Práctica - Programación en modo protegido

Organización del Computador 2

1er Cuatrimestre 2017

Introducción a modo protegido

Ejercicio 1

En modo real, calcular la dirección física indicada por `8123:FFEC`.

Ejercicio 2

Sea la siguiente GDT:

índice	Entrada GDT
0h	NULL
1h	Dir. Base = 71230h
2h	Dir. Base = 72230h
⋮	
Ah	Dir. Base = 7A230h
Bh	Dir. Base = 7B230h

a) En modo protegido, calcular la dirección física indicada por `0058:0000FFEC`. Recordar que los últimos tres bits del selector de segmento están reservados y por lo tanto no son utilizados para indexar la GDT.

b) ¿Y si la dirección fuera `0059:0000FFEC`?

Paginación

Ejercicio 3

Indicar las diferencias entre dirección de memoria lógica, lineal y física.

Ejercicio 4

Enumere toda la información que debe contener una entrada de un directorio o tabla de páginas.

Ejercicio 5

a) Describa como completaría las primeras entradas de la GDT respetando las siguientes características de los segmentos:

Comienzo	Tamaño	Uso	Atributos
0 Gb	2 Gb	código	No Lectura / nivel 0
2 Gb	1.5 Gb	datos	Escritura / nivel 3
512 Mb	4 Kb	datos	No Escritura / nivel 0
3 Gb	256 Mb	código	Lectura / nivel 3

Indique los valores `base` y `límite` en hexadecimal. Dibuje los segmentos indicando como se solapan.

b) Escriba todas las entradas de las estructuras que se requieran para construir el siguiente esquema de paginación, suponiendo que todas las entradas no mencionadas son nulas.

Rango virtual	Rango físico
0x20000000 a 0x20004FFF	0x5AA0A000 a 0x5AA0EFFF
0xC0000000 a 0xC0002FFF	0x000AA000 a 0x000ACFFF

Todos los rangos incluyen el último valor. Se deben setear los permisos como supervisor.

c) Resuelva las siguientes direcciones, desde lógica a física, pasando por lineal; utilizando las estructuras construidas en los ítems anteriores. Indique si se produce un error de protección y en qué unidad se genera.

- `0x0008:0x2000171A`
- `0x0010:0x40002832`
- `0x0018:0x0000071A`
- `0x0020:0x00000001`

d) Suponiendo que se construye un esquema de paginación donde ninguna página utiliza identity mapping, ¿es posible activar paginación en estas condiciones?

Ejercicio 6

Se desea tener una función que dada una dirección de memoria correspondiente a la base del directorio de páginas, y una dirección física, devuelva un valor correspondiente a la cantidad de direcciones virtuales distintas por las cuales se puede acceder a dicha dirección.

La signatura debe ser *unsigned int cantidad_direcciones(unsigned int base_dir, unsigned int fisica);*

a) Escriba el código en C de la función, contando solo direcciones que se puedan acceder en nivel Supervisor.

b) Modifique la función anterior para considerar las direcciones en nivel de Usuario.

c) Si el valor devuelto por los llamados a las funciones anteriores con una determinada dirección física son 0, ¿se puede suponer que la página que corresponde es una página física libre? Justifique.

Nota: Considerar que los marcos de página son de 4kb.

Ejercicio 7

Suponga una máquina con arquitectura IA-32, sin extensiones de memoria (utiliza 32 bits de direccionamiento), con un sistema operativo que trabaja con un modelo flat de segmentacién de memoria, cubriendo los 4GB, y utiliza paginación de 4KB en dos niveles (directorio de tablas de páginas y tabla de páginas).

Se tienen dos procesos ejecutando en nivel de privilegio 3, $Proc\_1$ y $Proc\_2$. Dichos procesos desean compartir espacio de memoria para realizar tareas colaborativamente. Como están en privilegio 3, ésto deben solicitarlo al sistema operativo. Cada proceso tiene su propio CR3 apuntando al directorio de página particular de ese proceso.

