiene su indicador `alive` en 1, estamos en un caso particular. Esto se da cuando hay solo una tarea corriendo para un tipo en particular. En esta situación cambiamos a esa tarea si su tipo es distinto al actual, para lo que indicamos que no estaremos en la tarea idle y actualizamos las variables de tipo e indice actuales antes de devolver el indice de la GDT correspondiente.
Si por el contrario el tipo es el mismo que el actual y el campo `alive` correspondiente es igual a 1, estamos en el caso en que solo hay una tarea corriendo, por lo que se sigue ejecutando y no necesitamos cambiar a otra. En estas circunstancias devolvemos 0, excepto que se estuviera ejecutando la tarea idle. Si así fuera, es necesario saltar a la única tarea existente, para lo que solo bajamos el flag `en_idle` y retornamos el valor indice de la GDT almacenado para esa tarea.

Siguiendo este procedimiento implementamos el comportamiento descrito las consignas de este trabajo para la distribución del tiempo de ejecución entre las tareas. A continuación detallamos los mecanismos que utilizamos para lanzar y dasalojar tareas.

2.5.1. Lanzado y desalojo de tareas

Como describimos en la sección anterior, dejando de lado el orden, el factor determinante para definir si se salta o no a una tarea es el campo `alive` en su entrada correspondiente en `tareasInfo`. Vale la pena recordar que, de acuerdo a lo mencionado en la sección **??**, este campo se deja en 0 al inicializar las estructuras.

Tal como se describió en la sección **??**, cuando un jugador presiona la tecla adecuada, se lanza una tarea de su tipo en la posición de su cursor.

3. Conclusiones y trabajo futuro