Para esto el sistema provee la siguiente llamada: *linear_address *getSharedPage(int other_process);*
y cuenta con las siguientes estructuras:

```
typedef struct {
        int id1;
        int id2;
        physical_address physicalAddress;
        int assigned;
} sharePage;

sharePage sharedPages[MAX_SHARED];
int firstFree = 0;
```

Dos procesos que desean compartir memoria deben ejecutar dicha llamada, pasando como parámetro el número del otro proceso:

a) Proc 1 ejecuta *getSharedPage(id Proc 2)*;
b) Proc 2 ejecuta *getSharedPage(id Proc 1)*;

Se pide:

a) Escribir el pseudo-código de la function *getSharedPage*, indicando claramente cómo el sistema actualiza las estructuras de paginación para este propósito. Cuenta con la primitiva: *getNextFreeFramePage()* que devuelve una dirección de memoria física libre para ubicar una página, y *getNextFreePageEntry()* que devuelve la próxima dirección lineal cuyo `Page_Entry` se encuentra disponible.

b) Suponiendo que la llamada al sistema se hiciera a través de la `int 0x80`, ¿cómo armaría el descriptor de la IDT para administrarla? Teniendo en cuenta el descriptor armado en el punto anterior, indicar qué debe resguardar el handler de la interrupción para asegurar el correcto funcionamiento del proceso llamador.

Ejercicio 8

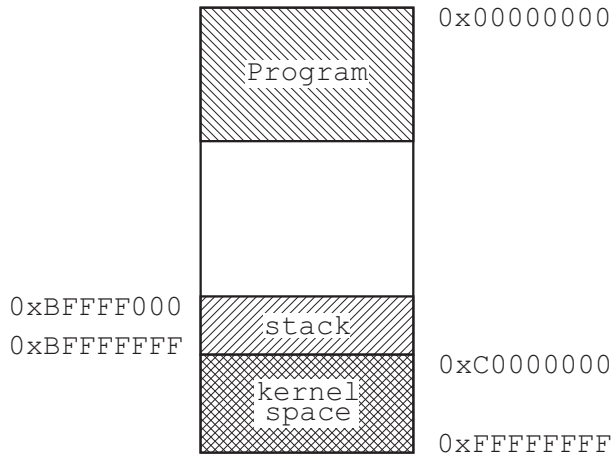
Escribir el pseudocódigo de una función para cargar un programa desde un espacio de memoria secundario.
La función deberá tener la siguiente aridad: *LoadProgram(Iterator)* → *PageDirectory*

Siendo `Iterator`, el iterador para acceder al espacio de memoria secundario y `PageDirectory` la dirección física del mapa de memoria virtual construido.

El `Iterador` guarda en la dirección virtual `BUFFER` los primeros 4kb del programa. Utilizando la función `NextBuffer` se cargan los siguientes 4kbs, además esta función retorna la cantidad de bytes cargados en el buffer. Los programas poseen como mínimo 4Kb y como máximo 2Gb.

LoadProgram deberá construir el siguiente mapa de memoria virtual:
Par esto, se poseen las siguientes funciones auxiliares:

- **GetNewKernelPage() → addr**
  Retorna la dirección virtual de una página de memoria libre de Kernel
- **GetNewUserPage() → paddr**
  Retorna la dirección física de una página de memoria libre de Usuario
- **Phy(addr) → paddr**
  Resuelve una dirección virtual en una física usando el CR3 actual
- **Clean(addr)**
  Setea a `0` la pagina apuntada por la dirección virtual `addr`



- **LoadKernelSpace(paddr)**
  Considera a `paddr` como la dirección física de un directorio de páginas de usuario y actualiza el mapa de memoria para mapear las páginas del kernel.

- **CopyPhysicalData(source_addr, dest_paddr, size)**
  Copia `size` bytes desde la dirección virtual `source_addr` a la dirección física `dest_paddr`

Interrupciones

Ejercicio 9

Cuando IBM diseñó la primera PC, utilizó interrupciones que figuraban como reservadas del procesador para otros dispositivos. Esto llevó a la superposición de interrupciones del procesador con interrupciones del sistema. ¿Cómo le parece que se puede resolver? Recuerde que el procesador Intel x86 tiene solamente un pin de interrupción de hardware y dos controladores de interrupción (PIC 8259) que manejan las 15 interrupciones del sistema.

Ejercicio 10

En un sistema tipo en dos niveles con segmentación *flat* y paginación activa se desea implementar una extraña funcionalidad. El sistema cuenta con un *scheduler* que se encarga de ejecutar una a una las tareas durante un tiempo. Las tareas son intercambiadas desde una rutina que comienza ejecutando la instrucción *PUSHAD*, siendo esta la única que afecta la pila de la tarea a ser desalojada. Todos estos registros son salvados en la pila y reestablecidos al continuar la ejecución.

La funcionalidad a implementar asociada a la interrupción `0x77`, consiste en desarrollar una rutina lea un registro de una tarea en particular y lo retorne a la tarea llamadora.

Se cuenta con la siguiente función,

- *tss* dame_tss(id task)*: retorna el puntero a la `tss` de la tarea `task`.

La función pedida debe respetar,

- En `eax` se pasa el `id` de la tarea cual leer.
- En `bl` se pasa número del registro a leer ordenado desde 1 en adelante como `edi`, `esi`, `ebp`, `esp`, `ebx`, `edx`, `ecx` y `eax`.

a) Escriba el código en ASM de la interrupción `0x77`.

Recordar:

```
  Code | Mnemonic  | Description
-------+-----------+-------------------------------------------------------
   60  | PUSHAD    | Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI
```

Ejercicio 11

Se cuenta con un sistema operativo básico al cual se le desea agregar una llamada al sistema para poder obtener la tecla presionada por el usuario. Sólo en el momento de ser presionada la tecla será capturada por el sistema, pasando sólo el caracter ascii al programa que *lo solicite*. La interrupción se la quiere implementar a través de la **interrupt gate** número **90** (*dame_tecla_presionada*). La tarea que la llama obtiene la tecla que presionó el usuario en el registro `al`. En caso de que el usuario no haya presionado ningúna tecla, la tarea se queda a la espera de que lo haga, es decir, la interrupción no retorna hasta que haya una tecla para devolver. Para ello, se marca como 'a la espera de una tecla' y se pone a correr la próxima tarea disponible en el sistema.

a) Describir la entrada en la IDT para esta nueva interrupción, la misma se debe poder acceder desde el anillo de protección 3.

b) Programar en Assembler la interrupción *dame_tecla_presionada*.

c) Programar en Assembler el handler de la interrupción de teclado.

   Se cuenta con las siguientes funciones y variables:

   - *task_wait_key*: Variable que indica el índice en la gdt que corresponde a la tarea a la espera por una tecla, de ser nulo no hay tarea a la espera.
   - *void add_to_scheduler(gdt_index i)*: Agrega la tarea indicada por el índice en la gdt al scheduler.
   - *void remove_from_scheduler(gdt_index i)*: Quita la tarea indicada por el índice en la gdt del scheduler.
   - *int get_key_from_buffer(char* a)*: Toma una tecla del buffer de teclado y la escribe en `a`. Si no hay tecla en el buffer retorna `0` y `1` en caso contrario.
   - *void add_key_to_buffer(char a)*: Agrega una tecla al buffer pasada por parámetro.
   - *unsigned short next_task()*: Retorna el selector de segmento de la TSS de la próxima tarea a ejecutar.
   - *char scancode_a_ascii(char scancode)*: Retorna el ascii del scancode que es pasado como parámetro.

Nota:

- Se garantiza que en ningún momento va a haber más de una tarea esperando por una tecla de manera simultánea.

Tareas

Ejercicio 12

Mencionar cual es la información mínima que debe contener una estructura de `TSS` y su descriptor en la `GDT`

Ejercicio 13

Se cuenta con un sistema operativo básico al cual se le desea agregar una llamada al sistema que permita crear `subtareas`. La misma se la quiere implementar a través de la **interrupt gate** número **60**. La llamada toma como parámetro la posición de memoria a partir de donde se va a empezar a ejecutar la subtarea una vez creada. También toma como parámetro un dato de 32 bits que se le va a pasar a la subtarea. Estos parámetros se pasan a través de los registros `eax` y `ebx`, respectivamente. El prototipo de la llamada es el siguiente:
void crear_subtarea(unsigned int eip, unsigned int dato)

Ejemplo de uso:

```
int main() {
  ...
  crear_subtarea(imprimir_valor, 0xBABA)
  ...
}
```

```
void imprimir_valor(int valor)
{
  printf("valor: %d\n", valor);

  exit(0);
}
```

Se pide:

a) Describir los datos correspondientes de la entrada en la IDT para esta nueva interrupción. La misma se debe poder acceder desde el anillo de protección 3.

b) Programar en C la función *crear_subtarea* para ser llamada desde la nueva interrupción. La subtarea se debe ejecutar en el anillo de protección 3, con el mismo mapa de memoria que la tarea padre. La pila será una página libre que se debe mapear con identity mapping.

   Cuenta con la definición de las estructuras de `gdt_entry`, `tss` y las siguientes funciones:

   - *dame_tss()*: Retorna el puntero a una TSS vacía.
   - *dame_cr3()*: Retorna el cr3 de la tarea actual.
   - *dame_gdt()*: Retorna el índice de una entrada libre en la GDT.
   - *dame_base_gdt()*: Retorna la dirección base de la GDT.
   - *dame_pagina_libre_usuario()*: Retorna la dirección física de una página libre de usuario.
   - *mapear_pagina(dir_virtual, dir_fisica, cr3, permisos)*: Realiza el mapeo entre la dirección virtual y la dirección física. Además, setea los permisos (bits 0 a 2).
   - *desmapear_pagina(dir_virtual, cr3)*: Elimina el mapeo entre la dirección virtual y la dirección física.
   - *dame_segmento_codigo_usuario*: Retorna índice en la GDT donde se encuentra el segmento de código de usuario.
   - *dame_segmento_datos_usuario*: Retorna índice en la GDT donde se encuentra el segmento de código de datos.
   - *agregar_a_scheduler(indice_gdt)*: Agrega una tarea, identificada por el índice en la GDT de su TSS, al scheduler.

Nota:

- La rutina creada se ejecuta en nivel 3 y SOLO puede ser interrumpida por rutinas a nivel 1 y 0.
- Todos los selectores de segmento de datos se setean con el mismo segmento de datos.

Ejercicio 14

En el sistema operativo Orga2SO, cada tarea se ejecuta durante una cantidad de ticks determinada por la constante `QUANTUM`. Cada tarea puede estar en distintos estados: `CORRIENDO`, `DURMIENDO`, `DIFUNTA` o `LISTA` (para correr). Para llevar cuenta de esto, las tareas se mantienen en un arreglo donde se almacena el índice en la GDT donde se encuentra su descriptor de TSS, el estado de la misma y la cantidad de ticks por el cuál la tarea va a estar dormida (0 si la tarea no está dormida). Las tareas se ejecutan (solo se pueden ejecutar aquellas que están en estado `LISTA`) en orden desde el principio del arreglo (y de manera cíclica). En caso de que ninguna tarea esté en condiciones de ejecutarse se ejecuta la tarea `IDLE`.

Las estructuras de datos utilizadas son la siguientes:

```
typedef enum {
  CORRIENDO = 0, DURMIENDO = 1, DIFUNTA = 2, LISTA = 3
} estado_t;

typedef struct {
  estado_t estado;
  unsigned short indice_gdt;
  unsigned short despertar_en;
} __attribute__ ((__packed__)) tarea_t;

tarea_t tareas[CANT_TAREAS];
unsigned short indice_tarea_actual;
```

Se pide:

a) Implementar en lenguaje ensamblador el código correspondiente al scheduler para que se ejecute en la interrupción del timer tick.

b) Se desea poder poner a dormir a una tarea mediante la interrupción `0x60`. Implementar en lenguaje ensamblador el handler de la interrupción, en `_isrDormir`, recibiendo por `ax` la cantidad de ticks de reloj que la tarea va estar dormida. Al ser llamada la interrupción, la tarea que la llamó se debe poner a dormir, y se debe pasar a ejecutar la siguiente tarea en estado `LISTA`.

c) Implementar en lenguaje ensamblador la interrupción de teclado de modo que cada vez que se presione la tecla `Del` se mate a la tarea actual, es decir, se cambie su estado a `DIFUNTA` y se pase a ejecutar la próxima tarea disponible.

Aclaraciones:

* El tamaño del tipo de datos `estado_t` es de 4 bytes.

* La tarea `IDLE` se encuentra en la posición 0 del arreglo de tareas y no se puede matar.

* Se puede llamar a una función `proxima_tarea_lista` que devuelve en `ax` el índice de la próxima tarea en estado `LISTA`. Si no hay ninguna tarea en estado `LISTA`, esta función devuelve la tarea `IDLE`.

* Se puede llamar a una función `decrementar_tick` que decrementa el campo `despertar_en` de cada tarea en el arreglo `tareas` que esté en estado `DURMIENDO`. En caso de que el campo `despertar_en` llegue a 0, esta función actualiza el campo `estado` a `LISTA`.

* El scan code de la tecla `Del` es `0x53`.

* Tener en cuenta que varias funcionalidades a implementar van a ser utilizadas en todos los ítems de este ejercicio.

Proteción

Ejercicio 15

Suponiendo que se está utilizando un modelo de segmentación flat con segmentos de 4 GB superpuestos, y se tiene activada paginación, ¿es posible proteger de alguna forma que el usuario modifique el código durante la ejecución del mismo? Justifique.

Ejercicio 16

La entrada en el directorio de páginas de una página tiene permisos de Supervisor y Sólo-Lectura; la entrada en la tabla de páginas de la misma página tiene permisos de Usuario y Sólo-Lectura.

a) ¿Qué sucede con los permisos efectivos de esta página? Suponga que el flag `CR0.WP` es cero, y considere los casos de acceso por parte de código corriendo con privilegio de Usuario, y código corriendo con privilegio de Supervisor.

b) ¿Qué sucede si el flag `CR0.WP` es uno? Explique el comportamiento de este flag.

Ejercicio 17

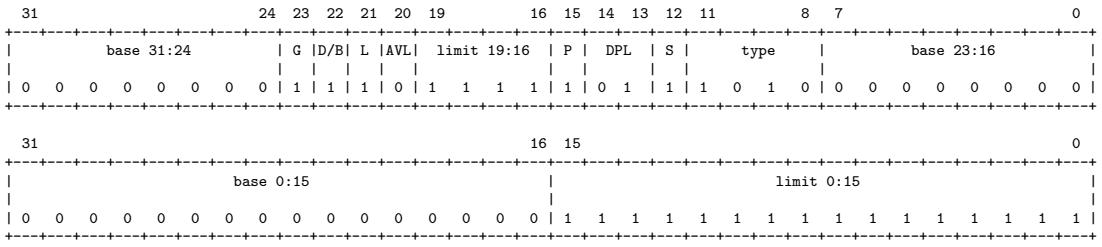
Conteste y justifique las siguientes preguntas:

a) En un sistema que sólo implementa segmentación, donde ninguno de los segmentos se solapa: ¿es posible ejecutar datos y modificar código? (sin modificar los segmentos).

b) Si en el item anterior algunos segmentos se solaparan, ¿sería posible?

c) En un sistema que implementa el modelo de segmentación flat: ¿es posible impedir la modificación de código por parte de un programa de usuario?

d) Dado el siguiente descriptor de segmento:



¿se puede ejecutar desde un segmento de código con CPL = 00?

Integradores

Ejercicio 18

Se tiene un sistema operativo básico, al que se le desea agregar hibernación. Para eso, los diseñadores tomaron la decisión de meterlo en la IRQ0, antes de ejecutar el *scheduler*. La interrupción IRQ0 se encuentra manejada con una **interrupt gate**. El PIC está mapeado a partir de la `int 0x30`.

El proceso de atención al hibernador consiste en:

a) Verificar cuantos quantum lleva el sistema de inactividad. Para eso se tiene la llamada al sistema en la `int 0x80`, que devuelve en eax la cantidad de quatum de inactividad. Esta funcionalidad está implementada con una **task gate**, con su correspondiente punto de entrada $quantum_{80}$.

b) Si la cantidad de quantum supera el umbral, se debe pasar a hibernación utilizando la `int 0x81`, sin parámetros. La hibernación está implementada como una **tarea** independiente, con TSS_{hiber}.

Se pide:

a) Escribir el pseudo-código de la rutina de atención de la interrupción 0.

b) Detallar TODOS los descriptores en la IDT y GDT necesarios para poder llevar adelante este procedimiento.

Ejercicio 19

En el sistema operativo *Orga2SO*, las tareas para ejecutar se mantienen en una lista circular. La idea es que cada tarea ejecute una cantidad de ticks o *QUANTUM* y luego se pase el control de la ejecución a la próxima. También se pueden agregar nuevas tareas para ejecutar. La lista circular tiene los siguientes métodos:

- *unsigned int actual(Lista* l)*: devuelve el índice del descriptor de la GDT para el TSS de la tarea actual.
- *void proximo(Lista* l)*: avanza al siguiente elemento dentro de la lista.
- *void agregarAtras(Lista* l, unsigned int indice_descriptor)*: agrega en la última posición de la lista el descriptor dado.

El puntero a la lista está ubicado en *running*. La GDT es un arreglo de *gdt_entry* y está ubicada en *gdt_base*. Además se cuenta con las funciones

- *unsigned int next_free_gdt_entry()*: devuelve un índice de una entrada libre en la GDT
- *void* get_free_tss()*: devuelve un puntero a una nueva TSS vacía (inicializada con ceros).

Se pide:

a) Implementar en lenguaje ensamblador el código correspondiente al scheduler para que se ejecute en la interrupción de timer tick. Para cambiar de tarea se debe hacer un *JMP FAR* a una etiqueta, ya que no se puede hacer a un registro. Se debe comenzar con el siguiente código:

```
extern actual
extern proximo
extern running
extern QUANTUM
```

b) Implementar una función en C, cuyo prototipo sea *void crear_tarea(unsigned short code_segment, unsigned int initial_eip)*, que realice lo necesario para agregar una nueva tarea a la lista de ejecución. El tamaño de la TSS es fijo e igual a 0x68 bytes. El DPL del descriptor de la TSS es 3. En su código, describa el contenido de los campos de la TSS. Se debe comenzar con el siguiente código:

```
extern gdt_base
extern running
extern agregarAtras
extern next_free_gdt_entry
extern get_free_tss
```

Ejercicio 20

Cansado de que los Sistemas Operativos comunes se queden colgados y anden lentamente, Jaimito decide hacer su propio S.O. (pidiéndole permiso a su mama). Como no es exigente, hace un sistema que va a correr tan solo dos tareas concurrentemente, las cuales nunca van a terminar. Todos los servicios de *Jaimito S.O.* se atienden por medio de la interrupción 74. (*)

a) Describir qué estructuras se requieren mínimamente para administrar tareas y memoria en el *Jaimito S.O.* Considerar que se opera en modo protegido, con paginación activa y tareas en dos niveles.

b) Presentar una instanciación posible de todas las estructuras mencionadas en el item anterior. (**)

c) Escribir el código ASM de la interrupción de reloj que se encarga de realizar el intercambio entre las dos unicas tareas del sistema.

(*) Código ascii de la "J".
(* *) Estructuras como GDT, IDT, TSS, CR3, PageDirectory, PageTable, u otra que requiera.

Ejercicio 21

Frigga, enojada porque Thor le rompió la nariz a su hermanastro Loki con su martillo, decide retarlo mandándolo a su cuarto con solo tres porciones de pizza. Thor, enojado, decide continuar con el desarrollo de un primitivo sistema con extrañas funcionalidades dignas de un dios.

ThorSO permite correr una única tarea denominada de prioridad "suprema", y muchas tareas esclavas. El sistema ejecutará a la única tarea suprema todo el tiempo. Mediante interrupciones, se podrán cambiar los registros de la tarea "suprema" informando de eventos. Las tareas esclavas serán despertadas cada un determinado número de *ticks* de reloj indicados por la misma tarea esclava. Las tareas esclavas utilizarán todos los *ticks* de reloj que necesiten hasta terminar su trabajo. Cuando estas terminen generarán una nueva interrupción indicando que finalizaron (int 0x66), dejando en `eax` la cantidad de *ticks* en los que debe ser llamada nuevamente la tarea.

La única interrupción externa que le interesa capturar a la tarea "suprema" es la interrupción de teclado. Cuando esta se produce, de estar corriendo la tarea "suprema" se debe modificar el registro `ecx` por 1. Considerando que todas las tareas corren en anillo 3, incluyendo a la tarea "suprema".

a) Indicar que estructuras son necesarias para administrar el sistema.

b) Indicar el tipo y los campos de las entradas en la IDT de las tres interrupciones, int 0x66, reloj y teclado.

c) Implementar las tres interrupciones en ASM.

Nota: Si más de una tarea esclava debe ser despertada en el mismo *tick* de reloj, no se debe considerar un orden especial.